Verificare il Green Pass europeo

Da una settimana, il Green Pass (che dimostra l'immunità o comunque il non contagio dal Covid-19) è obbligatorio per entrare in buona parte degli edifici in tutta Italia. Molti gestori di attività che si svolgono al chiuso sono preoccupati di non riuscire a controllare i Green Pass del pubblico. Ma è davvero così difficile? In realtà, no. L'EU Digital COVID Certificate è infatti un certificato digitale, le cui specifiche sono pubblicamente disponibili proprio per consentire a chiunque di verificare la validità di uno di questi QR code.

Esistono delle app per smartphone che svolgono questa operazione, ma richiederebbero comunque un operatore che scansioni i vari QR code degli avventori di un locale, e non sempre questo è praticabile. Esiste, però, la possibilità di automatizzare tutto il procedimento, gestendo l'accesso all'edificio tramite un meccanismo come un tornello, o una porta azionabile elettronicamente, e un semplice programma in Python che possiamo scrivere in breve tempo. Utilizzando un computer dotato di pin GPIO come il RaspberryPi è possibile realizzare un sistema completamente automatico: si può usare una webcam per riconoscere il QRcode, un lettore di smartcard per confrontare i dati con quelli della Tessera Sanitaria, e un relay per attivare il tornello (o la porta) soltanto nel caso in cui il Green Pass risulti valido. Il ricorso alla Tessera Sanitaria è fondamentale perché altrimenti un utente potrebbe presentarsi alla porta d'ingresso con un QRcode appartenente a un'altra persona, magari fotografato e condiviso tra tanti utenti. La Tessera Sanitaria è invece una sola per ogni cittadino, quindi permette di identificare automaticamente le persone e assicurarsi che ogni accesso sia legittimo. Naturalmente si potrebbero utilizzare altri

meccanismi, come la CIE o la Firma Digitale, ma la Tessera Sanitaria è l'unico ID digitale a disposizione di tutti i cittadini italiani. Il progetto che proponiamo si basa su un RaspberryPi2 o superiore, con RasperryOS Buster, una webcam USB, un lettore di smartcard USB, un altoparlante passivo con jack audio, e un eventuale modulo relay per far scattare l'apertura della porta.

Table of Contents

- Installare i requisiti
- Riconoscere il QRCode
- Le funzioni di servizio
- Decodificare il Green Pass
- Leggere la Tessera Sanitaria
- Verificare se il certificato è valido
- Catturare il QRcode dalla webcam
- Mettere tutto assieme

Installare i requisiti

I requisiti di questo software sono parecchi, ma possiamo installarli con una serie di comandi. Da notare che ci serve lo script https://github.com/panzi/verify-ehc, e quindi dovremo anche installare tutti i suoi requisiti. Possiamo farlo con questa serie di comandi:

In poche parole, prima di tutto installiamo le varie librerie necessarie per leggere le smartcard, poi quelle necessarie per prelevare immagini dalla webcam e manipolare le immagini (utilizzeremo OpenCV e PIL). Poi serviranno anche le librerie per gestire i pin GPIO del Raspberry, in modo da attivare un relay, e quelle per decodificare i QRCode. Infine, la libreria per emettere suoni (beepy) e i vari requisiti di Verify EHC.

Riconoscere il QRCode

Per prima cosa, scriveremo il codice che ci serve per riconoscere il QRcode, cioè per ottenere il testo (che poi tradurremo in JSON). Qui è necessario un piccolo hack: al momento, la versione di Debian disponibile come RaspberryOs è Buster. Purtroppo questa versione è ormai molto vecchia, quindi non è possibile avere le ultime versioni dei pacchetti. E la libreria qrtool è disponibile solo per Python2 (si trova su apt come **python-qrtools**). Esistono ovviamente divese altre librerie, ma questa è quella che abbiamo notato essere più efficiente e semplice da utilizzare. Quindi per ora metteremo le sue poche righe di codice in uno script a parte, che verrà interpretato da Python2 invece che da Python3: in futuro, usando la nuova versione di Debian, sarà possibile usare questa libreria nella versione per Python3. Il codice è questo:

Il codice è estremamente semplice: lo script si aspetta in argomento il percorso di un file contenente l'immagine di un QRcode da interpretare. Con questo file si può costruire un oggetto QR, decodificabile con la funzione **decode()**. A questo punto la variabile **.data** contiene il testo che è stato riconosciuto nel QRCode, che possiamo restituire sullo standard output. E che andremo a leggere dal programma principale.

Il Green Pass è un testo

firmato con le informazioni del paziente memorizzate in un JSON

Le funzioni di servizio

Ora iniziamo a scrivere lo script principale, quello che si occuperà di svolgere tutto il processo di verifica sia del Green Pass che della Tessera Sanitaria. Cominciamo dall'importazione delle librerie:

Librerie extra sono sostanzialmente quelle a cui accennavamo per lo script di installazione delle dipendenze, poi servono alcune librerie standard di Python per la gestione del JSON, dei sottoprocessi e dell'orario.

Continuiamo definendo alcuni oggetti che saranno utili per tutto lo script, e che quindi avranno valore globale. Per esempio, il percorso in cui trovare il file di configurazione, oppure quello in cui scrivere i log. Poi proviamo a importare le librerie per gestire i pin GPIO del RaspberryPi: saranno necessarie per attivare il relay, e quindi aprire automaticamente la porta o il tornello nel caso il Green Pass risulti valido. In realtà possiamo anche utilizzare lo script su un computer diverso dal Raspberry, e in quel caso non riusciremmo a importare le librerie dei GPIO. In questo caso impostiamo la variabile **rpi** a False, così sapremo che non ci troviamo su un Raspberry. Creiamo un dizionario vuoto per memorizzare la configurazione, e poi l'oggetto **reader**: questo rappresenterà il nostro punto di accesso al lettore di smartcard. Siccome dovrebbe essere possibile procedere anche senza il lettore, perché l'utente potrebbe decidere di usare solo la webcam per il riconoscimento del QR code e poi lasciare a un operatore l'identificazione della persona, se non riusciamo a trovare il lettore di Smart Card catturiamo

l’eccezione e andiamo avanti comunque.

Definiamo due funzioni “di servizio”, non fondamentali ma utili per definire due procedure e non preoccuparsene più. La prima si occupa di leggere il file di configurazione, che sarà nel formato JSON, e memorizzare il contenuto in un dizionario. La seconda è quella che apre la porta facendo scattare il relay: ci serve il numero del pin GPIO da attivare, ma vogliamo anche assicurarci di essere davvero su un Raspberry, perché altrimenti non ci sono i GPIO e non dobbiamo fare nulla.

Decodificare il Green Pass

Iniziamo con le cose “serie”: lo script **verify-ehc.py** si occupa di decodificare la stringa del Green Pass (che è un testo codificato in Base45).

Lo chiamiamo direttamente con os.system, scrivendo l’output in un file. Poi leggiamo il file, memorizzandolo come testo in una variabile. Utilizziamo os.system perché il modulo subprocess ha difficoltà a leggere tutte le righe, dal momento che lo script scrive l’output a intervalli non regolari.

L’output è diviso su più righe, e in realtà a noi interessano solo alcune. Nello specifico, ci interessa la riga **Is Expired** che, se presente, indica che il certificato era valido, ma ora è scaduto. E poi la riga **Signature Valid**, che è presente solo se la firma del certificato risulta corretta: questa indica che il certificato è stato generato da uno dei ministeri della salute dell’Unione Europea, e quindi possiamo considerarlo non contraffatto. Infine, cerchiamo la riga **Payload**, perché dopo di essa viene riportato l’intero contenuto del Green Pass vero e proprio, con i dati personali della persona. Questo payload è in formato JSON, quindi possiamo tranquillamente caricarlo

in un dizionario, assicurandoci di prendere il testo e rimuovere gli invii a capo per evitare che il modulo json di Python possa avere difficoltà a interpretarlo.

Leggere la Tessera Sanitaria

Purtroppo non esiste una documentazione pratica per l'utilizzo delle informazioni presenti nella Tessera Sanitaria italiana, solo delle specifiche tecniche. Noi ci siamo basati sul lavoro di decodifica fatto alcuni anni fa da MMXForge.

I dati su una tessera sanitaria sono memorizzati in un particolare filesystem, ed è possibile selezionare i file inviando una serie di comandi binari (che codifichiamo in esadecimale per leggiblità). La funzione per la lettura dei dati personali dalla tessera sanitaria deve quindi iniziare stabilendo una connessione con la smartcard e poi utilizzando quella connessione per inviare una serie di comandi.

La Tessera Sanitaria italiana contiene un microchip leggibile con un comune lettore di smartcard

Otteniamo come risposta una tupla di tre oggetti: il primo rappresenta i dati restituiti dalla smartcard, gli altri due eventuali codici per identificare errori. Se tutto va bene, sw1 e sw2 dovrebbero sempre contenere i valori 0x90 e 0x00 rispettivamente. Nel nostro caso non c'è bisogno di

verificarli, perché siamo solo interessati a estrarre i dati dal file corretto, qualsiasi cosa vada storta indica che la tessera inserita non era corretta e possiamo considerare nulla l'identificazione.

A questo punto, la variabile data contiene tutti i dati dell'utente, ma come byte. Dobbiamo convertirla in stringa e poi estrarre i singoli dati. I dati sono codificati in modo abbastanza semplice: i primi due caratteri contengono il numero di byte che costituiscono il successivo dato, così sappiamo sempre esattamente quando leggere. Quindi dobbiamo leggere i primi due caratteri, trasformarli in un numero intero, e leggere quel numero di byte per estrapolare il codice dell'emettitore della tessera. Poi leggiamo i due caratteri successivi per conoscere il numero di byte da leggere per avere il cognome. Segue il nome dell'intestatario della tessera.

Con la stessa logica possiamo continuare a leggere i dati personali dell'utente. Sono, in sequenza, sesso, statura, codice fiscale, cittadinanza, comune di nascita e stato di nascita (nel caso la persona non sia nata in Italia). Memorizziamo tutti questi dati in un dizionario, così sarà più facile accedere a quello che ci interessa.

Verificare se il certificato è valido

Iniziamo ora la funzione che ci permetterà di stabilire se il Green Pass sia valido.

I due oggetti che dobbiamo ricevere in argomento sono il dizionario con i dati del green pass e quello con i dati della tessera sanitaria. Possiamo considerare il green pass immediatamente non valido se dai suoi stessi dati risulta che

sia scaduto (expired) o se la sua firma non risulti correttamente apposta da uno dei ministeri della salute europei (in questo caso **signature_valid** sarebbe **False**).

Se è stata fornita una Tessera Sanitaria valida, possiamo confrontare i suoi dati con quelli del Green Pass. Dobbiamo solo fare una piccola conversione sulla data di nascita, perché nella TS è memorizzata nel formato GG/MM/YYYY, mentre nel GP è memorizzata come AAAA-MM-GG. Poi possiamo confrontare data di nascita, nome, e cognome: li confrontiamo in minuscolo, per evitare problemi con eventuali lettere mauscole non corrispondenti.

Se non è stata fornita una tessera sanitaria, per esempio perché la persona non è un cittadino italiano, e la configurazione consente comunque all'operatore di verificare l'identità della persona, facciamo apparire un semplice prompt per chiedere proprio all'operatore se il Green Pass appartenga davvero alla persona che si è presentata. Se l'operatore preme i tasti **y** oppure **s**, il Green Pass è considerato legittimo, ma segnaliamo comunque che non era stata fornita una tessera sanitaria. Così nell'eventuale log viene indicato che l'identificazione è stata manuale.

