# ./Scanner/arp a output.txt

[irrelevant content]

# ./Scanner/autopep8 cmd screenshot.txt

[irrelevant content]

# ./Scanner/background.png

[binary content]

# ./Scanner/CacheDecorators.py

from typing import Callable  
  
  
def memorise(f: Callable) -> Callable:  
 """A decorator that saves the return value of a function to a cache,  
 so that when it's called again (with the same arguments!),  
 no calculations are made and the result is returned from the cache.  
 The arguments to the function must be hashable,  
 so no lists or sets, but it can get ints or strings or tuples.  
  
 A cached datum that's retreived 10 seconds or more after it was calculated,  
 will be calculated again.  
 The amount of seconds can be varied with `TIME\_LIMIT`.  
  
 Args:  
 f (function): the function to decorate and add a cache to.  
  
 Returns:  
 function: the decorated function with a cache.  
 """  
 memory = {}  
 from time import time as now  
 TIME\_LIMIT: int = 180 # in seconds  
  
 def wrapper(\*args):  
 try:  
 # Do I have the cached result of these arguments?  
 if args in memory:  
 # If I do have the cached result, calculate whether I'm still within the TIME\_LIMIT.  
 value, time\_created = memory[args]  
 if now() - time\_created < TIME\_LIMIT:  
 # If I am, just return the cached value.  
 return value  
 else:  
 # If not, calculate the value again.  
 result = f(\*args)  
 memory[args] = (result, now())  
 return result  
 else:  
 # If I don't have the cached result, calculate it.  
 result = f(\*args)  
 memory[args] = (result, now())  
 return result  
 except TypeError:  
 print("@memorise function cannot receive unhashable types! (e.g. lists, sets)")  
 raise  
  
 # Make `wrapper` inherit `f`'s properties.  
 wrapper.\_\_name\_\_ = f.\_\_name\_\_  
 wrapper.\_\_doc\_\_ = f.\_\_doc\_\_  
 return wrapper  
  
  
def one\_cache(f: Callable) -> Callable:  
 """A decorator that saves the return value of a function to a cache,  
 so that when it's called again,  
 no calculations are made and the result is returned from the cache.  
 This completely ignores arguments, so use it only when:  
 1. There are no arguments; or  
 2. Arguments don't matter; or  
 3. The function is expected to be called with the same arguments.  
  
 The function must return some non-None value.  
  
 Args:  
 f (function): the function to decorate and add a cache to.  
  
 Returns:  
 function: the decorated function with a cache.  
 """  
 # This is a list for it to be a reference type.  
 memory = [None]  
  
 def wrapper(\*args):  
 # If I have cache, return it.  
 if memory[0] is not None:  
 return memory[0]  
 # If I don't have cache, call the function.  
 memory[0] = f(\*args)  
 if memory[0] is None:  
 raise ValueError("A @one\_cache function cannot return None!")  
 return memory[0]  
  
 # Make `wrapper` inherit `f`'s properties.  
 wrapper.\_\_name\_\_ = f.\_\_name\_\_  
 wrapper.\_\_doc\_\_ = f.\_\_doc\_\_  
 return wrapper  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module provides two cache-related function decorators.")  
 print("One is a multiargument cache with time limit, called @memorise.")  
 print("The other is a single-entry no-time-limit cache, called @one\_cache.")

# ./Scanner/colors.py

# SGR color constants  
# rene-d 2018  
# from  
# https://gist.githubusercontent.com/rene-d/9e584a7dd2935d0f461904b9f2950007/raw/e2e58ccf955475d8066338a4e538c52debc06a06/colors.py  
  
class Colors:  
 """ ANSI color codes """  
 BLACK = "\033[0;30m"  
 RED = "\033[0;31m"  
 GREEN = "\033[0;32m"  
 BROWN = "\033[0;33m"  
 BLUE = "\033[0;34m"  
 PURPLE = "\033[0;35m"  
 CYAN = "\033[0;36m"  
 LIGHT\_GRAY = "\033[0;37m"  
 DARK\_GRAY = "\033[1;30m"  
 LIGHT\_RED = "\033[1;31m"  
 LIGHT\_GREEN = "\033[1;32m"  
 YELLOW = "\033[1;33m"  
 LIGHT\_BLUE = "\033[1;34m"  
 LIGHT\_PURPLE = "\033[1;35m"  
 LIGHT\_CYAN = "\033[1;36m"  
 LIGHT\_WHITE = "\033[1;37m"  
 BOLD = "\033[1m"  
 FAINT = "\033[2m"  
 ITALIC = "\033[3m"  
 UNDERLINE = "\033[4m"  
 BLINK = "\033[5m"  
 NEGATIVE = "\033[7m"  
 CROSSED = "\033[9m"  
 END = "\033[0m"  
 # cancel SGR codes if we don't write to a terminal  
 if not \_\_import\_\_("sys").stdout.isatty():  
 for \_ in dir():  
 if isinstance(\_, str) and \_[0] != "\_":  
 locals()[\_] = ""  
 else:  
 # set Windows console in VT mode  
 if \_\_import\_\_("platform").system() == "Windows":  
 kernel32 = \_\_import\_\_("ctypes").windll.kernel32  
 kernel32.SetConsoleMode(kernel32.GetStdHandle(-11), 7)  
 del kernel32  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 for i in dir(Colors):  
 if i[0:1] != "\_" and i != "END":  
 print("{:>16} {}".format(i, getattr(Colors, i) + i + Colors.END))

# ./Scanner/CommandLineStyle.py

import os  
from import\_handler import ImportDefence  
from time import sleep  
with ImportDefence():  
 from scapy.config import conf  
 from scapy.sendrecv import sr1  
 from scapy.layers.inet import IP  
  
 from pygments import highlight, lexers, formatters  
 from json import dumps  
  
  
def cmdtitle(\*s, sep=''):  
 os.system(f'title {sep.join(s)}')  
  
  
def cmdcolor(c):  
 os.system(f'color {str(c).zfill(2)}')  
  
  
def print\_dict(x: dict) -> None:  
 """Prints a python dictionary using JSON syntax and console colouring.  
  
 Args:  
 x (dict): the dictionary to print.  
 """  
 formatted\_json = dumps(x, sort\_keys=False, indent=4)  
 colorful\_json = highlight(  
 formatted\_json,  
 lexers.JsonLexer(),  
 formatters.TerminalFormatter()  
 )  
 print(colorful\_json)  
  
  
def remove\_scapy\_warnings():  
 """Removes the "MAC address not found, using broadcast" warnings thrown by scapy.  
 These warnings occur when a packet is sent to an IP (layer 3) address, without an Ethernet (layer 2) MAC address,  
 and such an address cannot be found using ARP. Scapy thus uses the broadcast MAC instead.  
 """  
 conf.warning\_threshold = 1\_000\_000 # Time between warnings of the same source should be infinite (many seconds).  
 for \_ in range(3):  
 try:  
 sr1(IP(dst="255.255.255.255"), verbose=0, timeout=0.001)  
 except PermissionError:  
 input("Failure to send packets <IP dst=broadcast>.\nIf you're sure you've got everything correct, press any key to continue. . .")  
 return  
 sleep(0.01)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module is responsible for styling the CMD or console.")  
 print("It can change the title of the CMD window,")  
 print("and the colour of the text.")  
 print("A logical addition is the remover of scapy warnings, since all they do is clutter the CMD.")  
 print("Another logical addition is `print\_dict`, that turns a boring dictionary into a colourful JSON on the CMD.")

# ./Scanner/db.py

from import\_handler import ImportDefence  
with ImportDefence():  
 import sqlite3  
  
CREATE\_INFORMATION\_TABLE = """CREATE TABLE IF NOT EXISTS `information` (  
 `name` VARCHAR(49) NOT NULL,  
 `Description` TEXT,  
 `Time` FLOAT,  
 `Reward` VARCHAR(100),  
 `Certainty` TINYINT unsigned,  
 `Safety` TINYINT unsigned,  
 `Mode` INT,  
 `repeats` BOOLEAN,  
 PRIMARY KEY (`name`)  
);"""  
INSERT\_INFORMATION = "INSERT INTO information VALUES(?, ?, ?, ?, ?, ?, ?, ?)"  
GET\_ALL\_SCANS = "SELECT name FROM information WHERE mode=1"  
GET\_ALL\_ANALYSES = "SELECT name FROM information WHERE mode=0"  
SELECT\_SPECIFIC\_SCAN = "SELECT \* FROM information WHERE name=?"  
PATH = "scans.db"  
INFORMATION\_DATA = [  
 # name: str, description: longstr, time: float, reward: str, certainty: int, safety: int, mode: int, repeats: bool  
 # NOTE: `mode` is reserved for future use. # UPDATE: `mode` is used to distinguish screens -- 1 is Scan Screen; 0 is Know Screen.  
 # ("", "", 0.1, "", 0, 0, 0, True),  
 ("ICMP Sweep", "Sends a ping-echo-request packet, using the ICMP protocol, to all IP addresses in the local network. <IP dst=addresses\_in\_network> <ICMP type=8 (echo-request)>", 13, "IP Addresses of Active Devices", 75, 30, 1, True),  
 ("ARP Sweep", "Sends a who-has packet, using the ARP protocol, to all IP addresses in the local network. <Ether> <ARP pdst=addresses\_in\_network op=1 (who-has)>", 9, "MAC & IP addresses of active devices", 95, 10, 1, True),  
 ("Live ICMP", "Continually sends ping-echo-request packets, using the ICMP protocol, in two channels:\n\* Discovery -- to find new devices;\n\* Connection -- to check whether the known devices have disconnected. <IP dst=addresses\_in\_network/known\_addreses> <ICMP type=8 (echo-request)>", 0, "IP and Connection Status of Devices", 80, 40, 1, False),  
 ("Live ARP", "\*\*Step 1:\*\* executes `arp -a` to see which addresses are already saved on the device's lookup table.\n\*\*Step 2:\*\* Listens indefinitely to who-has/is-at packets, using the ARP protocol.\nCompletely silent -- sends no packets, just reads those that are received. <Ether> <ARP op=1 (who-has) or op=2 (is-at)>", 0, "MAC & IP addresses of active devices", 99, 100, 1, False),  
 ("Device Profile", "Log all the packets that were silently sniffed.", 0, "The packets are already saved, you're just viewing them.", 99, 100, 0, True),  
 ("OS-ID", "Identifies the Operating System (OS) of all known devices. This uses techniques called TCP/IP stack fingerprinting, and inspects the IP:TTL and TCP:window. This is very close to guessing, so be doubtful of the results.\nIt sends no packets by itself (i.e. it's safe),\nbut looks at the packets received from other (possibly unsafe) scans, or any packets that were sniffed silently.", 0, "Probable Operating System of devices", 30, 100, 0, False),  
 ("TCP Ports", "Transmission Control Protocol (TCP) Half-Open Port Scan, also known as TCP SYN Stealth Scan. A usual TCP 3-way handshake is (SYN) - (SYN+ACK) - (ACK); this scan sends a SYN packet, and waits for a SYN+ACK packet, but never completes the handshake. It asks for an \*\*IP address\*\* to direct the scan at, an amount of \*\*repeats\*\* (if it's found open once, that's enough), and a \*\*port range\*\* (by default from 0 to 1024). <IP dst=specific\_ip> <TCP SYN>", 33, "Open TCP ports on a device", 90, 80, 1, True),  
 # ("UDP Ports", "User Datagram Protocol (UDP) Scanning, can detect some closed ports, because they will respond with an ICMP Port Unreachable error packet. <IP dst=specific\_ip> <UDP> -> <IP> <ICMP type=3 (unreachable)>", 0.1, "", 0, 0, 1, True),  
 # ("woo!", "", 0.1, "", 0, 0, 1, True),  
 ("Public Address", "Gets the outside IP address of the router, using `https://api.ipify.org`. <IP> <TCP> <HTTPS GET / HTTP/2 [Host: api.ipify.org]>", 1.3, "Public IP address of the router", 97, 90, 1, True),  
 ("Traceroute", "Using ICMP and IP:ttl, find the route to a distant device.", 0.1, "IP addresses of all devices in the route", 0, 0, 1, True),  
 ("Reveal Myself", "This is one side of the Device Discovery ability of this software. The other side, the listener, is active automatically. Allows other devices that are running this software to discover that I'm running it too.\nThis doesn't let me discover others, if they want to remain hidden, it just reveals myself.\nDiscovering other devices is automatically started when the software starts, and it doesn't damage your hiddenness.\nImplementation-wise, this sends out broadcast UDP packets, continuously. <Ether> <IP dst=broadcast> <UDP> <Raw: name, OS, other identifying information>", 0, "You gain nothing on this device, but you gain knowledge about 'Who's running this software?' on other devices.", 95, 0, 1, False),  
 ("Log Packets", "Log all the packets that were silently sniffed.", 0, "The packets are already saved, you're just viewing them.", 99, 100, 0, True),  
 ("Vendor Mapping", "Maps each currently known MAC address of other devices in the network,\nto a vendor or manufacturer, using a website (like an API): `hwaddress.com`.\nAn organizationally unique identifier (OUI) is a 24-bit number that uniquely identifies a vendor, manufacturer, or other organization, bought from IEEE;\ncan be found in the MAC address in the first 3 octets.\nIf a new MAC address is found, run the analysis again. <IP> <TCP> <HTTP GET /q=mac>", 5, "The manufacturer's name for each network card.", 97, 90, 0, True)  
]  
  
  
def get\_information\_about\_scan(name: str) -> tuple[str, str, float, str, int, int, int, int]:  
 """Gets information about a scan or analysis, from the SQL database.  
 Takes in the identificator (name) of the action,  
 runs an SQL Select query on the database,  
 and returns a tuple containing:  
 \* name: str, the same string that was given as an argument, but from the database.  
 \* description: longstr, the description of the scan.  
 \* time: float, the rough amount of time for the scan to execute (per repeat).  
 \* reward: str, the information you gain from running this scan.  
 \* certainty: int, how certain (percentage to 100) are you that the data are correct.  
 \* safety: int, how safe (undetectable) is this scan, as a percentage to 100.  
 \* mode: int, whether this scan runs on Scan Screen (1) or an analysis from Know Screen (0).  
 \* repeats: bool, whether this scan can be repeated. This is actually returned as an int, with 1=True and 0=False.  
  
 Args:  
 name (str): the name of the scan/analysis to retrieve information about.  
  
 Raises:  
 FileNotFoundError: if the table (INSIDE the database file) was not found.  
  
 Returns:  
 tuple[str, str, float, str, int, int, int, int]: the entry of this scan. Values explained above.  
 """  
 connection = sqlite3.connect(PATH)  
 cursor = connection.cursor()  
 try:  
 cursor.execute(SELECT\_SPECIFIC\_SCAN, (name, ))  
 except sqlite3.OperationalError:  
 raise FileNotFoundError("SQL table 'information' was not found.")  
 result = cursor.fetchone()  
 connection.close()  
 return result  
  
  
def get\_scans() -> list[str]:  
 """Gets all the scans from the SQL database.  
 A scan is an action (entry) with `mode=1`.  
  
 Raises:  
 FileNotFoundError: if the SQL table (inside the file) was not found.  
  
 Returns:  
 list[str]: a list of the scans' names.  
 """  
 connection = sqlite3.connect(PATH)  
 cursor = connection.cursor()  
 try:  
 cursor.execute(GET\_ALL\_SCANS)  
 except sqlite3.OperationalError:  
 raise FileNotFoundError("SQL table 'information' was not found.")  
 # a list of tuples is returned, where each tuple has only a string within  
 # (at index 0).  
 result = cursor.fetchall()  
 # Convert it to a list of strings. list[tuple[single str]] -> list[str]  
 result = [item[0] for item in result]  
 connection.close()  
 return result  
  
  
def get\_analyses() -> list[str]:  
 """Gets all the analyses from the SQL database.  
 An analysis is an action (entry) with `mode=0`.  
  
 Raises:  
 FileNotFoundError: if the SQL table (inside the file) was not found.  
  
 Returns:  
 list[str]: a list of the analyses' names.  
 """  
 connection = sqlite3.connect(PATH)  
 cursor = connection.cursor()  
 try:  
 cursor.execute(GET\_ALL\_ANALYSES)  
 except sqlite3.OperationalError:  
 raise FileNotFoundError("SQL table 'information' was not found.")  
 # a list of tuples is returned, where each tuple has only a string within  
 # (at index 0).  
 result = cursor.fetchall()  
 # Convert it to a list of strings. list[tuple[single str]] -> list[str]  
 result = [item[0] for item in result]  
 connection.close()  
 return result  
  
  
def init():  
 """Recreates the database,  
 using the internal Python list `INFORMATION\_DATA`.  
 Do not use this function in the code.  
 It is used only in `refresh\_db.bat`,  
 through this `.py` file's main clause.  
 """  
 if len(INFORMATION\_DATA) == 0:  
 raise ValueError("This is a distribution build, it cannot recreate the database. If you're stuggling, re-install the software.")  
 connection = sqlite3.connect(PATH)  
 cursor = connection.cursor()  
 try:  
 cursor.execute(CREATE\_INFORMATION\_TABLE)  
 cursor.executemany(INSERT\_INFORMATION, INFORMATION\_DATA)  
 except sqlite3.OperationalError:  
 print("An error occured.")  
 raise  
 except sqlite3.IntegrityError:  
 print("Unique constaint failed.")  
 connection.commit()  
 connection.close()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 init()  
 print(get\_information\_about\_scan('ICMP Sweep'))  
 for scan in get\_scans():  
 print(f"Scan: {scan}")  
 for analysis in get\_analyses():  
 print(f"Analysis: {analysis}")

# ./Scanner/Do TCP Scan.bat

@echo off  
title TCP Scan  
color 0A  
:scan  
py -c "from scans.TCP import main; main()"  
pause  
goto scan

# ./Scanner/error log.txt

[irrelevant content]

# ./Scanner/example.scan

[binary content]

# ./Scanner/exe.py

\_\_author\_\_ = 'Shaked Dan Zilberman'  
  
from import\_handler import ImportDefence  
with ImportDefence():  
 import requests  
 import ipaddress  
 import scapy  
 import networkx  
 import numpy  
 import scipy  
 import PyQt5  
 import kivy  
 import tkinter  
 import markdown  
 import win32api  
 import re  
kivy.require('2.1.0')  
  
from NetworkStorage import \*  
from register import Register  
from PacketSniffer import PacketSniffer  
from threading import Thread, enumerate as enumerate\_threads  
from SimpleScan import simple\_scan  
from CommandLineStyle import cmdcolor, cmdtitle, remove\_scapy\_warnings  
from scans.PublicAddress import public\_address\_action  
from scans.ARP import scan\_ARP, scan\_ARP\_continuous  
from scans.ICMP import scan\_ICMP, scan\_ICMP\_continuous  
from scans.TCP import port\_scan\_TCP  
from scans.TraceRouter import traceroute  
from scans.Discovery import DeviceDiscoveryListener, reveal\_myself  
from analyses.OS\_ID import operating\_system\_fingerprinting  
from analyses.DeviceProfile import device\_profile  
from analyses.LogPackets import log\_packets  
from analyses.VendorMapping import vendor\_mapping  
from time import sleep  
import os  
import sys  
from gui.KivyFonts import add\_fonts  
from gui.StartApp import start\_tk, start\_kivy  
from gui.dialogs import PopupManager, get\_string  
  
  
def keep\_resolving\_storage():  
 def \_resolver():  
 from globalstuff import terminator  
 sleep(7)  
 while not terminator.is\_set():  
 sleep(5)  
 NetworkStorage().\_resolve()  
 # print(len(NetworkStorage()), G.copy())  
 from gui.Screens.KnowScreen import update\_know\_screen  
 update\_know\_screen(NetworkStorage())  
 from gui.Diagrams import Diagrams  
 Diagrams().update()  
 sys.exit()  
  
 \_resolver.\_\_name\_\_ = 'IntervalThread'  
 Thread(target=\_resolver).start()  
  
  
def register\_scans():  
 """Registers the scans into `Register()` dictionary."""  
 r = Register()  
 r["ICMP Sweep"] = simple\_scan(scan\_ICMP, 3)  
 r["ARP Sweep"] = simple\_scan(scan\_ARP, 3)  
 r["Live ICMP"] = lambda: scan\_ICMP\_continuous(  
 NetworkStorage()['ip'],  
 ipconfig()["All Possible Addresses"],  
 compactness=2  
 ), True  
 r["Live ARP"] = scan\_ARP\_continuous, True  
 r["TCP Ports"] = port\_scan\_TCP  
 r["OS-ID"] = operating\_system\_fingerprinting, True  
 r["Public Address"] = public\_address\_action  
 r["Traceroute"] = lambda: traceroute(get\_string("IP Destination", "Select the IP address:"))  
 r["Log Packets"] = log\_packets  
 r["Device Profile"] = device\_profile  
 r["Reveal Myself"] = reveal\_myself  
 r["Vendor Mapping"] = vendor\_mapping  
  
  
def main():  
 print("Loading...")  
 remove\_scapy\_warnings()  
 os.system('cls')  
 cmdcolor("0A")  
 print("Attempting to connect to an network-card interface...")  
 ipconfig()  
 cmdtitle(  
 "Network Scanner - ",  
 ipconfig()["Interface"],  
 " at ",  
 ipconfig()["IPv4 Address"]  
 )  
 from testing.tests import test  
 test()  
  
 NetworkStorage()  
 ipconfig.cache["All Possible Addresses"] = get\_all\_possible\_addresses()  
 from NetworkStorage import router, here  
 from globalstuff import G  
 NetworkStorage().add(router, here)  
 G.add\_node(router)  
  
 register\_scans()  
  
 # GUI initialisation  
 add\_fonts()  
 PopupManager() # starts the popup thread  
 DeviceDiscoveryListener() # starts the discovery thread  
   
 # Start tk (on main thread) and kivy (on different thread) and  
 # `NetworkStorage().\_resolve` (on a third thread)  
 PacketSniffer()  
 keep\_resolving\_storage()  
 Thread(target=start\_kivy).start()  
 start\_tk()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # sys.stdout = sys.stderr = open(os.devnull, "w")  
 try:  
 main()  
 except Exception as err:  
 from datetime import datetime  
 with open('error log.txt', 'a', encoding='utf-8') as f:  
 f.write(f'An exception occurred - {err}\n{datetime.now()}')  
 raise

# ./Scanner/favicon.ico

[binary content]

# ./Scanner/favicon.png

[binary content]

# ./Scanner/files.py

from time import sleep  
from import\_handler import ImportDefence  
with ImportDefence():  
 import tkinter.filedialog as dialogs  
  
import files\_cryptography  
  
  
def get\_password():  
 from gui.dialogs import get\_string  
 return get\_string("File Password", "Enter password:")  
  
  
def exporter():  
 filename = dialogs.asksaveasfilename(  
 title="Save As",  
 defaultextension=".scan",  
 filetypes=(("Scan files", "\*.scan"), ("All files", "\*.\*")),  
 )  
 print("Exporting to", filename)  
 if filename == "":  
 return  
 builder = ScanFileBuilder()  
  
 from ScanID import get\_scan\_id  
 scan\_id = get\_scan\_id().encode()  
 builder.add(scan\_id)  
  
 from NetworkStorage import NetworkStorage  
 network\_entities = [str(x).encode() for x in NetworkStorage()] # \*\*\*\*\*\*\*\*\* change this to `entity.encode()` and implement that method  
 builder.add\_many(network\_entities)  
  
 from register import Register  
 scan\_history = [  
 int(timestamp).to\_bytes(4, byteorder='big') + str(name).encode() + int(duration).to\_bytes(3, byteorder='big')  
 for (name, timestamp, duration) in Register().get\_history()  
 ]  
 builder.add\_many(scan\_history)  
  
 builder.set\_password(get\_password())  
 builder.write\_to(filename)  
 print("Done writing")  
 return filename  
  
  
def importer():  
 filename = dialogs.askopenfilename(  
 title="Open",  
 filetypes=(("Scan files", "\*.scan"), ("All files", "\*.\*")),  
 )  
 print("Importing from", filename)  
  
 builder = ScanFileBuilder()  
 builder.set\_password(get\_password())  
 result = builder.parse(filename)  
 # print(result)  
  
 from ScanID import parse\_scan\_id, get\_scan\_id  
 scan\_id = result["scan\_id"]  
 same\_network = scan\_id == get\_scan\_id()  
 same\_network\_message = "\nYou're currently in the same connection (computer, interface, network) as the scan file!" if same\_network else "\n"  
  
 scan\_id = parse\_scan\_id(scan\_id)  
  
 entities = result["network\_entities"]  
 entities = '\n'.join(entities)  
  
 history = result["scan\_history"]  
 from datetime import datetime  
  
 def format\_timestamp(t: int) -> str:  
 return datetime.fromtimestamp(t).strftime('%Y-%m-%d %H:%M:%S')  
  
 def format\_duration(t: int) -> str:  
 return f'for {t}[s]' if t > -1 else f'indefinitely'  
 history = [format\_timestamp(timestamp) + f' {name} {format\_duration(duration)}.' for (timestamp, name, duration) in history if name != '']  
 history = '\n'.join(history)  
  
 return f"""{scan\_id}{same\_network\_message}\n\n{entities}\n\n{history}"""  
  
  
def encrypt(message: bytes, password: str) -> bytes:  
 return files\_cryptography.password\_encrypt(message, password)  
  
  
def decrypt(token: bytes, password: str) -> bytes:  
 # decrypter\_with\_timeout = SetTimeout(files\_cryptography.password\_decrypt, timeout=10)  
 # is\_done, is\_timeout, erro\_message, results = decrypter\_with\_timeout(token, password)  
 # return results if is\_done else ""  
 return files\_cryptography.password\_decrypt(token, password)  
  
  
class ScanFileBuilder:  
 HEADER = b"SHZILSCAN"  
 SEP = b"\n"  
 COMMA = b","  
  
 def \_\_init\_\_(self):  
 self.parts = [self.HEADER]  
 self.password = None  
  
 def add(self, part: bytes):  
 self.parts.append(part)  
  
 def add\_many(self, parts):  
 self.parts.append(self.COMMA.join(parts))  
  
 def write\_to(self, path: str):  
 assert self.password is not None  
 with open(path, "xb") as f:  
 content = self.SEP.join(self.parts)  
 content = encrypt(content, self.password) if self.password != '' else content  
 f.write(content)  
  
 def set\_password(self, password: str):  
 self.password = password  
  
 def parse(self, path: str):  
 assert self.password is not None  
 self.parts = [b""] \* 3  
 with open(path, "rb") as f:  
 content = f.read()  
 content = decrypt(content, self.password) if self.password != '' else content  
 if content == b'':  
 raise ValueError(  
 "Couldn't decrypt the file. The password is wrong"  
 )  
 content = content.split(self.SEP)  
 if content[0] != self.HEADER:  
 if not path.endswith('.scan'):  
 raise ValueError(  
 "Invalid file format. The extension is also wrong."  
 )  
 # or wrong password  
 raise ValueError("Invalid file format, or wrong password.")  
  
 self.parts = content[1:]  
 scan\_id, entities, history, \*rest = self.parts  
 if len(rest) > 0:  
 def \_decode\_bytes(b):  
 try:  
 return b.decode('utf-8')  
 except UnicodeDecodeError:  
 return b.hex()  
 raise ValueError("The file has more content than expected: " + \_decode\_bytes(rest))  
  
 return {  
 "scan\_id": scan\_id.decode(),  
 "network\_entities": [x.decode() for x in entities.split(self.COMMA)], # \*\*\*\*\*\*\* change this to NetworkEntity.decode(x) and implement that method  
 "scan\_history": [(x[:4], x[4:-3].decode(), x[:-3]) for x in history.split(self.COMMA)]  
 }  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module is used for saving scans as files,")  
 print("and opening previously saved scans.")

# ./Scanner/files\_cryptography.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from Crypto.Cipher import AES  
 from Crypto.Hash import SHA256  
 from Crypto.Util.Padding import pad, unpad  
  
  
def password\_encrypt(message: bytes, password: str) -> bytes:  
 return Cipher\_CBC(password).encrypt(message)  
  
  
def password\_decrypt(token: bytes, password: str) -> bytes:  
 try:  
 return Cipher\_CBC(password).decrypt(token)  
 except ValueError as e:  
 print(e)  
 return b''  
  
  
class Cipher\_CBC:  
 def \_\_init\_\_(self, password: str):  
 """This function initializes the cipherer.  
  
 Args:  
 password (str): The password used to encrypt and decrypt the data.  
 """  
 hashed = SHA256.new(password.encode()).digest()  
 key, iv = hashed[:16], hashed[16:]  
  
 self.\_encryptor = self.\_decryptor = AES.new(key, AES.MODE\_CBC, iv)  
  
 def encrypt(self, msg: bytes) -> bytes:  
 """This function encrypts the message.  
  
 Args:  
 msg (bytes): The message to encrypt.  
  
 Returns:  
 bytes: The encrypted message.  
 """  
 return self.\_encryptor.encrypt(pad(msg, AES.block\_size))  
  
 def decrypt(self, ciphertext: bytes) -> bytes:  
 """This function decrypts the message.  
  
 Args:  
 ciphertext (bytes): The ciphertext to decrypt.  
  
