ICMP “Echo” requests in python

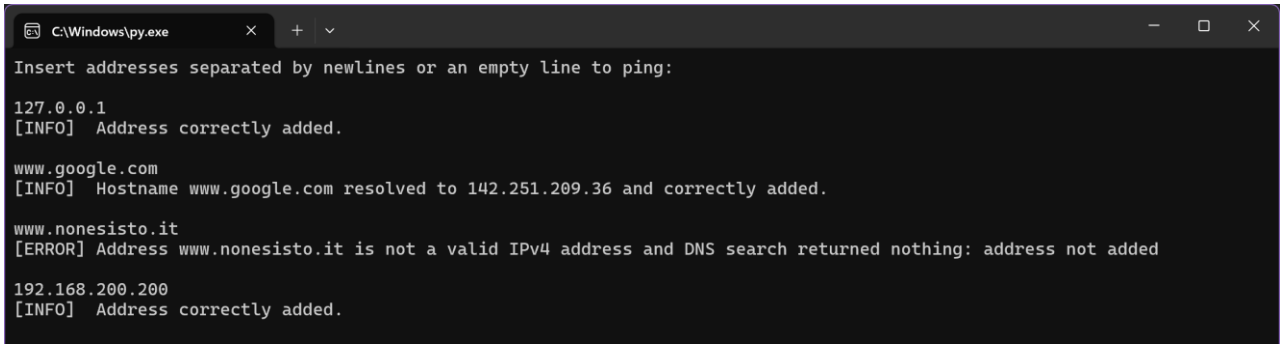
Relazione allegata all’elaborato rappresentante la soluzione sviluppata da Davide Fiocchi (matricola 0001070915) al problema esposto nella terza traccia:

Realizzare uno script Python per monitorare lo stato di una rete, controllando la disponibilità di uno o più host tramite il protocollo ICMP (ping). Lo script deve consentire all'utente di specificare gli indirizzi IP degli host da monitorare e deve visualizzare lo stato (online/offline) di ciascun host.

Sommario

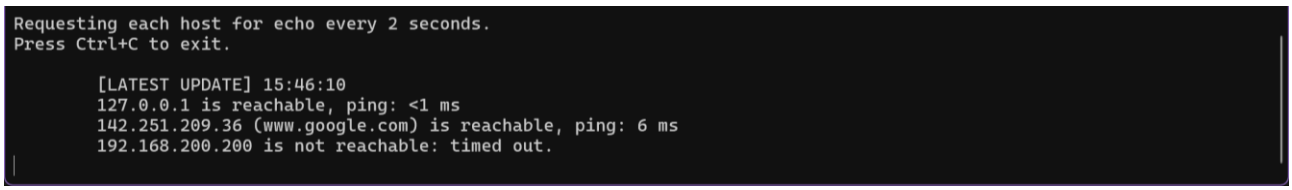
I – utilizzo dello script

Lo script si compone di un unico file composto da un main che gestisce l'interazione con l'utente attraverso una cli e diverse funzioni necessarie per raggiungere lo scopo che si prefigge. Sarà quindi sufficiente avviare l'esecuzione del file "ping.py".
Il modello di utilizzo è semplice: si inseriscono in un primo momento tutti gli indirizzi (ipv4 o mnemonici) che si vogliono monitorare, in seguito lo script monitorerà in modo ininterrotto questi stessi bersagli fino all'interruzione del processo (per esempio mediante interrupt da tastiera Ctrl+C o chiusura della finestra che ospita il tty).
Nello specifico i bersagli sono da inserire uno alla volta, andando a capo tra l'uno e l'altro. Dopo la pressione del tasto invio il programma fornirà un feedback sul risultato dell'elaborazione dell'input: se l'indirizzo era di tipo ipv4 sarà sicuramente aggiunto alla lista degli indirizzi monitorati, in caso contrario la sua aggiunta dipenderà dall'esito di una query DNS.



Output dello script in seguito all'inserimento di diversi indirizzi (192.168.200.200 non corrisponde a nessun host ma viene aggiunto lo stesso, sarà controllata in seguito la sua disponibilità tramite protocollo ICMP)

Una volta specificati tutti gli indirizzi desiderati, basta andare a capo su una linea vuota per segnalare allo script di procedere al controllo della disponibilità di host che rispondano agli indirizzi inseriti.
Lo script comincerà quindi a restituire a intervalli di tempo regolari (che dipendono anche dalla quantità di host da contattare e l'eventuale loro irraggiungibilità) informazioni sugli host che sono stati correttamente aggiunti.