Per leggere un QR code basta una comune webcam, anche non FullHD

Catturare il QRcode dalla webcam

Per catturare il QRcode creiamo una funzione che utilizza OpenCV, così è facile scattare foto in tempo reale dalla webcam.

L'immagine verrà inserita nella variabile **img**.

Ora utilizziamo OpenCV per scrivere l'immagine su un file temporaneo (sempre lo stesso, tanto possiamo gestire un solo ingresso alla volta). Poi cerchiamo di tradurre questa immagine nel testo del GreenPass usando lo script qrcodereader. Non utilizziamo direttamente a funzione di lettura del QR code di OpenCV perché non funziona bene con webcam a bassa definizione. Se abbiamo ottenuto qualcosa, lo scriviamo nella variabile **data**.

Se è disponibile una sessione di Xorg, il server grafico di GNU-Linux, mostriamo una finestra con l'anteprima della foto scattata dalla webcam, così l'utente può capire se ha allineato correttamente il QR code. Chiaramente non possiamo farlo quando non c'è uno schermo. La funzione fa un ciclo continuo finché non viene riconosciuto un QRcode valido.

Mettere tutto assieme

Nella routine principale del programma possiamo riunire le varie funzioni che abbiamo scritto seguendo il filo logico della verifica del Green Pass: lettura del QRcode, lettura della tessera, confronto dei dati, responso all'utente sotto forma di segnale audio, apertura della porta, e eventuale messaggio sullo schermo.

Nella routine principale del programma prima di tutto leggiamo la configurazione dall'apposito file. Poi attiviamo un ciclo continuo, che svolgerà le varie operazioni in sequenza: prima di tutto si riproduce un suono, per segnalare che siamo pronti a leggere un nuovo QRcode. Poi procediamo a avviare la funzione per la lettura delle immagini dalla webcam: quando questa avrà identificato e decodificato un QR code, potremo procedere a verificarne il contenuto. Fatto questo, andiamo a leggere l'eventuale Tessera Sanitaria presente nel lettore (se la tessera non è stata inserita, il dizionario risultante sarà vuoto).

Ora abbiamo tutto quello che potrebbe servirci, quindi possiamo procedere alla verifica delle credenziali. Come abbiamo visto, la funzione **isCertValid** ci restituisce una tupla di due oggetti. Il primo è un semplice booleano, chiamato **val**, che sarà True se il Green Pass è valido e corrispondente alla Tessera Sanitaria e False negli altri casi. Mentre **err** è una stringa che contiene l'eventuale codice di errore ottenuto. Se il Green Pass è valido non soltanto lo segnaliamo con un suono, così è subito palese se l'accesso sia consentito oppure no, ma inneschiamo anche l'apertura della porta o del tornello con la funzione **open_door**.

Per finire, gestiamo il caso in cui la configurazione preveda di loggare i dati, per esempio per identificare. Ovviamente questa è una eventualità che richiede una certa cautela, perché si tratta di memorizzare dati privati sensibili delle persone, quindi non è detto che qualcuno voglia attivarla. Se il log è attivo, quindi, generiamo una riga di log costituita dal timestamp, lo stato della validità del green pass (**OK** oppure **ERROR**), l'eventuale codice fiscale (che però è una stringa vuota se non è stata fornita una Tessera Sanitaria), e l'eventuale messaggio di errore.
Prima di ripetere il ciclo attendiamo un secondo, per dare all'utente il tempo di togliere la propria tessera sanitaria e il QRcode dai lettori. Poi siamo pronti per un altro ciclo,

verificando le credenziali di un’altro avventore..

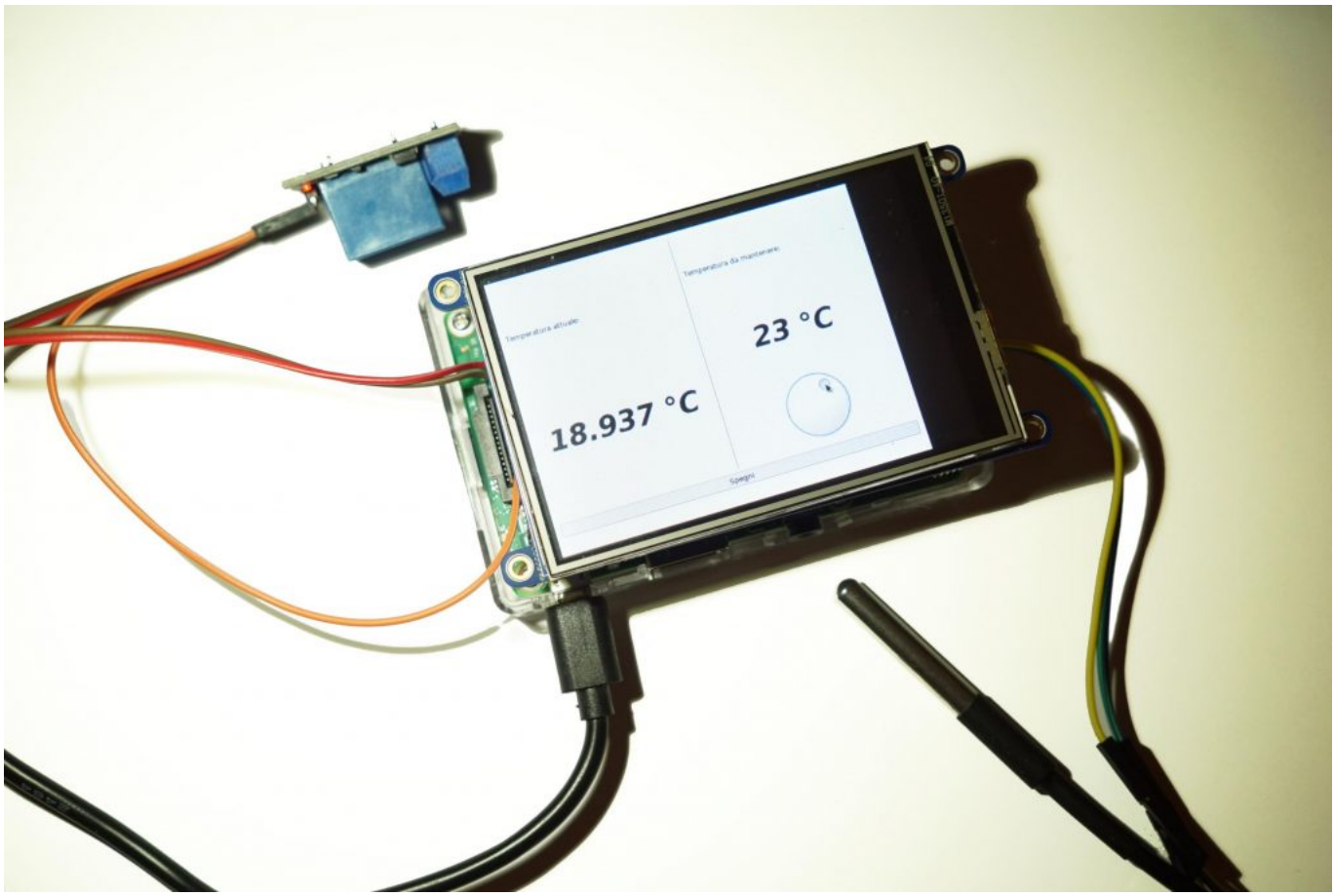
Il codice sorgente
Potete trovare il codice sorgente di questo strumento su GitHub:
https://github.com/zorbaproject/greenpass-turnstile
Il repository integra già la dipendenza https://github.com/panzi/verify-ehc, lo script che esegue la decodifica del Green Pass.

Un termostato touchscreen con RaspberryPi

I RaspberryPi sono una ottima piattaforma su cui costruire oggetti basati sull’idea dell’Internet of Things: dispositivi per uso più o meno domestico che siano connessi a reti locali o ad internet e possano essere facilmente controllati da altri dispositivi. Il vantaggio di un RaspberryPi rispetto a un Arduino è che è più potente, e permette quindi una maggiore libertà. E non solo in termini di collegamento al web, ma anche per quanto riguarda le interfacce grafiche. È un dettaglio del quale non si parla molto, perché tutti danno per scontato che un Raspberry venga usato solo come server, controllato da altri dispositivi con una interfaccia web, e non collegato a uno schermo. Ma non è sempre così. Per esempio, possiamo utilizzare un RaspberryPi per realizzare un termostato moderno. In questo caso non abbiamo più di tanto bisogno di accedervi dallo smartphone: sarebbe utile, ma di sicuro non è l’utilizzo principale che si fa di un termostato. Basterebbe avere uno schermo touchscreen, con una bella

grafica, che si possa fissare al muro per regolare la temperatura. Il punto su cui molti ideatori dell'IoT perdono parte del proprio pubblico è il mantenere le cose "con i piedi per terra". La gente, infatti, è abituata a regolare la temperatura della propria casa da un semplice pannello attaccato al muro, ed è questo che vuole. Magari un po' più futuristico e gradevole alla vista, ma comunque non troppo diverso da quello a cui si è già abituati. Non tutto ha bisogno di essere connesso al web, soprattutto considerando che è un pericolo: se il nostro termostato è raggiungibile da internet, prima o poi un pirata russo si divertirà ad abbassare la temperatura di casa nostra fino a farci raggiungere un congelamento siberiano. La strada più semplice e immediata, a volte, è la migliore. La domanda a questo punto è: come si realizza una interfaccia grafica per Raspberry? Esistono diverse opzioni, ovviamente, ma quella che abbiamo scelto è PySide2. Si tratta della versione Python delle librerie grafiche Qt5, le più comuni librerie grafiche cross platform. È l'opzione migliore semplicemente perché sono disponibili per la maggioranza delle piattaforme e dei sistemi operativi esistenti, quindi i progetti che realizziamo con queste librerie possono funzionare anche su dispositivi differenti dal Raspberry, e ciascuno può adattare il codice alle propri esigenze con poche modifiche.

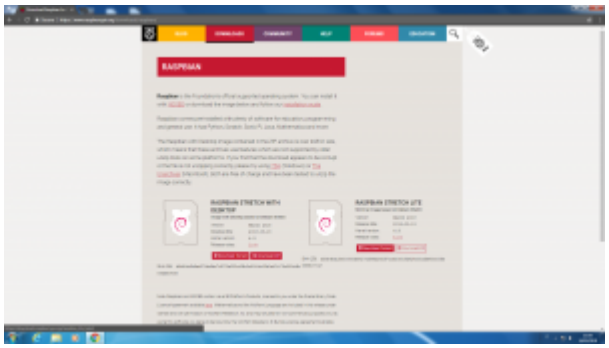
Il nostro dispositivo di riferimento sarà un RaspberryPi 3 B+, dal costo di 50 euro, con uno schermo TFT di Adafruit da 3.5 pollici. Come sistema operativo, si può utilizzare la versione di **Raspbian Buster con PySide2** preinstallato realizzata per Codice Sorgente (https://www.codice-sorgente.it/raspbian-buster-pyside2-lxqt/).



Ma le procedure presentate possono funzionare anche per il Raspberry Pi Zero W, con lo stesso schermo, bisogna solo aspettare che venga pubblicata la versione Buster di Raspbian per questo dispositivo (prevista tra qualche mese).