 Returns:  
 bytes: The decrypted message.  
 """  
 return unpad(self.\_decryptor.decrypt(ciphertext), AES.block\_size)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This is a module extending `files.py` with encryption.")  
 print("It supplies `files.py` with two methods: password\_encrypt and password\_decrypt.")  
 print("The internal implementation is irrelevant for `files.py`, and abstracted away.")  
 print("\n\nExample: message=\"Hello, world!\", password=\"A123\", then:")  
 message = "Hello, world!"  
 password = "A123"  
 encrypted = password\_encrypt(message.encode(), password)  
 print("Encrypted (HEX):", ' '.join(format(x, '02x').upper()  
 for x in encrypted))  
 decrypted = password\_decrypt(encrypted, password)  
 print("Decrypted:", decrypted)  
 print("Example: enter file path:")  
 try:  
 with open(input(), 'rb') as f:  
 print(  
 "Decrypted:",  
 password\_decrypt(  
 f.read(),  
 input("Password: ")  
 )  
 )  
 except Exception as e:  
 print(e)  
 input()

# ./Scanner/generally\_unimportant.txt

[irrelevant content]

# ./Scanner/globalstuff.py

import networkx  
from threading import Event  
  
# --- The Graph ---  
G = networkx.empty\_graph()  
  
# --- Inter-thread Communication ---  
terminator = Event()  
  
# --- GUI Global Values ---  
is\_kivy\_running = True  
  
  
# --- Design Settings ---  
bg\_color = (0, 0, .01) # tuple[float]: rgb  
fg\_color = (0.023, 0.92, 0.125) # tuple[float]: rgb  
button\_column\_background = [0.1, 1, 0.3, 1] # list[float]: rgba  
DIAGRAM\_DIMENSIONS = (300, 300) # tuple[int]: width, height  
# float; under `HoverReplace`, `new\_text\_size = HOVER\_REPLACE\_FACTOR \*  
# old\_text\_size`, e.g. fontsizeof("Information") = 0.75 \* fontsizeof("ℹ").  
HOVER\_REPLACE\_FACTOR = 0.75  
DIAGRAM\_POINT\_RADIUS = 5 # int: px  
BUTTON\_COLUMN\_FONT\_SIZE = 24 # int: px  
# tuple[float]: rgba; used as overlay, do not set alpha=1, because that  
# will hide the text.  
SCAN\_HIGHLIGHT = (0, 1, 0, 0.2)  
ANALYSIS\_HIGHLIGHT = (0, 0.2, 0.8, 0.2)  
OPERATION\_BUTTON\_FONT\_SIZE = 30 # int: px  
OPERATION\_BUTTON\_BACKGROUND = [0.8, 0.8, 0.8, 1] # list[float]: rgba  
# int: px; the padding used by the kivy diagram from the top, to avoid  
# hiding the title by overlapping it.  
TITLE\_HEIGHT = 70  
DIAGRAM\_SCALE = 1 / 2.3 # float  
PAGES\_BACKGROUND = [0, 0, 0, 0] # list[float]: rgba  
TITLE\_FONT\_SIZE = 30 # int: px  
GREEN = '00ff00' # str: hex color  
UNDER\_DIAGRAM\_FONT\_SIZE = 30 # int: px  
RIGHT\_COLUMN\_WIDTH = 300 # int: px; in Scan screen.  
SAVE\_BUTTONS\_HOVER\_BACKGROUND = [0, 0, 1, 1] # list[tuple]: rgba  
SAVE\_BUTTONS\_FONT\_SIZE = 50 # int: px  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module provides global values to the other modules.")  
 print("It does not depend on any other module, but on some libraries.")  
 print("Three kinds of global variables:")  
 print(" universal values between threads;")  
 print(" dynamic information custom objects;")  
 print(" hardcoded design settings.")

# ./Scanner/hostify.bat

@echo off  
color 0A  
  
REM if there's no command line argument, go to the `empty` label.  
if [%1]==[] goto empty  
  
:assign  
REM if there is a command line argument, put its value in %address%  
set address=%\*  
goto body  
  
:empty  
REM asks the user to input an IP address, store answer in %address%  
set /p "address=Enter IP Address: "  
echo.  
  
:body  
echo.  
  
(for %%a in (%address%) do (  
 REM method 1 calls `nslookup %address%` and selects only the line which starts with "Name:".  
 echo --- %%a ---  
 echo Method 1: C:\Windows\System32\nslookup.exe  
 FOR /F "Tokens=1,\* Delims==" %%A in (  
 'nslookup %%a ^| FINDSTR "Name:"'  
 ) DO (echo %%A)  
  
 echo.  
 REM method 2 calls this Python snippet which imports socket and calls socket.gethostbyaddr('%address%')  
 echo Method 2: Python.Socket.GetHostByAddr  
 py -c "import socket; print('Name: ' + socket.gethostbyaddr('%%a')[0])"  
  
 echo.  
 echo.  
))  
  
  
pause

# ./Scanner/hostify.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from subprocess import CalledProcessError, check\_output as read\_command, DEVNULL  
  
 from socket import gethostbyaddr as hostify\_base  
 from socket import herror as hostify\_error1  
 from socket import gaierror as hostify\_error2  
  
from CacheDecorators import memorise  
from PrintingContexts import NoPrinting  
from util import threadify  
  
  
@memorise  
def hostify(address: str):  
 """This function turns an IPv4 address to a host name using one of these methods:  
 1. Calling `>nslookup` with that address. If that fails,  
 2. Using `socket.gethostbyaddr` function. If that fails,  
 3. Returns "Unknown" since all the methods failed.  
  
 Args:  
 address (str): the IPv4 address to turn into a host.  
  
 Returns:  
 str: the host name.  
 """  
 host = "Unknown"  
  
 def use\_hostify\_base(address):  
 try:  
 # print("Method: socket.gethostbyaddr", end=' -- ')  
 return hostify\_base(address)[0]  
 except (hostify\_error1, hostify\_error2):  
 return "Unknown"  
  
 # First method -> nslookup  
 # If first method failed, second method -> socket.gethostbyaddr  
 try:  
 with NoPrinting():  
 lines = read\_command(['nslookup', address], stderr=DEVNULL).decode(  
 encoding='utf-8', errors='ignore').split('\n')  
 for line in lines:  
 if line.strip().startswith('Name:'):  
 host = line[len("Name:"):].strip()  
 # print("Method: nslookup", end=' -- ')  
 break  
 else:  
 host = use\_hostify\_base(address)  
 except CalledProcessError:  
 host = use\_hostify\_base(address)  
 # print("Hostified:", host)  
 return host  
  
  
hostify\_sync = threadify(hostify, silent=True)

# ./Scanner/ICMP continuous opacity algorithm.txt

[irrelevant content]

# ./Scanner/import\_handler.py

class ImportDefence:  
 """This context manager ensures all `import` statements were successful,  
 and if some weren't, it attempts a `pip install`.  
  
 Source: https://raw.githubusercontent.com/ShZil/network-utilities/main/Scanner/import\_handler.py  
  
 This function handles a ModuleNotFoundError,  
 attempting to install the not-found module using `pip install`,  
 and restarting the script / instructing the user.  
  
 Line-by-line breakdown of the ModuleNotFoundError handler:  
 - necessary imports: sys, os, subprocess  
  
 ```py  
 import sys  
 from subprocess import check\_call as do\_command, CalledProcessError  
 import os  
 ```  
  
 - print the failure  
  
 ```py  
 print(f"Module `{err.name}` was not found. Attempting `pip install {err.name}`...\n")  
 ```  
  
 - try to pip install it  
  
 ```py  
 try:  
 do\_command([sys.executable, "-m", "pip", "install", err.name])  
 ```  
  
 - if failed, request manual installation  
  
 ```py  
 except CalledProcessError:  
 print(f"\\nModule `{err.name}` could not be pip-installed. Please install manually.")  
 sys.exit(1)  
 ```  
  
 - if succeeded, restart the script  
  
 ```py  
 argv = ['\"' + sys.argv[0] + '\"'] + sys.argv[1:]  
 os.execv(sys.executable, ['python'] + argv)  
 ```  
  
 \*\*Usage:\*\*  
 ```py  
 from import\_handler import ImportDefence  
  
 with ImportDefence():  
 import module1  
 from module2 import some\_function  
 ```  
 """  
  
 def \_\_init\_\_(self):  
 pass  
  
 def \_\_enter\_\_(self):  
 return self  
  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 import os  
 # If no ModuleNotFoundError occured, clear the screen and print and  
 # return to original script.  
 if exc\_val is None:  
 os.system('cls')  
 print("All imports were successful.")  
 return  
 # Otherwise, raise the exception.  
 try:  
 raise exc\_val  
 # If the exception is of type ModuleNotFoundError, handle it:  
 except ModuleNotFoundError as err:  
 import sys  
 from subprocess import check\_call as do\_command, CalledProcessError  
  
 to\_install = err.name  
 # Some modules have `pip install X` and `import Y`, where `X != Y`.  
 # These have to be added manually, since there's no pattern.  
 if 'win32' in to\_install:  
 to\_install = 'pywin32'  
 if to\_install == 'cv2':  
 to\_install = 'opencv-python'  
 if to\_install == 'Crypto':  
 to\_install = 'pycryptodome'  
 if to\_install == 'PIL':  
 to\_install = 'pillow'  
 if to\_install == '\_curses':  
 to\_install = 'windows-curses'  
 to\_install = to\_install.split('.')[0]  
  
 print(  
 f"Module `{err.name}` was not found. Attempting `pip install {to\_install}`...\n"  
 )  
 try:  
 do\_command(  
 [sys.executable, "-m", "pip", "install", to\_install]  
 )  
 except CalledProcessError:  
 if\_different = f"" if to\_install == err.name else f" (from `{err.name}`)"  
 print(  
 f"\nModule `{to\_install}`{if\_different} could not be pip-installed. Please install manually."  
 )  
 sys.exit(1)  
 argv = ['\"' + sys.argv[0] + '\"'] + sys.argv[1:]  
 os.execv(sys.executable, ['python'] + argv)  
 # If the exception is of type ImportError, log an error and keep raising it.  
 except ImportError:  
 print("You've misnamed your import. Check it.")  
 raise  
  
# possible add pip updating, using:  
# `python.exe -m pip install --upgrade pip`

# ./Scanner/ipconfig.py

from import\_handler import ImportDefence  
from CacheDecorators import one\_cache  
  
with ImportDefence():  
 from scapy.interfaces import get\_working\_ifaces  
 from scapy.config import conf  
 from subprocess import CalledProcessError, check\_output as read\_command  
  
  
def read\_ipconfig():  
 """Read the command `>ipconfig /all` from console and decode it to UTF-8 text.  
  
 Returns:  
 list[str]: the command's output as a list of lines  
  
 Raises:  
 subprocess.CalledProcessError: if subprocess.check\_output fails.  
 """  
 try:  
 return read\_command(['ipconfig',  
 '/all']).decode(encoding='utf-8',  
 errors='ignore').split('\n')  
 except CalledProcessError:  
 print(">ipconfig /all raised an error.")  
 raise  
  
  
def dictify(text: list[str] | str) -> dict:  
 """Turn `text` to a python dictionary.  
 The nested dictionary is created according to the following rules:  
 - loop over the lines.  
 - if the line isn't indented, inistalise a new subdictionary.  
 This is a new network interface, e.g. Ethernet, Wireless LAN, Bluetooth; or general info ("Windows IP Configuration").  
 - otherwise,  
 - if the line is formatted like "key . . . . : value", add this pair to the current active dictionary.  
 - otherwise, convert the pair to a (key, list) pair, and add the line's contents as a new value.  
 - if the value is empty, use an empty list to represent it.  
  
 Example:  
 ```r  
 Windows IP Configuration  
 Host Name . . . . . . . . . . . . : MyComputer-007  
 Primary Dns Suffix . . . . . . . :  
 Node Type . . . . . . . . . . . . : Hybrid  
 IP Routing Enabled. . . . . . . . : No  
 ```  
 &darr;&darr;&darr;  
 ```json  
 {  
 "Windows IP Configuration": {  
 "Host Name": "MyComputer-007",  
 "Primary Dns Suffix": "",  
 "Node Type": "Hybrid",  
 "IP Routing Enabled": "No"  
 }  
 }  
 ```  
 Another example:  
 ```r  
 Wireless LAN adapter Wi-Fi:  
 Media State . . . . . . . . . . . : Media disconnected  
 Connection-specific DNS Suffix . : local  
 Description . . . . . . . . . . . : Wireless-ABCDE  
 Physical Addresses. . . . . . . . : AB-CD-EF-01-02-03  
 AB-CD-EF-01-02-04  
 AB-CD-EF-01-02-05  
 DHCP Enabled. . . . . . . . . . . : Yes  
 Autoconfiguration Enabled . . . . : Yes  
 ```  
 &darr;&darr;&darr;  
 ```json  
 {  
 "Wireless LAN adapter Wi-Fi:": {  
 "Media State": "Media disconnected",  
 "Connection-specific DNS Suffix": "local",  
 "Description": "Wireless-ABCDE",  
 "Physical Addresses": [  
 "AB-CD-EF-01-02-03",  
 "AB-CD-EF-01-02-04",  
 "AB-CD-EF-01-02-05"  
 ],  
 "DHCP Enabled": "Yes",  
 "Autoconfiguration Enabled": "Yes"  
 }  
 }  
 ```  
  
  
  
 Args:  
 text (list[str]): the text to be converted. Expected to be from ipconfig or similar.  
  
 Returns:  
 dict: the text in dictionary format.  
  
 Raises:  
 IndexError: if the format is invalid.  
 """  
 if isinstance(text, str):  
 text = text.split('\n')  
 result = {} # The dictionary to be returned.  
 # The current interface whose configuration values are being decoded.  
 interface = None  
 # The current title, inside the interface, whose value/s are being decoded.  
 title = None  
 for line in text:  
 if line.strip() == '':  
 continue  
  
 if line[0].strip() != '':  
 # New interface found. Initialise dictionary.  
 interface = line.strip(": \r")  
 result[interface] = {}  
  
 else:  
 # Adding information to current `interface`.  
 if '. :' in line or (not line.startswith(" ") and ':' in line):  
 # New property (title).  
 key, value = line.split(':', 1)  
 title, value = key.strip(' .'), value.strip().replace("(Preferred)", "")  
 if value.strip() == "":  
 result[interface][title] = []  
 else:  
 result[interface][title] = value  
 else:  
 # Last property is a list, appending item.  
 value = line.strip().replace("(Preferred)", "")  
 if not isinstance(result[interface][title], list):  
 result[interface][title] = [result[interface][title]]  
 result[interface][title].append(value)  
 if len(result[interface][title]) == 1:  
 result[interface][title] = result[interface][title][0]  
 return result  
  
  
def ipconfig() -> dict:  
 """Get information from `>ipconfig /all`,  
 select the first interface with a Default Gateway (i.e. online),  
 return its information as a dictionary. Has cache.  
  
 Guaranteed keys:  
 ```  
 "IPv4 Address"  
 "Subnet Mask"  
 "Default Gateway"  
 ```  
  
  
 Returns:  
 dict: containing the following information:  
 ```  
 {  
 \*\*information["Windows IP Configuration"],  
 \*\*information["Interface with Gateway"],  
 'Interface': interface,  
 'Auto-Selected Interface': auto\_select\_interface(...)  
 }  
 ```  
  
 Raises:  
 RuntimeError: if no Default Gateway is found, meaning the computer is disconnected from the Internet.  
 """  
 if hasattr(ipconfig, 'cache'):  
 return ipconfig.cache  
  
 information = dictify(read\_ipconfig())  
 possible\_interfaces = [  
 interface for interface,  
 info in information.items() if 'Default Gateway' in info.keys()  
 ]  
  
 if len(possible\_interfaces) <= 0:  
 raise RuntimeError("Computer is not connected to Internet.")  
 elif len(possible\_interfaces) == 1:  
 selected = possible\_interfaces[0]  
 else:  
 for i, interface in enumerate(possible\_interfaces):  
 print(f" ({i}) {interface}")  
 selected = get\_interface\_safe(possible\_interfaces)  
  
 print("Interface:", selected)  
 auto\_selected\_interface = auto\_select\_interface(  
 information[selected]["IPv4 Address"])  
 data = {  
 \*\*information["Windows IP Configuration"],  
 \*\*information[selected],  
 'Interface': selected,  
 'Auto-Selected Interface': auto\_selected\_interface  
 }  
 ipconfig.cache = data  
 return data  
  
  
@one\_cache  
def get\_interface\_safe(possible):  
 while True:  
 try:  
 num = int(input("Select: "))  
 except ValueError:  
 print("Not a number")  
 continue  
  
 if not 0 <= num < len(possible):  
 print("Not in range")  
 continue  
  
 return possible[num]  
  
  
def auto\_select\_interface(ip: str):  
 """Automatically selects the interface whose IP matches the given value.  
 Uses the list given in `scapy.interfaces.get\_working\_ifaces()`.  
 Sets the `scapy.config.conf.iface` to the correct value.  
  
 Args:  
 ip (str): the IPv4 address of the correct interface.  
  
 Returns:  
 str: `str(scapy.config.conf.iface)`  
 """  
 for iface in get\_working\_ifaces():  
 if iface.ip == ip:  
 conf.iface = iface  
 return str(conf.iface)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 ipconfig()  
 from CommandLineStyle import print\_dict  
 print\_dict(ipconfig())

# ./Scanner/ip\_handler.py

from ipconfig import ipconfig  
from CacheDecorators import one\_cache  
  
\_\_author\_\_ = 'Shaked Dan Zilberman'  
  
  
def bitify(address: str) -> str:  
 """This method turns an IPv4 address (0.0.0.0 to 255.255.255.255) to a string of 0s and 1s.  
 Each part in the address turns to 8 binary digits, thus the result must always be 32 characters long.  
  
 Examples:  
 0.0.0.0 -> '00000000000000000000000000000000' (32 0s)  
 255.255.255.255 -> '11111111111111111111111111111111' (32 1s)  
 192.168.0.1 -> '11000000101010000000000000000001'  
  
 Args:  
 address (str): the address to be converted.  
  
 Returns:  
 str: a binary-looking string.  
 """  
 result = ''  
 if len(address.split('.')) != 4:  
 raise TypeError(  
 f"Given address ({address}) is not IPv4 - not composed of 4 dot-separated parts."  
 )  
 for part in address.split('.'):  
 try:  
 result += "{0:08b}".format(int(part, base=10))  
 except ValueError:  
 raise TypeError(  
 f"Given address ({address}) is not IPv4 -- part \"{part}\" is not a valid integer."  
 )  
 return result  
  
  
def unbitify(binary: str) -> str:  
 """This method turns an IPv4 address represented as a string of binary digits to the regular representation.  
 Each part in the address turns from 8 binary digits to 1-3 decimal digits.  
 Acts as the inverse of `bitify`.  
  
 Examples:  
 '00000000000000000000000000000000' (32 0s) -> 0.0.0.0  
 '11111111111111111111111111111111' (32 1s) -> 255.255.255.255  
 '11000000101010000000000000000001' -> 192.168.0.1  
  
 Args:  
 address (str): the address to be parsed.  
  
 Returns:  
 str: an IPv4 address.  
 """  
 result = ''  
 for byte in [binary[i:i + 8] for i in range(0, len(binary), 8)]:  
 result += str(int(byte, base=2))  
 result += '.'  
 return result.strip('.')  
  
  
def subnet\_address\_range(subnet\_mask: str, \*some\_addresses: tuple[str]):  
 """This function computes the address range of a subnet (i.e. the Network ID, with all (\*)s in the Device ID section).  
  
 Code explanation:  
  
 Define a wrapper for `bitify` which also converts the result to an integer. Apply it to subnet\_mask.  
 ```  
 bits = lambda address: int(bitify(address), base=2)  
 mask = bits(subnet\_mask)  
 ```  
  
 Compute the \*bitwise logical AND\* of each address and the mask.  
 ```  
 base = [bits(address) & mask for address in some\_addresses]  
 ```  
  
 Raise exceptions if no unique values are found / more than one unique value is found.  
  
 Find the network ID: using the bitwise AND result, zfill it up to 32 characters, and cut only the part with 1s in the mask.  
 ```  
 mask = format(mask, 'b')  
 network = format(base[0], 'b').zfill(32)[:mask.count('1')]  
 ```  
  
 Find the lowest and highest possible addresses by filling the empty space with 0s or 1s respectively.  
 ```  
 lowest = unbitify(network + ('0' \* mask.count('0')))  
 highest = unbitify(network + ('1' \* mask.count('0')))  
 ```  
  
 For each part of the address (i.e. `A.B.C.D`, then A, B, C, and D are parts), find the correct format.  
 ```  
 if low == high:  
 result += low  
 elif low == '0' and high == '255':  
 result += '\*'  
 else:  
 result += f"{low}-{high}"  
 ```  
  
 Args:  
 subnet\_mask (str): the subnet mask, `like 255.255.255.0`.  
 \*some\_addresses (str, str, str...): some example addresses of devices in the network. Must be at least one.  
  
 Raises:  
 TypeError: If no example addresses are given.  
 ValueError: If the example addresses belong to different networks.  
  
 Returns:  
 str: the address range. Note: this is not a valid IPv4 address, it uses (\*)s and (-)s in the Device ID portion.  
 """  
 def bits(address):  
 return int(bitify(address), base=2)  
 mask = bits(subnet\_mask)  
 base = [bits(address) & mask for address in some\_addresses]  
 base = list(set(base))  
 if len(base) == 0:  
 raise TypeError("No addresses were given besides the mask.")  
 if len(base) > 1:  
 raise ValueError("The addresses given fit different networks.")  
 mask = format(mask, 'b')  
 network = format(base[0], 'b').zfill(32)[:mask.count('1')]  
 lowest = unbitify(network + ('0' \* mask.count('0')))  
 highest = unbitify(network + ('1' \* mask.count('0')))  
 result = ''  
 for low, high in zip(lowest.split('.'), highest.split('.')):  
 if low == high:  
 result += low  
 elif low == '0' and high == '255':  
 result += '\*'  
 else:  
 result += f"{low}-{high}"  
 result += '.'  
 return result.strip('.')  
  
  
def base\_subnet\_address(subnet\_mask: str, \*some\_addresses: tuple[str]) -> str:  
 """This function computes the base address of a network (i.e. the Network ID, with all 0s in the Device ID section)  
  
 Code explanation:  
  
 Define a wrapper for `bitify` which also converts the result to an integer. Apply it to subnet\_mask.  
 ```  
 bits = lambda address: int(bitify(address), base=2)  
 mask = bits(subnet\_mask)  
 ```  
  
 Compute the \*bitwise logical AND\* of each address and the mask.  
 ```  
 base = [bits(address) & mask for address in some\_addresses]  
 ```  
  
 Raise exceptions if no unique values are found / more than one unique value is found.  
  
 Turn `base[0]` (the only unique value) to a binary string, zfill it to 32 characters, pass it to `unbitify`, and return.  
 ```  
 return unbitify(format(base[0], 'b').zfill(32))  
 ```  
  
 Args:  
 subnet\_mask (str): the subnet mask, `like 255.255.255.0`.  
 \*some\_addresses (str, str, str...): some example addresses of devices in the network. Must be at least one.  
  
 Raises:  
 TypeError: If no example addresses are given.  
 ValueError: If the example addresses belong to different networks.  
  
 Returns:  
 str: the base address. Note: this is a valid IPv4 address, the lowest in the network.  
 """  
 def bits(address):  
 return int(bitify(address), base=2)  
 mask = bits(subnet\_mask)  
 base = [bits(address) & mask for address in some\_addresses]  
 base = list(set(base))  
 if len(base) == 0:  
 raise TypeError("No addresses were given besides the mask.")  
 if len(base) > 1:  
 raise ValueError("The addresses given fit different networks.")  
 return unbitify(format(base[0], 'b').zfill(32))  
  
  
def subnet\_slash\_notation(subnet\_mask: str, router: str) -> str:  
 subnet\_mask = bitify(subnet\_mask)  
 count = subnet\_mask.count('1')  
 return f"{router}/{count}"  
  
  
@one\_cache  
def get\_all\_possible\_addresses() -> list[str]:  
 """This method calculates all the possible IPv4 addresses in the current subnet,  
 according to this device's IP address and the Subnet Mask, both from `ipconfig()`.  
  
 Returns:  
 list[str]: a list of IPv4 addresses, that are all the possible ones in the current network.  
 """  
 this\_device\_ip = ipconfig()["IPv4 Address"]  
 subnet\_mask = ipconfig()["Subnet Mask"]  
  
 this\_device\_ip, subnet\_mask = bitify(this\_device\_ip), bitify(subnet\_mask)  
 unique, mutual = subnet\_mask.count('0'), subnet\_mask.count('1')  
  
 base = this\_device\_ip[:mutual]  
  
 def binary(number):  
 return bin(number)[2:].zfill(unique)  
  
 # All possible addresses in binary look like `[mutual part to all in network][special identifier]`,  
 # i.e. base + binary representation of i (where i ranges from (0) to (2 ^ unique))  
 return [unbitify(base + binary(i)) for i in range(2 \*\* unique)]  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This is a module for doing calculations on IPv4 addresses.")

# ./Scanner/known.txt

[irrelevant content]

# ./Scanner/mac\_vendors.db

[binary content]

# ./Scanner/NetworkStorage.py

from import\_handler import ImportDefence  
with ImportDefence():  
 import re  
 import ipaddress  
 from queue import Queue  
  
from PrintingContexts import JustifyPrinting  
from ipconfig import ipconfig  
from ip\_handler import get\_all\_possible\_addresses  
from globalstuff import G  
  
  
class NetworkEntity:  
 def \_\_init\_\_(self, mac, ip, ipv6, name):  
 self.mac = standard\_mac(mac)  
 self.ip = check\_ip(ip)  
 self.ipv6 = extend\_ipv6(ipv6)  
 self.name = name  
 self.\_compare = None  
  
 def equals(self, other: object) -> bool:  
 """This method checks equality between `self` and `other`.  
  
 Note: Transitive Property of Equality (A=B and B=C => A=C) doesn't necessarily apply here.  
 There can be cases where A = B and B = C but A != C.  
 This is because this method is based on (possibly) incomplete information in NetworkEntity-ies.  
  
 Examples:  
 ```  
 | MAC | IPv4 | IPv6 |  
 |---|-------------------|-------------|----------------------------------------|  
 | A | 00:00:5E:00:53:AF | 192.168.0.5 | 2001:db8:3333:4444:CCCC:DDDD:EEEE:FFFF |  
 | B | | 192.168.0.5 | 2001:db8:3333:4444:CCCC:DDDD:EEEE:FFFF |  
 | C | 00:00:5E:11:90:B1 | 192.168.0.5 | |  
 | D | | | 2001:db8:3333:4444:CCCC:DDDD:EEEE:FFFF |  
 ```  
 Here, for example, `A` contains the full information,  
 `B` contains only IPv4 and IPv6 addresses,  
 `C` contains the true IPv4, a false MAC (maybe from an ARP poisoning attack), and no IPv6;  
 and `D` contains only an IPv6 address.  
 - Comparing `A == B` will compare the IPv4 and IPv6, and return `True`.  
 - Comparing `B == C` will compare only the IPv4, and return `True`.  
 - Comparing `A == C` will compare the MAC (doesn't match) and IPv4 (does match), and return `False`.  
 - Comparing `D == C` will find no intersection between the address data, and return `False`.  
 - Comparing `E == nothing` will return `False` for every `NetworkEntity E` (including `E = nothing`!).  
  
 Args:  
 other (object): the object to compare to.  
  
 Returns:  
 bool: whether the NetworkEntity-ies are equal.  
 """  
 if not isinstance(other, NetworkEntity):  
 return False  
 intersection = []  
 for address in ["mac", "ip", "ipv6"]:  
 if self[address] != nothing[address] and other[address] != nothing[address]:  
 intersection.append(address)  
  
 if len(intersection) == 0:  
 return False  
  
 for address in intersection:  
 if self[address] != other[address]:  
 return False  
  
 return True  
  
 def \_\_getitem\_\_(self, key):  
 if key == "mac":  
 return self.mac  
 if key == "ip":  
 return self.ip  
 if key == "ipv6":  
 return self.ipv6  
 if key == "name":  
 return self.name  
 raise TypeError(  
 f"Subscripting in NetworkEntity must be `mac`, `ip`, `ipv6`, or `name`; got `{key}`"  
 )  
  
 def \_\_setitem\_\_(self, key, value):  
 if key == "mac":  
 self.mac = value  
 elif key == "ip":  
 self.ip = value  
 elif key == "ipv6":  
 self.ipv6 = value  
 elif key == "name":  
 self.name = value  
 else:  
 raise TypeError(  
 f"Item assignment in NetworkEntity must be `mac`, `ip`, `ipv6`, or `name`; got `{key}`"  
 )  
  
 def \_\_str\_\_(self):  
 return "< " + ' | '.join([self[field] for field in ["mac", "ip",  
 "ipv6", "name"] if self[field] != nothing[field]]) + " >"  
  
 def to\_string(self, sep=' '):  
 return sep.join([self[field] for field in ["mac", "ip",  
 "ipv6", "name"] if self[field] != nothing[field]])  
  
 def to\_dict(self):  
 return {field: self[field] for field in ["mac", "ip", "ipv6", "name"]}  
  
 def compare(self):  
 """Turns the values of the Entity's fields into integers to be used in a comparison.  
  
 Returns:  
 dict: field names as the keys, integers as the values.  
 """  
 if self.\_compare is None:  
 result = {}  
 result["ip"] = [int(part, base=10) for part in self.ip.split('.')]  
 result["ip"] = sum([result["ip"][-i] \* (256\*\*i) for i in range(4)])  
 result["mac"] = [int(part, base=16)  
 for part in self.mac.split('-')]  
 result["mac"] = sum([result["mac"][-i] \* (256\*\*i)  
 for i in range(6)])  
 result["ipv6"] = [int(part, base=16)  
 for part in self.ipv6.split(':')]  
 result["ipv6"] = sum([result["ipv6"][-i] \* (65536\*\*i)  
 for i in range(8)])  
 self.\_compare = result  
 return self.\_compare  
  
 def merge(self, other):  
 """Merges the information from two equal NetworkEntity-ies.  
 This method fills in any missing information in `self` with the information from `other`.  
 Note: they must be equal.  
 Note: merges right-into-left -- in `A.merge(B)`, A is full with information, and B is unchanged.  
  