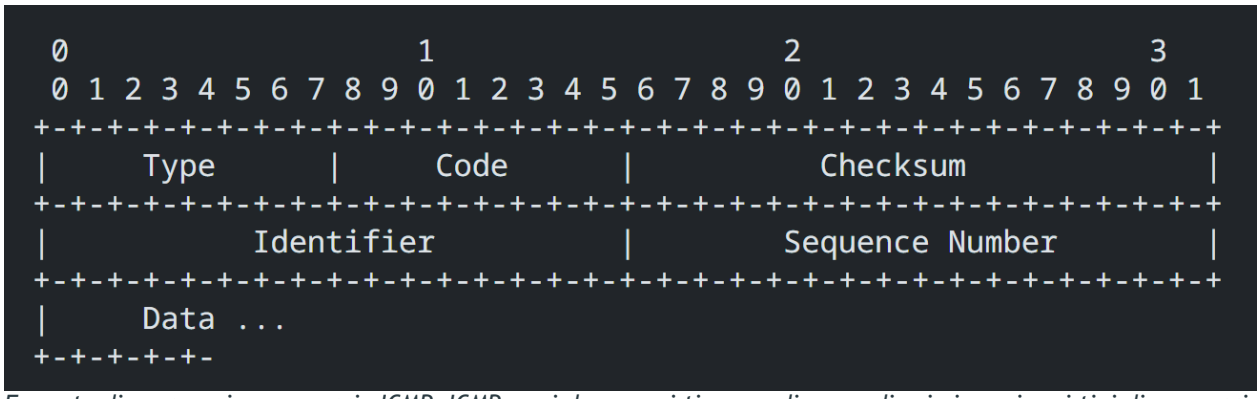


II – meccanismo

Lo script si basa sull'utilizzo del protocollo ICMP. Dall'RFC 729:

> *Occasionally a gateway or destination host will communicate with a source host, for example, to report an error in datagram processing. For such purposes this protocol, the Internet Control Message Protocol (ICMP), is used. ICMP, uses the basic support of IP as if it were a higher level protocol, however, ICMP is actually an integral part of IP, and must be implemented by every IP module.*

Questo protocollo permette di inviare alcuni specifici messaggi ad altri host che implementano IP impostando il campo protocollo a 1 (il numero scelto per indicare il protocollo ICMP), in particolare il tipo di messaggio che ci permette di verificare la disponibilità di un altro host è il cosiddetto messaggio di "Echo" a cui l'host ricevente dovrebbe (in condizioni normali) rispondere con un messaggio di "Echo Reply".



Formato di un generico messaggio ICMP. ICMP usa i due campi tipo e codice per discriminare i vari tipi di messaggio che il protocollo può trasferire.

Facendo riferimento all'immagine, un messaggio di richiesta Echo dovrebbe avere il campo "Type" impostato a 8, e il campo "Code" a 0; per una risposta invece anche "Type" è impostato a 0:

> *The address of the source in an echo message will be the destination of the echo reply message. To form an echo reply message, the source and destination addresses are simply reversed, the type code changed to 0, and the checksum recomputed.*

I campi "Identifier" e "Sequence Number" aiutano a identificare una risposta Echo nel caso si abbiano mandato più richieste (magari a host diversi) in un breve periodo: questi campi, infatti, assieme alla sezione "Data" devono essere contenuti identicamente nella risposta Echo inviata da un host che abbia ricevuto una richiesta di Echo e sia collaborativo rispetto a questo meccanismo:

> *The data received in the echo message must be returned in the echo reply message. The identifier and sequence number may be used by the echo sender to aid in matching the replies with the echo requests. For example, the identifier might be used like a port in TCP or UDP to identify a session, and the sequence number might be incremented on each echo request sent. The echoer returns these same values in the echo reply.*

Il checksum invece, come definito in questa RFC, è il complemento a uno della somma in complemento a uno dei byte componenti il messaggio ICMP raggruppati a due a due in parole da 2 byte, senza considerare il campo checksum (o considerandolo pari a zero):

> *The checksum is the 16-bit ones's complement of the one's complement sum of the ICMP message starting with the ICMP Type. For computing the checksum , the checksum field should be zero.*