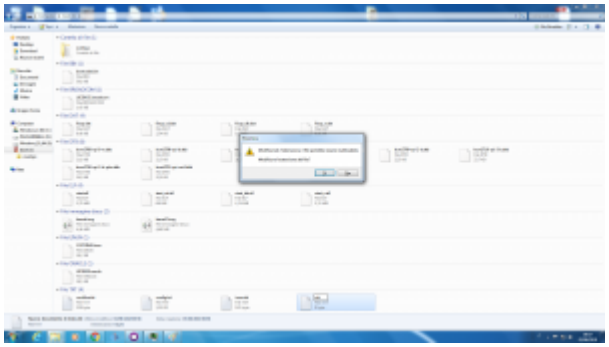
Preparare la scheda sd

Per prima cosa si scarica l'immagine di Raspbian: al momento si può recuperare dal sito ufficiale la versione Raspbian Stretch per qualsiasi tipo di Raspberry. Se invece avete un Raspberry Pi 3 o 3B+ (o anche un Raspberry Pi 2) potete utilizzare **la versione che ho realizzato personalmente di Raspbian Buster, con le librerie Qt già installate: https://www.codice-sorgente.it/raspbian-buster-pyside2-lxqt/**. Questa è la strada consigliabile, visto che Raspbian Stretch è molto datato e non è possibile installare le librerie Qt più recenti, su cui si basa il progetto di questo articolo.



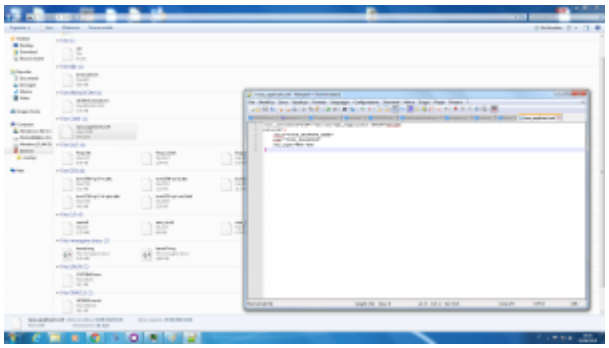
Ottenuta l'immagine del sistema operativo, la si scrive su una scheda microSD con il programma Win32Disk Imager:

Dopo il termine della scrittura, si può inserire nella scheda SD il file **ssh**, un semplice file vuoto senza estensione (quindi ssh.txt non funzionerebbe):

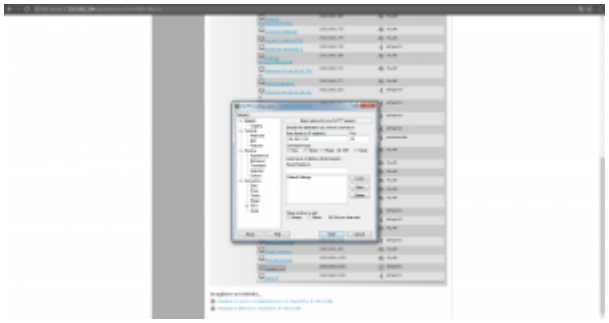


Sempre nella scheda SD si può creare il file di testo **wpa_supplicant.conf**, inserendo un testo del tipo

Bisogna però assicurarsi che il file abbia il formato Unix per il fine riga (LF, invece del CRLF di Windows). Lo si può stabilire con Notepad++ o Kate, degli editor di testo decisamente più avanzati di Blocco Note:



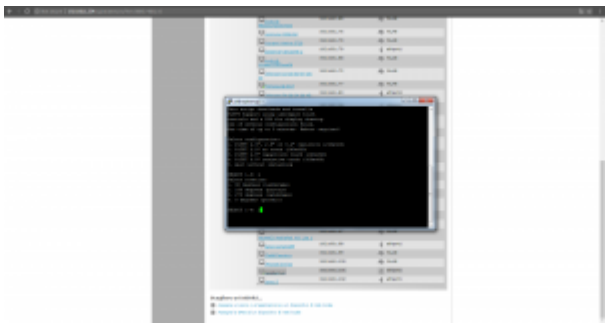
Dopo avere collegato il proprio Raspberry all'alimentazione (e eventualmente alla rete, se si sta usando l'ethernet), si può accedere al terminale remoto conoscendo il suo indirizzo IP. L'indirizzo Il programma che simula un terminale remoto SSH su Windows si chiama Putty, basta indicare l'indirizzo e premere **Open**. Quando viene richiesto il login e la password, basta indicare rispettivamente **pi** e **raspberry**. Per chi non lo sapesse, la password non compare mentre le si scrive, come da tradizione dei sistemi Unix.



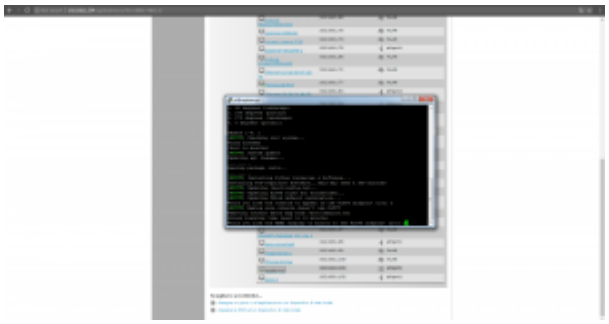
Se il login è corretto si accede al terminale del Raspberry:

Per configurare il TFT di Adafruit bisogna dare i seguenti comandi:

Quando la configurazione inizia, vengono richieste due informazioni: una sul modello di schermo che si sta usando (nell'esempio il numero 4, quello da 3.5 pollici), e una sull'orientamento con cui è fissato sul Raspberry (tipicamente la 1, classico formato orizzontale).



Alla fine dell'installazione dei vari software necessari, viene anche richiesto come usare lo schermo: alla prima richiesta, quella sulla console, conviene rispondere **n**, mentre alla seconda (quella sul mirroring HDMI) si deve rispondere **y**. In questo modo sullo schermo touch verrà presentata la stessa immagine che si può vedere dall'uscita HDMI.



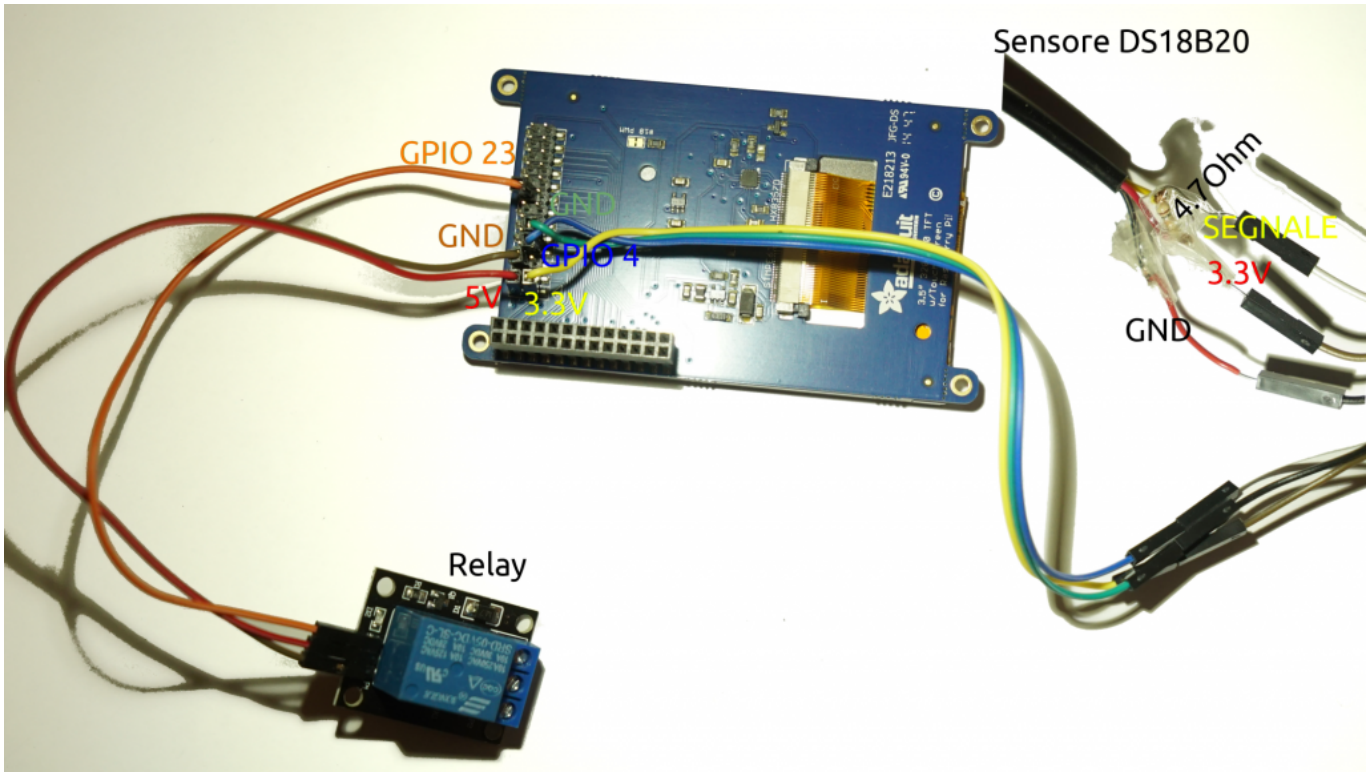
Un appunto (temporaneo) su Raspbian Buster
Se state usando l'immagine che abbiamo fornito all'inizio dell'articolo, basata su Raspbian Stretch, vi sarete accorti che lo script di Adafruit non funziona. Questo perché al momento il pacchetto tslib, necessario per lo script, non è disponibile per Buster, visto che si tratta di una versione non stabile. Per aggirare il problema, si può utilizzare

questa versione modificata dello script: https://codice-sorgente.it/cgit/termostato-raspberry.git/tree/adafruit-pitft.sh. Basta scaricarlo con il comando

Le altre istruzioni non cambiano. In futuro, quando Raspbian Buster verrà rilasciato ufficialmente, non sarà necessario usare questa versione modificata e si potrà fare riferimento all'originale.

I collegamenti del relè e del termometro

Quando si monta lo schermo sul Rasberry, i vari pin della scheda vengono coperti. È tuttavia possibile continuare a usarli perché lo schermo ce li ha doppi. Il sensore di temperatura e il relay potranno quindi essere collegati direttamente ai pin maschi che si trovano sullo schermo, seguendo lo stesso ordine dei pin sul Raspberry.



Ovviamente, un Raspberry offre molti pin, per collegare sensori e dispositivi, ma bisogna stare attenti a non usare lo stesso pin per due cose diverse. Per esempio, se stiamo usando il TFT da 3.5 pollici, possiamo verificare sul sito pinout che i contatti che usa sono il **18,24,25,7,8,9,10,11**. Rimane quindi perfettamente libero sul lato da 5Volt il pin 23, mentre sul lato a 3Volt è libero il pin 4. Il primo verrà usato come segnale di output per il relay, mentre il secondo come segnale di input del termometro. Il relay va collegato anche al pin 5V e GND, mentre il sensore va collegato al pin 3V e al GND.