 Args:  
 other (NetworkEntity): the entity to be merged with.  
 """  
 if not self.equals(other):  
 raise ValueError(  
 "In NetworkEntity.merge(self,other), the entities must be equal."  
 )  
 for field in ["mac", "ip", "ipv6", "name"]:  
 if other[field] != nothing[field]:  
 self[field] = other[field]  
 self.\_compare = None  
  
 def \_\_hash\_\_(self):  
 return hash((self.mac, self.ip, self.ipv6, self.name))  
  
 def \_\_eq\_\_(self, other):  
 # Use `.equals` for usual comparisons!  
 if other is None:  
 return False  
 return self.mac == other.mac and self.ip == other.ip and self.ipv6 == other.ipv6 and self.name == other.name  
  
 def tablestring(self, lengths):  
 padded = []  
 fields = [  
 self.mac,  
 self.ip,  
 ipaddress.ip\_address(self.ipv6).compressed,  
 self.name  
 ]  
 titles = ["mac", "ip", "ipv6", "name"]  
 # longests = ["FF-FF-FF-FF-FF-FF", "255.255.255.255", "0000:0000:0000::0000:0000", "NamesShouldntBeThisLong"]  
 for title, field, length in zip(titles, fields, lengths):  
 length = min(23, length)  
 if field == nothing[title]:  
 padded.append(" " \* length)  
 else:  
 if len(str(field)) > length:  
 padded.append(str(field)[:length - 3] + '...')  
 else:  
 padded.append(str(field).ljust(length))  
 return ' | '.join(padded)  
  
  
def standard\_mac(mac: str) -> str:  
 MAC\_REGEX = r'^([0-9A-F]{2}-){5}([0-9A-F]{2})$'  
 # using the IEEE Std 802-2014 definition.  
 mac = mac.replace(':', '-').upper()  
 if not re.match(MAC\_REGEX, mac):  
 raise ValueError(f"Invalid MAC address: \"{mac}\"")  
 return mac  
  
  
def check\_ip(ip: str) -> str:  
 IP\_REGEX = r'^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$'  
 if not isinstance(ip, str):  
 raise ValueError(f"Invalid IP address type: \"{ip}\"")  
 if not re.match(IP\_REGEX, ip):  
 raise ValueError(f"Invalid IP address: \"{ip}\"")  
 return ip  
  
  
def filterIPv4(addresses: list[str]) -> list[str]:  
 if isinstance(addresses, str):  
 return [addresses]  
  
 def isip(ip: str) -> bool:  
 try:  
 return check\_ip(ip) == ip  
 except ValueError:  
 return False  
  
 return list(filter(isip, addresses))  
  
  
def extend\_ipv6(ipv6: str) -> str:  
 IPV6\_REGEX = r'(([0-9a-fA-F]{1,4}:){7,7}[0-9a-fA-F]{1,4}|([0-9a-fA-F]{1,4}:){1,7}:|([0-9a-fA-F]{1,4}:){1,6}:[0-9a-fA-F]{1,4}|([0-9a-fA-F]{1,4}:){1,5}(:[0-9a-fA-F]{1,4}){1,2}|([0-9a-fA-F]{1,4}:){1,4}(:[0-9a-fA-F]{1,4}){1,3}|([0-9a-fA-F]{1,4}:){1,3}(:[0-9a-fA-F]{1,4}){1,4}|([0-9a-fA-F]{1,4}:){1,2}(:[0-9a-fA-F]{1,4}){1,5}|[0-9a-fA-F]{1,4}:((:[0-9a-fA-F]{1,4}){1,6})|:((:[0-9a-fA-F]{1,4}){1,7}|:)|fe80:(:[0-9a-fA-F]{0,4}){0,4}%[0-9a-zA-Z]{1,}|::(ffff(:0{1,4}){0,1}:){0,1}((25[0-5]|(2[0-4]|1{0,1}[0-9]){0,1}[0-9])\.){3,3}(25[0-5]|(2[0-4]|1{0,1}[0-9]){0,1}[0-9])|([0-9a-fA-F]{1,4}:){1,4}:((25[0-5]|(2[0-4]|1{0,1}[0-9]){0,1}[0-9])\.){3,3}(25[0-5]|(2[0-4]|1{0,1}[0-9]){0,1}[0-9]))'  
 ipv6 = ipv6.lower()  
 if not re.match(IPV6\_REGEX, ipv6):  
 raise ValueError(f"Invalid IPv6 address: \"{ipv6}\"")  
 return ipaddress.ip\_address(ipv6).exploded.lower()  
  
  
def match(address: str) -> NetworkEntity:  
 try:  
 standard\_mac(address)  
 return NetworkEntity(mac=address, ip=nothing.ip, ipv6=nothing.ipv6, name="Unknown")  
 except ValueError:  
 pass  
  
 try:  
 check\_ip(address)  
 return NetworkEntity(mac=nothing.mac, ip=address, ipv6=nothing.ipv6, name="Unknown")  
 except ValueError:  
 pass  
  
 try:  
 extend\_ipv6(address)  
 return NetworkEntity(mac=nothing.mac, ip=nothing.ip, ipv6=address, name="Unknown")  
 except ValueError:  
 pass  
  
 raise ValueError("The address is not MAC, not IP, and not IPv6.")  
  
  
class LockedNetworkEntity(NetworkEntity):  
 def \_\_setitem\_\_(self, key, value):  
 raise TypeError(  
 f"Item assignment in NetworkEntity cannot be done on a locked entity.")  
  
 def merge(self, other):  
 # Since we know LockedNetworkEntities will have complete information,  
 # there's no need to merge additions into them (plus it causes errors).  
 pass  
  
  
# Special Entities: LockedNetworkEntity  
nothing = LockedNetworkEntity(  
 mac="00:00:00:00:00:00",  
 ip="0.0.0.0",  
 ipv6="::",  
 name="Unknown"  
)  
localhost = None  
mDNS = None  
multicast = None  
broadcast = None  
router = None  
local\_broadcast = None  
here = None  
  
specials = []  
  
  
class NetworkStorage:  
 data = []  
 waiting = Queue()  
 connections = Queue()  
  
 # Use Singleton pattern:  
 def \_\_new\_\_(cls):  
 if not hasattr(cls, 'instance'):  
 cls.instance = object.\_\_new\_\_(cls)  
 # Initialise special entities  
 global nothing, localhost, mDNS, multicast, broadcast, router, local\_broadcast, here  
 localhost = LockedNetworkEntity(  
 mac=nothing.mac,  
 ip="127.0.0.1",  
 ipv6="::1",  
 name="loopback"  
 )  
  
 mDNS = LockedNetworkEntity(  
 mac=nothing.mac,  
 ip="224.0.0.251",  
 ipv6="ff02::fb",  
 name="multicast DNS"  
 )  
  
 multicast = LockedNetworkEntity(  
 mac=nothing.mac,  
 ip="224.0.0.2",  
 ipv6="ff00::",  
 name="multicast"  
 ) # hostify returns '\*.mcast.net' (differs for 224.0.0.\*)  
  
 broadcast = LockedNetworkEntity(  
 mac="FF-FF-FF-FF-FF-FF",  
 ip="255.255.255.255",  
 ipv6=nothing.ipv6,  
 name="broadcast"  
 )  
  
 router = NetworkEntity(  
 mac=nothing.mac,  
 ip=filterIPv4(  
 ipconfig()["Default Gateway"])[0],  
 ipv6=nothing.ipv6,  
 name="router"  
 )  
 SpecialInformation()[router, 'role'] = 'router'  
  
 local\_broadcast = LockedNetworkEntity(  
 mac=nothing.mac,  
 ip=get\_all\_possible\_addresses()[-1],  
 ipv6=nothing.ipv6,  
 name="local broadcast"  
 )  
  
 here = LockedNetworkEntity(  
 mac=ipconfig()["Physical Address"],  
 ip=ipconfig()["IPv4 Address"],  
 ipv6=ipconfig()["IPv6 Address"] if 'IPv6 Address' in ipconfig() else nothing.ipv6,  
 name=ipconfig()["Host Name"]  
 )  
 SpecialInformation()[here, 'role'] = 'here'  
  
 cls.instance.special\_add(  
 localhost,  
 mDNS,  
 multicast,  
 broadcast,  
 router,  
 local\_broadcast,  
 here  
 )  
 # print(nothing, \*specials, sep="\n")  
  
 cls.instance.\_give\_names()  
 return cls.instance  
  
 def \_give\_names(self):  
 for method in dir(self):  
 if not method.startswith('\_'):  
 try:  
 getattr(self, method).\_\_func\_\_.\_\_name\_\_ = "NetworkStorage." + method  
 except AttributeError:  
 continue  
  
 def add(self, \*args, mac=nothing.mac, ip=nothing.ip, ipv6=nothing.ipv6, name=nothing.name):  
 if len(args) == 0:  
 self.waiting.put(NetworkEntity(mac, ip, ipv6, name))  
 else:  
 for entity in args:  
 if isinstance(entity, NetworkEntity) \  
 or isinstance(entity, LockedNetworkEntity):  
 self.waiting.put(entity)  
  
 def special\_add(self, \*entities):  
 """Adds a special LockedNetworkEntity to the `specials` list.  
 NOT THREAD-SAFE. Only use in non-parellel code.  
 Intended for LockedNetworkEntities but regular NetworkEntities are allowed too.  
  
 Args:  
 entities (list[NetworkEntity]): the special entities to be added.  
 """  
 for entity in entities:  
 specials.append(entity)  
   
 def connect(self, ip1, ip2):  
 self.connections.put((ip1, ip2))  
  
 def \_resolve(self):  
 def append(entity):  
 for other in self.data:  
 if entity is other:  
 return  
 if other.equals(entity):  
 other.merge(entity)  
 return  
 if not isinstance(entity, LockedNetworkEntity):  
 for special in specials:  
 if entity.equals(special):  
 entity.merge(special)  
 self.data.append(entity)  
 G.add\_node(entity)  
 if entity in LAN:  
 G.add\_edge(router, entity)  
  
 from hostify import hostify\_sync, hostify  
 hostify\_sync([  
 entity.ip  
 for entity in list(self.waiting.queue)  
 if entity.ip != nothing.ip  
 ])  
 for entity in list(self.waiting.queue):  
 if entity.name == nothing.name:  
 entity.name = hostify(entity.ip)  
 append(entity)  
 ips = {entity.ip: entity  
 for entity in self.data  
 if entity.ip != nothing.ip}  
 for ip1, ip2 in list(self.connections.queue):  
 if (ip1 not in ips) or (ip2 not in ips):  
 continue  
 G.add\_edge(ips[ip1], ips[ip2])  
   
  
 def sort(self, key="ip"):  
 self.\_resolve()  
 try:  
 others = ['ip', 'mac', 'ipv6']  
 others.remove(key)  
 except ValueError:  
 raise ValueError(  
 'Sorting key must be either `mac`, `ip`, or `ipv6`.'  
 )  
 keys = [key] + others  
  
 if len(self.data) == 0:  
 return [nothing]  
 return sorted(self.data,  
 key=lambda entity: tuple(entity.compare()[field]  
 for field in keys))  
  
 def organise(self, key="ip"):  
 """Converts the storage into a dictionary, with the key being one of the fields, and the values -- the whole entity.  
 Example:  
 ```  
 data = [networkEntity1, networkEntity2, networkEntity3]  
 organise('ip') = {  
 "1.1.1.1": networkEntity1,  
 "2.2.2.2": networkEntity2,  
 "1.0.0.3": networkEntity3  
 }  
 ```  
  
 Args:  
 key (str, optional): the key for the dictionary. Must be `mac`, `ip`, `ipv6`, or `name`. Defaults to `ip`.  
  
 Returns:  
 dict: the dictionary as described above.  
  
 Raises:  
 TypeError: if the key is invalid.  
 """  
 self.\_resolve()  
 if key not in ["mac", "ip", "ipv6", "name"]:  
 raise TypeError(  
 f"NetworkStorage.organise's key must be `mac`, `ip`, `ipv6`, or `name`; got `{key}`"  
 )  
 return {entity[key]: entity  
 for entity in self.data  
 if entity[key] != nothing[key]}  
  
 def \_\_iter\_\_(self):  
 self.\_resolve()  
 for elem in self.sort():  
 yield elem  
  
 def \_\_len\_\_(self):  
 return len(self.data)  
  
 def \_\_getitem\_\_(self, key):  
 """Gets a single "column" (key, property, field) as a list from all the NetworkEntity-ies stored.  
 Note: this will not include empty values. E.g., asking for `lookup['ipv6']` will not return any data from entities without an IPv6 datum.  
  
 Args:  
 key (str): the key to select from all the entities. Must be 'mac', 'ip', 'ipv6', or 'name'.  
  
 Raises:  
 TypeError: if the key is invalid.  
  
 Returns:  
 list: a list containing all the requested data.  
 """  
 self.\_resolve()  
 if key in ["mac", "ip", "ipv6", "name"]:  
 self.\_resolve()  
 return [entity[key]  
 for entity in self.data  
 if entity[key] != nothing[key]]  
 raise TypeError(f"Subscripting in NetworkStorage must be `mac`, `ip`, `ipv6`, or `name`; got `{key}`")  
  
 def print(self):  
 self.\_resolve()  
 with JustifyPrinting():  
 for entity in self:  
 print(entity)  
  
 def tablestring(self):  
 lengths = [max(map(lambda x: len(str(x)), self[field]), default=4)  
 for field in ["mac", "ip", "ipv6", "name"]]  
 lengths = [min(length, 23) for length in lengths]  
 titles = ["MAC", "IPv4", "IPv6", "Name"]  
 titles = "| " + ' | '.join([title.center(length)  
 for title, length in zip(titles, lengths)]) + " |"  
  
 width = sum(lengths) + 11  
 top = "/" + ("-" \* width) + "\\"  
 subtitles = "|" + ('-' \* width) + "|"  
 bottom = "\\" + ("-" \* width) + "/"  
 return [top, titles, subtitles,  
 \*["| " + x.tablestring(lengths) + " |" for x in self.sort()], bottom]  
  
  
class LAN:  
 def \_\_contains\_\_(self, entity):  
 return entity.ip in get\_all\_possible\_addresses()  
  
  
LAN = LAN()  
  
class SpecialInformation(dict):  
 # singleton  
 \_instance = None  
  
 def \_\_new\_\_(cls):  
 if not cls.\_instance:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 return cls.\_instance  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 def \_\_setitem\_\_(self, keys: tuple[NetworkEntity, str], value):  
 if isinstance(keys, NetworkEntity) and isinstance(value, dict):  
 super().\_\_setitem\_\_(keys, value)  
 return  
 if not isinstance(keys[0], NetworkEntity):  
 raise TypeError("First key must be of type NetworkEntity.")  
 if not isinstance(keys[1], str):  
 raise TypeError("Second key must be a string.")  
 entity, info\_key = keys  
 if entity not in self:  
 super().\_\_setitem\_\_(entity, {})  
 self[entity][info\_key] = value  
  
 def \_\_getitem\_\_(self, key):  
 # This function trys to find any NetworkEntities in the dict's keys that `.equals` with `key`,  
 # and merges all their information to a single dict that is returned.  
 if isinstance(key, NetworkEntity):  
 merged = {}  
 for entity, value in self.items():  
 if key.equals(entity):  
 merged.update(value)  
 self[key] = merged  
 return merged  
 elif isinstance(key, tuple):  
 entity, info\_key = key  
 merged = {}  
 for item, value in self.items():  
 if entity.equals(item):  
 merged.update(value)  
 self[entity] = merged  
 return merged[info\_key]  
 elif isinstance(key, str):  
 entities = []  
 for entity, value in self.items():  
 if key in value:  
 entities.append(entity)  
 return entities  
 else:  
 raise TypeError("Key must be NetworkEntity or tuple or key.")  
  
 def \_\_contains\_\_(self, item):  
 if isinstance(item, NetworkEntity):  
 return any(item.equals(entity) for entity in self.keys())  
 entity, info\_key = item  
 if entity not in self:  
 return False  
 return info\_key in self[entity]  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module is responsible for the Network Storage data,")  
 print("and all the infrastructure that supports it.")  
 NetworkStorage()  
 print([str(entity) for entity in specials])  
 print(str(SpecialInformation()))

# ./Scanner/nmap udp scan on router results.txt

[irrelevant content]

# ./Scanner/nslookup results.txt

[irrelevant content]

# ./Scanner/packets.db

[binary content]

# ./Scanner/PacketSniffer.py

from import\_handler import ImportDefence  
from typing import TypeVar, Any, GenericAlias, ClassVar  
import sys  
\_T = TypeVar("\_T")  
\_S = TypeVar("\_S")  
with ImportDefence():  
 from scapy.all import IP  
 import sqlite3  
 import pickle  
 from typing\_extensions import SupportsIndex  
 from collections.abc import Callable, Iterable, Iterator  
 from queue import Queue  
 from threading import Thread  
 from time import sleep  
from Sniffer import Sniffer  
  
  
class ListWithSQL:  
 CREATE = '''CREATE TABLE IF NOT EXISTS list\_with\_sql (id INTEGER PRIMARY KEY AUTOINCREMENT, item BLOB)'''  
 INSERT = "INSERT INTO list\_with\_sql (item) VALUES (?)"  
 CLEAR = "DELETE FROM list\_with\_sql"  
 RESET\_AUTOINCREMENT = "UPDATE SQLITE\_SEQUENCE SET SEQ=0 WHERE NAME='list\_with\_sql'"  
  
 def \_\_init\_\_(self, path: str, maxram: int = 100):  
 self.path = path  
 self.ram = list()  
 self.length = 0  
 self.maxram = maxram  
  
 with sqlite3.connect(self.path) as conn:  
 cursor = conn.cursor()  
 cursor.execute(ListWithSQL.CREATE)  
 cursor.execute(ListWithSQL.CLEAR)  
 cursor.execute(ListWithSQL.RESET\_AUTOINCREMENT)  
 conn.commit()  
  
 def copy(self):  
 copied = ListWithSQL(self.path, "list\_with\_sql\_copy", self.maxram)  
 for item in self:  
 copied.append(item)  
 return copied  
  
 def append(self, \_\_object: \_T) -> None:  
 self.ram.append(\_\_object)  
 self.length += 1  
 if len(self.ram) >= self.maxram:  
 self.\_flush\_to\_sql()  
  
 def \_flush\_to\_sql(self) -> None:  
 to\_database = [(pickle.dumps(p),) for p in self.ram]  
 with sqlite3.connect(self.path) as conn:  
 cursor = conn.cursor()  
 cursor.executemany(ListWithSQL.INSERT, to\_database)  
 conn.commit()  
 self.ram = []  
  
 def extend(self, \_\_iterable: Iterable[\_T]) -> None:  
 for item in \_\_iterable:  
 self.append(item)  
  
 def pop(self, \_\_index: SupportsIndex = -1) -> \_T:  
 raise NotImplementedError("I don't think you should pop elements from a ListWithSQL.")  
  
 def index(self, \_\_value: \_T, \_\_start: SupportsIndex = 0, \_\_stop: SupportsIndex = sys.maxsize) -> int:  
 if \_\_start < 0:  
 \_\_start += self.length  
 if \_\_stop < 0:  
 \_\_stop += self.length  
 if \_\_start < 0:  
 \_\_start = 0  
 if \_\_stop > self.length:  
 \_\_stop = self.length  
  
 for i in range(\_\_start, \_\_stop):  
 item = self[i]  
 if item == \_\_value:  
 return i  
 else:  
 raise ValueError(f"{\_\_value} is not in list")  
  
 def count(self, \_\_value: \_T) -> int:  
 with sqlite3.connect(self.path) as conn:  
 cursor = conn.cursor()  
 cursor.execute(f"SELECT COUNT(\*) FROM list\_with\_sql WHERE item=?", (pickle.dumps(\_\_value),))  
 sql\_count = cursor.fetchone()[0]  
 ram\_count = self.ram.count(\_\_value)  
 return ram\_count + sql\_count  
  
 def insert(self, \_\_index: SupportsIndex, \_\_object: \_T) -> None:  
 raise NotImplementedError("Inserting manually is not supported for ListWithSQL. Please use `append` instead.")  
  
 def remove(self, \_\_value: \_T) -> None:  
 raise NotImplementedError("I don't think you should remove elements from a ListWithSQL.")  
  
 def sort(self: list, \*, key: None = None, reverse: bool = False) -> None:  
 raise NotImplementedError("Sorting is not supported for ListWithSQL. Please use `\_\_iter\_\_` and sort manually instead.")  
  
 def \_\_len\_\_(self) -> int:  
 return self.length  
  
 def \_\_iter\_\_(self) -> Iterator[\_T]:  
 for i in range(self.length):  
 yield self[i]  
  
 \_\_hash\_\_: ClassVar[None] # type: ignore[assignment]  
  
 def \_\_getitem\_\_(self, \_\_i: SupportsIndex | slice) -> \_T:  
 if isinstance(\_\_i, slice):  
 return [self[j] for j in range(\*\_\_i.indices(len(self)))]  
  
 if not (-len(self) <= \_\_i < len(self)):  
 raise IndexError("list index out of range")  
  
 if \_\_i < 0:  
 \_\_i += len(self)  
  
 if \_\_i >= len(self) - len(self.ram):  
 return self.ram[\_\_i - len(self) + len(self.ram)]  
  
 with sqlite3.connect(self.path) as conn:  
 cursor = conn.cursor()  
 cursor.execute(f"SELECT item FROM list\_with\_sql WHERE id = ?", (\_\_i + 1,))  
 res = cursor.fetchone()  
 if res is None:  
 raise IndexError("list index out of range")  
 return pickle.loads(res[0])  
  
 def \_\_setitem\_\_(self, \_\_key: SupportsIndex, \_\_value: \_T) -> None:  
 # Convert negative indices to positive indices  
 if isinstance(\_\_key, int) and \_\_key < 0:  
 \_\_key += len(self)  
  
 # If index is in range of SQL data, update it in the SQL table  
 if \_\_key < len(self) - len(self.ram):  
 with sqlite3.connect(self.path) as conn:  
 cursor = conn.cursor()  
 cursor.execute(f"UPDATE list\_with\_sql SET item=? WHERE id=?", (pickle.dumps(\_\_value), \_\_key + 1))  
  
 # If index is in range of RAM data, update it in RAM  
 elif \_\_key < len(self):  
 self.ram[\_\_key - len(self) + len(self.ram)] = \_\_value  
  
 # If index is out of range, raise an IndexError  
 else:  
 raise IndexError('list assignment index out of range')  
  
 def \_\_delitem\_\_(self, \_\_key: SupportsIndex | slice) -> None:  
 raise NotImplementedError("I don't think you should remove (\_\_delitem\_\_) elements from a ListWithSQL.")  
  
 def \_\_contains\_\_(self, \_\_key: object) -> bool:  
 if \_\_key in self.ram:  
 return True  
 with sqlite3.connect(self.path) as conn:  
 cursor = conn.cursor()  
 cursor.execute("SELECT COUNT(\*) FROM list\_with\_sql WHERE item=?", (pickle.dumps(\_\_key),))  
 return cursor.fetchone()[0] > 0  
  
 def \_\_reversed\_\_(self) -> Iterator[\_T]:  
 for item in reversed(self.ram):  
 yield item  
 with sqlite3.connect(self.path) as conn:  
 cursor = conn.cursor()  
 cursor.execute("SELECT item FROM list\_with\_sql ORDER BY id DESC")  
 while True:  
 res = cursor.fetchone()  
 if res is None:  
 break  
 yield pickle.loads(res[0])  
  
  
class ObserverPublisher:  
 """This is the implementation of the Observer Behavioural Design Pattern,  
 which is used when a centralised source of data (the publisher) needs to send out updates (notifications)  
 to many code pieces (observers).  
  
 This specific implementation, being focused on not blocking the `add\_datum` calls too much,  
 uses a separate thread to notify observers, and an internal queue to save the data in the meantime.  
  
 Just extend this class, make sure to call `.add\_datum` when new data arrives,  
 and you can use `add\_observer` to attach observers!  
 """  
 def \_\_init\_\_(self):  
 if not hasattr(self, 'observer\_thread'):  
 self.data\_queue = Queue()  
 self.observers = []  
 self.observer\_thread = Thread(target=self.notify\_all)  
 self.observer\_thread.start()  
   
 def notify\_all(self) -> None:  
 from globalstuff import terminator  
 while not terminator.is\_set():  
 if self.data\_queue.empty():  
 sleep(0.3)  
 continue  
 datum = self.data\_queue.get()  
 for observer in self.observers:  
 observer(datum)  
   
 def add\_observer(self, observer: Callable) -> None:  
 if not callable(observer):  
 raise TypeError("Observer must be callable.")  
 if observer not in self.observers:  
 self.observers.append(observer)  
   
 def add\_datum(self, datum):  
 self.data\_queue.put(datum)  
  
  
class PacketSniffer(ObserverPublisher):  
 \_instance = None  
 DB\_PATH = 'packets.db'  
 SQL\_CREATE\_TABLE = '''CREATE TABLE IF NOT EXISTS packets (id INTEGER PRIMARY KEY AUTOINCREMENT, packet BLOB, proto TEXT, src TEXT, dst TEXT, ttl INTEGER, flags TEXT, options BLOB, timestamp INTEGER)'''  
 INSERT\_STATEMENT = "INSERT INTO packets(packet, proto, src, dst, ttl, flags, options, timestamp) VALUES (?, ?, ?, ?, ?, ?, ?, ?)"  
 CLEAR\_TABLE = '''DELETE FROM packets;'''  
  
 def \_\_new\_\_(cls, max\_packets=100):  
 if cls.\_instance is None:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 cls.\_instance.max\_packets = max\_packets  
 cls.\_instance.packets = []  
 cls.\_instance.sniff\_thread = Sniffer(prn=cls.\_instance.\_packet\_handler, lfilter=cls.\_instance.\_ip\_filter)  
 cls.\_instance.sniff\_thread.start()  
 cls.\_instance.length = 0  
 cls.\_instance.initialised = False  
 return cls.\_instance  
  
 def \_\_init\_\_(self, max\_packets=100):  
 super().\_\_init\_\_()  
 if not self.initialised:  
 self.initialised = True  
 self.max\_packets = max\_packets  
 with sqlite3.connect(self.DB\_PATH) as conn:  
 cursor = conn.cursor()  
 cursor.execute(self.SQL\_CREATE\_TABLE)  
 cursor.execute(self.CLEAR\_TABLE)  
  
 def stop(self):  
 if self.sniff\_thread:  
 if self.sniff\_thread.running:  
 self.sniff\_thread.stop()  
 self.sniff\_thread = None  
  
 self.\_flush\_packets()  
  
 def get\_packet(self, i: int):  
 packet\_row = None  
 with sqlite3.connect(self.DB\_PATH) as conn:  
 cursor = conn.cursor()  
 packet\_row = cursor.execute('SELECT packet FROM packets WHERE id = ?', (i,)).fetchone()  
 return pickle.loads(packet\_row[0])  
  
 def \_packet\_handler(self, packet):  
 from time import time as now  
 if IP in packet:  
 fields = packet[IP].fields  
 self.packets.append({'packet': packet, \*\*fields, 'timestamp': int(now())})  
 self.length += 1  
 self.add\_datum(packet)  
 if len(self.packets) >= self.max\_packets:  
 self.\_flush\_packets()  
  
 def \_\_len\_\_(self):  
 return self.length  
  
 def \_flush\_packets(self):  
 packets\_to\_insert = [(pickle.dumps(p['packet']), p['proto'], p['src'], p['dst'], int(p['ttl']), str(p['flags']), pickle.dumps(p['options']), int(p['timestamp'])) for p in self.packets]  
  
 with sqlite3.connect(self.DB\_PATH) as conn:  
 try:  
 cursor = conn.cursor()  
 cursor.executemany(PacketSniffer.INSERT\_STATEMENT, packets\_to\_insert)  
 conn.commit()  
 except sqlite3.OperationalError:  
 return  
 self.packets = []  
  
 def \_ip\_filter(self, packet):  
 return IP in packet  
  
 def \_\_iter\_\_(self):  
 packets = self.packets.copy()  
  
 # Yield all packets from the SQL database  
 for i in range(self.length - len(packets)):  
 packet = self.get\_packet(i)  
 if packet is None:  
 continue  
 yield packet  
  
 # Yield all the packets from the `self.packets` (but `copy()`ied earlier).  
 for packet in packets:  
 yield packet['packet']  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module contains the PacketSniffer class.")  
 import time  
 packet\_sniffer = PacketSniffer(max\_packets=40)  
 time.sleep(5)  
 print(f"{len(packet\_sniffer)} packet(s) were sniffed.")  
 for packet in packet\_sniffer:  
 print(packet)  
 packet\_sniffer.stop()

# ./Scanner/PrintingContexts.py

from import\_handler import ImportDefence  
with ImportDefence():  
 import os  
 import sys  
 from io import StringIO  
  
  
class \_Printing:  
 """This context manager delays and stores all outputs via `print`s.  
 It is not meant to be used directly, but other classes can inherit it.  
 """  
  
 def \_\_init\_\_(self):  
 pass  
  
 def \_\_enter\_\_(self):  
 self.real\_stdout = sys.stdout  
 return self  
  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 sys.stdout = self.real\_stdout  
  
  
class InstantPrinting(\_Printing):  
 """This context manager delays and stores all outputs via `print`s, and prints everything when closed.  
 Usage:  
 ```py  
 with InstantPrinting():  
 # do some stuff here including printing  
 # Here, exiting the context, the printing will all happen immediately.  
 ```  
 """  
  
 def \_\_init\_\_(self):  
 self.output = StringIO()  
  
 def \_\_enter\_\_(self):  
 super().\_\_enter\_\_()  
 sys.stdout = self.output  
 return self  
  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 super().\_\_exit\_\_(exc\_type, exc\_val, exc\_tb)  
 print(self.output.getvalue())  
  
  
class NoPrinting(\_Printing):  
 """This context manager prevents all output (through `sys.stdout`, e.g. normal `print` statements) from showing.  
 Usage:  
 ```py  
 with NoPrinting():  
 # do some stuff here including printing  
 # Nothing will actually display  
 ```  
  
 Technical note: this just inherits `\_Printing` with no additional behaviour.  
 """  
 pass  
  
  
class \_SplitStringIO:  
 """This class is like the io.StringIO, but it splits different `write` statements.  
 Internally, this is a `list` of `StringIO`s.  
 Not meant for use outside the `util` module.  
  