III – implementazione

Composizione del messaggio ICMP

Per prima cosa sarà necessario comporre il messaggio ICMP da inviare ai vari host, di questo si occupa la seguente funzione:

```
def compose_echo_message(identifier, sequence_number):
    temp_message = struct.pack("!BBHHH", TYPE, CODE, 0, identifier,
sequence_number)
    cs = checksum(temp_message)
    message = struct.pack("!BBHHH", TYPE, CODE, cs, identifier, sequence_number)
```

In un pacchetto ICMP di richiesta Echo minimale (che non trasporta dati aggiuntivi) gli unici campi da specificare sono ”Identifier” e “Sequence Number”, quindi la funzione li richiede in ingresso e procede ad assemblarli. Un modo idiomatico in python per interfacciarsi con strutture dati è il modulo struct.py, citando dalla documentazione:

> *The module’s functions and objects can be used for two largely distinct applications, data exchange with external sources (files or network connections), or data transfer between the Python application and the C layer.*

La funzione struct.pack richiede in input una stringa di formato che descrive il modo in cui i dati in input devono essere disposti e i dati che formeranno il pacchetto di byte restituito; nel nostro caso la stringa formato è “!BBHHH”:

- ! specifica che l’ordinamento dei byte (little-endian / big-endian) deve essere quello generalmente utilizzato dalla rete
- B sta per unsigned char (1 byte senza segno)
  - 1°B = il campo type (caricato da una costante TYPE con valore 8)
  - 2°B = il campo code (caricato da una costante CODE con valore 0)
- H sta per unsigned short (2 byte senza segno)
  - 1°H = il campo checksum impostato temporaneamente a 0 per il calcolo del checksum effettivo
  - 2°H = il campo identifier
  - 3°H = il campo sequence_number

Prima di concludere l’analisi di questa funzione scendiamo nei dettagli del calcolo del checksum.

Calcolo dell’Internet checkusm

Per ogni messaggio composto, dobbiamo calcolare il checksum, in quanto esso varia al variare dei campi identifier e sequence number, di questo si occupano le funzioni:

```
def checksum(data):
    if (len(data) % 2) != 0:
        raise Exception("Data must have an even number of bytes")
    else:
        ocs = ones_complement_sum(data)
        return (~ocs) & 0xffff
```

```
def ones_complement_sum(data):
    if (len(data) % 2) != 0:
        raise Exception("Data must have an even number of bytes")
    else:
        ocs = 0
        for i in range(0, len(data), 2):
            ocs += ((data[i] << 8) + data[i + 1])
        while ocs >> 16:
            ocs = (ocs & 0xffff) + (ocs >> 16)
        return ocs
```

Dalla RFC 1071:

> *In outline, the Internet checksum algorithm is very simple:*
>
> *(1) Adjacent octets to be checksummed are paired to form 16-bit integers, and the 1's complement sum of these 16-bit integers is formed.*
>
> *(2) To generate a checksum, the checksum field itself is cleared, the 16-bit 1's complement sum is computed over the octets concerned, and the 1's complement of this sum is placed in the checksum field.*
>
> *(3) To check a checksum, the 1's complement sum is computed over the same set of octets, including the checksum field. If the result is all 1 bits (-0 in 1's complement arithmetic), the check succeeds.*

La seconda funzione riportata si occupa di calcolare la somma in complemento a uno del messaggio i cui byte sono considerati raggruppati in parole di 16 bit (punto 1). La funzione non somma i riporti a mano a mano che si generano ma li somma in un unico ciclo alla fine, possibilità specificata nella stessa RFC di cui sopra:

> *Depending upon the machine, it may be more efficient to defer adding end-around carries until the main summation loop is finished.*

Il primo passaggio del punto 2 (the checksum field is cleared) è già stato svolto sui dati in input, anche se nel nostro caso specifico, per l'allineamento delle parole la somma in complemento a uno non cambierebbe anche omettendo il campo checksum (si salterebbe la somma di una parola composta da soli zeri). Il checksum a questo punto non è altro che il complemento a uno di questo risultato, di ciò si occupa il return della funzione checksum (& 0xffff necessario per come python gestisce i numeri interi).