La libreria per accedere ai pin GPIO dovrebbe già essere presente su Raspbian, mentre per installare quella relativa al sensore di temperatura si può dare il comando

Bisogna anche abilitare il modulo nel sistema operativo:

Dopo il riavvio diventa possibile utilizzare la libreria w1 per l'accesso al sensore di temperatura.

Un codice di test dal terminale

Possiamo verificare il funzionamento del relay e del termometro con un programma molto semplice da eseguire sul terminale:

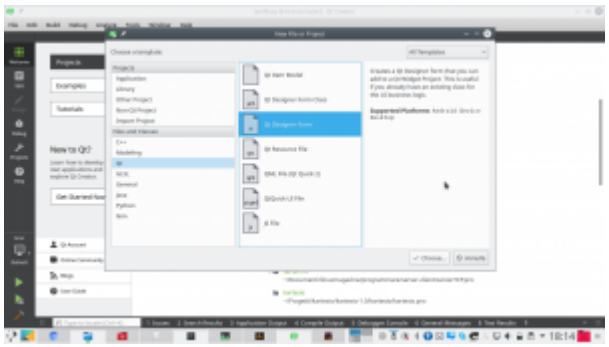
Per provare il programma basta dare i seguenti comandi:

Che cosa fa questo programma? Prima di tutto si assicura che siano caricati i moduli necessari per accedere ai pin GPIO e al sensore di temperatura. Poi esegue un ciclo infinito accendendo il relay, leggendo la temperatura attuale, aspettando 5 secondi, spegnendo il relay, leggendo ancora la

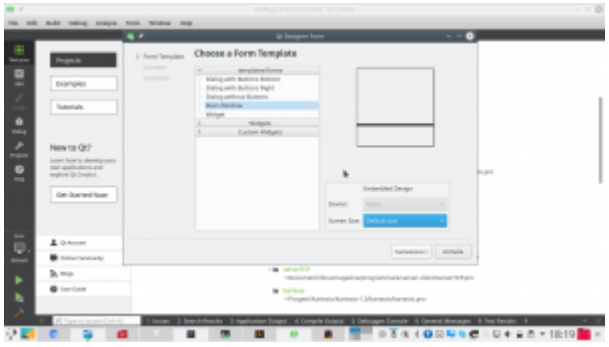
temperatura, e attendendo altri 5 secondi. Per terminare il programma, basta premere **Ctrl+C**.

L’interfaccia grafica

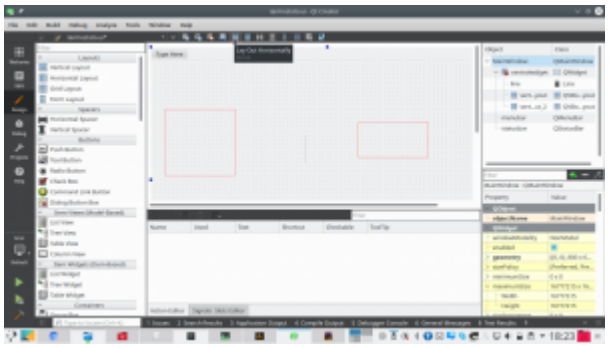
L’interfaccia grafica del nostro termostato è disponibile, assieme al resto del codice, nel repository Git: https://codice-sorgente.it/cgit/termostato-raspberry.git/tree/termostato.ui. Tuttavia, per chi volesse disegnarla da se, ecco i passi fondamentali. Prima di tutto si apre l’IDE QtCreator, creando un nuovo file di tipo Qt Designer Form.



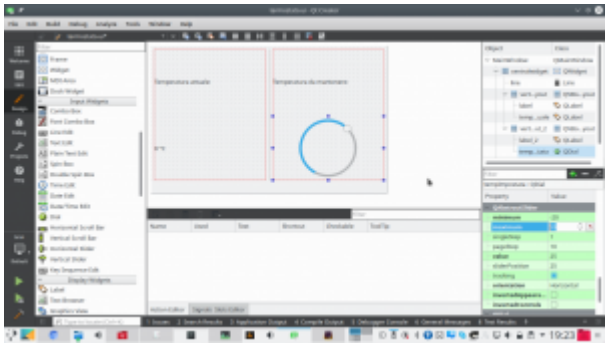
Il template da utilizzare è Main Window, perché quella che andiamo a realizzare è la classica finestra principale di un programma. Per il resto, la procedura guidata chiede solo dove salvare il file che verrà creato.



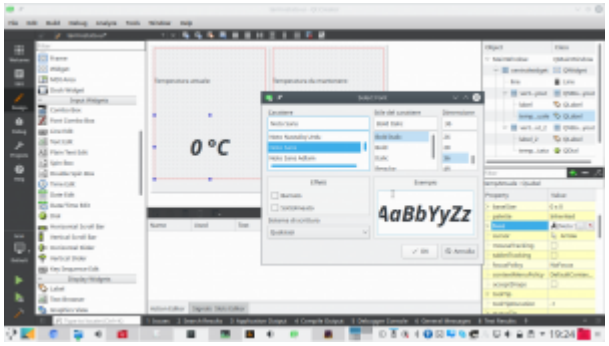
Quando si disegna una finestra, la prima cosa da fare è dividere il contenuto in layout. Considerando il nostro progetto, possiamo aggiungere due oggetti **Vertical Layout**, affiancandoli. Poi, con la barra degli strumenti in alto, impostiamo il form con un **Layout Horizontally**. I due layout verticali sono ora dei contenitori in cui possiamo cominciare ad aggiungere oggetti.



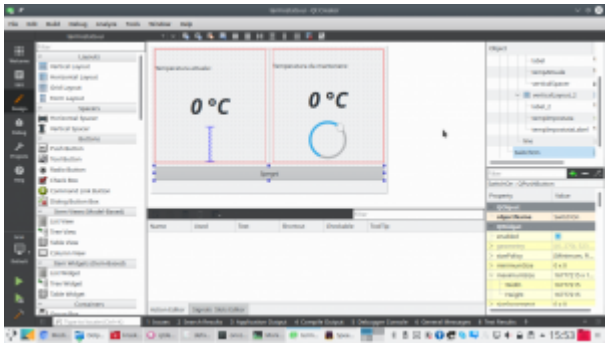
Inseriamo un paio di **label**: in particolare, una dovrà essere chiamata **tempAttuale**, e un'altra **tempImpostataLabel**. Queste etichette conterranno rispettivamente il valore della temperatura attualmente registrata dal termometro e quello che si è deciso di raggiungere (cioè la temperatura da termostatare). Nello stesso layout di **tempImpostataLabel** inseriamo un oggetto **Dial**, che chiamiamo **tempImpostata**. Si tratta di una classica rotella, proprio come quelle normalmente presenti sui termostati fisici. Tra le proprietà di questa dial, indichiamo i valori che desideriamo come minimo (**minimum**), massimo (**maximum**), e predefinito **(value).** Poi possiamo cliccare col tasto destro sulla **menubar** del form e scegliere di rimuoverla, così da lasciare spazio ai vari oggetti dell'interfaccia, tanto non useremo i menù.



Cliccando col tasto destro sui vari oggetti, è possibile personalizzarne l'aspetto. Per esempio, può essere una buona idea dare alle etichette che conterranno le due temperature una formattazione del testo facilmente riconoscibile, con una dimensione del carattere molto grande.



Infine, si può aggiungere, dove si preferisce ma sempre usando dei layout, un **Push Button** chiamato **SwitchOn**. Questo pulsante servirà per spegnere il termostato manualmente, in modo che il relay venga disattivato a prescindere dalla temperatura. Questa è una buona idea, perché se si sta via da casa per molto tempo non ha senso che il termostato scatti per tenere la casa riscaldata sprecando energia. Il pulsante che abbiamo inserito deve avere le proprietà **checkable** e **checked** attivate (basta cliccare sulla spunta), in modo da farlo funzionare non come un pulsante ma come un interruttore, che mantiene il proprio stato acceso/spento.



L'interfaccia è ora pronta, tutti i componenti fondamentali sono presenti. Per il resto, è sempre possibile personalizzarla aggiungendo altri oggetti o ridisegnando i layout.

Il codice del termostato

Cominciamo ora a scrivere il codice Python che permetterà il funzionamento del termostato:

All'inizio del file si inserisce la classica shebang, il

cancelletto con il punto esclamativo, per indicare l'interprete da usare per avviare automaticamente questo script Python3 (che non va quindi confuso col vecchio Python2). Si indica anche la codifica del file come utf-8, utile se si vogliono usare lettere accentate nei testi. Poi, si importano le varie librerie che abbiamo già visto essere utili per accedere ai pin GPIO del Raspberry, al termometro, e alle funzioni di sistema per misurare il tempo.

Ora dobbiamo aggiungere le librerie PySide2, in particolare quella per la creazione di una applicazione (**QApplication**). In teoria potremmo farlo con una sola riga di codice. In realtà, è preferibile usare questo sistema di blocchi try-except, perché lo utilizziamo per installare automaticamente la libreria usando il sistema di pacchetti **pip**. In poche parole, prima di tutto si prova (try) a importare la libreria **QApplication**. Se non è possibile (except), significa che la libreria non è installata, quindi si utilizza l'apposita funzione di pip per installare automaticamente la libreria. E siccome ci vorrà del tempo è bene far apparire un messaggio che avvisi l'utente di aspettare. È necessario usare due formule differenti perché, anche se al momento nella versione 3.6 di Python il primo codice funziona, in futuro sarà necessario utilizzare la seconda forma. Visto che la transizione potrebbe richiedere ancora qualche anno, con sistemi che usano ancora la versione 3.6 e altri con la 3.7, è bene avere entrambe le opzioni. Il vero vantaggio di questo approccio è che se si sta realizzando un programma realizzato con alcune librerie non è necessario preoccuparsi di distribuirle col programma: basta lasciare che sia Python stesso a installarla al primo avvio del programma. L'installazione è comunque possibile solo per le piattaforme supportate dai rilasci ufficiali su pip della libreria, e nel caso di Raspbian conviene sempre installare le librerie a parte.

Se l’importazione della libreria QApplication è andata a buon fine, significa che PySide2 è installato correttamente. Quindi possiamo importare tutte le altre librerie di PySide che ci torneranno utili nel programma.

Definiamo una variabile globale da usare per tutte le prossime classi: questa variabile, chiamata **toSleep**, contiene l’intervallo (in secondi) ogni cui controllare la temperatura.

Per prima cosa creiamo una classe di tipo QThread. Ovviamente, i programmi Python vengono sempre eseguiti in un unico thread, quindi se il processore è impegnato a svolgere una serie di calcoli e operazioni in background (come il raggiungimento della temperatura) non può occuparsi anche dell’interfaccia grafica. Il risultato è che l’interfaccia risulterebbe bloccata, un effetto sgradevole per l’utente e poco pratico. La soluzione consiste nel dividere le operazioni in più thread: il thread principale sarà sempre assegnato all’interfaccia grafica, e creeremo dei thread a parte che possano occuparsi delle altre operazioni. Creare dei thread in Python non è semplicissimo, ma per fortuna usando i QThread diventa molto facile. Il QThread che stiamo preparando ora si chiama TurnOn, e servirà per accendere e spegnere il relay in modo da raggiungere la temperatura impostata. All’inizio della classe si definisce un **segnale** con valore booleano (**True** oppure **False**) chiamato **TempReached**. Potremo emettere questo segnale per far sapere al thread principale (quello dell’interfaccia grafica) che la temperatura fissata è stata raggiunta e il termostato ha fatto il suo lavoro. Nella funzione che costruisce il thread (cioè **__init__**), ci sono le istruzioni necessarie per accedere ai pin GPIO del relay e e al sensore. Il pin cui è collegato il relay viene memorizzato nella variabile **self.relayPin**, mentre il sensore sarà raggiungibile tramite l’oggetto **self.sensor**. La funzione prende in argomento **w**, un oggetto che rappresenta la finestra del programma (che passeremo al thread dell’interfaccia

grafica stessa). In questo modo le funzione del thread potranno accedere in tempo reale all'interfaccia e interagire.