 Implements:  
 `\_\_init\_\_`: initialises an empty list.  
 `write`: adds a StringIO to the list, and writes the data into it.  
 `getvalue`: returns a list of all the `.getvalue`s of the `StringIO`s.  
 `flush`: does nothing.  
 """  
  
 def \_\_init\_\_(self):  
 self.content = []  
  
 def write(self, data):  
 self.content.append(StringIO())  
 self.content[-1].write(data)  
  
 def getvalue(self):  
 return [string.getvalue() for string in self.content]  
  
 def flush():  
 pass  
  
  
class JustifyPrinting(InstantPrinting):  
 """This context manager delays and stores all outputs via `print`s, and prints everything when closed,  
 justifying every print statement to form a nice-looking block of text, where each line is centred and as widespread as is allowed.  
  
 Note: Messing with `print`'s default values (`sep=' ', end='\\n'`) is not recommended,  
 since this context manager treats space-separated strings as belonging to the same statement,  
 and newline-separated string as belonging to different statements.  
  
 Usage:  
 ```py  
 with JustifyPrinting():  
 # do some stuff here including printing  
 # Here, exiting the context, the printing will all happen immediately and (hopefully) nicely.  
 ```  
 """  
  
 def \_\_init\_\_(self):  
 self.output = \_SplitStringIO()  
  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 \_Printing.\_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb)  
 blocks = self.output.getvalue()  
 width = int(os.get\_terminal\_size().columns)  
  
 MIN\_SEP = 3 # There must be at least one space between blocks.  
 MAX\_SEP = 10 # There cannot be more than ten spaces between blocks.  
  
 statements = [""]  
 for block in blocks:  
 if block == '\n':  
 statements.append("")  
 else:  
 statements[-1] += block  
 blocks = statements  
  
 lines = [[]]  
 for block in blocks:  
 # Lengths of all previous blocks  
 # + Length of current block  
 # + assuming `MIN\_SEP` spaces in-between (thus, #spaces = #blocks \* MIN\_SEP)  
 # > width of console in characters  
 all\_previous\_blocks = sum(map(len, lines[-1]))  
 spaces\_in\_between = len(lines[-1]) \* MIN\_SEP  
 if all\_previous\_blocks + len(block) + spaces\_in\_between > width:  
 lines.append([])  
 lines[-1].append(block)  
  
 for line in lines:  
 # Optimal case: total\_length + total\_separator\_length = width  
 # total\_separator\_length = sep \* (len(line) - 1)  
 # => sep = (width - total\_length) // (len(line) - 1)  
 line = [part for part in line if part.strip() != '']  
 if len(line) == 1:  
 print(line[0].center(width))  
 continue  
 total\_length = sum([len(block) for block in line])  
 sep = (width - total\_length) // (len(line) - 1)  
 if sep > MAX\_SEP:  
 sep = MAX\_SEP  
 sep \*= ' '  
 print(sep.join(line).center(width))  
  
  
class TablePrinting(InstantPrinting):  
 """This context manager delays and stores all outputs via `print`s, and prints everything when closed,  
 justifying every print statement to form a nice-looking table.  
  
 Note: Messing with `print`'s default values (`sep=' ', end='\\n'`) is not recommended,  
 since this context manager treats space-separated strings as belonging to the same statement,  
 and newline-separated string as belonging to different statements.  
  
 Usage:  
 ```py  
 with TablePrinting():  
 # do some stuff here including printing  
 # Here, exiting the context, the printing will all happen immediately and (hopefully) nicely.  
 ```  
 """  
 aligns = {  
 'left': lambda s, w: s.ljust(w),  
 'center': lambda s, w: s.center(w),  
 'right': lambda s, w: s.rjust(w)  
 }  
  
 def \_\_init\_\_(self, align='center'):  
 self.output = \_SplitStringIO()  
 self.align = TablePrinting.aligns["center"]  
 if align in TablePrinting.aligns.keys():  
 self.align = TablePrinting.aligns[align]  
  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 \_Printing.\_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb)  
 output = self.output.getvalue()  
 width = int(os.get\_terminal\_size().columns)  
  
 # Separate `print statements`.  
 # Every block which is just a newline is (probably) a different print,  
 # so I'll treat it as such.  
 blocks = [""]  
 for block in output:  
 if block == '\n':  
 blocks.append("")  
 else:  
 blocks[-1] += block  
  
 # Split the blocks into a `chunk list` (e.g. [a, b, c, d, e, f] + n=2  
 # -> [[a, b], [c, d], [e, f]])  
 lengths = [len(block) for block in blocks]  
 try:  
 n = max(width // max(lengths), 3)  
 except ZeroDivisionError:  
 # There is no content, only empty strings  
 print()  
 return  
 lines = [blocks[i:i + n] for i in range(0, len(blocks), n)]  
 for line in lines:  
 for part in line:  
 w = width // n  
 print(self.align(part, w), end="")  
 print()  
  
  
class AutoLinebreaks(InstantPrinting):  
 """This context manager delays and stores all outputs via `print`s, and prints everything when closed,  
 wrapping lines only when nessessary to maintain integrity.  
 In short: Applies CSS's `word-wrap: normal;` (whereas the console is usually `word-wrap: break-word;`).  
  
 Note: Messing with `print`'s default values (`sep=' ', end='\\n'`) is not recommended,  
 since this context manager treats space-separated strings as belonging to the same statement,  
 and newline-separated string as belonging to different statements.  
 You may do so after familiarising yourself with the code, in order to not induce annoying bugs.  
  
 Usage:  
 ```py  
 with AutoLinebreaks():  
 # do some stuff here including printing  
 # Here, exiting the context, the printing will all happen immediately and (hopefully) nicely.  
 ```  
 """  
  
 def \_\_init\_\_(self):  
 self.output = \_SplitStringIO()  
  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 \_Printing.\_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb)  
 output = self.output.getvalue()  
 width = int(os.get\_terminal\_size().columns)  
  
 # Separate `print statements`.  
 # Every block which is just a newline is (probably) a different print,  
 # so I'll treat it as such.  
 blocks = [""]  
 for block in output:  
 if block == '\n':  
 blocks.append("")  
 else:  
 blocks[-1] += block  
  
 counter = width  
 for block in blocks:  
 if block.strip() == "":  
 continue  
 if counter - len(block) <= 0:  
 counter = width  
 print('\n' + block, end="")  
 else:  
 counter -= len(block)  
 print(block, end="")  
 print()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module provides some Printing context managers,")  
 print("That allow you to decide how to format your console output")  
 print("and whether to even show it!")

# ./Scanner/README.md

[binary content]

# ./Scanner/RecommendProbabilitiesPrimitive.py

import networkx as nx  
import matplotlib.pyplot as plt  
import matplotlib.colors as mcolors  
import matplotlib.cm as cm  
  
G = nx.DiGraph()  
probabilities = {}  
  
  
def construct\_graph():  
 global probabilities, G  
 G.add\_nodes\_from(["ARP Sweep", "ICMP Sweep", "ARP Live", "ICMP Live", "OS-ID", "Public Address"])  
 G.add\_weighted\_edges\_from([  
 # (v, u, w: float)  
 ("ARP Sweep", "ARP Live", 0.4),  
 ("ICMP Sweep", "ICMP Live", 0.4),  
 ("ARP Sweep", "ICMP Sweep", 0.2),  
 ("ICMP Sweep", "ARP Sweep", 0.3),  
 ("ARP Live", "ICMP Live", 0.05),  
 ("ICMP Sweep", "OS-ID", 0.5),  
 ("ICMP Live", "OS-ID", 0.6)  
 ])  
 G.add\_weighted\_edges\_from(list((n, n, -0.9) for n in G.nodes))  
 # positive values are "Yeah, if you executed `v`, consider executing `u`".  
 # negative values are "If you executed `v` please do not execute `u`".  
 probabilities = {node: 1 for node in G}  
  
  
def normalise():  
 global probabilities  
 s = sum(probabilities.values())  
 probabilities = {node: float(i) / s for node, i in probabilities.items()}  
  
  
def render\_ax1(fig, ax1):  
 pos = nx.circular\_layout(G)  
 node\_values = list(probabilities.values())  
  
 # Node colormap and colorbar  
 cmap\_nodes = cm.PiYG\_r  
 norm\_nodes = plt.Normalize(vmin=0, vmax=1)  
 sm\_nodes = cm.ScalarMappable(norm=norm\_nodes, cmap=cmap\_nodes)  
  
 node\_colors = [sm\_nodes.to\_rgba(value) for value in node\_values]  
 nodes = nx.draw\_networkx\_nodes(G, pos, node\_size=100, node\_color=node\_colors, ax=ax1)  
  
 # Edge colormap and colorbar  
 colors = [w['weight'] for v, u, w in G.edges(data=True)]  
 cmap\_edges = plt.cm.coolwarm\_r  
 norm\_edges = plt.Normalize(vmin=-1, vmax=1)  
 sm\_edges = plt.cm.ScalarMappable(norm=norm\_edges, cmap=cmap\_edges)  
 sm\_edges.set\_array([])  
  
 edges = nx.draw\_networkx\_edges(  
 G,  
 pos,  
 node\_size=100,  
 arrowstyle="->",  
 arrowsize=10,  
 edge\_color=colors,  
 edge\_cmap=cmap\_edges,  
 edge\_vmin=min(colors),  
 edge\_vmax=max(colors),  
 width=2,  
 arrows=True,  
 ax=ax1  
 )  
  
 plt.colorbar(sm\_edges, ax=ax1, label='Edge weights')  
 plt.colorbar(sm\_nodes, ax=ax1, label='Node probabilities')  
  
 y\_off = -0.13  
  
 pos\_higher = {k: (v[0], v[1] + y\_off) for k, v in pos.items()}  
 labels = nx.draw\_networkx\_labels(G, pos\_higher, ax=ax1)  
 ax1.set\_title("Scan's Influence On Each Other")  
  
 for node, value in probabilities.items():  
 ax1.annotate(f"{value:.2f}", xy=pos[node], xytext=(-10, -15), textcoords="offset points")  
  
  
def render\_ax2(fig, ax2):  
 adj\_matrix = nx.adjacency\_matrix(G)  
 adj\_array = adj\_matrix.toarray()  
  
 im = ax2.imshow(adj\_array, cmap='coolwarm\_r', interpolation='nearest', vmin=-1, vmax=1)  
 ax2.set\_title("Presented as Adjacency Matrix")  
 # fig.colorbar(im, ax=ax2)  
  
 node\_names = list(G.nodes())  
 ax2.set\_xticks(range(len(node\_names)))  
 ax2.set\_yticks(range(len(node\_names)))  
 ax2.set\_xticklabels(node\_names, rotation=45)  
 ax2.set\_yticklabels(node\_names)  
  
 for i in range(adj\_array.shape[0]):  
 for j in range(adj\_array.shape[1]):  
 ax2.text(j, i, f"{adj\_array[i,j]:.2f}", ha="center", va="center", color="black")  
  
  
def render\_graph():  
 fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(17, 7), gridspec\_kw={'width\_ratios': [1.5, 1]})  
  
 render\_ax1(fig, ax1)  
  
 render\_ax2(fig, ax2)  
  
 plt.show()  
  
  
def step(node):  
 edges = G.edges(node, data=True)  
 edges = [(dst, data['weight']) for src, dst, data in edges]  
 print(edges)  
 p = probabilities[node]  
 for scan, weight in edges:  
 probabilities[scan] += weight \* p  
 print(f"Changed {scan} by {weight \* p}")  
 normalise()  
  
  
def main():  
 construct\_graph()  
 normalise()  
 render\_graph()  
 step("ICMP Sweep")  
 render\_graph()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

# ./Scanner/refresh\_db.bat

@echo off  
title Refresh scans.db  
del scans.db  
py db.py  
echo Done.  
pause

# ./Scanner/register.py

from time import time as now  
from threading import Thread  
  
from gui.dialogs import popup  
  
  
class Register(dict):  
 """This class managers the connection between names (strings)\* and python methods (or lambdas; any callables) that execute these scans.  
 Usage (i.e. this is a dictionary):  
 ```  
 Set: Register()["Scan Name"] = lambda: ...  
 Set: Register()["Scan Name"] = execute\_scan # no parentheses  
 Set: Register()["Scan Name"] = execute\_infinite\_scan, True  
 Get: x = Register()["Scan Name"]  
 ```  
  
 This class implements the singleton pattern.  
  
 \* formerly GUI Buttons, abstracted by `class Scan`.  
 """  
 \_instance = None  
 threads: dict[str, Thread] = {}  
 infinites = set()  
 history = [] # list[list[str, int, int]]: [name, start time [unix timestamp], duration [seconds]]  
  
 def \_\_new\_\_(cls):  
 if cls.\_instance is None:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 return cls.\_instance  
  
 def \_\_setitem\_\_(self, key, value):  
 if not isinstance(key, str):  
 raise TypeError(f"Key must be of type str")  
 if isinstance(value, tuple):  
 value, is\_infinite = value  
 if is\_infinite:  
 self.infinites.add(key)  
 if not callable(value):  
 raise TypeError(f"Value must be callable")  
 super().\_\_setitem\_\_(key, value)  
  
 def \_\_getitem\_\_(self, key: str):  
 try:  
 return super().\_\_getitem\_\_(key)  
 except KeyError:  
 return lambda: popup("Coming Soon", "This scan is not implemented yet.")  
 # raise KeyError(f"Key \"{key}\" not found in register. Try adding it :)")  
  
 def start(self, name: str, action, callback) -> None:  
 def \_add\_callback(action, callback):  
 entry = [name, int(now()), -1]  
 self.history.append(entry)  
 action()  
 callback()  
 if not self.is\_infinite\_scan(name):  
 entry[2] = int(now()) - entry[1]  
  
 \_add\_callback.\_\_name\_\_ = action.\_\_name\_\_ + "\_with\_callback"  
 self.threads[name] = t = Thread(target=\_add\_callback, args=(action, callback))  
 t.start()  
  
 def is\_running(self, name: str) -> bool:  
 if name not in self.threads:  
 return False  
 if self.threads[name].is\_alive():  
 return True  
 self.threads.pop(name)  
 return  
  
 def is\_infinite\_scan(self, name: str):  
 # `name` may contain '...' in the end.  
 return name in self.infinites or name[:-3] in self.infinites  
  
 def get\_history(self):  
 return [tuple(item) for item in self.history]

# ./Scanner/run.bat

@echo off  
color 0A  
title Network Scanner  
  
REM Check Python version  
python --version 2>NUL | findstr /C:"Python 3.10" >NUL  
if %errorlevel% neq 0 (  
 echo WARNING: Python 3.10 is required to run this script.  
 pause  
)  
  
REM Reduce console output from kivy (graphical library)  
set "KIVY\_NO\_CONSOLELOG=true"  
  
REM run the python file  
python exe.py  
  
REM rem log the error level of exiting  
echo.  
echo Process exited with code %errorlevel%.  
pause

# ./Scanner/ScanID.py

from import\_handler import ImportDefence  
with ImportDefence():  
 import ipaddress  
  
from CacheDecorators import one\_cache  
from ipconfig import ipconfig  
  
  
@one\_cache  
def get\_scan\_id() -> str:  
 """Generates the current scan's identifier, based on `ipconfig()` info.  
 The format:  
 [Host Name]@[Interface]@[Gateway IPv4][Subnet Mask][Physical Address]  
 All in Base64, with integer values whenever possible.  
  
 Returns:  
 str: the scan ID, encoded in base64, as a regular string.  
 """  
 from NetworkStorage import router, here  
 import base64  
  
 # '@' == (char)64 == '\x40'  
  
 host = here.name.replace('@', '\x02').encode()  
 iface = ipconfig()["Interface"].replace('@', '\x02').encode()  
  
 gateway = int(ipaddress.IPv4Address(router.ip)).to\_bytes(4, 'big')  
  
 mask = ipconfig()["Subnet Mask"]  
 mask = sum(bin(int(x)).count('1')  
 for x in mask.split('.')).to\_bytes(1, 'big')  
  
 physical = here.mac  
 physical = int(physical.replace('-', ''), 16).to\_bytes(6, 'big')  
  
 return base64.b64encode(  
 host + b'\x40' + iface + b'\x40' + gateway + mask + physical  
 ).decode()  
  
  
def parse\_scan\_id(scan\_id: str) -> str:  
 """Decodes a scan ID.  
 Reversing the logic in `get\_scan\_id`.  
 Here, the `scan\_id` doesn't have to be of the current network,  
 but any valid scan ID.  
 The returned string is of this format:  
 "Here: {host name}, {host mac}, via {network interface}  
 Router: {gateway ipv4}/{mask}"  
 (yes, it contains a newline character)  
  
 Args:  
 scan\_id (str): the scan ID to parse.  
  
 Returns:  
 str: a textual description of the network.  
 """  
 import base64  
 decoded = base64.b64decode(scan\_id)  
 host, iface, others = decoded.split(b'\x40')  
  
 host = host.decode().replace('\x02', '@')  
 iface = iface.decode().replace('\x02', '@')  
  
 gateway, mask, physical = others[:4], others[4:5], others[5:]  
 gateway = int.from\_bytes(gateway, 'big')  
 gateway = str(ipaddress.ip\_address(gateway))  
 mask = int.from\_bytes(mask, 'big')  
 physical = hex(int.from\_bytes(physical, 'big'))[2:].upper()  
 physical = '-'.join(a + b for a, b in zip(physical[::2], physical[1::2]))  
  
 router = f"{gateway}/{mask}"  
  
 return f"Here: {host}, {physical}, via {iface}\nRouter: {router}"  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file is responsible for the concept of Scan ID.")  
 print("Scan ID is a base64 string that uniquely represents a network and the device in it running the software.")  
 print("It's built from:")  
 print(" - The host computer's name")  
 print(" - The network interface used")  
 print(" - The router's IPv4")  
 print(" - The subnet mark of the network")  
 print(" - The physical (MAC) address of the host computer")  
 print("This file provides both an encoder: get\_scan\_id() -> scan ID,")  
 print("and a decoder: parse\_scan\_id(scan ID) -> textual description of the network")

# ./Scanner/scans.db

[binary content]

# ./Scanner/SimpleScan.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from typing import Callable, Union  
  
from ipconfig import ipconfig  
  
  
def do\_simple\_scan(scan: Callable, all\_possible\_addresses: list[str], \*, results=False, repeats=3) -> list[str]:  
 """This is a wrapper for simple\* scans, like ARP or ICMP.  
  
 (\*) Simple means they are standardised:  
 [X] Have been @threadify-ied.  
 [X] Get list[str] of IPv4 addresses as input (post-threadify).  
 [X] Output list[bool] indicating online-ness of the addresses (index-correlated) (post-threadify).  
  
 Note: index-correlatedness, lists as input, and lists as output are handled by @threadify.  
 The requirements for the base function are just that it's of the form: `scan: str (IPv4 address) -> bool (connectivity)`.  
  
  
 Args:  
 scan (function): a @threadify-ied method from list[str] all\_possible\_addresses -> list[bool] online.  
 all\_possible\_addresses (list[str]): a list of IPv4 to test connectivity to.  
 results (bool, optional): Decides whether to print the results. Defaults to True.  
 repeats (int, optional): How many times should the full-range of addresses be scanned? Defaults to 3.  
  
 Returns:  
 list[str]: the addresses which replied, at least once, to the scan.  
 """  
 # if the amount of repeats is non-positive, the result is always empty.  
 if repeats < 1:  
 return []  
  
 # Parsing the title & protocol.  
 title = scan.\_\_name\_\_  
 protocol = "".join(char for char in title if char.isupper())  
  
 # Define a <lambda> that returns a list[str] of connectable addresses.  
 def get\_new():  
 return [  
 address for address, online  
 in zip(all\_possible\_addresses, scan(all\_possible\_addresses))  
 if online  
 ]  
  
 # Call it `repeats` times and unite all results into a set.  
 connectable\_addresses = set()  
 for \_ in range(repeats):  
 connectable\_addresses = connectable\_addresses.union(get\_new())  
  
 # Turn it into a sorted list (just for convenience, order doesn't matter).  
 connectable\_addresses = sorted(  
 connectable\_addresses,  
 key=lambda x: int(''.join(x.split('.')))  
 )  
  
 # Print if asked  
 if results:  
 print(  
 "There are",  
 len(connectable\_addresses),  
 protocol,  
 "connectable addresses in this subnet:"  
 )  
 print(' ' + '\n '.join(connectable\_addresses))  
  
 return connectable\_addresses  
  
  
def standardise\_simple\_scans(scans: list[Union[tuple[Callable, int], Callable]]) -> list[Callable]:  
 """This function standardises a collection of simple scans.  
 The argument is a list of tuples, where each tuple has two items:  
 (1) The scan (Callable), (2) The amount of repeats.  
  
 If a Callable is provided instead of a tuple,  
 it takes the default value of 1 repeat.  
  
 If a non-positive (i.e. negative or zero) amount of repeats is provided,  
 the scan is ignored.  
  
 It returns a list of actions, provided by `does\_simple\_scan`,  
 with updated names and docstrings.  
 `does\_simple\_scan` is a wrapper around `do\_simple\_scan`,  
 that provides the arguments: scan, ipconfig()["All Possible Addresses"], and repeats.  
  
 Args:  
 scans (list[tuple[Callable, int] + Callable]): a list of the scans.  
  
 Returns:  
 list[Callable]: a list of the actions -- scans times repeats.  
 """  
 # If `repeats` is not provided, default it to 1.  
 scans = [scan if isinstance(scan, tuple) else (scan, 1) for scan in scans]  
 # Filter only scans with a positive amount of repeats.  
 scans = [scan for scan in scans if scan[1] > 0]  
  
 # Wrapper around `do\_simple\_scan`, that provides the wanted arguments and returns a <lambda>.  
 def does\_simple\_scan(scan):  
 scan, repeats = scan  
 return (  
 lambda: do\_simple\_scan(  
 scan,  
 ipconfig()["All Possible Addresses"],  
 repeats=repeats)  
 )  
 lambdas = [does\_simple\_scan(scan) for scan in scans]  
  
 # Handle all the \_\_name\_\_s and \_\_doc\_\_s.  
 for (scan, repeats), method in zip(scans, lambdas):  
 prefix = f"{repeats} × " if repeats > 1 else ""  
 method.\_\_name\_\_ = prefix + scan.\_\_name\_\_  
 method.\_\_doc\_\_ = prefix + scan.\_\_doc\_\_  
 return lambdas  
  
  
def simple\_scan(scan: Callable, repeats: int) -> Callable:  
 """Wrapper around `do\_simple\_scan`,  
 that handles the `\_\_name\_\_` and `\_\_doc\_\_`.  
  
 Args:  
 scan (Callable): the scan to standardise and repeat.  
 repeats (int): the amount of repeats.  
  
 Returns:  
 Callable: a standardised simple scan.  
 """  
 def result():  
 return do\_simple\_scan(scan,  
 ipconfig()["All Possible Addresses"],  
 repeats=repeats)  
 prefix = f"{repeats} × " if repeats > 1 else ""  
 result.\_\_name\_\_, result.\_\_doc\_\_ = prefix + scan.\_\_name\_\_, prefix + scan.\_\_doc\_\_  
 return result  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module is responsible for the conceptual type called Simple Scan.")  
 print("It represents a scan that answers a list of criteria:")  
 print("[X] Have been @threadify-ied.")  
 print("[X] Get list[str] of IPv4 addresses as input (post-threadify).")  
 print("[X] Output list[bool] indicating online-ness of the addresses (index-correlated) (post-threadify).")  
 print("\nNote: index-correlatedness, lists as input, and lists as output are handled by @threadify.")  
 print("The requirements for the base function are just that it's of the form: `scan: str (IPv4 address) -> bool (connectivity)`.")

# ./Scanner/Sniffer.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from scapy.sendrecv import AsyncSniffer  
  
class Sniffer(AsyncSniffer):  
 references = []  
  
 def \_\_init\_\_(self, \*args, \*\*kwargs) -> None:  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
 Sniffer.references.append(self)  
   
 @staticmethod  
 def stopall():  
 for sniffer in Sniffer.references:  
 if sniffer.running:  
 sniffer.stop()  
 Sniffer.references = []  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file contains the Sniffer class,")  
 print("which extends scapy's AsyncSniffer,")  
 print("and upgrades it with a `.stopall` static function.")

# ./Scanner/test.db

[binary content]

# ./Scanner/tests\_log.txt

[irrelevant content]

# ./Scanner/times.txt

[irrelevant content]

# ./Scanner/util.py

from typing import Callable  
from import\_handler import ImportDefence  
with ImportDefence():  
 from io import StringIO  
 from math import floor, ceil  
 from queue import Queue  
 import sys  
 from threading import Thread, active\_count  
 from time import sleep  
 import sys  
 import os  
  
  
\_\_author\_\_ = 'Shaked Dan Zilberman'  
MAX\_THREADS: int = 50  
  
  
def color\_to\_hex(rgb) -> str:  
 """This function converts between two common representations of RGB colours.  
 It takes in a tuple or list (any iterable) that has 3 items, floats between 0 and 1,  
 and converts it into a hex string with hash symbol, like #00FF00 (green).  
  
 Args:  
 rgb (Iterable[float]): the input colour, as an iterable with 3 items, as floats between 0 and 1, representing the RGB channel strengths.  
  
 Returns:  
 str: a hex representation of an RGB colour, with 7 characters (#RRGGBB).  
 """  
 return '#%02x%02x%02x' % tuple([int(c \* 255) for c in rgb])  
  
  
def barstyle(name: str) -> str:  
 """Selects a style for @threadify's progress bar.  
  
 Available styles:  
 Dot Fill : ███████∙∙∙∙∙∙∙  
 Default : [─────── ]  
 Unstyled : [------- ]  
 Hash Fill : |####### |  
  
 Usage:  
 ```  
 def function():  
 # some code  
  
 function.options = {"format": style("default")}  
 function = threadify(function)  
 ```  
  
 For `name`:  
 Whitespaces are ignored, capital letters are `.lower`ed.  
 If the name was not found, uses the Default style.  
  
 Args:  
 name (str): the name of the style.  
  
 Returns:  
 str: the format.  
 """  
 try:  
 return {  
 "dotfill": " █∙ ",  
 # "squarefill": " ▣▢ ",  
 # "circlefill": " ◉◯ ",  
 "default": "[─ ]",  
 "unstyled": "[- ]",  
 "hashfill": "|# |"  
 }[name.lower().replace(' ', '')]  
 except NameError:  
 return "[─ ]"  
  
  
def threadify(f: Callable, silent=False):  
 """This function turns methods (tasks) into thread-based on-list execution.  
 Therefore, the execution will be faster.  
 Returns the same result as `[f() for \_ in input]`, but faster.  
  
 Return values from the task are saved in an array, which is the return value of the decorated function.  
 The values are not messy (which is what normally happens with threading), but organised according to the `args`.  
 Meaning, `returned[0] = f(args[0]); returned[1] = f(args(1))...`.  
  
 \*\*This function is blocking.\*\*  
 Its internals run asynchronously, but calling this will wait until all the tasks are done.  
 \*\*It will slurp up any printing done by other threads!\*\*  
 It will stop generating new threads if `globalstuff`'s `terminator` is set.  
  
 Also, if the arguments were `f(a, b, c)`, the new argument is `f(list[tuple(a, b, c)])`.  
 E.g., the function `add(x: int, y: int)`, if it's `@threadify`-ied, will be called by `add([(x0, y0), (x1, y1), (x2, y2)...])`.  
 If the function receives a single not-tuple argument, you can just put it in the list `[a, b, c, d]` -> f(a) + f(b) + f(c) + f(d).  
 If the function receives a single tuple as an argument, wrap it in another tuple: `[a: tuple, b: tuple, c: tuple, d: tuple]` -> `f([(a, ), (b, ), (c, ), (d, )])`.  
 If a code-based explanation for the logic is better, here are the relevant (pseudo)code pieces:  
 ```py  
 for arg in args:  
 a = arg if isinstance(arg, tuple) else (arg, )  
 f(\*a)  
 ```  
 Doesn't support keyword-arguments.  
 The amount of threads per decorated function call is limited by `MAX\_THREADS`.  
  
 The function can have an `f.options` dict as an attribute, overriding any of these:  
 ```  
 options={  
 # daemon: bool -- if True, the threads forcibly end when main ends.  
 # Else, they can continue running in the background.  
 "daemon": False,  
 # printing: bool -- should there be a progress bar?  
 "printing": True,  
 # min\_printing\_length: int -- the minimal size of the progress bar (in characters). The actual length can be larger, if the terminal is wide enough.  
 "min\_printing\_length": 10,  
 # format: str -- the format of the progress bar.  
 # For printing\_length=6 and this format, after half the execution, the bar would look like "[--- ] (50%)".  
 "format": "[- ]",  
 # output: bool -- should the output (via print()s) of the tasks be logged?  
 "output": True,  
 # give: str -- decides what the function returns: "results" is the return value of the tasks; "output" is the printing of the tasks; "both" is a tuple of both.  
 # Any other value defaults to "results".  
 "give": "results"  
 }  
 ```  
  
 Args:  
 f (function): The task to be turned into a threaded task.  
 silent (bool): forces `options["output"]` and `options["printing"]` to be `False`.  
  