Ottenuto il checksum possiamo comporre il messaggio definitivo nello stesso modo di prima, sostituendo allo 0 il codice appena calcolato.
A fini didattici ed espositivi nella funzione compose_echo_message si verifica anche la corretta computazione del checksum (punto 3):

```
check = ones_complement_sum(message)
if check == 0xffff:
    return message
else:
    raise ValueError("Error while computing checksum.")
```

Questo conclude la composizione di un pacchetto di richiesta Echo minimale, che in condizioni normali viene ritornato dalla funzione.

Lettura di una risposta ICMP

Se tutto va bene, dovremmo ricevere una risposta dagli host a cui abbiamo mandato una richiesta Echo, quindi è necessario poter analizzare il contenuto di un messaggio ICMP, di questo si occupa la seguente funzione:

```
def read_icmp_message(data):
    if ones_complement_sum(data) == 0xffff:
        try:
            message_type, code, cs, identifier, sequence_number =
struct.unpack("!BBHHH", data)
            if message_type == 0 and code == 0:
                return identifier, sequence_number
        except Exception:
            raise ValueError("data contains a number of bytes != 8")
    return None, None
```

questa prende in input una sequenza di byte che si assume essere un messaggio ICMP, verifica l'assenza di errori mediante checksum e tenta di scomporla mediante struct.unpack(). Notare che questa funzione alza un'eccezione in caso i byte non siano del numero giusto, condizione che non dovrebbe verificarsi in quanto gli host dovrebbero rispedire nella Echo reply solo i dati che sono stati inviati nella richiesta, a una simile situazione corrisponde quindi l'impossibilità di interpretare la risposta e la funzione ritorna None, None. Lo stesso risultato si avrà se i byte non costituiscono una risposta echo (campo type o code diversi da 0) o se il checksum fallisce.
In caso i byte in input corrispondano a una risposta echo, la funzione restituisce i campi identifier e sequence number che permetteranno di accertarsi che la risposta sia inerente alla chiamata attuale e non sia, per esempio la risposta ad una richiesta eseguita in precedenza che ha tardato a raggiungere il nostro terminale.

Richiesta di Echo ad un host

Avendo tutti gli ingredienti necessari per scrivere e leggere i messaggi ICMP necessari all'implementazione delle funzionalità, non ci resta che inviare una richiesta e attendere una risposta, di questo si occupa la seguente funzione:

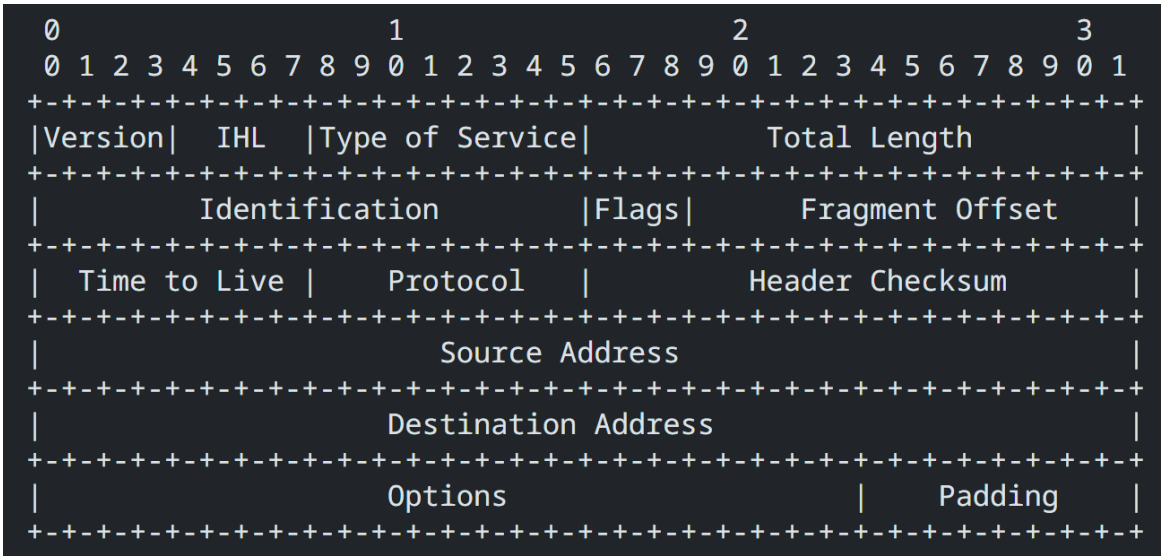
```
ddef ping_once(address, identifier, sequence_number):
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)
        s.settimeout(TIMEOUT)  # timeout for recvfrom()
    except socket.error:
        return SOCKET_ERROR_CODE  # error in creating the socket
    try:
        sent = compose_echo_message(identifier, sequence_number)
    except ValueError:
        return ICMP_ERROR_CODE  # error in composing the message

    try:
        s.sendto(sent, (address, 0))
        send_time = time.time
        answer, recv_addr = s.recvfrom(RECEIVE_MAX_SIZE)
        receive_time = time.time() - send_time
        ip_version = answer[0] >> 4
        protocol = answer[9]
        if recv_addr[0] == address and ip_version == 4 and protocol == 1:
            header_length = (answer[0] & 0x0f) * 4
            message = answer[header_length:]
            try:
                answer_identifier, answer_sequence_number =
read_icmp_message(message)
                if answer_identifier == identifier and answer_sequence_number ==
sequence_number:
                    return int(receive_time * 1000)
            except ValueError:
                pass
        return WRONG_ANSWER_CODE
    except socket.timeout:
        return TIMEOUT_CODE
    except socket.error:
        return SOCKET_ERROR_CODE
```