La funzione **run** viene eseguita automaticamente quando avviamo il thread: potremmo inserire le varie operazioni al suo interno, ma per tenere il codice pulito ci limitiamo a usarla per chiamare a sua volta la funzione **reachTemp**. Sarà questa a svolgere le operazioni vere e proprie. Prima di vederla, definiamo un'altra funzione: **readTemp**. Questa funzione non fa altro che leggere la temperatura attuale, scriverla sul terminale per debug, e restituirla. Ci tornerà utile per leggere facilmente la temperatura e controllare se è stata raggiunta quella prefissata. Nella funzione **reachTemp** per prima cosa si dichiara di avere bisogno della variabile globale **toSleep**. Poi si inserisce un blocco try-except: in questo modo, se per qualche motivo l'attivazione del relay dovesse fallire, verrà lanciato il segnale **TempReached** con valore **False** e nell'interfaccia grafica potremo tenerne conto per avvisare l'utente che qualcosa è andato storto. Se invece va tutto bene, si inizia un ciclo while, che rimarrà attivo finché il pulsante presente nell'interfaccia grafica (che abbiamo chiamato SwitchOn) è premuto (cioè nello stato **checked**). Ciò significa che appena l'utente cliccherà sul pulsante per farlo passare allo stato "spento" (cioè "non checked") il ciclo si fermerà. Nel ciclo, si controlla se la temperatura attuale (ottenuta grazie alla funzione **readTemp**) sia inferiore alla temperatura impostata per il termostato. In caso positivo bisogna assicurarsi che il relay sia acceso, quindi il suo pin si imposta con valore **HIGH**. E poi si attende il tempo prefissato prima di fare un altro ciclo e quindi controllare di nuovo la temperatura. Se, invece, la temperatura attuale è maggiore di quella da raggiungere, vuol dire che possiamo spegnere il relay impostando il suo pin al valore **LOW**, ed emettere il segnale **TempReached** col valore **True**, visto che è andato tutto bene. Abbiamo quindi un ciclo che si ripete, per esempio, ogni 10 secondi finché viene

raggiunta la temperatura impostata, e a quel punto si interrompe.

Poi creiamo un'altra classe di tipo QThread, stavolta chiamata **ShowTemp**. In questa classe non facciamo altro che leggere la temperatura attuale, dal sensore, e inserirla nella **label** che abbiamo dedicato proprio a presentare questo valore all'utente. Avremmo potuto inserire questa funzione nello stesso QThread già creato per regolare la temperatura, visto che poi lo possiamo lanciare più volte e con argomenti diversi. Quindi avremmo potuto usare un QThread unico con le due funzioni da attivare separatamente. Tuttavia, quando si fanno operazioni diverse è una buona idea tenere il codice pulito e creare QThread separati. Esattamente come per il QThread precedente, la funzione di costruzione della classe (la solita **__init__**) richiede come argomento **w**, l'oggetto che rappresenta la finestra dell'interfaccia grafica. Viene anche creato l'oggetto **sensor**, per accedere al sensore di temperatura. Anche in questa classe inseriamo la funzione **readTemp**, uguale a quella del QThread precedente, e per semplificare stavolta scriviamo le istruzioni direttamente nella funzione **run**. Anche in questo caso si accede alla variabile globale **toSleep**. Il codice che svolge davvero le operazioni è un semplice ciclo infinito (**while True**) che imposta un nuovo valore come testo della label presente nell'interfaccia grafica (cioè **self.w**). L'etichetta in questione si chiama **tempAttuale**, e possiamo assegnarle un testo usando la funzione **setText**. Il testo viene creato prendendo la temperatura (ottenuta dalla funzione **readTemp** e traducendo il numero in una stringa di testo) e aggiungendo il simbolo dei gradi Celsius. Poi, si aspetta il tempo preventivato prima di procedere a ripetere il ciclo.

Il risultato che otterremo, con l'interfaccia grafica interattiva

La classe **MainWidow**, di tipo QMainWindow, conterrà il codice necessario a far apparire e funzionare la nostra interfaccia grafica.

La prima cosa da fare, nella funzione che costruisce la classe (la solita **__init__**) è caricare, in lettura (**QFile.ReadOnly**) il file che contiene l'interfaccia grafica stessa. Il file in questione si trova nella stessa cartella dello script attuale, e si chiama **termostato.ui**, quindi possiamo scoprire il suo percorso completo estraendo dal nome dello script (**sys.argv[0]**) la cartella in cui si trova (con la funzione **os.path.dirname**) e risalire al percorso assoluto con la funzione **os.path.abspath**. L'interfaccia grafica viene interpretata con la libreria **QUiLoader**, e memorizzata nell'oggetto **self.w**. Da questo momento sarà quindi possibile accedere ai vari componenti dell'interfaccia grafica usando questo oggetto. Affinché l'interfaccia grafica venga utilizzata per la finestra che stiamo costruendo, bisogna impostare l'oggetto **self.w** come **CentralWidget**. Possiamo impostare un titolo per la finestra con la funzione **self.setWindowTitle**, tipica di ogni QMainWindow. Ora possiamo cominciare a rendere interattiva l'interfaccia grafica: per farlo dobbiamo collegare i segnali degli oggetti dell'interfaccia alle funzioni che si occuperanno di gestirli. Per esempio, dobbiamo collegare il segnale **clicked** del pulsante **SwitchOn** a una funzione che chiameremo **self.StopThis**. Lo facciamo usando la funzione **connect** del segnale di questo

pulsante. La scrittura è molto semplice: si tratta semplicemente di un sistema a scatole cinesi. Per esempio, anche quando colleghiamo il movimento della rotella (la QDial per impostare la temperatura) alla funzione **setTempImp** non facciamo altro che prendere l'oggetto che rappresenta l'interfaccia grafica, cioè **self.w**, e puntare sulla QDial al suo interno, che avevamo chiamato **tempImpostata**. All'interno di questo oggetto, andiamo a recuperare il segnale **valueChanged**, che viene emesso dalla QDial stessa nel momento in cui l'utente modifica il suo valore spostando la rotella, e per questo segnale chiamiamo la funzione **connect**. Alla funzione bisogna soltanto assegnare il nome (completo di **self.**, non dimentichiamo che è un membro della classe Python che stiamo scrivendo) della funzione che dovrà essere chiamata. Va indicato solo il nome, senza le parentesi, quindi si scrive **self.setTempImp** e non **self.setTempImp()**.

Sempre nella funzione **__init__** si provvede a dare i comandi necessari per attivare i moduli di sistema che forniscono il controllo del sensore e dei pin GPIO. Poi impostiamo la variabile **self.alreadyOn** a False: si tratta di un semplice flag che useremo per capire se il relay sia già stato attivato, o se sia necessario attivarlo, quindi ovviamente all'avvio del programma è False perché il relay è ancora spento. Ora si può accedere alla QDial e impostare manualmente il suo valore iniziale, con la funzione **setValue**, per esempio a 25 gradi. Poi va chiamata manualmente la funzione **setTempImp**, con lo stesso valore in argomento, per essere sicuri che il programma controlli se sia necessario accendere o spegnere il relay per raggiungere la temperatura in questione. La variabile **self.stoponreached** verrà utilizzata soltanto per decidere se disattivare il termostato una volta raggiunta la temperatura: di norma non è necessario, anzi, si preferisce che il termostato rimanga vigile per riaccendere il relay qualora la temperatura dovesse scendere nuovamente. Ma per funzioni di test o casi di abitazioni con un isolamento

davvero buono può avere senso impostare questa variabile a True. L'ultima cosa da fare prima di concludere la funzione di costruzione dell'interfaccia grafica è creare il thread che si occupa di leggere la temepratura attuale dal sensore e scriverla nell'apposita label. Basta creare un oggetto di tipo **ShowTemp**, perché questo è il nome che abbiamo scelto per la classe di questo QThread, indicando l'oggetto **self.w** come argomento, così le funzioni del thread potranno accedere all'interfaccia grafica di questa finestra. Il thread viene avviato usando la funzione **start**. È importante non confondersi: quando si scrive la classe del QThread il codice va messo nella funzione **run**, ma quando lo si avvia si chiama la funzione **start**, perché così vengono eseguiti una serie di controlli prima dell'effettivo inizio delle operazioni.

Definiamo due funzioni: una è **reached**, e l'altra **itIsOff**. La funzione **reached** verrà chiamata automaticamente quando la temperatura impostata per il termostato viene raggiunta. A questo punto possiamo decidere cosa fare: se la variabile **stoponreached** è impostata a True, chiameremo la funzione **itIsOff**, così da disattivare il termostato. In caso contrario, non è necessario fare nulla, ma volendo si potrebbe modificare l'aspetto dell'interfaccia grafica (per esempio colorando di verde l'etichetta con la temperatura) per segnalare che la temperatura è stata raggiunta. La funzione **itIsOff**, come abbiamo già suggerito, si occupa di disattivare il termostato. Per farlo, imposta come False il pulsante presente nell'interfaccia grafica: siccome si comporta come un interruttore, se il suo stato **checked** è falso il pulsante è disattivato. E, come avevamo visto nella classe del thread TurnOn, il ciclo che si occupa di controllare se sia necessario tenere il relay rimane attivo solo se il pulsante è **checked**. Poi viene chiamata la funzione StopThis, che si occupa di modificare il pulsante (che da "Spegni" deve diventare "Accendi").

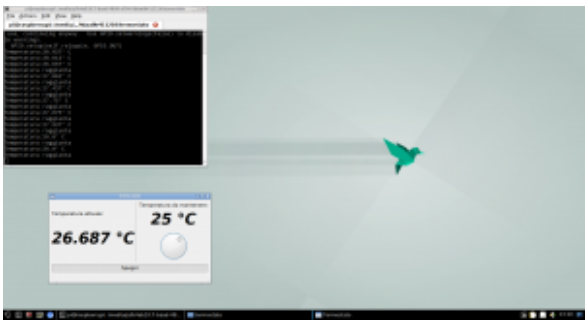
La funzione **dostuff** è quella che si occupa effettivamente di creare il thread dedicato al controllo del relay. Prima di tutto, si controlla che il thread non sia già stato avviato, usando il flag **alreadyOn** che avevamo creato all'inizio del codice di questa classe. Se il thread non è già attivo, lo si crea passandogli l'oggetto **self.w** così da permettere al thread l'accesso all'interfaccia grafica. Poi colleghiamo il segnale **TempReached**, che avevamo creato per il thread **TurnOn**, alla funzione **self.reached**. Si collega anche il segnale **finished**, dello stesso thread, alla funzione **itItOff**, così se per un motivo o l'altro il thread dovesse terminare la funzione adeguerebbe lo stato del pulsante presente nell'interfaccia grafica. Alla fine, si avvia il thread e si imposta il flag **alreadyOn** come True.