 Returns:  
 list: a list of values returned from the multiple calls to the function, sorted by the call order (i.e. not disorganised by the threads).  
 """  
 # Set up the options dictionary with default values.  
 options = {  
 "daemon": False,  
 "printing": True,  
 "min\_printing\_length": 10,  
 "format": "[─ ]",  
 "output": True,  
 "give": "results"  
 }  
 # Add options set via `f.options`, if such an attribute exists.  
 try:  
 options = {\*\*options, \*\*f.options}  
 except AttributeError:  
 pass  
  
 name = f.\_\_name\_\_  
 name = name.replace("\_base", "")  
 if silent:  
 options["output"] = False  
 options["printing"] = False  
  
 def wrapper(args: list[tuple] | list) -> list | str | tuple[list, str]:  
 if not isinstance(args, list):  
 raise TypeError(  
 "Threadify-ied functions must receive a single argument of type list."  
 )  
  
 # The return values from the function calls.  
 values = [None] \* len(args)  
 # The exceptions raised during tasks thread-safe queue.  
 fails = Queue(maxsize=len(args))  
  
 # Define a task inner wrapper for `f`  
 def task(func, arg, index):  
 # Convert arg to tuple if needed ("If the function receives a  
 # single not-tuple argument,..." in docstring)  
 args = arg if isinstance(arg, tuple) else (arg, )  
 try:  
 # Execute the function & save to `values` list.  
 values[index] = func(\*args)  
 except Exception as e:  
 # Catch any exception and add it to the `fails` queue  
 fails.put(e)  
 return  
  
 # Rename `task` to user-friendly name  
 task.\_\_name\_\_ = f.\_\_name\_\_ + '\_task'  
  
 real\_stdout = sys.stdout  
 output = StringIO()  
 if options["output"]:  
 # Redirect printing  
 sys.stdout = output  
  
 # Create Thread objects  
 threads = [Thread(target=task, args=(f, x, i), daemon=options["daemon"])  
 for i, x in enumerate(args)]  
  
 def threadify\_start\_threads(threads: list):  
 from globalstuff import terminator  
 # Activate the threads, waiting for threads to be freed if needed.  
 for thread in threads:  
 thread.start()  
 while active\_count() >= MAX\_THREADS and MAX\_THREADS > 0:  
 # print(active\_count(), "threads active.", file=real\_stdout)  
 sleep(0.02)  
 if terminator.is\_set():  
 return  
 if terminator.is\_set():  
 return  
  
 # Rename `threadify\_start\_threads` to user-friendly name  
 threadify\_start\_threads.\_\_name\_\_ = f.\_\_name\_\_ + '\_threadify\_master'  
  
 starter = Thread(target=threadify\_start\_threads, args=(threads, ))  
 starter.start()  
  
 # Print a progress bar if requested  
 if options["printing"]:  
 width = os.get\_terminal\_size().columns - \  
 len(f"@threadify: {name} ")  
 if width < options["min\_printing\_length"]:  
 width = options["min\_printing\_length"]  
 while any([thread.is\_alive() for thread in threads]):  
 ratio = sum([thread.is\_alive()  
 for thread in threads]) / len(args)  
  
 print(  
 f"@threadify: {name} {progress(ratio, options['format'], width)} \r",  
 end='',  
 file=real\_stdout  
 )  
 sleep(0.1)  
 print(  
 f"@threadify: {name} {progress(0, options['format'], width)} \r",  
 end='',  
 file=real\_stdout  
 )  
 print("\n")  
  
 # Join all threads  
 # You have to join `starter` first, because if somehow some thread is  
 # still not active, joining it will raise a RuntimeError.  
 # Also, if the `terminator` was set, not all threads will have been started, and you need to catch those RuntimeErrors.  
 starter.join()  
 for thread in threads:  
 try:  
 thread.join()  
 except RuntimeError:  
 pass  
  
 # Restore printing  
 sys.stdout = real\_stdout  
  
 # If any exceptions happened, print them orderly and raise another.  
 if not fails.empty():  
 while not fails.empty():  
 err = fails.get()  
 print(type(err), err, sep="\n")  
 raise Exception(  
 "@threadify-ied function has raised some exceptions."  
 )  
  
 # Handle the output from the tasks  
 output = output.getvalue()  
 if options["output"] and output.strip() != "":  
 print("\n\nTasks' output:\n", output, sep='')  
 print("\n\n")  
 # print()  
  
 # Returning logic.  
 if options["give"] == "output":  
 return output  
 if options["give"] == "both":  
 return values, output  
 return values  
  
 # Make `wrapper` inherit `f`'s properties.  
 wrapper.\_\_name\_\_ = f.\_\_name\_\_  
 wrapper.\_\_doc\_\_ = f.\_\_doc\_\_  
 return wrapper  
  
  
def progress(ratio: float, form: str, width: int) -> str:  
 """This function generates a printable progress bar.  
 The bar is filled according to `ratio`, which ranges from 0 (all done) to 1 (nothing done).  
 Notice, 0 means the bar is completely filled, and 1 means the bar is completely empty, not the opposite.  
 The ratio should decrease from 1 to 0, in order for the bar to fill up.  
 The characters are taken from `form`, which is a 4-chraracter string specifying the style of the bar.  
 `form` contains the start character, the filler, the no-filler, and the end character; in that order.  
 `width` is the amount of characters the progress bar can expand into, usually the width of the console, minus some padding.  
  
 Example:  
 progress(0.3, '[- ]', 20) -> '[-------- ] (70%)' (19 characters long)  
  
 Args:  
 ratio (float): the ratio of unfilled/all of the bar. Ranges from 0 to 1.  
 form (str): the characters of the progress bar. 4 characters long.  
 width (int): the amount of characters the bar may expand into and take up.  
  
 Returns:  
 str: a progress bar of this format: `{start}{done}{waiting}{end} ({percent}%)`  
 """  
 if len(form) < 4:  
 form = form.rjust(4)  
 start, fill, nofill, end, \*ignore = tuple(form)  
 width -= len("[] (100%)")  
  
 done\_length: int = ceil(width \* (1 - ratio))  
 done: str = fill \* done\_length  
  
 wating\_length: int = floor(width \* (ratio))  
 waiting: str = nofill \* wating\_length  
  
 percent: int = ceil(100 \* (1 - ratio))  
  
 return f"{start}{done}{waiting}{end} ({percent}%)"  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This is a utility module.")  
 print("It provides various methods, content managers, and decorators,")  
 print("That are used all throughout the software.")

# ./Scanner/wireshark 2023-03-04.pcapng

[binary content]

# ./Scanner/\_\_download\_from\_git.bat

@echo off  
color 0A  
title Clone from Git  
if exist "network-utilities\" rmdir /S /Q network-utilities  
"D:\Git\bin\git.exe" clone https://github.com/ShZil/network-utilities.git network-utilities  
pause

# ./Scanner/analyses\DeviceProfile.py

from typing import Any  
from gui.dialogs import get\_string, popup  
  
  
def \_match\_device(address):  
 from NetworkStorage import NetworkStorage, SpecialInformation, match  
 try:  
 entity = match(address)  
 for item in NetworkStorage():  
 if entity.equals(item):  
 return item  
 except ValueError:  
 name = address.lower()  
 if name == "unknown":  
 return None  
 for item in NetworkStorage():  
 if item.name.lower() == name:  
 return item  
 role = address.lower()  
 have\_roles = SpecialInformation()['role']  
 for entity in have\_roles:  
 if SpecialInformation()[entity, 'role'].lower() == role:  
 return entity  
 return None  
  
  
def \_construct\_content(info: dict) -> str:  
 from NetworkStorage import nothing  
  
 def \_transform\_item(item: tuple[str, str]) -> tuple[str, str]:  
 key, value = item  
  
 try:  
 if nothing[key] == value:  
 return '', ''  
 except TypeError: # occurs on properties which `nothing` doesn't have, e.g. `os`. i.e. anything from Special Information.  
 pass  
  
 if key in ['mac', 'ipv6']:  
 value = value.upper()  
   
 if key in ['mac']:  
 key = key.upper()  
 elif key.startswith('ip'):  
 key = 'IP' + key[2:]  
 elif not key.isupper():  
 key = ' '.join([word if word.isupper() else word.capitalize() for word in key.split(' ')]) # title case, but doesn't affect acronyms.  
   
 return key, value  
  
 info = dict(map(\_transform\_item, info.items()))  
 markdowned = [  
 f"### {key}:\n`{value}`"  
 for key, value in info.items()  
 if key != ''  
 ]  
 return '\n\n'.join(markdowned)  
  
  
def device\_profile(\*\_):  
 address = get\_string("Device Profile", "Insert device's MAC / IP / IPv6 Address, or Device Name, or Role:")  
 entity = \_match\_device(address)  
 if entity is None:  
 popup("Device Profile", f"The device was not found.\nCheck whether you wrote the address correctly.\nThe address: `{address}`", warning=True)  
 return  
 regular\_info = entity.to\_dict() # contains mac, ip, ipv6, and name.  
 from NetworkStorage import SpecialInformation  
 special\_info = SpecialInformation()[entity] # contains possible additional information, like OS (from OS-ID), or Device Discovery status, or role (e.g. router).  
 information = {\*\*regular\_info, \*\*special\_info} # merge all information to one dictionary  
 popup("Device Profile", \_construct\_content(information), info=True)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("This module contains the `device\_profile` function,")  
 print("i.e. the code for the Device Profile analysis.")

# ./Scanner/analyses\LogPackets.py

from PacketSniffer import PacketSniffer  
  
  
def log\_packets():  
 from gui.dialogs import popup  
 packets = [str(packet.summary()) for packet in PacketSniffer()]  
 packets = [packet.replace('>', '&gt;').replace('<', '&lt;').split('/') for packet in packets]  
 packets = ['<span class="ip">['.join(layers) + ' ]</span>'\*len(layers) for layers in packets]  
 packets = [f"<tr><td><span>[ {packet}</td></tr>" for packet in packets]  
 packets = '\n'.join(packets)  
 packets = f"<table>{packets}</table>"  
 popup("Packets", packets, info=True)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module contains the `log\_packets` function,")  
 print("i.e. the code of the Log Packets analysis.")

# ./Scanner/analyses\OS\_ID.py

def operating\_system\_fingerprinting() -> None:  
 """This function does the OS-ID scan-like action.  
 It's an infinite action, so this method starts a thread to run that action.  
 (Updated: this method creates an observer of PacketSniffer)  
 For further explanation, execute this file alone, or read OS-ID's information.  
 """  
 from NetworkStorage import here, SpecialInformation, NetworkEntity, nothing  
 from PacketSniffer import PacketSniffer  
 from scapy.all import Ether, IP  
  
 def \_determine\_os(packet):  
 # do more testing on this \*\*\*\*\*\*\*\*\*\*\*\*\*  
 return "Linux or Android" if packet.ttl <= 64 else "Windows"  
  
 def fingerprint(packet):  
 # for each packet,  
 # if it originates at some other computer...  
 if packet.src == here.ip:  
 return  
 entity = NetworkEntity(mac=packet[Ether].src, ip=packet[IP].src, ipv6=nothing.ipv6, name="Unknown")  
 # ...that has not yet been OS-ID'd  
 if entity in SpecialInformation()['OS']:  
 return  
 # Then guess what it's OS is, using `\_determine\_os`, and save it to `SpecialInformation`.  
 SpecialInformation()[entity, 'OS'] = \_determine\_os(packet)  
   
 PacketSniffer().add\_observer(fingerprint)  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file is responsible for the OS-ID scan-like action.")  
 print("This action guesses the Operating System of scanned devices,")  
 print("and of devices that sent an IP packet (for any reason) that was sniffed.")  
 print("It touches on two pieces of architecture:")  
 print(" PacketSniffer - get the packets as input.")  
 print(" SpecialInformation - output the OS guess information.")

# ./Scanner/analyses\VendorMapping.py

from util import threadify  
from import\_handler import ImportDefence  
with ImportDefence():  
 import sqlite3  
 import requests  
 from threading import Lock  
  
  
class MACVendorDict:  
 \_instance = None  
 path = 'mac\_vendors.db'  
 CREATE\_QUERY = '''CREATE TABLE IF NOT EXISTS mac\_vendors (mac TEXT PRIMARY KEY, vendor TEXT)'''  
 SELECT\_QUERY = 'SELECT vendor FROM mac\_vendors WHERE mac=?'  
 INSERT\_QUERY = 'INSERT INTO mac\_vendors (mac, vendor) VALUES (?, ?)'  
  
 def \_\_new\_\_(cls):  
 if cls.\_instance is None:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 cls.\_instance.lock = Lock()  
 with sqlite3.connect(cls.path) as conn:  
 cursor = conn.cursor()  
 cursor.execute(cls.CREATE\_QUERY)  
 return cls.\_instance  
  
 def \_\_contains\_\_(self, mac):  
 with self.lock, sqlite3.connect(self.path) as conn:  
 cursor = conn.cursor()  
 cursor.execute(self.SELECT\_QUERY, (mac,))  
 return cursor.fetchone() is not None  
  
 def \_\_setitem\_\_(self, mac, text):  
 with self.lock, sqlite3.connect(self.path) as conn:  
 cursor = conn.cursor()  
 cursor.execute(self.INSERT\_QUERY, (mac, text))  
 conn.commit()  
  
 def \_\_getitem\_\_(self, mac):  
 with self.lock, sqlite3.connect(self.path) as conn:  
 cursor = conn.cursor()  
 cursor.execute(self.SELECT\_QUERY, (mac,))  
 result = cursor.fetchone()  
 return result[0] if result else None  
  
 def get(self, mac, default=None):  
 return self[mac] if mac in self else default  
  
  
def vendor\_mapper(mac):  
 try:  
 url\_mac = '%3A'.join(mac.replace(':', '-').split('-'))  
 response = requests.get(f'https://hwaddress.com/?q={url\_mac}')  
 except requests.exceptions.ConnectTimeout:  
 return  
 if response.status\_code != 200:  
 return  
 html = response.text  
 a\_tag = html.split('<tr><th>Company</th><td>')[1].split('</td></tr>')[0]  
 vendor = a\_tag.split('>')[1].split('<')[0]  
 return vendor  
  
  
@threadify  
def mapper\_wrapper(entity):  
 from NetworkStorage import nothing, SpecialInformation  
 if entity.mac == nothing.mac:  
 return  
 if entity.mac in MACVendorDict():  
 return  
 try:  
 vendor = vendor\_mapper(entity.mac)  
 except IndexError:  
 return  
 if vendor is None:  
 return  
 MACVendorDict()[entity.mac] = vendor # I could save just the first 3 bytes, but, sometimes the OUI extends further, so I'd rather save the entire MAC.  
 SpecialInformation()[entity, 'Network Card Vendor'] = vendor  
  
  
def vendor\_mapping():  
 from NetworkStorage import NetworkStorage  
 MACVendorDict()  
 mapper\_wrapper(list(NetworkStorage()))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This analysis uses an API to map MAC physical network card address,")  
 print("to a vendor / manufacturer according to the organizationally unique identifier (OUI) found in the start of the MAC address,")  
 print("using `hwaddress.com`.")  
 print("Saves a cache using SQL to `mac\_vendors.db`.")

# ./Scanner/analyses\\_\_init\_\_.py

# ./Scanner/analyses\\_\_pycache\_\_\DeviceProfile.cpython-310.pyc

[binary content]

# ./Scanner/analyses\\_\_pycache\_\_\LogPackets.cpython-310.pyc

[binary content]

# ./Scanner/analyses\\_\_pycache\_\_\OS\_ID.cpython-310.pyc

[binary content]

# ./Scanner/analyses\\_\_pycache\_\_\VendorMapping.cpython-310.pyc

[binary content]

# ./Scanner/analyses\\_\_pycache\_\_\\_\_init\_\_.cpython-310.pyc

[binary content]

# ./Scanner/drawio-diagrams\ability.drawio

[binary content]

# ./Scanner/drawio-diagrams\ARPSweep.drawio

[binary content]

# ./Scanner/drawio-diagrams\db.drawio

[binary content]

# ./Scanner/drawio-diagrams\ICMPSweep.drawio

[binary content]

# ./Scanner/drawio-diagrams\LiveARP.drawio

[binary content]

# ./Scanner/drawio-diagrams\LiveICMP.drawio

[binary content]

# ./Scanner/drawio-diagrams\memorise.drawio

[binary content]

# ./Scanner/drawio-diagrams\modules extended.drawio

[binary content]

# ./Scanner/drawio-diagrams\modules.drawio

[binary content]

# ./Scanner/drawio-diagrams\NetworkStorage.drawio

[binary content]

# ./Scanner/drawio-diagrams\overview.drawio

[binary content]

# ./Scanner/drawio-diagrams\PacketSniffer.drawio

[binary content]

# ./Scanner/drawio-diagrams\register.drawio

[binary content]

# ./Scanner/drawio-diagrams\Screens.drawio

[binary content]

# ./Scanner/drawio-diagrams\TCP-Ports.drawio

[binary content]

# ./Scanner/drawio-diagrams\threadify.drawio

[binary content]

# ./Scanner/drawio-diagrams\done\ARPSweep.png

[binary content]

# ./Scanner/drawio-diagrams\done\db.png

[binary content]

# ./Scanner/drawio-diagrams\done\ICMPSweep.png

[binary content]

# ./Scanner/drawio-diagrams\done\LiveARP.png

[binary content]

# ./Scanner/drawio-diagrams\done\LiveICMP.png

[binary content]

# ./Scanner/drawio-diagrams\done\modules extended.png

[binary content]

# ./Scanner/drawio-diagrams\done\modules.png

[binary content]

# ./Scanner/drawio-diagrams\done\NetworkStorage.png

[binary content]

# ./Scanner/drawio-diagrams\done\overview.png

[binary content]

# ./Scanner/drawio-diagrams\done\PacketSniffer.png

[binary content]

# ./Scanner/drawio-diagrams\done\register.png

[binary content]

# ./Scanner/drawio-diagrams\done\Screens.png

[binary content]

# ./Scanner/drawio-diagrams\done\TCP-Ports.png

[binary content]

# ./Scanner/drawio-diagrams\done\threadify.png

[binary content]

# ./Scanner/fonts\BainsleyBold.ttf

[binary content]

# ./Scanner/fonts\Consolas.ttf

[binary content]

# ./Scanner/fonts\Segoe UI Symbol.ttf

[binary content]

# ./Scanner/fonts\Symbola.ttf

[binary content]

# ./Scanner/gui\Activation.py

from register import Register  
from gui.ScanClasses import DummyScan, Analysis, Scan  
from gui.dialogs import popup  
from gui.AppState import State  
  
  
def activate(\*x):  
 if not State().ask\_for\_permission():  
 return  
  
 s: Scan = State().highlighted\_scan  
 if Register().is\_running(s.name) or s.is\_running:  
 import db  
 from datetime import timedelta  
 time = int(db.get\_information\_about\_scan(s.name)[2])  
 indefinite\_message = "running indefintely forever" if Register().is\_infinite\_scan(s.name) else "not running indefinitely, and will stop at some point soon."  
 if time == 0:  
 time\_message = f"The datebase lists this scan's time estimation as infinite."  
 else:  
 time\_message = f"The database lists this scan at roughly {time} second(s). So it'll probably be done in {str(timedelta(seconds=time))}[^1]"  
 popup(  
 "Cannot run scan",  
 f"\*\*This scan is already running!\*\*\n{s.name} \n\  
 This scan is {indefinite\_message}. \n\  
 {time\_message} \  
 \n\n\*\*TIP:\*\* You can see which scans are currently running according to the ellipsis (...)! \n\  
 \n\n\*\*Note:\*\* Time estimate might be off, depending on your network size and scan configuration.",  
 error=True  
 )  
 return  
  
 # print(f"Play {s.name}!")  
 if s is DummyScan():  
 popup("Cannot start scan", f"Select a scan first!\nYou can select a scan by left-clicking on a name in the right column of the screen.\n\  
 You can read all about each scan by clicking on it, and then on the Information button on top.", warning=True)  
 if not isinstance(s, Analysis):  
 if not popup("Start scan", f"Starting the scan: {s.name}", cancel=True):  
 popup(  
 "Canceled scan",  
 "The scan has been cancelled.\n\n<sub><sup>Cancel culture, I'm telling ya.</sup></sub>",  
 warning=True  
 )  
 return  
 Register().start(s.name, s.act, s.finished)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file is responsible for any methods called by the Activation Button (▶) in Scan Screen.")

# ./Scanner/gui\App.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from kivy.app import App  
 from kivy.uix.screenmanager import ScreenManager, FadeTransition  
  
from gui.AppState import State  
from gui.Screens.StartScreen import StartScreen  
from gui.Screens.ScanScreen import ScanScreen  
from gui.Screens.SaveScreen import SaveScreen  
from gui.Screens.KnowScreen import KnowScreen  
from gui.Screens.ViewScreen import ViewScreen  
from gui.Hover import Hover  
  
  
class MyApp(App):  
 """The main application, using `kivy`.  
 Includes five screens:  
 1. ScanScreen  
 2. SaveScreen  
 3. KnowScreen  
 4. StartScreen  
 5. ViewScreen (only accessible through `SaveScreen.ImportButton`)  
  
  
 Args:  
 App (kivy): the kivy base app.  
 """  
  
 def build(self):  
 self.title = 'Local Network Scanner'  
 self.icon = 'favicon.png'  
 from kivy.core.window import Window  
 Window.size = (1300, 800)  
  
 screens = ScreenManager(transition=FadeTransition())  
 State().setScreenManager(screens)  
  
 screens.add\_widget(StartScreen())  
 screens.add\_widget(ScanScreen())  
 screens.add\_widget(SaveScreen())  
 screens.add\_widget(KnowScreen())  
 screens.add\_widget(ViewScreen())  
  
 State().screen('Start')  
  
 Hover.start()  
  
 return screens  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file handles a class called MyApp (yes, tutorial-ly, I know) which extends kivy's App.")  
 print("It builds the kivy GUI, plain and simple.")

# ./Scanner/gui\AppState.py

from import\_handler import ImportDefence  
with ImportDefence():  
 import win32api  
 from kivy.clock import Clock  
  
from gui.ScanClasses import DummyScan  
  
  
class State:  
 \_instance = None  
  
 def \_\_new\_\_(cls):  
 if cls.\_instance is None:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 cls.\_instance.screenManager = None  
 cls.\_instance.currentScreen = None  
 cls.\_instance.permission = False  
 cls.\_instance.highlighted\_scan = DummyScan()  
 cls.\_instance.scans = [DummyScan()]  
 return cls.\_instance  
  
 def setScreenManager(self, screens):  
 self.screenManager = screens  
  
 def screen(self, name=None):  
 if name is None:  
 return self.currentScreen  
 from gui.Hover import Hover  
 Hover.enter(name)  
 self.screenManager.current = name  
 self.currentScreen = name  
  
 def resize\_callback(self, \*\_):  
 # Called from Hover's `.\_bind`.  
 def \_resize\_callback(\*\_):  
 self.scan(self.highlighted\_scan)  
 self.highlighted\_scan.select(0)  
  
 Clock.schedule\_once(\_resize\_callback, 0.15)  
  
 def scan(self, scan):  
 if scan not in self.scans:  
 self.scans.append(scan)  
 self.highlighted\_scan = scan  
 for scan in self.scans:  
 scan.deselect()  
  
 def ask\_for\_permission(self):  
 if not self.permission:  
 self.permission = win32api.MessageBox(  
 0,  
 'Do you have legal permission to execute scans on this network?',  
 'Confirm permission',  
 0x00000004  
 ) == 6  
 return self.permission  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file is responsible for the State class/object (Singleton Pattern).")  
 print("It keeps track of the state in terms of:")  
 print(" - Screen Manager of kivy, and which screen is currently displayed.")  
 print(" All calls to change screen must pass through here.")  
 print(" - Permission to scan the network is handled here.")  
 print(" - Highlighting a scan button is handled here.")

# ./Scanner/gui\Configuration.py

from gui.AppState import State  
from gui.ScanClasses import DummyScan  
from gui.dialogs import popup  
  
  
def display\_configuration(\*\_):  
 if State().highlighted\_scan is None:  
 name = "scans"  
 elif State().highlighted\_scan is DummyScan():  
 name = "scans"  
 else:  
 name = State().highlighted\_scan.name  
 popup(f"Configuration of {name}", "Coming soon.")  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file is responsible for any methods called by the Configuration Button (⚙) in Scan Screen.")

# ./Scanner/gui\Diagrams.py

from import\_handler import ImportDefence  
with ImportDefence():  
 import networkx as nx  
 import tkinter as tk  
 from kivy.graphics import Color, Ellipse, Rectangle, Line  
 from kivy.clock import Clock  
 from matplotlib import pyplot as plt  
  
from globalstuff import \*  
from util import color\_to\_hex  
from CacheDecorators import one\_cache  
  
from abc import ABC, abstractmethod  
from typing import ContextManager  
  
  
class Diagrams:  
 """This class handles all the Diagrams that present the network graph (`G`).  
 To use, simply create an instance (uses Singleton), and do `.add` to your diagram.  
 """  
 \_instance = None  
  
 def \_\_new\_\_(cls):  
 """Override the \_\_new\_\_ method to create only one instance of the class -- Singleton pattern."""  
 if not cls.\_instance:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 cls.\_instance.diagrams = []  
 return cls.\_instance  
  
 def add(self, diagram):  
 if isinstance(diagram, Diagram) \  
 and isinstance(diagram, ContextManager):  
 self.diagrams.append(diagram)  
  
 def update(self):  
 for diagram in self.diagrams:  
 diagram.update()  
  
  
class Diagram(ABC):  
 """This is an abstarct base class for diagrams that render the network.  
 """  
 @abstractmethod  
 def \_\_init\_\_(self):  
 pass  
  
 @abstractmethod  
 def update(self):  
 pass  
  
 @abstractmethod  
 def color(self, r, g, b):  
 pass  
  
 @abstractmethod  
 def rectangle(self, x, y, w, h):  
 pass  
  
 @abstractmethod  
 def circle(self, x, y, node):  
 pass  
  
 @abstractmethod  
 def line(self, x0, y0, x1, y1, stroke):  
 pass  
  
 @abstractmethod  
 def \_\_contains\_\_(self, pos):  
 pass  
  
  
class TKDiagram(Diagram, ContextManager):  
 """A diagram under `tkinter` window.  
 Uses tk.Canvas.  
 Doesn't actually close until `is\_kivy\_running` is set to False, and another closing is attempted.  
 Uses the Singleton pattern.  
  
 Extends:  
 Diagram (abstract class): allows for this class to be used as a diagram.  
 ContextManager (type): allows for this class to be used as a context manager (for rendering).  
 """  
 \_instance = None  
  
 def \_\_new\_\_(cls):  
 if not cls.\_instance:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 cls.\_instance.root = tk.Tk()  
 cls.\_instance.root.title("Network Diagram")  
 cls.\_instance.width, cls.\_instance.height = DIAGRAM\_DIMENSIONS  
  
 cls.\_instance.canvas = tk.Canvas(  
 cls.\_instance.root,  
 bg=color\_to\_hex(bg\_color),  
 height=cls.\_instance.height,  
 width=cls.\_instance.width,  
 borderwidth=0,  
 highlightthickness=0  
 )  
 cls.\_instance.canvas.pack(expand=True, fill='both')  
  
 cls.\_instance.radius = DIAGRAM\_POINT\_RADIUS  
 cls.\_instance.color\_cache = '#000000'  
  
 cls.\_instance.canvas.bind('<Configure>', cls.\_instance.resize)  
 cls.\_instance.update()  
  
 cls.\_instance.hide()  
 cls.\_instance.root.protocol("WM\_DELETE\_WINDOW", cls.\_instance.try\_close)  
 return cls.\_instance  
  
 def \_\_init\_\_(self):  
 pass  
  
 def \_\_exit\_\_(self, exception\_type, exception\_value, exception\_traceback):  
 pass  
  
 def color(self, r, g, b):  
 self.color\_cache = color\_to\_hex((r, g, b))  
  
 def rectangle(self, x, y, w, h):  
 self.canvas.create\_rectangle(x, y, w, h, fill=self.color\_cache)  
  
 def circle(self, x, y, node):  
 r = self.radius  
 x0 = x - r  
 y0 = y - r  
 x1 = x + r  
 y1 = y + r  
 self.canvas.create\_text(  
 x0,  
 y0 - 30,  
 text=node.to\_string('\n'),  
 fill=self.color\_cache,  
 font=("Consolas", 10),  
 justify=tk.CENTER  
 )  
 return self.canvas.create\_oval(  
 x0, y0, x1, y1, fill=self.color\_cache)  
  
 def line(self, x0, y0, x1, y1, stroke):  
 self.canvas.create\_line(  
 x0, y0, x1, y1, width=stroke, fill=self.color\_cache)  
  
 def \_\_contains\_\_(self, pos):  
 return True  
  
 # Window management -- closing, hiding, showing, resizing.  
 def try\_close(self):  
 """To prevent the user from really closing this window if the source (kivy) is still open."""  
 from globalstuff import is\_kivy\_running  
 if is\_kivy\_running:  
 self.hide()  
 else:  
 self.root.destroy()  
  
 def hide(self):  
 self.root.withdraw()  
  
 def show(self):  
 self.root.update()  
 self.root.deiconify()  
  
 def resize(self, event):  
 # `geomery` is of the form "{width}x{height}+{x}+{y}"  
 geometry = self.root.geometry().replace('+', 'x')  
 self.width, self.height, \*\_ = map(int, geometry.split('x'))  
 self.update()  
  
 def update(self):  
 try:  
 self.root.after(0, lambda \*\_: render\_diagram(self, 0, 0, self.width, self.height, bg\_color, fg\_color, -50))  
 except RuntimeError:  
 pass # raised when terminating because tkinter cannot find its main thread.  
  
  
class PlotDiagram(Diagram, ContextManager):  
 """A diagram in a `matplotlib.pyplot` window.  
 Uses the Singleton pattern.  
  
 Extends:  
 Diagram (abstract class): allows for this class to be used as a diagram.  
 ContextManager (type): allows for this class to be used as a context manager (for rendering).  
 """  
  
 \_instance = None  
  
 def \_\_new\_\_(cls):  
 if not cls.\_instance:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 fig, ax = plt.subplots()  
 cls.\_instance.fig = fig  
 cls.\_instance.ax = ax  
 cls.\_instance.manager = plt.get\_current\_fig\_manager()  
 fig.canvas.mpl\_connect('close\_event', cls.\_instance.hide)  
 return cls.\_instance  
  
 def \_\_init\_\_(self):  
 pass  
  
 def \_\_enter\_\_(self):  
 return self  
  
 def \_\_exit\_\_(self, exception\_type, exception\_value, exception\_traceback):  
 pass  
  
 def color(self, r, g, b):  
 pass  
  
 def rectangle(self, x, y, w, h):  
 pass  
  
 def circle(self, x, y, node):  
 pass  
  
 def line(self, x0, y0, x1, y1, stroke):  
 pass  
  
 def \_\_contains\_\_(self, pos):  
 return True  
  
 def update(self):  
 self.ax.clear()  
 H = G.copy()  
 pos = nx.kamada\_kawai\_layout(H)  
 nx.draw(H, pos, ax=self.ax)  
   
 plt.draw()  
  
 def show(self):  
 self.manager.window.show()  
   
 def hide(self):  
 self.manager.window.hide()  
  
  
class KivyDiagram(Diagram, ContextManager):  
 """A diagram under `kivy` window, specifically the `Scan` screen, more specifically the `ScanScreenMiddleDiagram` widget.  
 Uses kivy's widget.canvas.  
 Linked to a kivy widget with `self.widget` and `.set\_widget(widget)`.  
 There has to be a widget set before rendering (entering the context manager)!  
 You cannot change the widget once you set it.  
 Uses the Singleton pattern.  
  