La funzione prende in input l'indirizzo a cui inviare la richiesta e i campi identifier e sequence number da inserire nei rispettivi campi, e restituisce il tempo trascorso tra l'invio della richiesta e la ricezione della risposta in millisecondi oppure un codice di errore con valore minore di 0.
I messaggi ICMP sfruttano il livello di rete per essere inviati, dobbiamo quindi inizializzare una socket di tipo RAW che ci permette di specificare il numero di protocollo da inserire nell'header IP come terzo argomento, socket.IPPROTO_ICMP (che corrisponde a 1). Impostiamo anche il timeout dell'attesa di risposta al valore di TIMEOUT secondi in modo da rendere l'applicazione più reattiva.
Procediamo poi a comporre il messaggio ICMP da inviare come visto precedentemente e lo inviamo all'indirizzo specificato come argomento. Salviamo anche l'orario di invio della richiesta per poter calcolare il valore del "ping" e ci mettiamo in ascolto sulla socket per un'eventuale risposta. Se avviene un timeout o un errore nell'invio del massaggio, la relativa eccezione è catturata e l'opportuno codice di errore è restituito dalla funzione.

Se abbiamo ricevuto una risposta, salviamo l'orario di ricezione per restituire, in caso la risposta fosse corretta, il tempo intercorso e passiamo all'analisi del messaggio ricevuto.

Analisi della risposta ricevuta



Formato di un pacchetto IPv4 (fonte RFC 791)

Perché una risposta (sotto forma di pacchetto IP) sia correttamente interpretata dal nostro semplice script devono essere vere le seguenti condizioni:

- l'indirizzo IPv4 del mittente deve corrispondere al destinatario della nostra richiesta
- il campo Version deve contenere il valore 4 (lo script non supporta IPv6)
- il campo Protocol deve contenere il valore 1, relativo al protocollo ICMP

Se una di queste non è vera o se il payload del pacchetto non corrisponde alla risposta ICMP che ci aspettiamo la funzione restituisce il codice di errore WRONG_ANSWER_CODE.

```
receive_time = time.time() - send_time
ip_version = answer[0] >> 4
protocol = answer[9]
if recv_addr[0] == address and ip_version == 4 and protocol == 1:
    header_length = (answer[0] & 0x0f) * 4  # ihl is saved in 32-bit words
    message = answer[header_length:]
    try:
        answer_identifier, answer_sequence_number = read_icmp_message(message)
        if answer_identifier == identifier and answer_sequence_number ==
sequence_number:
            return int(receive_time * 1000)
    except ValueError:
        pass
```

Questa sezione della funzione si occupa di estrarre la versione (primi quattro bit ovvero i quattro bit più significativi del primo byte), la lunghezza dell'header (secondi quattro bit ovvero i quattro bit meno significativi del primo byte della risposta) che è espressa in numero di parole da 32 bit, e viene quindi moltiplicata per 4 per ottenere la lunghezza in byte.
Procediamo poi a estrarre il numero di protocollo, contenuto nel 10° byte dell'header (indice 9) e se corrisponde a quello cercato estraiamo il payload del pacchetto (che inizia dalla byte di numero header_length + 1, con indice header_length) e lo passiamo alla funzione incaricata di verificarne il contenuto.
Solo nel caso in cui la risposta sia conforme a ciò che lo script si aspetta (vedere sopra), e i campi identifier e sequence number corrispondano alla richiesta inviata, la funzione ritorna finalmente il tempo intercorso tra richiesta e risposta in millesimi di secondo (time.time() restituisce un double che esprime il tempo in secondi quindi è necessario moltiplicare per 1000 e un cast a int per eliminare le cifre decimali indesiderate).