La funzione **StopThis** controlla lo stato attuale del pulsante: se è attivato, il suo testo deve essere impostato a "Spegni", e bisogna ovviamente chiamare la funzione **dostuff** in modo che venga lanciato il thread, se necessario. Viceversa, se il pulsante è disattivato, il suo testo deve essere "Accendi", così l'utente capirà che premendo il pulsante può attivare il sistema, e il flag **alreadyOn** va impostato a False, per segnalare che al momento il thread è disattivato (ricordiamo che se il pulsante è disattivato, anche il thread **TurnOn**, inserito in questa finestra col nome **self.myThread**, si disattiva automaticamente).

L'ultima funzione che inseriamo nella classe della finestra è **setTempImp**, ed è la funzione che abbiamo collegato al segnale **valueChanged** della QDial. Quando l'utente sposta la rotella, verrà chiamata questa funzione. Si può notare che questa funzione è leggermente diversa dalle altre che abbiamo scritto finora per reagire ai segnali dell'interfaccia grafica: ha un argomento, chiamato **arg1**. Questo perché il segnale **valueChanged** offre alla funzione chiamata il valore attuale della QDial. Nel nostro caso, quindi, **arg1** contiene la

temperatura che l’utente vuole impostare, quindi possiamo direttamente inserirla nell’etichetta **tempImpostataLabel**, che abbiamo creato nell’interfaccia grafica proprio per presentare la temperatura selezionata. Poi, per sicurezza, chiamiamo la funzione **dostuff** in modo da attivare il thread che fa funzionare il relay se è necessario.

Terminate le classi, possiamo scrivere il codice principale del programma. Per prima cosa, il programma crea una **QApplication**, necessaria per le librerie Qt. Poi possiamo creare una istanza della finestra principale, memorizzandola nell’oggetto **w**. Ora possiamo impostare alcune caratteristiche della finestra. Per esempio, la dimensione: se stiamo usando lo schermo PiTFT3.5, la risoluzione ideale è 640×480: naturalmente, si possono modificare i parametri sulla base delle proprie esigenze. Infine, si fa apparire la finestra con la funzione **show**. E quando questa verrà chiusa (cosa che nel nostro caso non dovrebbe succedere, ma è un buona idea tenere in considerazione l’ipotesi), si chiude normalmente il programma, fornendo alla riga di comando il risultato dell’esecuzione dell’applicazione (quindi eventuali codici d’errore se qualcosa dovesse non funzionare correttamente).



L’applicazione eseguita dal desktop del Raspberry, per provare il suo funzionamento

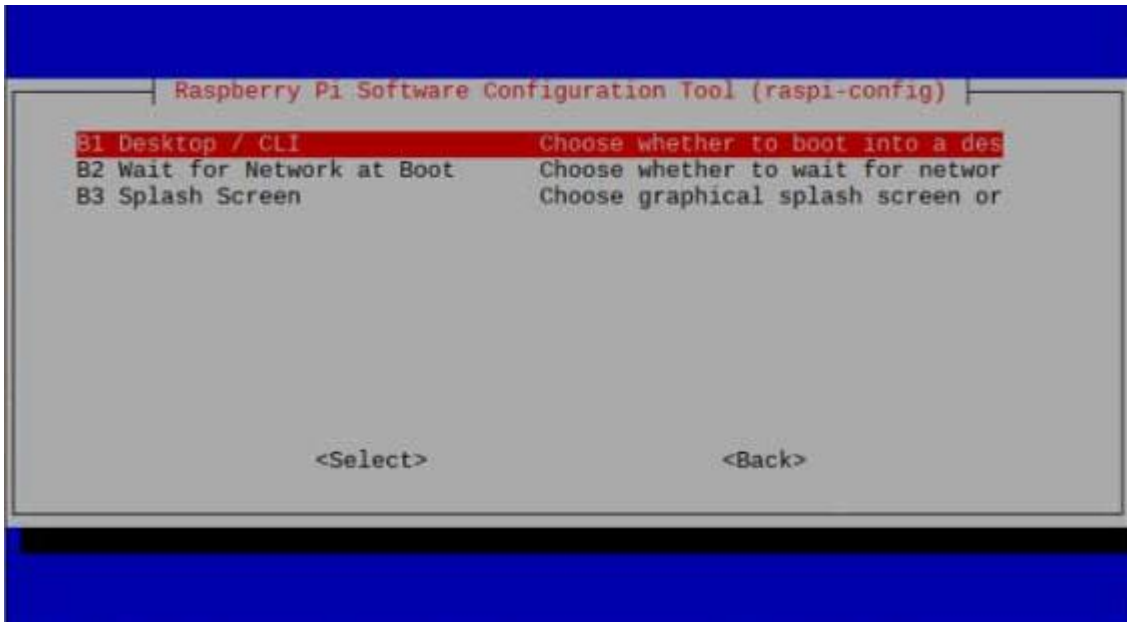
Avvio automatico

Siccome la nostra applicazione è pensata per apparire sullo schermo all'avvio del sistema, dobbiamo preparare un piccolo script per automatizzare la sua installazione:

Prenderemo come riferimento l'utente pi, che è sempre disponibile su Raspbian. Con queste prime righe di codice creiamo un servizio di sistema che esegue il login automatico per l'utente pi sul terminale tty1. Così, all'avvio del sistema non sarà necessario digitare la password, si avrà subito un terminale funzionante.

Si modificano due file di configurazione: con la modifica a xinit aggiungiamo il comando cat all'avvio del server grafico: questo permette di tenerlo in stallo e evitare eventuali procedure automatiche. La modifica a bashrc ci permette di fare un controllo appena viene aperto un terminale per l'utente **pi**. Se il nome del terminale è tty1 (su GNU/Linux ci sono diversi terminali disponibili) lanciamo sia lo script di avvio del nostro programma, sia il server grafico Xorg. Aver fatto questo controllo è importante, perché così il programma termostato verrà avviato automaticamente solo sul terminale tty1, quello che ha l'autologin, e non su tutti gli altri.

Infine, si scrive lo script di avvio del programma termostato: inizialmente, lo script attende un secondo, in modo da essere sicuro che il server grafico sia pronto a funzionare. Poi, lancia Python3 con il nostro programma.



C’è un dettaglio: con l’immagine di Raspbian Buster fornita all’inizio dell’articolo l’autologin potrebbe non funzionare correttamente, e questo perché Raspbian usa carica una immagine di splash per il boot che impedisce il corretto avvio del server grafico per come lo abbiamo configurato. La soluzione è disabilitare il boot con splash grafica, visto che comunque non ci serve, usando il comando **sudo raspi-config** nella sezione **Boot**, selezionando l’opzione **Splash Screen** e impostandola come disabilitata.

Il codice completo
Potete trovare il codice completo nel repository git https://codice-sorgente.it/cgit/termostato-raspberry.git/tree/
Per scaricarlo potete dare il comando

da un terminale GNU/Linux come quello del Raspberry, oppure usare le varie interfacce grafiche di Git disponibili. O, anche, scaricare i file singolarmente dalla pagina https://codice-sorgente.it/cgit/termostato-raspberry.git/plain/. Nel repository è presente un README con i comandi da eseguire per installare correttamente il programma sulla

versione di **Raspbian Buster** che proponiamo. C'è da dire che il codice che abbiamo presentato è pensato per un uso accademico, non professionale: una applicazione per un termostato può essere più semplice, il codice è prolisso in diversi punti per facilitare la comprensione a chi non sia pratico delle librerie Qt.

Facebook Scraping: scaricare tutti i post delle pagine Facebook

Da quando sono esplosi gli scandali relativi all'uso dei dati dei social network, come quello di Cambridge Analytica, Facebook ha messo in piedi un sistema di controllo delle applicazioni. In poche parole, ora qualsiasi applicazione voglia accedere a ogni tipo di dato degli utenti deve prima superare un controllo in cui, in teoria, Facebook dovrebbe verificare che l'app non usi i dati in modo contrario alle norme di condotta previste dal social network.
Il problema è che, al momento, il sistema non funziona bene: ovviamente è appena partito, e si presume che in futuro verrà migliorato, ma ha una serie di difetti fondamentali che saranno difficili da correggere a meno che Facebook non sia pronto a spendere davvero molte risorse finanziarie. Già adesso, infatti, ci sono delle persone incaricate di analizzare ogni app che viene sottoposta alla verifica, ma hanno migliaia di app da seguire e non hanno quindi il tempo di entrare nei dettagli. Soprattutto, nessuno controlla il codice sorgente delle app, quindi non c'è davvero una garanzia che questo controllo serva a impedire utilizzi impropri dei dati del social network. Allo stesso tempo, inoltre, i

meccanismi di autorizzazione delle app offerti al momento sono insufficienti a coprire alcune delle app più legittime: parliamo di quelle che collezionano dati pubblici per realizzare statistiche (per pubblica amministrazione, università, e ricerca). Al momento è molto difficile farsi approvare una app di tipo desktop, l'opzione non è prevista e gli script non sono visti di buon occhio. Tuttavia, una università, un istituto di statistica, o una redazione giornalistica hanno in genere bisogno di accedere soltanto a dati come i vari post delle pagine dei personaggi famosi (per analizzare il loro linguaggio e capire come cambi la comunicazione in caso di eventi importanti, o controllare la veridicità delle affermazioni). Un esempio semplice è un team di ricercatori che voglia controllare sistematicamente la percentuale di verità dei post di un politico, il cosiddetto fact checking. In questi casi si vuole accedere soltanto a dati che sono già pubblici, e che quindi possono essere raccolti senza violare la privacy di nessuno.

Uno script con Python

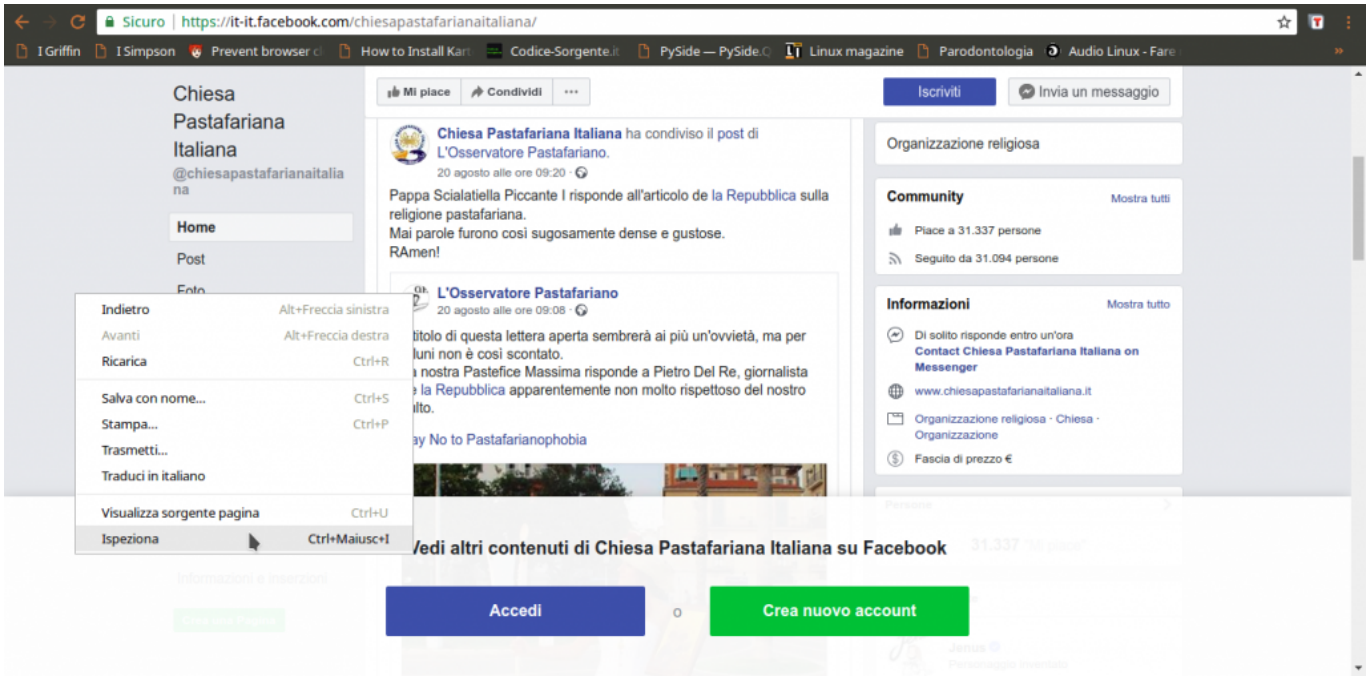
Per il nostro script utilizziamo Python3: nonostante sul sito ufficiale venga ancora presentato il vecchio Python2 per questioni di retrocompatibilità, la versione 3 presenta delle differenze importanti che rendono alcune funzioni incompatibili. Per essere sicuri di poter utilizzare lo script che presentiamo (alla fine dell'articolo trovare un link all'intero codice sorgente), bisogna installare sul proprio pc almeno la versione 3.6 di Python.