 Extends:  
 Diagram (abstract class): allows for this class to be used as a diagram.  
 ContextManager (type): allows for this class to be used as a context manager (for rendering).  
 """  
  
 \_instance = None  
  
 def \_\_new\_\_(cls):  
 if not cls.\_instance:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 cls.\_instance.widget = None  
 cls.\_instance.radius = DIAGRAM\_POINT\_RADIUS  
 return cls.\_instance  
  
 def \_\_init\_\_(self):  
 pass  
  
 @one\_cache  
 def set\_widget(self, widget):  
 self.widget = widget  
 return widget  
  
 def \_\_enter\_\_(self):  
 assert self.widget is not None  
 self.widget.canvas.\_\_enter\_\_()  
 return self  
  
 def \_\_exit\_\_(self, exception\_type, exception\_value, exception\_traceback):  
 assert self.widget is not None  
 self.widget.canvas.\_\_exit\_\_()  
  
 def color(self, r, g, b):  
 Color(r, g, b)  
  
 def rectangle(self, x, y, w, h):  
 Rectangle(pos=(x, y), size=(w, h))  
  
 def circle(self, x, y, node):  
 r = self.radius  
 Ellipse(pos=(x - r, y - r), size=(2 \* r, 2 \* r))  
  
 def line(self, x0, y0, x1, y1, stroke):  
 Line(points=(x0, y0, x1, y1), width=stroke)  
  
 def \_\_contains\_\_(self, pos):  
 assert self.widget is not None  
 x, y = pos  
 r = self.radius  
 collides = lambda x0, y0: self.widget.collide\_point(x0, y0)  
 if r == 0:  
 return collides(x, y)  
 return collides(x + r, y + r) and collides(x - r, y - r)  
  
 def update(self, \*\_):  
 assert self.widget is not None  
 Clock.schedule\_once(lambda \*\_: render\_diagram(  
 self,  
 \*self.widget.pos,  
 \*self.widget.size,  
 bg\_color,  
 fg\_color,  
 -TITLE\_HEIGHT  
 )  
 )  
  
  
def render\_diagram(draw, x, y, w, h, bg, fg, dh=0):  
 """An abstract representation of the algorithm used to render the diagram.  
 To interface with ~reality~ the screen, it uses the `draw` argument,  
 which is a context manager supporting various methods.  
 Currently, there are two implementations: `KivyDiagram` and `TKDiagram`.  
 """  
 scale = min(w, h + dh) \* DIAGRAM\_SCALE  
 stroke = 1  
 H = G.copy()  
  
 pos = nx.kamada\_kawai\_layout(H, center=(x + w / 2, y + h / 2), scale=scale)  
  
 with draw:  
 draw.color(\*bg)  
 draw.rectangle(x, y, w, h)  
  
 draw.color(\*fg)  
 for node, (x0, y0) in pos.items():  
 if (x0, y0) in draw:  
 draw.circle(x0, y0, node)  
  
 for edge in H.edges:  
 x0, y0 = pos[edge[0]]  
 x1, y1 = pos[edge[1]]  
 if (x0, y0) in draw and (x1, y1) in draw:  
 draw.line(x0, y0, x1, y1, stroke)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file handles Diagrams that present the network graph.")  
 print("Currently two types: KivyDiagram and TKDiagram.")  
 print("To create a new diagram, simply inherit Diagram and typing.ContextManager,")  
 print("and implement all the methods.")

# ./Scanner/gui\dialogs.py

from time import sleep  
from typing import Callable  
from import\_handler import ImportDefence  
with ImportDefence():  
 import win32api  
 import win32con  
 import markdown2  
 import PySimpleGUIQt as sg  
 import threading  
 import time  
 from enum import Enum  
 import queue  
  
  
class IconType(Enum):  
 INPUT = 5  
 ERROR = 4  
 WARNING = 3  
 QUESTION = 2  
 INFO = 1  
 NOTHING = 0  
  
POPUP\_WINDOW\_SIZE = (1000, 600)  
def get\_CSS():  
 try:  
 return f"<style>{open('./gui/popup\_style.css', 'r').read()}</style>"  
 except OSError:  
 return f"<style></style>"  
POPUP\_WINDOW\_LOOP\_TIMEOUT\_MS = 500  
  
  
class PopupManager:  
 \_instance = None # Private class variable to store the singleton instance  
  
 def \_\_new\_\_(cls):  
 if cls.\_instance is None:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 sg.theme('DarkGreen5')  
 cls.\_instance.waiting = queue.Queue()  
 cls.\_instance.popup\_thread = threading.Thread(target=cls.\_instance.\_popup\_loop, name="PopupThread")  
 cls.\_instance.\_stop\_thread = threading.Event()  
 cls.\_instance.popup\_thread.start()  
 return cls.\_instance  
  
 def add(self, popup):  
 self.waiting.put(popup)  
  
 def render\_popup(self, popup):  
 title, message, icon, \*\_ = popup  
 icon = IconType(int(icon))  
 if icon == IconType.INPUT:  
 callback = popup[3]  
 callback(self.\_get\_input(title, message))  
 else:  
 self.\_show\_text(title, message, icon)  
  
 def \_get\_input(self, title: str, message: str):  
 layout = [[sg.Text(message)],  
 [sg.Input(key='-IN-')],  
 [sg.Button('Submit')]]  
  
 window = sg.Window(title, layout)  
  
 while True:  
 event, values = window.read()  
 if event == sg.WINDOW\_CLOSED:  
 return ''  
 elif event == 'Submit':  
 result = values['-IN-']  
 window.close()  
 return result  
  
 def \_show\_text(self, title: str, message: str, icon: IconType):  
 lines = message.split('\n')  
 lines = [  
 item for line in lines  
 for item in (  
 line.split('. ') if len(line) > 200 else [line]  
 )  
 ]  
 lines = [  
 item for line in lines  
 for item in (  
 line.split(', ') if len(line) > 150 else [line]  
 )  
 ]  
 lines = [line.strip() for line in lines]  
 message = '\n\n'.join(lines)  
 # message = message.replace('\n', '\n\n')  
 markdown\_text = markdown2.markdown(message)  
 html\_text = f"{get\_CSS()}<div class=\"limit-width\">{markdown\_text}</div>"  
  
 layout = [[  
 sg.Column(  
 [[sg.Text('', key='\_HTML\_')]],  
 size=POPUP\_WINDOW\_SIZE,  
 scrollable=(False, True)  
 )  
 ]]  
 window = sg.Window(title, layout, finalize=True, resizable=False)  
 window['\_HTML\_'].update(html\_text.replace('\x00', '\\0')) # remove null characters  
 # window['\_HTML\_'].Widget.setOpenExternalLinks(True) # Allow links to be clicked  
  
 while True:  
 event, values = window.read(timeout=POPUP\_WINDOW\_LOOP\_TIMEOUT\_MS)  
 if event == sg.WIN\_CLOSED:  
 break  
  
 window.close()  
 return -1  
   
 def stop(self):  
 self.\_stop\_thread.set()  
  
 def \_popup\_loop(self):  
 """  
 Private method that runs continuously as the popup thread.  
 Waits for popups to arrive and displays them when available.  
 """  
 while not self.\_stop\_thread.is\_set():  
 try:  
 popup = self.waiting.get(block=False)  
 self.render\_popup(popup)  
 except queue.Empty:  
 time.sleep(0.1)  
  
  
def popup(title: str, message: str, \*, error=False, warning=False, question=False, info=False, cancel=False):  
 """This function creates a visual UI popup, with `title` as the window's title, and `message` in the body.  
 The popup itself isn't too large.  
 You can add `error`, `warning`, `question`, `info`, to set an icon next to the content.  
 You can add `cancel` to give the user a choice between "OK" (returns True) and "Cancel" (returns False).  
  
 This function unifys two separate ideas, under an abstracted interface.  
 One is icnoned, with markdown support, and slower.  
 One is iconless, plaintext, Win API, and faster.  
  
 | Popup Type | Markdowned | Plaintext |  
 |:------------------:|:-----------------------------------------------------------:|:------------------------------------:|  
 | Supports plaintext | Yes | Yes |  
 | Supports Markdown | Yes | No |  
 | Supports HTML/CSS | Yes | No |  
 | Supports Icons | Yes (soon) | No |  
 | Graphical Library | PySimpleGUI | Win32 API |  
 | Graphical Object | `PySimpleGUIQt as sg; sg.Window, sg.Column, sg.Text` | `win32api.MessageBox` |  
 | Return Value | None | bool or None |  
 | Arguments Used | title, message, error, warning, question, info | title, message, cancel |  
 | Immediate | No, uses a Queue on a separate thread | Yes |  
 | Blocking | No | Yes, until the user closes the popup |  
 | How to apply | set either of these to True: error, warning, question, info | Don't apply the other option |  
  
 Note: if you supply markdown content into `message`, and you wish for it to not display as plaintext,  
 set one of the icons, e.g. `info=True`.  
 If multiple icons are set, the icon is chosen by this priority list:  
 error > warning > question > info  
 (`error` overpowers all, `info` gets overpowered by all).  
 If no icon is chosen, the text is displayed as plaintext, not markdown.  
  
 The function returns:  
 - None for all `PySimpleGUI` dialogs, i.e. anything with an icon.  
 - Boolean (True/False) indicating whether the Cancel button wasn't pressed, if `cancel=True`.  
 - None if no keyword arguments were set, after displaying a `MessageBox`.  
 Notice: the function will return immediately for `Markdowned`, it will be blocking for `Plaintext` (even if the return value will be `None` eventually).  
  
 Args:  
 title (str): the title of the window.  
 message (str): the content of the window (plaintext or markdown).  
 error (bool, optional): whether to display a Critical icon. Defaults to False.  
 warning (bool, optional): whether to display a Warning icon. Defaults to False.  
 question (bool, optional): whether to display a Question icon. Defaults to False.  
 info (bool, optional): whether to display an Information icon. Defaults to False.  
 cancel (bool, optional): if no icon is chosen, this determines the type of the MessageBox: whether there'll be a Cancel button. Defaults to False.  
  
 Returns:  
 (bool | None): `False` if the Cancel Button was pressed, `True` if not, `None` if irrelevant. See above.  
 """  
 if not isinstance(message, str):  
 raise TypeError("Popup message must be a string.")  
 if not isinstance(title, str):  
 raise TypeError("Popup title must be a string.")  
 if not (error or warning or question or info):  
 if cancel:  
 return win32api.MessageBox(0, message, title, win32con.MB\_OKCANCEL) != win32con.IDCANCEL  
 else:  
 win32api.MessageBox(0, message, title, win32con.MB\_OK)  
 return None  
   
 icon = IconType.NOTHING  
 if error:  
 icon = IconType.ERROR  
 elif warning:  
 icon = IconType.WARNING  
 elif question:  
 icon = IconType.QUESTION  
 elif info:  
 icon = IconType.INFO  
  
 PopupManager().add((title, message, icon.value))  
  
  
def get\_string(title: str, prompt: str) -> None:  
 """This function prompts the user to give a string input.  
 It's basically like `input` in plain Python,  
 but with a GUI.  
 The window's title will be `title`.  
 \*\*Blocking\*\* until user input is given.  
  
 Args:  
 title (str): the title of the small window.  
 prompt (str): the question to ask the user.  
 """  
 l = [] # Python has lists as reference type, so they can store data in just the way I need  
 def \_save(l ,x): # used as callback, to store the result in the list.  
 l.append(x)  
 PopupManager().add((title, prompt, IconType.INPUT.value, lambda x: \_save(l, x)))  
 while len(l) == 0: # blocking until `l` is given an item with the string.  
 sleep(0.1)  
 return l[0]  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file gives visual dialog actions,")  
 print("that are useful for displaying information to the user")  
 print("as well as getting the user's input.")

# ./Scanner/gui\Hover.py

from globalstuff import \*  
from CacheDecorators import one\_cache  
  
  
class Hover:  
 """Enables hovering cursor and behaviours. Uses singleton structure (because it accesses a system function of changing cursor).  
 Includes two lists: `items`, where each item can change the cursor to `pointer` if hovered (`item.collide\_point(x, y) -> True`);  
 and `behaviors`, where each item is a `HoverBehavior`, and they do more exotic stuff, abstracted by `.show()` and `.hide()`.  
  
 Raises:  
 AttributeError: raised when `.add(item)` receives an `item` that has no method `.collide\_point(int,int)`.  
 TypeError: raised when `.add\_behavior(behavior)` receives a `behavior` that is not of type `HoverBehavior`.  
 """  
 items = {}  
 behaviors = {}  
 current\_screen = ""  
  
 @staticmethod  
 @one\_cache  
 def \_bind():  
 from kivy.core.window import Window  
 from gui.AppState import State  
 Window.bind(mouse\_pos=Hover.update)  
 Window.bind(size=State().resize\_callback)  
 return 0 # to comply with @one\_cache's rule: A @one\_cache function cannot return None!  
  
 @staticmethod  
 def add(instance):  
 Hover.\_bind()  
 if Hover.current\_screen == "":  
 raise KeyError("Hover cannot add without screen")  
 try:  
 instance.collide\_point(0, 0)  
 except AttributeError:  
 raise AttributeError(  
 "The instance passed to `Hover.add` doesn't support `.collide\_point(int,int)`.")  
 Hover.items[Hover.current\_screen].append(instance)  
  
 @staticmethod  
 def add\_behavior(behavior):  
 Hover.\_bind()  
 if Hover.current\_screen == "":  
 raise KeyError("Hover cannot add without screen")  
 if not isinstance(behavior, HoverBehavior):  
 raise TypeError(  
 "The behavior passed to `Hover.add\_behavior` isn't a `HoverBehavior`.")  
 Hover.behaviors[Hover.current\_screen].append(behavior)  
 # A behaviour should support 3 methods: `collide\_point(int,int)`,  
 # `show()`, and `hide()`, and that's enforced by the HoverBehaviour  
 # interface.  
  
 def update(window, pos):  
 if any([item.collide\_point(\*pos)  
 for item in Hover.items[Hover.current\_screen]]):  
 window.set\_system\_cursor("hand")  
 else:  
 window.set\_system\_cursor("arrow")  
  
 for behavior in Hover.behaviors[Hover.current\_screen]:  
 if behavior.collide\_point(\*pos):  
 behavior.show()  
 else:  
 behavior.hide()  
  
 def enter(screen: str):  
 Hover.current\_screen = screen  
 if screen not in Hover.items:  
 Hover.items[screen] = []  
 if screen not in Hover.behaviors:  
 Hover.behaviors[screen] = []  
  
 @staticmethod  
 def start():  
 # Hide everything when the screen loads. Kinda misleading name -- this  
 # function is called last in initalisation -- it marks the start of the  
 # UI.  
 for screen in Hover.behaviors.keys():  
 for behavior in Hover.behaviors[screen]:  
 behavior.hide()  
  
  
class HoverBehavior:  
 """  
 Inherit from this class to create behaviours,  
 and pass the instances to `Hover.add\_behavior(...)`.  
 """  
  
 def show(self):  
 raise NotImplementedError()  
  
 def hide(self):  
 raise NotImplementedError()  
  
 def collide\_point(self, x, y):  
 raise NotImplementedError()  
  
  
class HoverReplace(HoverBehavior):  
 """A `HoverBehavior` that replaces the text shown on a label.  
 When hovered, it displays the string in `text`,  
 otherwise, it displays the initial string.  
 """  
  
 def \_\_init\_\_(self, widget, text, font\_size, font="Arial"):  
 self.widget = widget  
 self.text = text  
 self.save = self.widget.text  
 self.font\_size = font\_size  
 self.save\_font = self.widget.font\_name  
 self.font = font  
 Hover.add\_behavior(self)  
  
 def show(self):  
 self.widget.text = self.text  
 self.widget.font\_name = self.font  
 self.widget.font\_size = self.font\_size \* HOVER\_REPLACE\_FACTOR  
  
 def hide(self):  
 self.widget.text = self.save  
 self.widget.font\_name = self.save\_font  
 self.widget.font\_size = self.font\_size  
  
 def collide\_point(self, x, y):  
 return self.widget.collide\_point(x, y)  
  
  
class HoverReplaceBackground(HoverReplace):  
 """A `HoverBehavior` that replaces the text shown on a label.  
 When hovered, it displays the string in `text` (AND a different background colour),  
 otherwise, it displays the initial string.  
 """  
  
 def \_\_init\_\_(self, widget, text, font\_size, new\_bg, font="Arial"):  
 super().\_\_init\_\_(widget, text, font\_size, font)  
 self.save\_bg = self.widget.background\_color  
 self.bg = new\_bg  
  
 def show(self):  
 super().show()  
 self.widget.background\_color = self.bg  
  
 def hide(self):  
 super().hide()  
 self.widget.background\_color = self.save\_bg  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module creates the Hovering mechanism for the kivy gui.")

# ./Scanner/gui\Information.py

from import\_handler import ImportDefence  
with ImportDefence():  
 import re  
 from threading import enumerate as enumerate\_threads  
  
from gui.dialogs import popup  
import db  
from ipconfig import ipconfig  
from gui.ScanClasses import DummyScan  
from gui.AppState import State  
  
  
def display\_information(\*\_):  
 highlighted = State().highlighted\_scan  
 if highlighted is DummyScan() or highlighted is None:  
 popup(f"General Information", general\_information(), info=True)  
 else:  
 name = highlighted.name  
 information = information\_about(name)  
 if information == '': return  
 popup(f"Information about {name}", information, info=True)  
  
  
def information\_about(name: str) -> str:  
 # Get the entry about the scan and destructure it  
 entry = db.get\_information\_about\_scan(name)  
 if entry is None:  
 popup(f"Information about {name}", "It appears the database has no information regarding this scan.\  
 \n\nPlease try updating your installation!", warning=True)  
 return ""  
 name0, description, time, reward, certainty, safety, mode, repeats = entry  
 if name != name0:  
 raise ValueError(  
 f"Weird name problem: key is `{name}`, database says `{name0}`."  
 )  
  
 # Generate phrases  
 perrepeat = " per repeat" if repeats else ""  
 hasrepeats = "Repeatable" if repeats else "Not repeatable"  
 certainty\_prompt = "That's pretty uncertain" if certainty <= 50 else "That's mildly certain" if certainty <= 80 else "That's pretty certain" if certainty <= 100 else "???"  
 safety\_prompt = "That's really unsafe" if safety <= 30 else "That's pretty unsafe" if safety <= 70 else "That's quite safe" if safety < 100 else "That's perfectly safe -- completely undetectable"  
  
 # If the description includes a packet model, escape it into a code block.  
 if not description.endswith(('.', '. ', '>')):  
 description += '.'  
   
 def \_html\_from\_osi\_model(message: str) -> str:  
 import re  
  
 pattern = r"<([A-Z][^>]\*)>"  
 replacement = r"<tr><td>\1</td></tr>"  
  
 new\_string = re.sub(pattern, replacement, message)  
 new\_string = re.sub(r"<tr><td>([^<>]\*)</td></tr>\s\*<tr><td>", r"<tr><td>\1</td></tr>", new\_string)  
  
 new\_string = re.sub(r"<tr>", "<table><tr>", new\_string, count=1)  
 new\_string = new\_string.rsplit("</tr>", 1)[0] + "</tr></table>"  
   
 return new\_string  
 description = \_html\_from\_osi\_model(description)  
 # description = re.sub(  
 # "<[^<>]+>",  
 # "<br>```\\g<0>```",  
 # description,  
 # flags=re.DOTALL  
 # )  
  
 # Return everything in markdown format.  
 return '\n'.join([  
 f"# {name}",  
 f"## Description",  
 f"{description}",  
 f"## Time estimate",  
 f"{time}s{perrepeat}",  
 f"## Risk and Reward",  
 f"### What you get",  
 f"{reward}",  
 f"### How reliable is that?",  
 f"{certainty}% certain that the data are correct.",  
 f"{certainty\_prompt}.",  
 f"### Safety",  
 f"Running this is {safety}% safe.",  
 f"{safety\_prompt}.",  
 f"## Others",  
 f"- {hasrepeats}",  
 f"- Mode: {mode}"  
 ])  
  
  
def general\_information() -> str:  
 computers\_in\_network = len(ipconfig()["All Possible Addresses"])  
 interface = ipconfig()["Interface"]  
 mask = ipconfig()["Subnet Mask"]  
  
 def \_get\_readable\_threads():  
 def find\_between(s):  
 # "some(str)ing"  
 # "some", "str)ing"  
 # "str)ing"  
 # "str", "ing"  
 # "str"  
 return (s.split('('))[1].split(')')[0]  
  
 threads = enumerate\_threads()  
 threads = [thread.name for thread in threads]  
 threads = [name if '(' not in name else find\_between(name) for name in threads]  
 uniques = set(threads)  
 counts = [threads.count(name) for name in uniques]  
 threads = [  
 f"{count} × {name}" if count > 1 else name for name,  
 count in zip(uniques, counts)  
 ]  
 threads = [f'\* `{name}`' for name in threads]  
 return threads  
  
 from NetworkStorage import here, router  
 return '\n\n'.join([  
 f"# General Information",  
 f"## This device",  
 f"IPv4 Address: `{here.ip}`",  
 f"Network Interface: `{interface}`",  
 f"## Router (Gateway)",  
 f"IPv4 Address: `{router.ip}`",  
 f"## Local Network",  
 f"Subnet mask: `{mask}`",  
 f"Possible IPv4 addresses: `{computers\_in\_network} addresses`",  
 f"## Others",  
 f"Premission to scan: `{State().permission}`",  
 f"## Current Threads",  
 \*\_get\_readable\_threads()  
 ])  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file is responsible for any methods called by the Information Button (ℹ) in Scan Screen.")

# ./Scanner/gui\KivyExtensions.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from kivy.uix.button import Button  
 from kivy.utils import escape\_markup  
 from kivy.uix.gridlayout import GridLayout  
 from kivy.uix.widget import Widget  
  
from globalstuff import \*  
from gui.Diagrams import KivyDiagram  
from gui.Hover import Hover, HoverReplace  
  
  
# --- Kivy Extensions ---  
class ButtonColumn(GridLayout):  
 """Organises buttons in a column  
  
 Args:  
 GridLayout (kivy): the superclass.  
 """  
  
 def \_\_init\_\_(self, width: int):  
 super().\_\_init\_\_(cols=1, width=width, size\_hint=(  
 None, 1), spacing=[-3], padding=[-1, -3, -1, -3])  
 self.buttons = [] # list of tuples `(button, callback)`  
 self.background\_color = button\_column\_background  
 self.font\_size = BUTTON\_COLUMN\_FONT\_SIZE  
  
 def add(self, text: str, callback=None):  
 btn = Button(  
 text=text,  
 font\_size=self.font\_size,  
 background\_color=self.background\_color,  
 font\_name="Roboto"  
 )  
 if callback is not None:  
 btn.bind(on\_press=callback)  
 super().add\_widget(btn)  
 Hover.add(btn)  
 self.buttons.append((btn, callback))  
 return btn  
  
 def add\_raw(self, button):  
 super().add\_widget(button)  
 self.buttons.append((button, None))  
  
  
class MyPaintWidget(Widget):  
 """Responsible for the middle diagram (object #9).  
 It is the caller's responsibility to set this as the `KivyDiagram()`'s widget.  
  
 Args:  
 Widget (kivy widget): the superclass.  
 """  
  
 def init(self):  
 pass  
  
 def on\_touch\_down(self, touch):  
 KivyDiagram().update()  
  
  
class GreenButton(Button):  
 """A button that has green background, and also adds itself to `Hover`."""  
  
 def \_\_init\_\_(self, text, \*\*kwargs):  
 super().\_\_init\_\_(  
 text=f'[color={GREEN}]{escape\_markup(text)}[/color]',  
 markup=True,  
 \*\*kwargs  
 )  
 Hover.add(self)  
  
  
class OperationButton(Button):  
 """A button that has grey background, adds itself to `Hover`, defines a `HoverReplace` on itself, and uses font `Symbols`."""  
  
 def \_\_init\_\_(self, text, long\_text, onclick, \*\*kwargs):  
 super().\_\_init\_\_(  
 text=text,  
 background\_color=OPERATION\_BUTTON\_BACKGROUND,  
 font\_name="Symbols",  
 \*\*kwargs  
 )  
 Hover.add(self)  
 HoverReplace(self, long\_text, OPERATION\_BUTTON\_FONT\_SIZE)  
 self.bind(on\_press=onclick)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This modules provides some classes that inherit from kivy's classes,")  
 print("and adds behaviour or simplfies constructors.")

# ./Scanner/gui\KivyFonts.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from kivy.core.text import LabelBase  
  
  
def add\_fonts():  
 """Loads a font (`kivy`'s), from the folder `fonts/`."""  
 def \_add\_font(path, name, fallback='fonts/Segoe UI Symbol.ttf'):  
 path = f'fonts/{path}'  
 try:  
 LabelBase.register(name=name, fn\_regular=path)  
 except OSError:  
 LabelBase.register(name=name, fn\_regular=fallback)  
  
 \_add\_font('BainsleyBold.ttf', 'Symbols')  
 \_add\_font('Segoe UI Symbol.ttf', 'Symbols+')  
 \_add\_font('Symbola.ttf', 'Symbols++')  
 \_add\_font('Consolas.ttf', 'Monospace')  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file is responsible for loading the fonts for the kivy gui.")

# ./Scanner/gui\popup\_style.css

[binary content]

# ./Scanner/gui\ScanClasses.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from kivy.uix.button import Button  
 from kivy.graphics import Color, Rectangle  
  
from gui.dialogs import popup  
from globalstuff import \*  
from register import Register  
from gui.Hover import Hover  
  
  
class Scan:  
 font\_size = BUTTON\_COLUMN\_FONT\_SIZE  
 background\_color = button\_column\_background  
  
 def \_\_init\_\_(self, name, action, parent):  
 self.name = name  
 self.action = action  
 self.x = 0  
 self.is\_running = False  
  
 self.button = Button(  
 text=name,  
 font\_size=Scan.font\_size,  
 background\_color=Scan.background\_color,  
 font\_name="Roboto"  
 )  
 self.button.bind(on\_press=lambda x: self.select(x))  
 parent.add\_raw(self.button)  
 Hover.add(self.button)  
  
 def select(self, x):  
 from gui.AppState import State  
 if State().highlighted\_scan == self:  
 State().scan(DummyScan())  
 return  
 State().scan(self)  
 self.paint\_highligh()  
 self.x = x  
   
 def paint\_highligh(self):  
 with self.button.canvas.after:  
 self.highlight = Color(\*SCAN\_HIGHLIGHT)  
 self.highlight\_rect = Rectangle(  
 pos=(self.button.x, self.button.y),  
 size=(self.button.width, self.button.height)  
 )  
  
 def deselect(self):  
 self.button.canvas.after.clear()  
  
 def act(self):  
 self.is\_running = True  
 self.button.text += '...'  
 try:  
 self.action()  
 except TypeError:  
 self.action(self.x)  
  
 def finished(self):  
 self.is\_running = False  
 if self.button.text.endswith('...') and not Register().is\_infinite\_scan(self.button.text):  
 self.button.text = self.button.text[:-3]  
  
  
class DummyScan(Scan):  
 \_instance = None  
  
 def \_\_new\_\_(cls):  
 if cls.\_instance is None:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 return cls.\_instance  
  
 def \_\_init\_\_(self):  
 self.name = "Dummy"  
 self.is\_running = False  
  
 def select(self, x):  
 pass  
  
 def deselect(self):  
 pass  
  
 def act(self):  
 # win32api.MessageBox(0, "You must first select a scan to run.", "Running without a scan", 0x00000000)  
 popup("Running without a scan", "You must first select a scan to run.")  
 # raise NotImplementedError("A DummyScan cannot be `.act`ed upon.")  
  
 def finished(self):  
 pass  
  
  
class Analysis(Scan):  
 """This is identical to a scan, except for cosmetic changes.  
 It's intended to be used under Know Screen.  
  
 Another difference, present in `Activation.py`, is that Analyses do not require confirmation popup;  
 just permission, which is half what other Scans need (permission + confirmation).  
  