Il main

Abbiamo ormai discusso le parti interessanti dell'applicazione, ciò che rimane sono alcune funzioni di aiuto nella formattazione: get_desc, pinfo, perror, pheader e pstatus. La prima si occupa di tradurre i codici di errore in una descrizione testuale da mostrare all'utente le altre funzioni formattano diversi tipi di dati in modo uniforme per essere stampati (vedere i commenti nello script per i dettagli).
Come già accennato, l'esecuzione dello script si articola in due fasi che corrispondono a due loop nella funzione main, uno per raccogliere gli input da parte dell'utente e uno per eseguire richieste di Echo indefinitamente.

```
def main():
    sequence_number = random.randint(0, 0xffff + 1)
    identifier = (os.getpid() & 0xffff)
    names = {}
    status = {}
    print("Insert addresses separated by newlines or an empty line to ping: \n")
    while True:
        host = input()
        if len(host):
            try:
                address = socket.gethostbyname(host)
                if address != host:
                    names[address] = host
                    pinfo("Hostname " + host + " resolved to " + address + " and
correctly added.")
                else:
                    names[address] = None
                    pinfo("Address correctly added.")
            except socket.gaierror:
                perror(
                    "Address " + host +
                    " is not a valid IPv4 address and DNS search returned
nothing: address not added")
        else:
            if(len(names.keys())):
                pinfo("Addresses accepted.")
                break
            else:
                perror("No address has been added yet.")

    while True:
        for address in names:
            status[address] = ping_once(address, identifier, sequence_number)
        pheader()
        for address in names:
            pstatus(address, status[address], names[address])
        time.sleep(WAIT_TIME)
```

Dopo aver inizializzato il numero di sequenza a un numero casuale e l'identificatore ai primi quattro byte del process id, si procede a richiedere l'input dall'utente come descritto nella prima sezione. Ogni indirizzo inserito è passato attraverso socket.gethostbyname() che restituisce l'indirizzo stesso se si tratta di un indirizzo IPv4 o esegue una ricerca DNS in caso contrario. L'indirizzo così determinato è inserito come chiave nel dizionario names, il valore sarà l'eventuale indirizzo mnemonico specificato dall'utente (per visualizzarlo in seguito) o None se l'indirizzo era già IPv4. L'unico caso in cui un indirizzo non viene aggiunto è se non si tratta di un indirizzo IPv4 valido e la ricerca DNS non produce risultati: in questa situazione la clausola except si occupa di catturare l'eccezione lanciata da socket.gethostbyname() e avvertire l'utente che nessun indirizzo è stato aggiunto.

L’inserimento di una riga vuota conclude la fase di raccolta dati (se si ha inserito almeno un indirizzo altrimenti produce un errore e la raccolta continua) e comincia il monitoraggio degli host corrispondenti agli indirizzi inseriti.

Il secondo loop del main non termina mai se non viene interrotto il processo e a intervalli di WAIT_TIME secondi aggiorna lo stato di ogni indirizzo salvandolo in un dizionario per poi stampare tutti gli stati in una volta sola in un secondo momento. La funzione pstatus() si occupa di interpretare il valore contenuto nel dizionario (che potrebbe essere maggiore di 0 nel caso di ping riuscito o minore di 0 nel caso di un errore) e stampare una stringa che lo comunichi all’utente, ma per questi dettagli e per qualsiasi altro dubbio vi rimando al codice sorgente e i commenti presenti.

Grazie per l’attenzione.

P.S: il codice e i relativi commenti sono stati scritti in lingua inglese per adeguarsi alla prassi comune.

P.P.S: per creare un illusione di “refresh” dell’output del terminale lo script stampa diverse righe vuote (in quanto system(“clr”) non funziona su tutti gli ide), questo però può portare alcuni ide (per esempio IDLE) a effettuare uno “squeeze” delle righe di output che risultano quindi solo parzialmente visibili senza espandere ciò che è stato nascosto.