Abbiamo quindi pensato di realizzare uno script in Python che si occupi di eseguire lo scraping delle pagine Facebook pubbliche. Lo scraping è, per chi non lo sapesse, un insieme di tecniche di estrazione di informazioni da pagine web e

altri documenti, in modo automatico, ripulendole da ciò che non serve. Un ricercatore universitario potrebbe scaricarsi i post di una pagina Facebook aprendola col browser e scorrendola verso il basso fino a visualizzarli tutti, selezionando il testo di ciascuno e copiandoselo. Ma sarebbe una operazione lunghissima e noiosa. Analizzando il codice delle pagine HTML che Facebook fornisce, invece, possiamo automatizzare l'estrazione dei testi (o delle immagini, se volete scaricarvi i meme delle vostre pagine preferite, basta modificare lo script per cercare i tag **img** invece dei tag **p**). E ovviamente lo script che realizziamo non richiede alcun accesso alle API o approvazione da parte di Facebook, perché di fatto faremo la stessa cosa che fa ogni utente che vuole guardare una pagina Facebook, permettendoci di bypassare tutte le verifiche che Facebook ha messo in piedi per le app.
Non dobbiamo, infatti, dimenticare un concetto fondamentale: se una informazione è disponibile, c'è sempre un modo non ufficiale per ottenerla. Quando carichiamo una pagina Facebook in Google Chrome, l'interfaccia realizzata con HTML e Javascript carica solo un certo numero di post. Quando "scrolliamo" la pagina, scendendo verso il basso, deve esserci una qualche funzione che si accorge che stiamo scendendo e quindi richiede al server un certo numero di nuovi post da visualizzare. È abbastanza ovvio che questa richiesta debba essere fatta, dalla pagina HTML+JS, con una richiesta HTTP (usando il meccanismo Ajax, quindi la funzione xmlhttprequest di Javascript). Se ne deduce che da qualche parte all'interno della pagina ci deve essere un riferimento a un'altra pagina che fornisce un elenco di post da visualizzare. L'accesso a questi dati da parte di script automatici invece che dai normali browser web è una cosa che, a prescindere dai suoi sforzi, Facebook non potrà mai impedire.

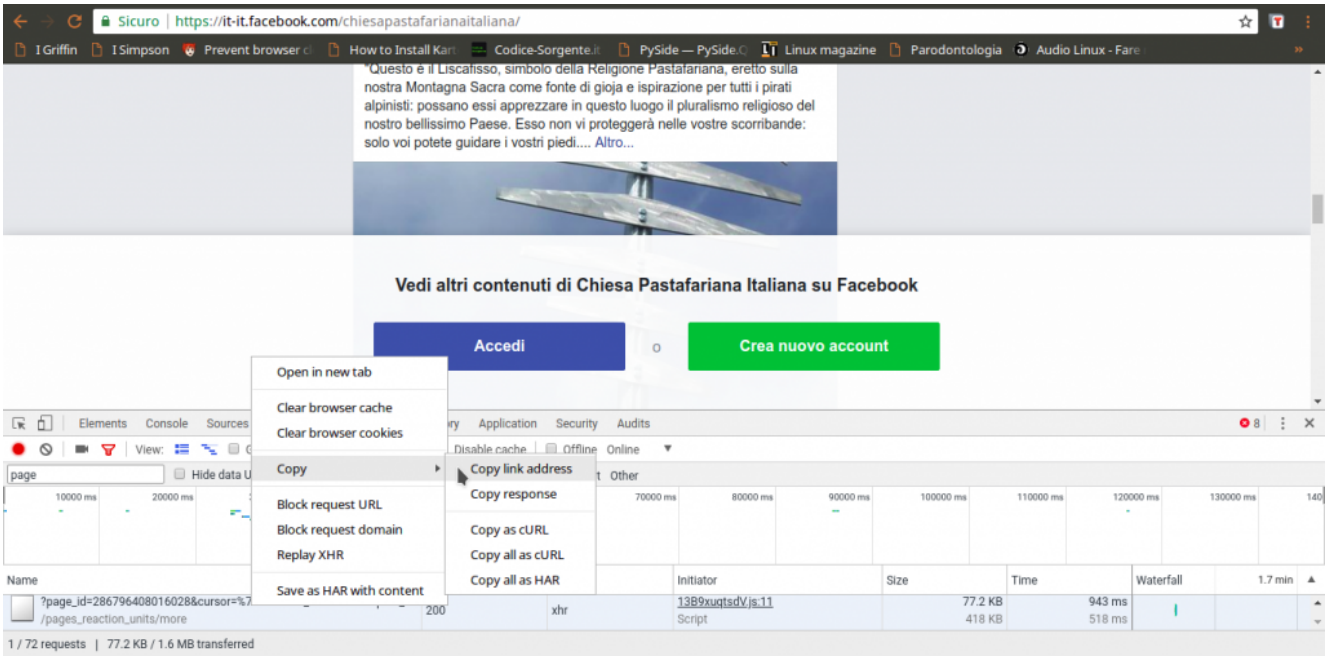
Scoprire l'indirizzo per ottenere i post

Per prima cosa dobbiamo scoprire come funzioni Facebook, cioè come vengano recuperati i vari post. In poche parole, bisogna conoscere il proprio obiettivo. Siccome si tratta di un sito web, la cosa migliore da fare è aprire una pagina Facebook (per esempio https://it-it.facebook.com/chiesapastafarianaitaliana/) col proprio browser, come Google Chrome, cliccando poi col tasto destro sulla pagina per scegliere la voce **Ispeziona**.



Tra le varie schede disponibili, quella che serve per capire cosa succeda è quella chiamata **Network**: si occupa di presentare in tempo reale le varie richieste HTTP che vengono fatte. Siccome è ovvio che la pagina di Facebook, per caricare altri post, abbia bisogno di fare una richiesta al server di Facebook per ottenerli, è anche ovvio che apparirà qui. Tutto quello che dobbiamo fare a questo punto è scorrere la pagina verso il basso, per obbligarla a caricare altri post: nel pannello vedremo comparire una richiesta a una pagina chiamata **page_reaction_units**. Sembra proprio che abbiamo

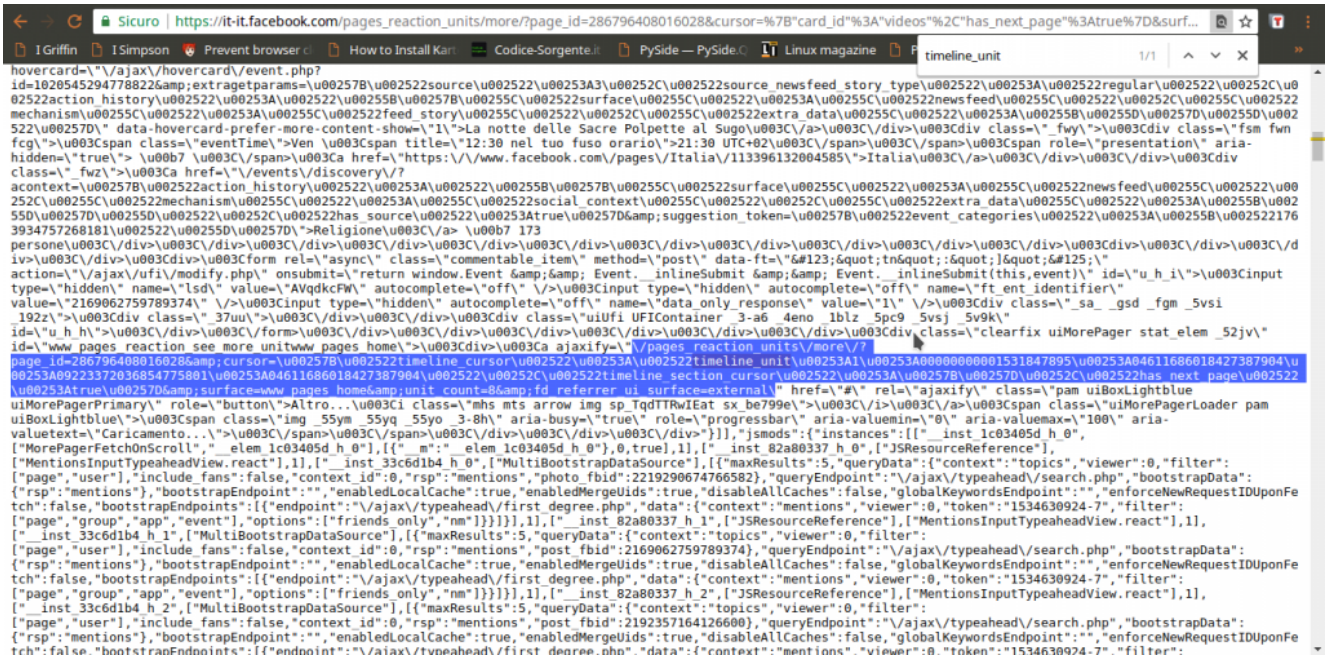
trovato ciò che ci interessava: le altre eventuali richieste sono tutte relative a file accessori, come le immagini.



L’indirizzo della pagina contiene una serie di informazioni importanti, in particolare l’ID della pagina, ed è l’unica richiesta di questo tipo. Possiamo leggere il suo intero URL cliccandoci sopra col tasto destro del mouse e scegliendo **Copy link address**. Aprendo il link, si può capire che forma abbia la risposta: è una sorta di array JSON, una lista di oggetti vari, tra i quali il codice HTML necessario a presentare i post che sono stati richiesti. Si può facilmente distinguere il testo dei post in mezzo a tutto il codice. Alcuni caratteri vengono codificati come Unicode, inclusi alcuni pezzi dei tag HTML, e c’è sempre l’escape per i simboli /, che appaiono come \/, quindi è importante ricordarsi di convertirli in caratteri veri e propri, per poterli riconoscere facilmente (per esempio, \u003C\/p> è in realtà

).