 Changes:  
 - background colour is more blue.  
 - highlight colour is ANALYSIS\_HIGHLIGHT.  
 """  
 def \_\_init\_\_(self, name, action, parent):  
 super().\_\_init\_\_(name, action, parent)  
 self.button.background\_color = [0.2, 0.4, 1, 1]  
   
 def paint\_highligh(self):  
 with self.button.canvas.after:  
 self.highlight = Color(\*ANALYSIS\_HIGHLIGHT)  
 self.highlight\_rect = Rectangle(  
 pos=(self.button.x, self.button.y),  
 size=(self.button.width, self.button.height)  
 )

# ./Scanner/gui\StartApp.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from kivy.config import Config  
  
import sys  
from datetime import datetime  
import traceback  
from gui.App import MyApp  
from gui.Diagrams import Diagrams, TKDiagram, PlotDiagram  
import threading  
  
  
def start\_kivy():  
 """Starts the `kivy` app, and handles the `tkinter` diagram's closing."""  
 global is\_kivy\_running  
 try:  
 is\_kivy\_running = True  
 # disable multi-touch emulation  
 Config.set('input', 'mouse', 'mouse,multitouch\_on\_demand')  
 MyApp().run()  
 except Exception as e:  
 traceback.print\_exc()  
 print(  
 f"{type(e).\_\_name\_\_} at {\_\_file\_\_}:{e.\_\_traceback\_\_.tb\_lineno}: {e}\n{datetime.now()}",  
 file=open('error log.txt', 'a', encoding='utf-8')  
 )  
 raise  
 finally:  
 is\_kivy\_running = False  
 try:  
 TKDiagram().show()  
 except RuntimeError:  
 pass  
 root = TKDiagram().root  
 root.after(1, root.destroy)  
 from globalstuff import terminator  
 terminator.set()  
 from gui.dialogs import PopupManager  
 PopupManager().stop()  
 from PacketSniffer import PacketSniffer  
 PacketSniffer().stop()  
 from Sniffer import Sniffer  
 Sniffer.stopall()  
 print('\n'.join([str(thread) for thread in threading.enumerate()]))  
  
 # sys.exit()  
  
  
start\_kivy.\_\_name\_\_ = 'GUIThread'  
  
  
def start\_tk():  
 """Starts the `tkinter` diagram in the background.  
 This has to be on the main thread (because `tkinter` said so)."""  
 # PlotDiagram().show()  
 Diagrams().add(TKDiagram())  
 TKDiagram().root.mainloop() # main thread waits here until the user leaves the application.  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file is responsible for starting the kivy and tkinter gui.")  
 print("The callers are responsible for thread handling: start\_tk on main, start\_kivy on separate thread.")

# ./Scanner/gui\\_\_init\_\_.py

# ./Scanner/gui\Screens\KnowScreen.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from kivy.uix.scrollview import ScrollView  
 from kivy.uix.label import Label  
 from kivy.uix.button import Button  
 from kivy.uix.screenmanager import Screen  
 from kivy.uix.boxlayout import BoxLayout  
 from kivy.uix.relativelayout import RelativeLayout  
  
from threading import Thread  
from gui.Hover import Hover  
from gui.Screens.Pages import Pages  
from gui.KivyExtensions import ButtonColumn, OperationButton  
from gui.dialogs import get\_string, popup  
from gui.Configuration import display\_configuration  
from gui.Information import display\_information  
from gui.Activation import activate  
from gui.ScanClasses import Analysis  
from register import Register  
from globalstuff import \*  
import db  
  
  
def update\_know\_screen(text):  
 return print("update\_know\_screen:", text)  
  
  
class KnowScreen(Screen):  
 """Builds an interface that looks like this:  
  
 ```md  
 The Window (Unicode Box Art):  
 ┌────────────────────────────────────────────┐  
 │ [#1 Title] │  
 │ #4 Save. │  
 │ #5 Scan. │  
 │ #6 Know. │  
 │ #3 Device Profile │  
 │ #2 Data │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ │  
 │ │  
 └────────────────────────────────────────────┘  
 ```  
  
 Args:  
 Screen (kivy): the base class for a screen.  
 """  
  
 def \_\_init\_\_(self, \*\*kw):  
 name = 'Know'  
 super().\_\_init\_\_(name=name, \*\*kw)  
 Hover.enter(name)  
  
 everything = RelativeLayout()  
 title = Label(  
 text=f"[color={GREEN}]Knowledge about Network[/color]",  
 size=(0, TITLE\_HEIGHT),  
 size\_hint=(1, None),  
 font\_size=TITLE\_FONT\_SIZE,  
 underline=True,  
 pos\_hint={'center\_x': .42, 'top': 1},  
 markup=True  
 )  
 everything.add\_widget(title)  
  
 body = BoxLayout(orientation='horizontal')  
 body.add\_widget(KnowScreenInfoLabel(pos\_hint={'top': .86}))  
 body.add\_widget(KnowScreenRightColumn())  
 everything.add\_widget(body)  
 everything.add\_widget(Pages())  
  
 self.add\_widget(everything)  
  
  
class KnowScreenRightColumn(ButtonColumn):  
 """Builds the right column used in the screen 'Know'.  
  
 ```md  
 Know Screen  
 . . . ─╥───────────────────────────┐  
 ║ [#2 Conf] [#3 Info] │  
 ╟───────────────────────────┤  
 ║ [#7 KnowA] │  
 ║ │  
 ║ [#8 KnowB] │  
 ║ │  
 ║ [#9 KnowC] │  
 ║ │  
 ║ [#10 KnowD] │  
 ║ │  
 ║ . . . │  
 ║ │  
 . . . ─╨───────────────────────────┘  
 ```  
  
 Args:  
 ButtonColumn (GridLayout): this inherits from ButtonColumn.  
 """  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_(width=RIGHT\_COLUMN\_WIDTH \* 0.8)  
  
 # Objects #2, #3 -- two operations on each analysis  
 operations = BoxLayout(orientation='horizontal', spacing=-3, size\_hint=(0.3, None))  
 operations.add\_widget(OperationButton('⏻', "Analyse", activate))  
 operations.add\_widget(OperationButton('⚙', "Config", display\_configuration))  
 operations.add\_widget(OperationButton('ℹ', "Info", display\_information))  
  
 self.add\_widget(operations)  
  
 # Objects #7 - #10  
 for name in db.get\_analyses():  
 Analysis(name, Register()[name], self)  
  
  
class KnowScreenInfoLabel(ScrollView):  
 """Holds the requested data in string format, displayed to the user.  
 Has a scrolling mechanic.  
  
 Args:  
 Label (kivy): the base class from kivy.  
 """  
  
 def \_\_init\_\_(self, \*\*kwargs):  
 super().\_\_init\_\_(width=1200, \*\*kwargs)  
 self.label = Label(  
 text='Loading data...',  
 color=(1, 1, 1, 1),  
 font\_size=20,  
 font\_name="Monospace",  
 size\_hint\_y=None,  
 text\_size=(self.width, None),  
 halign='center'  
 )  
 self.label.bind(texture\_size=self.label.setter('size'))  
 self.add\_widget(self.label)  
 global update\_know\_screen  
 update\_know\_screen = self.data  
  
 def data(self, text):  
 if isinstance(text, str):  
 self.label.text = text  
 else:  
 try:  
 items = text.tablestring()  
 items = [f"There are {len(items) - 4} Network Entities known."] + items  
 # print('\n'.join(items))  
 except AttributeError:  
 items = [str(x) for x in text]  
 self.label.text = '\n'.join(items)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file provides the Know Screen for the gui.\n")  
 print("""  
 ┌────────────────────────────────────────────┐  
 │ [#1 Title] │  
 │ #4 Save. │  
 │ #5 Scan. │  
 │ #6 Know. │  
 │ #3 Device Profile │  
 │ #2 Data │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ │  
 │ │  
 └────────────────────────────────────────────┘""")

# ./Scanner/gui\Screens\Pages.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from kivy.uix.boxlayout import BoxLayout  
  
from globalstuff import \*  
from gui.AppState import State  
from gui.KivyExtensions import GreenButton  
  
  
class Pages(BoxLayout):  
 def \_\_init\_\_(self, \*\*kwargs):  
 super().\_\_init\_\_(orientation='vertical', size\_hint=(.15, .15),  
 pos\_hint={'x': 0, 'top': 1}, \*\*kwargs)  
  
 labels = ['Save.', 'Scan.', 'Know.']  
 actions = [  
 lambda \_: State().screen("Save"),  
 lambda \_: State().screen("Scan"),  
 lambda \_: State().screen("Know")  
 ]  
 buttons = [  
 GreenButton(  
 text=label,  
 font\_size=20,  
 background\_color=PAGES\_BACKGROUND,  
 font\_name="Arial"  
 )  
 for label in labels]  
 for button, action in zip(buttons, actions):  
 button.bind(on\_press=action)  
 self.add\_widget(button)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module provides a GUI widget (BoxLayout),")  
 print("That should be used on each screen to transition between Save, Scan, and Know.")

# ./Scanner/gui\Screens\SaveScreen.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from kivy.uix.screenmanager import Screen  
 from kivy.uix.relativelayout import RelativeLayout  
 from kivy.uix.label import Label  
  
from globalstuff import \*  
from threading import Thread  
from files import importer, exporter  
from gui.Hover import Hover, HoverReplaceBackground  
from gui.dialogs import popup  
from gui.KivyExtensions import GreenButton  
from gui.Screens.Pages import Pages  
from gui.AppState import State  
  
  
class SaveScreenExportButton(GreenButton):  
 def \_\_init\_\_(self, \*\*kwargs):  
 super().\_\_init\_\_('↥', size\_hint=(.15, .15), pos\_hint={'x': 0.2, 'top': 0.75},  
 font\_name="Symbols+", background\_color=(0, 0, 0, 1), \*\*kwargs)  
 HoverReplaceBackground(  
 self,  
 'Export',  
 SAVE\_BUTTONS\_FONT\_SIZE,  
 SAVE\_BUTTONS\_HOVER\_BACKGROUND  
 )  
 self.bind(on\_press=self.export)  
  
 def export(self, \_):  
 def \_exporting():  
 try:  
 path = exporter()  
 print("Showing popup")  
 popup("Exported", f"Saved the scan to {path}")  
 except FileExistsError:  
 popup(  
 "File Exists Error",  
 "A file already exists in that path.",  
 error=True  
 )  
 except FileNotFoundError:  
 popup(  
 "File Error",  
 "A file cannot be written in that location.",  
 error=True  
 )  
 Thread(target=\_exporting).start()  
  
  
class SaveScreenImportButton(GreenButton):  
 def \_\_init\_\_(self, \*\*kwargs):  
 super().\_\_init\_\_('⭳', size\_hint=(.15, .15), pos\_hint={'x': 0.65, 'top': 0.75},  
 font\_name="Symbols++", background\_color=(0, 0, 0, 1), \*\*kwargs)  
 HoverReplaceBackground(  
 self,  
 'Import',  
 SAVE\_BUTTONS\_FONT\_SIZE,  
 SAVE\_BUTTONS\_HOVER\_BACKGROUND  
 )  
 self.bind(on\_press=self.do\_import)  
  
 def do\_import(self, \_):  
 def \_importing():  
 content = ''  
 try:  
 content = importer()  
 except FileNotFoundError:  
 popup("File Not Found", "The file you selected was not found.")  
 except ValueError as e:  
 popup("Error in parsing Scan File", e.args[0])  
 try:  
 from gui.Screens.ViewScreen import update\_view\_screen  
 if content != '':  
 update\_view\_screen(content)  
 else:  
 update\_view\_screen("Couldn't decrypt file.")  
 except Exception as e:  
 print(e)  
  
 Thread(target=\_importing).start()  
 State().screen("View")  
  
  
class SaveScreen(Screen):  
 """Builds an interface that looks like this:  
  
 ```md  
 The Window (Unicode Box Art):  
 ┌────────────────────────────────────────────┐  
 │ [#1 Title] │  
 │ #4 Save. │  
 │ #5 Scan. │  
 │ #6 Know. │  
 │ │  
 │ #2 Export #3 Import │  
 │ │  
 │ │  
 │ │  
 │ │  
 │ │  
 │ │  
 └────────────────────────────────────────────┘  
 ```  
  
 Args:  
 Screen (kivy): the base class for a screen.  
 """  
  
 def \_\_init\_\_(self, \*\*kw):  
 name = 'Save'  
 super().\_\_init\_\_(name=name, \*\*kw)  
 Hover.enter(name)  
  
 everything = RelativeLayout()  
 title = Label(  
 text=f"[color={GREEN}]Saving and Opening Scans[/color]",  
 size=(0, TITLE\_HEIGHT),  
 size\_hint=(1, None),  
 font\_size=TITLE\_FONT\_SIZE,  
 underline=True,  
 pos\_hint={'center\_x': .5, 'top': 1},  
 markup=True  
 )  
 everything.add\_widget(title)  
 everything.add\_widget(SaveScreenExportButton())  
 everything.add\_widget(SaveScreenImportButton())  
 # \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* add the recents  
 # everything.add\_widget(SaveScreenRecents())  
 everything.add\_widget(Pages())  
  
 self.add\_widget(everything)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file provides the Save Screen for the gui.\n")  
 print("""  
 ┌────────────────────────────────────────────┐  
 │ [#1 Title] │  
 │ #4 Save. │  
 │ #5 Scan. │  
 │ #6 Know. │  
 │ │  
 │ #2 Export #3 Import │  
 │ │  
 │ │  
 │ │  
 │ │  
 │ │  
 │ │  
 └────────────────────────────────────────────┘""")

# ./Scanner/gui\Screens\ScanScreen.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from kivy.uix.relativelayout import RelativeLayout  
 from kivy.uix.label import Label  
 from kivy.uix.screenmanager import Screen  
 from kivy.uix.boxlayout import BoxLayout  
  
from globalstuff import \*  
import db  
from gui.Screens.Pages import Pages  
from register import Register  
from gui.Activation import activate  
from gui.Diagrams import \*  
from gui.KivyExtensions import GreenButton, MyPaintWidget, OperationButton, ButtonColumn  
from gui.Configuration import display\_configuration  
from gui.Hover import Hover  
from gui.ScanClasses import Scan  
from gui.Information import display\_information  
  
  
class ScanScreenMiddleDiagram(RelativeLayout):  
 """Builds the middle diagram used in the screen 'Scan'.  
  
 ```md  
 Scan Screen  
 ┌────────────────────────────────────────────╥─ . . .  
 │ [#1 Title] ║  
 │ #4 Save. ║  
 │ #5 Scan. ║  
 │ #6 Know. ║  
 │ #9 D ║  
 │ I ║  
 │ A ║  
 │ G ║  
 │ R ║  
 │ A ║  
 │ M ║  
 │ [#15 Play] [#16 Fullscreen] ║  
 └────────────────────────────────────────────╨─ . . .  
 ```  
  
 Args:  
 RelativeLayout (kivy): the diagram is a type of a Relative Layout, since widgets are placed sporadically.  
 """  
  
 def \_\_init\_\_(self, \*\*kw):  
 super().\_\_init\_\_(\*\*kw)  
  
 # Object #1  
 title = Label(  
 text=f"[color={GREEN}]Local Network Scanner[/color]",  
 size=(0, TITLE\_HEIGHT),  
 size\_hint=(1, None),  
 font\_size=TITLE\_FONT\_SIZE,  
 underline=True,  
 pos\_hint={'center\_x': .5, 'top': 1},  
 markup=True  
 )  
  
 # Object #15  
 play\_button = GreenButton(text='▶',  
 font\_size=UNDER\_DIAGRAM\_FONT\_SIZE,  
 background\_color=PAGES\_BACKGROUND,  
 size\_hint=(.1, .1),  
 pos\_hint={'x': 0, 'y': 0},  
 font\_name="Symbols")  
 play\_button.bind(on\_press=activate)  
  
 # Object #16 # Previous icon: 🔍  
 open\_diagram = GreenButton(text='⛶',  
 font\_size=UNDER\_DIAGRAM\_FONT\_SIZE,  
 background\_color=PAGES\_BACKGROUND,  
 size\_hint=(.1, .1),  
 pos\_hint={'right': 1, 'y': 0},  
 font\_name="Symbols")  
 open\_diagram.bind(on\_press=lambda \_: TKDiagram().show())  
  
 # Object #9  
 paint = MyPaintWidget(  
 size\_hint=(1, 1),  
 pos\_hint={'center\_x': .5, 'center\_y': .5}  
 )  
 KivyDiagram().set\_widget(paint)  
 Diagrams().add(KivyDiagram())  
 paint.bind(pos=KivyDiagram().update, size=KivyDiagram().update)  
  
 # Unite all widgets of the middle diagram.  
 self.add\_widget(paint)  
 self.add\_widget(open\_diagram)  
 self.add\_widget(play\_button)  
 self.add\_widget(title)  
  
  
class ScanScreenRightColumn(ButtonColumn):  
 """Builds the right column used in the screen 'Scan'.  
  
 ```md  
 Scan Screen  
 . . . ─╥───────────────────────────┐  
 ║ [#2 Conf] [#3 Info] │  
 ╟───────────────────────────┤  
 ║ [#7 ScanA] │  
 ║ │  
 ║ [#8 ScanB] │  
 ║ │  
 ║ [#9 ScanC] │  
 ║ │  
 ║ [#10 ScanD] │  
 ║ │  
 ║ . . . │  
 ║ │  
 . . . ─╨───────────────────────────┘  
 ```  
  
 Args:  
 ButtonColumn (GridLayout): this inherits from ButtonColumn.  
 """  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_(width=RIGHT\_COLUMN\_WIDTH)  
  
 # Objects #2, #3 -- two operations on each scan  
 operations = BoxLayout(orientation='horizontal', spacing=-3)  
 operations.add\_widget(  
 OperationButton(  
 '⚙',  
 "Configure",  
 lambda \_: display\_configuration()  
 )  
 )  
 operations.add\_widget(  
 OperationButton(  
 'ℹ',  
 "Information",  
 lambda \_: display\_information()  
 )  
 ) # Consider a '?' instead  
  
 self.add\_widget(operations)  
  
 # Objects #7 - #10  
 for name in db.get\_scans():  
 Scan(name, Register()[name], self)  
 # print(name)  
 # Scan('ICMP Sweep', lambda: print("ICMP!!!"), self)  
 # Scan('ARP Sweep', lambda: print("ARP!!!"), self)  
 # Scan('Live ICMP', lambda: print("ICMP..."), self)  
 # Scan('Live ARP', lambda: print("ARP..."), self)  
 # Scan('OS-ID', lambda: print("It's fun to stay in the O-S-I-D"), self)  
 # Scan('TCP Ports', lambda: print("TCP! TCP! TCP!"), self)  
 # Scan('UDP Ports', lambda: print("Uridine DiPhosphate (UDP) -- glycogen synthesis polymer"), self)  
 # Scan('woo!', temp\_increase\_graph\_degree, self)  
  
  
class ScanScreen(Screen):  
 """Builds an interface that looks like this:  
  
 ```md  
 The Window (Unicode Box Art):  
 ┌────────────────────────────────────────────╥───────────────────────────┐  
 │ [#1 Title] ║ [#2 Conf] [#3 Info] │  
 │ #4 Save. ╟───────────────────────────┤  
 │ #5 Scan. ║ [#7 ScanA] │  
 │ #6 Know. ║ │  
 │ #9 D ║ [#8 ScanB] │  
 │ I ║ │  
 │ A ║ [#9 ScanC] │  
 │ G ║ │  
 │ R ║ [#10 ScanD] │  
 │ A ║ │  
 │ M ║ . . . │  
 │ [#15 Play] [#16 Fullscreen] ║ │  
 └────────────────────────────────────────────╨───────────────────────────┘  
 ```  
  
 Args:  
 Screen (kivy): the base class for a screen.  
 """  
  
 def \_\_init\_\_(self, \*\*kw):  
 name = 'Scan'  
 super().\_\_init\_\_(name=name, \*\*kw)  
 Hover.enter(name)  
  
 everything = RelativeLayout()  
  
 body = BoxLayout(orientation='horizontal')  
 body.add\_widget(ScanScreenMiddleDiagram())  
 body.add\_widget(ScanScreenRightColumn())  
  
 everything.add\_widget(body)  
 # Objects #4, #5, #6 -- Page frippery (top left corner)  
 everything.add\_widget(Pages())  
  
 self.add\_widget(everything)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file provides the Scan Screen for the gui.\n")  
 print("""  
 ┌────────────────────────────────────────────╥───────────────────────────┐  
 │ [#1 Title] ║ [#2 Conf] [#3 Info] │  
 │ #4 Save. ╟───────────────────────────┤  
 │ #5 Scan. ║ [#7 ScanA] │  
 │ #6 Know. ║ │  
 │ #9 D ║ [#8 ScanB] │  
 │ I ║ │  
 │ A ║ [#9 ScanC] │  
 │ G ║ │  
 │ R ║ [#10 ScanD] │  
 │ A ║ │  
 │ M ║ . . . │  
 │ [#15 Play] [#16 Fullscreen] ║ │  
 └────────────────────────────────────────────╨───────────────────────────┘""")

# ./Scanner/gui\Screens\StartScreen.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from kivy.uix.relativelayout import RelativeLayout  
 from kivy.uix.label import Label  
 from kivy.uix.screenmanager import Screen  
 from gui.Screens.Pages import Pages  
 from kivy.uix.image import Image  
 from globalstuff import \*  
 from gui.Hover import Hover  
  
class StartScreen(Screen):  
 """Builds an interface that looks like this:  
  
 ```md  
 The Window (Unicode Box Art):  
 ┌────────────────────────────────────────────┐  
 │ [#1 Title] │  
 │ #4 Save. │  
 │ #5 Scan. #2 instructions │  
 │ #6 Know. │  
 │ │  
 │ │  
 │ #3 Content │  
 │ │  
 │ │  
 └────────────────────────────────────────────┘  
 ```  
  
 Args:  
 Screen (kivy): the base class for a screen.  
 """  
  
 def \_\_init\_\_(self, \*\*kw):  
 name = 'Start'  
 super().\_\_init\_\_(name=name, \*\*kw)  
 Hover.enter(name)  
  
 everything = RelativeLayout()  
  
 from kivy.core.window import Window  
 background = Image(  
 source='background.png',  
 pos\_hint={'center\_x': .5, 'center\_y': .5},  
 size\_hint=(None, None),  
 size=Window.size,  
 allow\_stretch=True,  
 keep\_ratio=False  
 )  
 everything.add\_widget(background)  
  
 logo = Image(source='favicon.png', pos\_hint={'center\_x': .695, 'center\_y': .906})  
 everything.add\_widget(logo)  
  
 title = Label(  
 text=f"[color={GREEN}]Network Scannıng[/color]", # `logo` replaces the i dot.  
 size=(0, 3 \* TITLE\_HEIGHT),  
 size\_hint=(1, None),  
 font\_size=3 \* TITLE\_FONT\_SIZE,  
 underline=True,  
 pos\_hint={'center\_x': .5, 'top': 1},  
 markup=True  
 )  
 everything.add\_widget(title)  
 instructions = Label(  
 text=f"<--- Click on any screen to enter it.",  
 font\_size=20,  
 size=(0, TITLE\_HEIGHT),  
 size\_hint=(1, None),  
 pos\_hint={'center\_x': .22, 'top': 1},  
 markup=True  
 )  
 everything.add\_widget(instructions)  
 main = Label(  
 text=f"Welcome to the Network Scanner!\nEnjoy your stay.",  
 font\_size=30,  
 pos\_hint={'center\_x': .5, 'center\_y': 0.6},  
 markup=True,  
 halign="center", valign="middle"  
 )  
 everything.add\_widget(main)  
 more = Label(  
 text=f"Use `Scan` to execute scans on your local network.\nUse `Save` to save the results of those scans.\nUse `Know` to analyse your knowledge.\n\nMake sure you have permission to execute actions on the network!",  
 font\_size=20,  
 pos\_hint={'center\_x': .5, 'center\_y': 0.4},  
 markup=True,  
 halign="center", valign="middle"  
 )  
 everything.add\_widget(more)  
  
 # Objects #4, #5, #6 -- Page frippery (top left corner)  
 everything.add\_widget(Pages())  
  
 self.add\_widget(everything)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file provides the Start Screen for the gui.\n")  
 print("""  
 ┌────────────────────────────────────────────┐  
 │ [#1 Title] │  
 │ #4 Save. │  
 │ #5 Scan. #2 instructions │  
 │ #6 Know. │  
 │ │  
 │ │  
 │ #3 Content │  
 │ │  
 │ │  
 └────────────────────────────────────────────┘""")

# ./Scanner/gui\Screens\ViewScreen.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from kivy.uix.scrollview import ScrollView  
 from kivy.uix.label import Label  
 from kivy.uix.screenmanager import Screen  
 from kivy.uix.boxlayout import BoxLayout  
  
from gui.Hover import Hover  
from globalstuff import \*  
from gui.Screens.Pages import Pages  
  
  
def update\_view\_screen(text):  
 return print("update\_view\_screen:", text)  
  
  
class ViewScreenInfo(ScrollView):  
 """Holds the requested data in string format, displayed to the user.  
 Has a scrolling mechanic.  
  
 Args:  
 Label (kivy): the base class from kivy.  
 """  
  
 def \_\_init\_\_(self, \*\*kwargs):  
 super().\_\_init\_\_(width=1100, \*\*kwargs)  
 self.label = Label(  
 text='Loading...',  
 color=(1, 1, 1, 1),  
 font\_size=20,  
 font\_name="Monospace",  
 size\_hint\_y=None,  
 text\_size=(self.width, None),  
 halign='center'  
 )  
 self.label.bind(texture\_size=self.label.setter('size'))  
 self.add\_widget(self.label)  
 global update\_view\_screen  
 update\_view\_screen = self.data  
  
 def data(self, text):  
 if isinstance(text, str):  
 self.label.text = text  
 else:  
 raise TypeError()  
  
  
class ViewScreen(Screen):  
 """Builds an interface that looks like this:  
  
 ```md  
 The Window (Unicode Box Art):  
 ┌────────────────────────────────────────────┐  
 │ [#1 Title] │  
 │ #3 Know. │  
 │ #4 Scan. │  
 │ #5 Know. │  
 │ │  
 │ #2 Data │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ │  
 │ │  
 └────────────────────────────────────────────┘  
 ```  
  
 Args:  
 Screen (kivy): the base class for a screen.  
 """  
  
 def \_\_init\_\_(self, \*\*kw):  
 name = 'View'  
 super().\_\_init\_\_(name=name, \*\*kw)  
 Hover.enter(name)  
  
 everything = BoxLayout(orientation='vertical')  
 title = title = Label(  
 text=f"[color={GREEN}]View Past Scan[/color]",  
 size=(0, TITLE\_HEIGHT),  
 size\_hint=(1, None),  
 font\_size=TITLE\_FONT\_SIZE,  
 underline=True,  
 pos\_hint={'center\_x': .5, 'top': 1},  
 markup=True  
 )  
 everything.add\_widget(title)  
 everything.add\_widget(Pages())  
 everything.add\_widget(ViewScreenInfo())  
  
 self.add\_widget(everything)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file provides the View Screen for the gui.")  
 print("It is only accessable through the Import button on the Save Screen.\n")  
 print("""  
 ┌────────────────────────────────────────────┐  
 │ [#1 Title] │  
 │ #3 Know. │  
 │ #4 Scan. │  
 │ #5 Know. │  
 │ │  
 │ #2 Data │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ [\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] │  
 │ │  
 │ │  
 └────────────────────────────────────────────┘""")

# ./Scanner/gui\Screens\\_\_init\_\_.py

# ./Scanner/gui\Screens\\_\_pycache\_\_\KnowScreen.cpython-310.pyc

[binary content]

# ./Scanner/gui\Screens\\_\_pycache\_\_\Pages.cpython-310.pyc

[binary content]

# ./Scanner/gui\Screens\\_\_pycache\_\_\SaveScreen.cpython-310.pyc

[binary content]

# ./Scanner/gui\Screens\\_\_pycache\_\_\ScanScreen.cpython-310.pyc

[binary content]

# ./Scanner/gui\Screens\\_\_pycache\_\_\StartScreen.cpython-310.pyc

[binary content]

# ./Scanner/gui\Screens\\_\_pycache\_\_\ViewScreen.cpython-310.pyc

[binary content]

# ./Scanner/gui\Screens\\_\_pycache\_\_\\_\_init\_\_.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\Activation.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\App.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\AppState.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\Configuration.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\Diagrams.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\dialogs.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\Hover.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\Information.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\KivyExtensions.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\KivyFonts.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\ScanClasses.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\StartApp.cpython-310.pyc

[binary content]

# ./Scanner/gui\\_\_pycache\_\_\\_\_init\_\_.cpython-310.pyc

[binary content]

# ./Scanner/scans\ARP.py

import re  
from import\_handler import ImportDefence  
import sys  
  
with ImportDefence():  
 from scapy.sendrecv import sendp  
 from scapy.layers.l2 import Ether, ARP  
  
from util import threadify  
from NetworkStorage import NetworkStorage, NetworkEntity, SpecialInformation, nothing  
from ipconfig import ipconfig  
from Sniffer import Sniffer  
  
  
def scan\_ARP(addresses: list[str]) -> list[str]:  
 """This function tests whether it's possible to connect to other IPv4 addresses `addresses`,  
 using Address Resolution Protocol (ARP) who-has requests.  
 If an is-at response is detected, save the IP and MAC to the NetworkStorage,  
 and return only a list of IPs.  
  
 Args:  
 addresses (list[str]): the IPv4 addresses to try connecting to.  
  
 Returns:  
 list[str]: the IPv4 addresses that sent an is-at response.  
 """  
 results = []  
 appender = lambda x: results.append((x[ARP].hwsrc, x[ARP].psrc))  
 filter\_is\_at\_ARP = lambda x: ARP in x and x[ARP].op == 2 and x[ARP].psrc != ipconfig()["IPv4 Address"]  
 sniffer = Sniffer(prn=appender, lfilter=filter\_is\_at\_ARP, store=False)  
 sniffer.start()  
  
 send\_ARP = lambda packet: sendp(packet, verbose=0)  
 send\_ARP.\_\_name\_\_ = "can\_connect\_ARP\_base"  
 packets = [Ether() / ARP(pdst=address) for address in addresses]  
 threadify(send\_ARP)(packets)  
  
 from scapy.error import Scapy\_Exception as NoNpcap  
 try:  
 if sniffer.running:  
 sniffer.stop()  
 except NoNpcap as e:  
 if e.args[0] == "Unsupported (offline or unsupported socket)":  
 print("Npcap / WinPcap aren't installed. Please install either one lol")  
 sys.exit(1)  
 else:  
 raise  
  
 for result in results:  
 NetworkStorage().add(mac=result[0], ip=result[1])  
  
 return [result[1] for result in results]  
  
  
def scan\_ARP\_continuous():  
 """Starts a passive ARP scan.  
 Detects and extracts information from both `ARP who-has` and `ARP is-at` packets.  
 NetworkStorage handles duplicates.  
  
 Terminates quickly, since `scapy`'s `AsyncSniffer` (updated--inside `Sniffer.Sniffer`) opens its own thread.  
 """  
 # Stage 1  
 from subprocess import CalledProcessError, check\_output as read\_command, DEVNULL  
 try:  
 lines = read\_command(['arp', '-a'], stderr=DEVNULL).decode(encoding='utf-8', errors='ignore').split('\n')  
 for line in lines:  
 if re.match(r'^ \*[0-9.]\* \*[0-9A-Fa-f-]\* \*(dynamic|static)', line):  
 parts = [x for x in line.split(' ') if x not in ("", '\r')]  
 NetworkStorage().add(mac=parts[1], ip=parts[0])  
 entity = NetworkEntity(mac=parts[1], ip=parts[0], ipv6=nothing.ipv6, name="Unknown")  
 SpecialInformation()[entity, 'ARP cache entry type'] = parts[2]  
 except CalledProcessError:  
 pass  
  
 # Stage 2  
 filter\_ARP = lambda x: ARP in x  
  
 def save\_to\_storage(packet):  
 # print(packet.summary())  
 if packet[ARP].op == 2:  
 NetworkStorage().add(mac=packet[ARP].hwsrc, ip=packet[ARP].psrc)  
 NetworkStorage().add(mac=packet[ARP].hwdst, ip=packet[ARP].pdst)  
 elif packet[ARP].op == 1:  
 NetworkStorage().add(mac=packet[ARP].hwsrc, ip=packet[ARP].psrc)  
  
 sniffer = Sniffer(prn=save\_to\_storage, lfilter=filter\_ARP, store=False)  
 sniffer.start()

# ./Scanner/scans\Discovery.py

from import\_handler import ImportDefence  
with ImportDefence():  
 from scapy.sendrecv import send  
 from scapy.all import IP, UDP, Raw  
   
 from time import sleep  
 from threading import Thread  
  
  
DST\_PORT = 3581  
  
def reveal\_myself():  
 from gui.dialogs import get\_string  
 name = get\_string("Reveal Myself As", "Starting Device Discovery\nInsert the name you wish to reveal to others:")  
 from globalstuff import terminator  
 from NetworkStorage import broadcast  
 while not terminator.is\_set():  
 packet = IP(dst=broadcast.ip)  
 packet /= UDP(dport=DST\_PORT)  
 packet /= Raw(load="Hello there, sir " + name)  
 send(packet, verbose=0)  
 sleep(3)  
  
  
class DeviceDiscoveryListener:  
 \_instance = None  
 def \_\_new\_\_(cls):  
 if cls.\_instance is None:  
 cls.\_instance = super().\_\_new\_\_(cls)  
   
 return cls.\_instance  
  
 def \_\_init\_\_(self):  
 from PacketSniffer import PacketSniffer  
 PacketSniffer().add\_observer(self.check\_packet)  
  
 def check\_packet(self, packet):  
 from NetworkStorage import broadcast, SpecialInformation, NetworkEntity, nothing  
 if UDP not in packet:  
 return  
 if packet[IP].dst != broadcast.ip:  
 return  
 if packet[UDP].dport != DST\_PORT:  
 return  
 entity = NetworkEntity(mac=nothing.mac, ip=packet[IP].src, ipv6=nothing.ip6, name=nothing.name)  
 SpecialInformation()[entity, 'discovery'] = packet[Raw]  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This file is responsible for the Device Discovery scan,")  
 print("which reveals this computer to others that use this software")

# ./Scanner/scans\ICMP.py

from import\_handler import ImportDefence  
  
with ImportDefence():  
 from util import threadify, barstyle  
 from PrintingContexts import InstantPrinting, TablePrinting, JustifyPrinting, AutoLinebreaks  
 from ipconfig import ipconfig  
 from NetworkStorage import NetworkStorage  
 from colors import Colors  
 from hostify import hostify, hostify\_sync  
 from ip\_handler import subnet\_address\_range  
  
 from queue import Queue  
 from threading import Thread  
 from time import sleep  
 from math import floor  
 import os  
  
 from scapy.sendrecv import sr1  
 from scapy.layers.inet import IP, ICMP  
  
  
continuous\_pause\_seconds = 1.1  
save\_count = 60  
  
  
def shift(seq: list, n: int) -> list:  
 """Shifts / Rotates / Rolls a list `seq` by `n` places.  
  
 Example:  
 shift([1, 2, 3, 4], 1) -> [2, 3, 4, 1]  
  
 Args:  
 seq (list): The list to shift.  
 n (int): The amount of places to shift by.  
  
 Returns:  
 list: the shifted list.  
 """  
 if len(seq) == 0:  
 return []  
 if len(seq) == 1:  
 return seq  
 if n == 0:  
 return seq  
 n = n % len(seq)  
 return seq[n:] + seq[:n]  
  
  
def render\_opacity(percent: int | float):  
 """Returns a character to display a given opacity/fillness,  
 according to a percent:  
 0%-20% -> " "  
 ...  
 40%-60% -> "▒"  
 ...  
 80%-100% -> "█"  
  
 if the percent is outside the range [0, 100] inclusive, returns "X".  
  
 ```  
 -10% X  
 -5% X  
 0%  
 5%  
 10%  
 15%  
 20%  
 25% ░  
 30% ░  
 35% ░  
 40% ░  
 45% ▒  
 50% ▒  
 55% ▒  
 60% ▒  
 65% ▓  
 70% ▓  
 75% ▓  
 80% ▓  
 85% █  
 90% █  
 95% █  
 100% █  
 105% X  
 110% X  
 115% X  
 120% X  
 ```  
  
 Args:  
 percent (int | float): the percent to match.  
  
 Returns:  
 str: a single character representing that percent of fillness.  
 """  
 if not (0 <= percent <= 100):  
 return "X"  
  
 characters = " ░▒▓█"  
 # characters = " -─=≡▄█"  
  
 # the 0.1 doesn't change the results,  
 # it just prevents an `IndexError: string index out of range` for `percent=100`.  
 jump = 0.1 + 100 / len(characters)  
 level = floor(percent / jump)  
 return characters[level]  
  
  
def can\_connect\_ICMP\_base(address: str) -> bool:  
 """This function tests whether it's possible to connect to another IPv4 address `address`,  
 using an Internet Control Message Protocol (ICMP) ping request.  
 If the address given is localhost, `return False`.  
  
 Args:  
 address (str): the IPv4 address to try pinging.  
  
 Returns:  
 bool: a boolean indicating whether the echo ping had been successfully sent, and a response was received.  
 """  
 if address == ipconfig()["IPv4 Address"]:  
 return False  
 packet = IP(dst=address) / ICMP()  
 response = sr1(packet, verbose=0, timeout=1)  
 if response is not None:  
 # print(response[IP].show())  
 if response[ICMP].type == 0:  
 NetworkStorage().add(ip=response[IP].src)  
 return True  
 return False  
  
  
can\_connect\_ICMP\_base.options = {"format": barstyle("Dot Fill")}  
scan\_ICMP = threadify(can\_connect\_ICMP\_base)  
  
  
def calculate\_opacity(connections: list[bool]) -> float:  
 """This function calculates the opacity of a given connection list (a list of booleans indicating some contacting attempts' successes),  
 according to a probabilistic formula derived in an attached TXT file.  
  
 Args:  
 connections (list[bool]): a list of contact attempts' successes, taking the form of `[...True, True, False, True, False]`  
  
 Returns:  
 float: a value between `0.0` (disconnected) and `1.0` (connected) representing certainty that the device is still connected (a.k.a its opacity).  
 """  
  
 # # │++ │+++++++++  
 # # │ ++ │ +++++  
 # # │ ++ │ ++  
 # # │ ++ │ ++  
 # # │ ++ => │ +  
 # # │ ++ │ +  
 # # │ ++ │ +  
 # # │ ++ │ +  
 # # │ ++ │ +  
 # # │ ++ │ +  
 # # ──┼───────────────────────── ──┼────────────────────────  
 # # │ │  
  
 # GONE\_AFTER: int = 11  
  
 # if len(connections) == 0: return 0.0  
 # if not any(connections): return 0.0  
 # distance\_to\_last = list(reversed(connections)).index(True)  
 # # Change this function? (see art above)  
 # opacity = 1.0 - distance\_to\_last / GONE\_AFTER  
 # if opacity < 0: return 0.0  
 # # Maybe calculate the average amount of disconnected time for devices? And not just choose some random numbers?  
 # return opacity  
 if len(connections) == 0:  
 return 1.0  
 if not any(connections):  
 return 0.0  
 n = list(reversed(connections)).index(True)  
 a = connections.count(True) / len(connections)  
 return a \*\* n  
  
  
def scan\_ICMP\_continuous(addresses, all\_possible\_addresses, parallel\_device\_discovery=True, compactness=0):  
 # compactness=0 -> "255.255.255.255 (Smartphone-Galaxy-S90-5G) █████ █ ███████ █ ███ ████┅ [█]".  
 # compactness=1 -> "255.255.255.255 (Smartphone-Galaxy-S90-5G) [█]".  
 # compactness=2 -> "<ff:ff:ff:ff:ff:ff | 255.255.255.255 | Smartphone-Galaxy-S90-5G>" (text colour changes depending on opacity).  
 # otherwise -> "255.255.255.255 (Smartphone-Galaxy-S90-5G)" (text colour changes depending on opacity).  
 if not isinstance(addresses, list):  
 addresses = list(addresses)  
 table = {address: [] for address in addresses}  
 waiting = Queue()  
  
 network = subnet\_address\_range(ipconfig()["Subnet Mask"], ipconfig()["IPv4 Address"])  
 # from NetworkStorage import router  
 # network = subnet\_slash\_notation(ipconfig()["Subnet Mask"], router.ip)  
  
 if parallel\_device\_discovery:  
 # How many threads should be dedicated to the detection of new devices?  
 # The iteration shifts in different threads by `shifting = thread\_index \* 71 mod 255`, to ensure efficiency.  
 # Range of values: 1 to 18 (inclusive).  
 # Optimal values: 18, 6, 3, 2, 1  
 SCANNER\_THREADS = 6  
  
 def ICMP\_live\_device\_discovery(order: int):  
 all\_addresses = shift(all\_possible\_addresses, 71 \* order)  
 while True:  
 for address in all\_addresses:  
 if address in table.keys():  
 continue  
 if address in waiting.queue:  
 continue  
 if can\_connect\_ICMP\_base(address):  
 waiting.put(address)  
  
 for i in range(SCANNER\_THREADS):  
 Thread(target=ICMP\_live\_device\_discovery, args=(i, )).start()  
  
 def resolve\_queue():  
 lookup = NetworkStorage()  
 while not waiting.empty():  
 address = waiting.get()  
 if address not in addresses:  
 addresses.append(address)  
 lookup.add(ip=address)  
 print("Adding address", address)  
  
 def sweep\_scan():  
 for address, online in zip(addresses, scan\_ICMP(addresses)):  
 if address not in table:  
 table[address] = []  
 table[address].append(online)  
 if len(table[address]) > save\_count:  
 table[address] = table[address][-save\_count:]  
  
 def print0(sorted\_table):  
 print(f"Connection testing (ICMP ping) to {network}\n")  
  
 with InstantPrinting():  
 example\_length = len("255.255.255.255 (Smartphone-Galaxy-S90-5G)")  
 bar\_length = os.get\_terminal\_size().columns - example\_length - len(": ") - len("┅ [ ]")  
 for address in sorted\_table:  
 print(  
 f"{address} ({hostify(address)}): ".rjust(example\_length),  
 (''.join(['█' if x else ' ' for x in table[address][-bar\_length:]]) + "┅ ").rjust(bar\_length),  
 f"[{render\_opacity(100 \* calculate\_opacity(table[address]))}]"  
 )  
 print()  
  
 def print1(sorted\_table):  
 print(f"Connections (ICMP ping) to {network}\n")  
 with TablePrinting():  
 for address in sorted\_table:  
 print(  
 address,  
 f"({hostify(address)})",  
 f"[{render\_opacity(100 \* calculate\_opacity(table[address]))}]"  
 )  
  
 def print2(sorted\_table):  
 print(f"ICMP ping sweep over {network}\n")  
 with JustifyPrinting():  
 opacities = [Colors.BLACK, Colors.DARK\_GRAY, Colors.LIGHT\_GRAY, Colors.LIGHT\_WHITE]  
 data = NetworkStorage().organise('ip')  
 for address in sorted\_table:  
 opacity = calculate\_opacity(table[address])  
 index = floor(opacity \* (len(opacities) - 1))  
 if index == 0:  
 continue  
 color = opacities[index]  
 try:  
 print(f"{color}{data[address]}{Colors.END} ")  
 except KeyError:  
 print(f"{color}{address} ({hostify(address)}){Colors.END} ")  
  
 def print3(sorted\_table):  
 print(f"ICMP continuous: {network}\n")  
 with AutoLinebreaks():  
 opacities = [Colors.BLACK, Colors.DARK\_GRAY, Colors.LIGHT\_GRAY, Colors.LIGHT\_WHITE]  
 for address in sorted\_table:  
 opacity = calculate\_opacity(table[address])  
 index = floor(opacity \* (len(opacities) - 1))  
 if index == 0:  
 continue  
 color = opacities[index]  
 print(f"{color}{address} ({hostify(address)}){Colors.END} ")  
  
 while True:  
 sleep(continuous\_pause\_seconds)  
  
 resolve\_queue()  
 sweep\_scan()  
  
 hostify\_sync(list(table.keys()))  
 # os.system("cls")  
  
 # sorted\_table = sorted(table.keys(), key=lambda x: int(''.join(x.split('.'))))  
  
 # try:  
 # [print0, print1, print2][compactness](sorted\_table)  
 # except KeyError:  
 # print3()

# ./Scanner/scans\PublicAddress.py

from import\_handler import ImportDefence  
with ImportDefence():  
 import requests  
  
from NetworkStorage import NetworkStorage  
from CacheDecorators import one\_cache  
  
  
def public\_address\_action():  
 NetworkStorage().add(get\_public\_ip())  
  
  
@one\_cache  
def get\_public\_ip():  
 from NetworkStorage import nothing, NetworkStorage, LockedNetworkEntity  
 ip = requests.get('https://api.ipify.org').text  
 ipv6 = requests.get('https://api64.ipify.org').text  
 ipv6 = ipv6 if ipv6 != ip else nothing.ipv6  
 try:  
 outside = LockedNetworkEntity(  
 mac=nothing.mac,  
 ip=ip,  
 ipv6=ipv6,  
 name="Public Address"  
 )  
 except ValueError: # api64.ipify.org might not return the IPv6, and instead say "gateway timeout"  
 outside = LockedNetworkEntity(  
 mac=nothing.mac,  
 ip=ip,  
 ipv6=nothing.ipv6,  
 name="Public Address"  
 )  
 NetworkStorage().special\_add(outside)  
 return outside  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This scan-like action fetches the public IP of the router,")  
 print("using https://api.ipify.org API.")  
 print("It has to be act on outside the network, of course.")

# ./Scanner/scans\TCP.py

from import\_handler import ImportDefence  
  
with ImportDefence():  
 from scapy.all import sr1, TCP, IP  
 from random import randint  
 import ipaddress  
  
 from util import threadify  
 from PrintingContexts import InstantPrinting  
 from gui.dialogs import get\_string, popup  
 from NetworkStorage import match, SpecialInformation  
  
  
# A range for the scanned ports.  
PORT\_RANGE = list(range(0, 1024))  
  
# Enum-like  
OPEN = 0  
CLOSE = 1  
RESET = 2  
  
  
def scan\_port(ip: str, port: int) -> int:  
 seq = randint(0, 1000)  
 syn\_segment = TCP(dport=port, seq=seq, flags='S')  
 syn\_packet = IP(dst=ip) / syn\_segment  
 syn\_ack\_packet = sr1(syn\_packet, timeout=3, verbose=False)  
 if syn\_ack\_packet is None:  
 return CLOSE  
 else:  
 if 'R' in syn\_ack\_packet[TCP].flags:  
 return RESET  
 else:  
 return OPEN  
  
  
scan\_ports = threadify(scan\_port, silent=False)  
  
  
def scan\_TCP(ip: str, repeats: int) -> dict:  
 ports = [(ip, port) for port in PORT\_RANGE]  
 result = {port: False for port in PORT\_RANGE}  
  
 for \_ in range(repeats):  
 one\_scan = scan\_ports(ports)  
 for i, port in enumerate(PORT\_RANGE):  
 if one\_scan[i] == OPEN:  
 result[port] = True  
  
 return result  
  
  
def port\_scan\_TCP():  
 address = get\_string("Choose IP", "Which IP address? ")  
  
 def is\_ipv4(string):  
 try:  
 ipaddress.IPv4Network(string)  
 return True  
 except ValueError:  
 return False  
  
 while not is\_ipv4(address):  
 popup("Invalid IP", "This does not appear to be a valid IPv4 address.", error=True)  
 address = get\_string("Choose IP", "Which IP address? ")  
  
 try:  
 repeats = int(get\_string("Repeats", "How many repeats? [default=3]"))  
 except ValueError:  
 repeats = 3  
  
 try:  
 minimum = int(get\_string("Port Range - Minimum", "Choose min port [default=0 to 1024, min=0]: "))  
 maximum = int(get\_string("Port Range - Maximum", "Choose max port [default=0 to 1024, max=65536]: "))  
 if maximum < minimum:  
 raise ValueError("Maximum can't be smaller than minimum.")  
 if minimum < 0:  
 minimum = 0  
 if maximum > 65536:  
 maximum = 65536  
 global PORT\_RANGE  
 PORT\_RANGE = list(range(minimum, maximum))  
 except ValueError:  
 pass  
  
 entity = match(address)  
 results = scan\_TCP(address, repeats)  
 open\_ports = [port for port, res in results.items() if res]  
 SpecialInformation()[entity, "Open TCP ports"] = open\_ports  
  
# This is not supposed to be under an `if \_\_name\_\_ == '\_\_main\_\_`. It's called from ./Do TCP Scan.bat.  
# If one did execute it from this context, Python would be unable to `import util` for example, as that's an outside file.  
def main(addr=''):  
 import ipaddress  
  
 if addr:  
 address = addr  
 else:  
 address = input("Which IP address? ")  
  
 def is\_ipv4(string):  
 try:  
 ipaddress.IPv4Network(string)  
 return True  
 except ValueError:  
 return False  
  
 while not is\_ipv4(address):  
 print("This does not appear to be a valid IPv4 address.")  
 address = input("Which IP address? ")  
  
 try:  
 repeats = int(input("How many repeats? "))  
 except ValueError:  
 repeats = 3  
  
 try:  
 minimum = int(input("Choose min port: "))  
 maximum = int(input("Choose max port: "))  
 if maximum < minimum:  
 raise ValueError("Maximum can't be smaller than minimum.")  
 if minimum < 0:  
 minimum = 0  
 if maximum > 65536:  
 maximum = 65536  
 global PORT\_RANGE  
 PORT\_RANGE = list(range(minimum, maximum))  
 except ValueError:  
 pass  
  
 print()  
 print(f"Open TCP ports in {address} in range {PORT\_RANGE[0]} → {PORT\_RANGE[-1]} ({repeats} repeats):")  
 with InstantPrinting():  
 for port, res in scan\_TCP(address, repeats).items():  
 if res:  
 print(port)

# ./Scanner/scans\TraceRouter.py

from import\_handler import ImportDefence  
import sys  
  
with ImportDefence():  
 from scapy.sendrecv import sr1  
 from scapy.layers.inet import IP, ICMP  
  
 from util import threadify  
 from NetworkStorage import NetworkStorage  
 from ipconfig import ipconfig  
  
# This is intentionally not @threadify-ied.  
# The ICMP echoes in each hop are supposed to be temporally separated.  
def hop(ttl, dst):  
 TIMEOUT = 0.1  
 REPEATS = 3  
 results = set()  
 for \_ in range(REPEATS):  
 packet = IP(ttl=ttl, dst=dst) / ICMP()  
 res = sr1(packet, timeout=TIMEOUT, verbose=False)  
 if res is None:  
 continue  
 ip = res[IP].src  
 if ip == dst:  
 return dst  
 results.add(ip)  
 if len(results) == 0:  
 return 'Timed Out'  
 if len(results) >1:  
 return 'Undefined'  
 return results.pop()  
  
  
def traceroute(dst):  
 ROUTE\_MAX = 20  
 path = []  
 for ttl in range(1, ROUTE\_MAX):  
 path.append(hop(ttl, dst))  
 for previous, address in zip(path, path[1:]):  
 if address == 'Timed Out':  
 continue  
 NetworkStorage().add(ip=address)  
 NetworkStorage().connect(previous, address)  
 return path

# ./Scanner/scans\\_\_init\_\_.py

# ./Scanner/scans\\_\_pycache\_\_\Analyses.cpython-310.pyc

[binary content]

# ./Scanner/scans\\_\_pycache\_\_\ARP scans.cpython-310.pyc

[binary content]

# ./Scanner/scans\\_\_pycache\_\_\ARP.cpython-310.pyc

[binary content]

# ./Scanner/scans\\_\_pycache\_\_\Discovery.cpython-310.pyc

[binary content]

# ./Scanner/scans\\_\_pycache\_\_\ICMP.cpython-310.pyc

[binary content]

# ./Scanner/scans\\_\_pycache\_\_\OS\_ID.cpython-310.pyc

[binary content]

# ./Scanner/scans\\_\_pycache\_\_\PublicAddress.cpython-310.pyc

[binary content]

# ./Scanner/scans\\_\_pycache\_\_\TCP.cpython-310.pyc

[binary content]

# ./Scanner/scans\\_\_pycache\_\_\TraceRouter.cpython-310.pyc

[binary content]

# ./Scanner/scans\\_\_pycache\_\_\\_\_init\_\_.cpython-310.pyc

[binary content]

# ./Scanner/testing\tests.py

import os  
import sys  
  
from ip\_handler import bitify, unbitify  
from ipconfig import dictify, ipconfig  
from PrintingContexts import NoPrinting  
from scans.ICMP import shift  
from util import threadify  
import inspect  
  
  
def dictify\_example1():  
 x = 'Windows IP Configuration\n Host Name . . . . . . . . . . . . : MyComputer-007\n Primary Dns Suffix . . . . . . . :\n Node Type . . . . . . . . . . . . : Hybrid\n IP Routing Enabled. . . . . . . . : No'  
 return dictify(x) == {'Windows IP Configuration': {'Host Name': 'MyComputer-007', 'Primary Dns Suffix': [], 'Node Type': 'Hybrid', 'IP Routing Enabled': 'No'}}  
  
  
def dictify\_example2():  
 x = 'Wireless LAN adapter Wi-Fi:\n Media State . . . . . . . . . . . : Media disconnected\n Connection-specific DNS Suffix . : local\n Description . . . . . . . . . . . : Wireless-ABCDE\n Physical Addresses. . . . . . . . : AB-CD-EF-01-02-03\n AB-CD-EF-01-02-04\n AB-CD-EF-01-02-05\n DHCP Enabled. . . . . . . . . . . : Yes\n Autoconfiguration Enabled . . . . : Yes'  
 return dictify(x) == {'Wireless LAN adapter Wi-Fi': {'Media State': 'Media disconnected', 'Connection-specific DNS Suffix': 'local', 'Description': 'Wireless-ABCDE', 'Physical Addresses': ['AB-CD-EF-01-02-03', 'AB-CD-EF-01-02-04', 'AB-CD-EF-01-02-05'], 'DHCP Enabled': 'Yes', 'Autoconfiguration Enabled': 'Yes'}}  
  
  
def ipconfig\_data():  
 data = ipconfig()  
 for key in ["IPv4 Address", "Subnet Mask"]:  
 if key not in data:  
 print("ipconfig() has no key \"" + key + "\".")  
 return False  
 return True  
  
  
def bitify\_examples():  
 return bitify("0.0.0.0") == '00000000000000000000000000000000' \  
 and bitify("255.255.255.255") == '11111111111111111111111111111111' \  
 and bitify("192.168.0.1") == '11000000101010000000000000000001'  
  
  
def unbitify\_examples():  
 return unbitify('00000000000000000000000000000000') == "0.0.0.0" \  
 and unbitify('11111111111111111111111111111111') == "255.255.255.255" \  
 and unbitify('11000000101010000000000000000001') == "192.168.0.1"  
  
  
def valid\_subnet\_mask():  
 mask = ipconfig()["Subnet Mask"]  
 mask = bitify(mask)  
 counting, ones, zeros = "ones", 0, 0  
 for c in mask:  
 if c == '1':  
 if not counting == "ones":  
 return False  
 ones += 1  
 elif c == '0':  
 counting = "zeros"  
 zeros += 1  
 else:  
 return False  
 return ones + zeros == 32  
  
  
def threadify\_echo\_test():  
 echo = lambda x: x  
 echo = threadify(echo, silent=True)  
 return echo([1, 2, 3, 4, 5]) == [1, 2, 3, 4, 5]  
  
  
def shift\_list\_test():  
 a = [1, 2, 3]  
 return shift(a, 1) == [2, 3, 1] and shift(a, 2) == [3, 1, 2] and shift(a, 3) == a  
  
  
def does\_winpcap\_exist():  
 """WinPcap / Npcap aren't installed. It is essential that you install either one. https://npcap.com/#download"""  
 try:  
 from scapy.all import sendp, Ether, IP, ICMP  
 except (ImportError, ModuleNotFoundError):  
 return False  
  
 try:  
 with NoPrinting():  
 sendp(Ether() / IP() / ICMP(), verbose=0) # Sends a default to ICMP packet to localhost (so no network traffic generated).  
 except RuntimeError:  
 return False  
 return True  
  
  
def does\_fallback\_font\_exist():  
 """A necessary font (Segoe UI Symbol) was not found under `fonts/Segoe UI Symbol.ttf`. Please check your installation of the software."""  
 import os.path  
 return os.path.isfile(r".\fonts\Segoe UI Symbol.ttf")  
  
  
def is\_win32\_pip\_installed():  
 """`pywin32` is a module necessary for the GUI. It was not installed. Installing it failed."""  
 # This is supposed to be resolved with the updates to ImportDefence,  
 # but you can never be too safe.  
 try:  
 import win32api  
 return True  
 except ModuleNotFoundError:  
 from subprocess import check\_call as do\_command, CalledProcessError  
 try:  
 do\_command([sys.executable, "-m", "pip", "install", "pywin32"])  
 return True  
 except CalledProcessError:  
 return False  
  
  
def is\_sqlite\_table\_information\_present():  
 """The SQLite table `information` was not found. Please execute `python db.py` to solve this issue, and re-run the software."""  
 try:  
 import db  
 db.get\_scans()  
 return True  
 except FileNotFoundError:  
 return False  
  
  
def test\_append\_and\_getitem():  
 from PacketSniffer import ListWithSQL  
 db\_path = "test.db"  
 try:  
 list\_with\_sql = ListWithSQL(db\_path, maxram=2)  
  
 # Append some items to the list  
 list\_with\_sql.append("foo")  
 list\_with\_sql.append("bar")  
  
 # Check that the items were appended correctly  
 assert list\_with\_sql[0] == "foo"  
 assert list\_with\_sql[1] == "bar"  
  
 # Append some more items to the list  
 list\_with\_sql.append("baz")  
 list\_with\_sql.append("qux")  
  
 # Check that the items were appended correctly  
 assert list\_with\_sql[2] == "baz"  
 assert list\_with\_sql[3] == "qux"  
  
 # Change an item in the list  
 list\_with\_sql[1] = "spam"  
  
 # Check that the item was changed correctly  
 assert list\_with\_sql[1] == "spam"  
  
 # Check that the list length is correct  
 assert len(list\_with\_sql) == 4  
 except (AssertionError, ValueError, IndexError, NotImplementedError):  
 return False  
 return True  
  
  
def test() -> None:  
 os.system("") # Enables ANSI colouring  
 # get all functions defined in this module (tests.py)  
 tests = inspect.getmembers(sys.modules[\_\_name\_\_], inspect.isfunction)  
 # extract the functions, if they're defined here (i.e. not imported),  
 # and exclude `test` (because that'll cause some infinite recursion issues.)  
 tests = [func for name, func in tests if func.\_\_module\_\_ == \_\_name\_\_ and name != 'test']  
 # sort by alphabetical order  
 tests.sort(key=lambda func: func.\_\_name\_\_)  
 # Each element is a boolean function. False means the test failed.  
 results = [not run() for run in tests]  
 # Log all tests, both successful and unsuccessful.  
 with open('tests\_log.txt', 'w') as log:  
 log.write('\n'.join([  
 test.\_\_name\_\_ + " " + ("Successful" if not result else "Unsuccessful")  
 for test, result in zip(tests, results)  
 ]))  
 if not any(results):  
 log.write("\n\nAll tests were successful.")  
 # If any tests failed, print them to the user, and ask for confirmation.  
 if any(results):  
 print("Failed tests:")  
 for i in [index for index, bad\_result in enumerate(results) if bad\_result]:  
 test = tests[i]  
 print(" •", test.\_\_doc\_\_ if test.\_\_doc\_\_ else test.\_\_name\_\_)  
 print("The software might work incorrectly or crash.\nContinue execution only if you're sure.\nOtherwise, close this window.")  
 input("Press any key to continue. . . ")  
 else:  
 print("All tests were successful.")  
 print(sys.version)  
 print("\033[0m") # End colours  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print("This module runs a few tests.")  
 print("You can define additional tests here according to these guidelines:")  
 print(" - the return value is boolean: True means all good, False means the test failed.")  
 print(" - You can define a docstring as a custom error message, otherwise, the function name will be displayed.")

# ./Scanner/testing\\_\_pycache\_\_\tests.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\CacheDecorators.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\colors.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\CommandLineStyle.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\db.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\Decorators.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\exe.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\files.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\files\_cryptography.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\globalstuff.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\gui.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\gui.cpython-38.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\hostify.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\hover.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\import\_handler.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\import\_handler.cpython-38.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\ipconfig.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\ip\_handler.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\main.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\NetworkStorage.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\PacketSniffer.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\PrintingContexts.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\register.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\ScanID.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\SimpleScan.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\Sniffer.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\starter.cpython-310.pyc

[binary content]

# ./Scanner/\_\_pycache\_\_\util.cpython-310.pyc

[binary content]