Guardando meglio la risposta di **pages_reaction_units** si scoprono alcune cose interessanti. Innanzitutto, tutti i post che si sono ottenuti vengono presentati tra il testo **{“__html”:** e il testo **]]}**, quindi possiamo selezionarlo facilmente. Inoltre, ogni post è preceduto dalla sua data, in vari formati. In particolare, c’è la forma Unix Time, che è molto comoda da gestire ed è sempre identificata dalla dicitura **data-utime**, quindi potremo distinguere i vari post dividendo l’intero HTML in più pezzi dopo ogni occorrenza della parola **data-utime**. E non solo: si può anche capire come ottenere ulteriori post, facendo un’altra richiesta a questa stessa pagina. Se infatti cerchiamo **pages_reaction_units** all’interno della risposta possiamo notare che c’è l’intero indirizzo di una richiesta come quella che abbiamo appena inviato, ma contenente anche il blocco **timeline_cursor**, con i riferimenti della timeline di Facebook relativi ai prossimi post. Possiamo quindi facilmente estrarre questi riferimenti per confezionare la nostra prossima richiesta.



Ora, tutto quello che è rimasto da scoprire è come costruire l'indirizzo da contattare per ottenere i vari post: sappiamo che serve **pages_reaction_units** più una serie di argomenti (che vediamo nella richiesta estrapolata dal browser). Non tutti gli argomenti sono però necessari, e possiamo scoprire quali siano superflui semplicemente provando a cancellarli uno alla volta e vedendo in quali casi si ottiene comunque il risultato desiderato. Scopriamo quindi che l'indirizzo necessario è qualcosa del tipo: **https://it-it.facebook.com/pages_reaction_units/more/?page_id=286796408016028&cursor={"timeline_cursor":"timeline_unit:1:000000000015286240 41:04611686018427387904:09223372036854775793:04611686018427387904","timeline_section_cursor":{},"has_next_page":true}&surface=www_pages_home&unit_count=8&dpr=1&__user=0&__a=1**. Di questo indirizzo, abbiamo capito che la **timeline_unit** può essere scoperta all'interno di una richiesta precedente, mentre per scoprire l'ID della pagina Facebook basta scorrere il codice HTML della pagina stessa (che si può vedere nel browser Chrome con il prefisso **view-source:**) e cercare proprio **pages_reaction_units**, e subito dopo la parola **page_id**. Giocando un po' con **unit_count** scopriamo che per la prima richiesta possiamo ottenere fino a 300 post, mentre per tutte le successive (quelle in cui si specifica la **timeline_unit**) il massimo che

si può chiedere è 8 post. Tutto il resto è solo un insieme di argomenti vari sempre uguali, che nel nostro programma potremo quindi memorizzare sotto forma di variabili.

La procedura per scaricare tutti i post sarà quindi intuitiva: si accede alla pagina leggendo il suo codice HTML per scoprire l'ID. Con questo si forma il primo URL da contattare per ottenere gli ultimi post. All'interno della risposta si prendono i post dividendoli e salvandoli separatamente, e si cercano anche i riferimenti della timeline_unit per poter fare una nuova richiesta e ottenere altri post, più vecchi. Poi si ripetono continuamente gli ultimi passaggi, leggendo la risposta, salvando i post, costruendo il nuovo indirizzo, e facendo una nuova richiesta, finché Facebook non fornisce più alcun post (il che significa che siamo arrivati all'inizio della pagina e i post sono finiti). Ora dobbiamo tradurre questa idea in uno script Python.

Leggere le pagine web

Cominciamo lo script, tutto in un unico file che chiamiamo **scrapefb.py**:

L’inizio è dato dalla shebang (#!), che su sistemi Unix è utile per automatizzare l’avvio dello script trattandolo come un eseguibile. Poi si devono importare tutte le librerie necessarie: urllib permette di scaricare il contenuto degli URL, e socket permette di stabilire un timeout sulle connessioni per chiuderle se sono inattive. Per fare il parsing della pagina web, cioè per leggere il suo contenuto distinguendo i vari “pezzi”, utilizziamo le espressioni regolari con la libreria re. Abbiamo anche bisogno di lavorare con data e ora, e accedere a funzioni relative al sistema operativo (per lettura e scrittura dei file).

Definiamo uno user agent: si tratta di una semplice stringa di testo che ogni sito richiede a chi vuole ricevere le pagine web, per capire di chi si tratti. Siccome possiamo scriverla come vogliamo, nessuno controlla davvero che stiamo dicendo la verità, possiamo scriverla in modo da convincere Facebook che il nostro script è in realtà il browser web Mozilla Firefox.

Definiamo una funzione che ci aiuti a scaricare tutto il contenuto di una pagina web, e la chiamiamo **geturl**. Prima di tutto, specifichiamo che ci serve lo useragent che abbiamo dichiarato nella sezione globale dello script. Poi ci assicuriamo di non procedere se l’url fornito alla funzione è vuoto, così evitiamo errori inutili. Costruiamo la richiesta HTTP utilizzando l’url. Servono anche un array di dati, che però in questo caso non è necessario visto che non abbiamo un form HTML da fornire alla pagina, e una intestazione. L’intestazione viene costruita con lo user agent, così Facebook ci scambierà per un browser web e non bloccherà la richiesta.

La richiesta HTTP può essere inviata usando la famosa funzione urlopen, e impostiamo anche un timeout. Il timeout è utile per non rimanere bloccati in eterno nel caso la connessione dovesse essere troppo lenta. Con un tempo di 300 secondi,

sappiamo che al massimo dopo 5 minuti la situazione verrà sbloccata. La risposta del server alla nostra richiesta può essere letta con la funzione **read**, e nel caso qualcosa non abbia funzionato impostiamo la risposta (variabile **ft**) come vuota.

In teoria potremmo tenere la risposta così com'è, ma non è una buona idea: il web è una giungla di codifiche, e se non gestiamo la cosa rischiamo di ottenere testi illeggibili. Soprattutto per Facebook, che spesso codifica le varie emoticon sotto forma di caratteri speciali Unicode. Cerchiamo quindi prima di tutto di capire se il server ci suggerisca la codifica della pagina che ci sta inviando. In caso negativo, proviamo a decodificare il testo con la classica codepage di Windows 1252, uno standard sui sistemi Microsoft precedenti a Windows 10. Se non dovesse funzionare, proviamo a decodificare tutti i caratteri usando l'utf-8 togliendo però gli slash inutili (che spesso i server web forniscono per facilitare i browser), e altrimenti cerchiamo di tradurre direttamente l'intera pagina in una stringa python. Comunque sia andata, quindi, avremo una più o meno corretta stringa python piena di tutto il codice html della pagina. Per leggere meglio il suo contenuto, utilizziamo la funzione **html.unescape** per decodificare anche le varie entità dell'html (per esempio, &gt e &lt sono rispettivamente > e <, preziosi per interpretare il codice). L'unescape delle entità html non è fondamentale, ma rende il nostro lavoro più comodo.

Cercare l'ID della pagina Facebook

Cominciamo a scrivere la funzione vera e propria per lo scraping delle pagine di Facebook. La funzione richiede, come argomenti, l'indirizzo della pagina da scaricare, la cartella in cui salvare il risultato, e se si debba salvare il

risultato come tabella CSV invece che come testo TXT.

Innanzitutto ci sono un paio di informazioni, che possiamo memorizzare in alcune variabili. Potremmo anche scriverle direttamente nelle funzioni che le usano, come vedremo, ma tenendole nelle variabili è molto più facile modificarle in futuro se dovesse essere necessario a causa di modifiche nel funzionamento di Facebook. La variabile TOSELECT_FB contiene la stringa da cercare dentro la pagina Facebook per conoscere l'URL che fornisce i vari post. Le due successive variabili sono le stringhe che delimitano l'inizio e la fine dei post nella risposta. Infatti, Facebook non fornisce solo l'elenco dei post della pagina, ma anche una serie di altre informazioni che non ci servono. Per non complicarsi la vita, bisogna avere un output pulito, quindi toglieremo tutto ciò che non ci serve isolando solo il testo presente tra quei due delimitatori. Stabiliamo poi il numero di risultati che vogliamo: il massimo consentito da Facebook (al momento) per la prima richiesta è di 300 post. Inoltre, specifichiamo un periodo di attesa prima di inviare le richieste successive, per evitare che il server possa accorgersi che ne stiamo facendo troppe tutte assieme. Le ultime due rappresentano l'inizio e la fine del link per ottenere i vari post: vedremo tra un po' come costruirlo nella sua interezza.

In questo momento siamo pronti per eseguire la prima richiesta e scaricare la pagina Facebook. L'indirizzo che contattiamo è qualcosa del tipo **https://it-it.facebook.com/chiesapastafarianaitaliana/**. Ovviamente otteniamo soltanto gli ultimi post, proprio quello che un utente normale vede quando carica la pagina. Di per se i post che appaiono non ci interessano, li otterremo contattando direttamente l'URL che fornisce tutti i post. Il codice HTML di questa pagina ci interessa soltanto perché possiamo estrarre delle informazioni. In particolare, vogliamo scoprire il dominio di Facebook, cioè tutto quello che è

compreso tra **https://** e il primo / successivo. Nel caso in esempio è **it-it.facebook.com**, ovviamente è diverso per ogni paese (una pagina spagnola non inizierà con it-it). Cerchiamo anche di capire il nome della pagina, che è tutto ciò che segue il dominio: siccome lo useremo come nome del file in cui salvare i risultati è fondamentale che non ci siano caratteri strani. Usando una espressione regolare, cancelliamo (sostituiamo con "") tutti i caratteri che non siano lettere o numeri. Fondamentale per poter proseguire è il **pageid**, cioè il numero identificativo della pagina che vogliamo scaricare: questa informazione si può recuperare dalla pagina stessa perché è sempre presente in essa un link che contiene tale numero. Il link in questione ha la forma **?page_id=286796408016028&cursor**, quindi possiamo scoprire l'ID cercando ciò che segue la parola **page_id=** e arriva fino al simbolo **&**. Ci si potrebbe chiedere come mai per cercare i vari delimitatori utilizziamo direttamente la funzione index, molto pratica e veloce, mentre per cercare la posizione del **'pages_reaction_units'**, che determina l'inizio del link in cui troviamo la pageid, usiamo le RegEx. La risposta è semplice: per ora trovare questa stringa è facile, ma in futuro potrebbe essere necessario usare una espressione regolare. In questo modo, lo script è già pronto per future modifiche.

Ora che abbiamo tutte le informazioni necessarie, possiamo costruire il nome del file in cui andremo a scrivere i post recuperati. Il nome è dato dalla cartella in cui salvare i file più **fb_** e il nome della pagina. Ovviamente, se l'utente vuole un TXT l'estensione del file sarà TXT, e se vuole un CSV l'estensione sarà CSV. Creiamo anche un altro file, con stesso nome la estensione **.tmp**. Questo è il file in cui andremo ad inserire i vari link già visitati, così se si deve riprendere lo scaricamento dei post di una pagina Facebook non lo si ricomincia da capo ogni volta, ma si riprende da dove ci si era interrotti. Per l'appunto, nel caso il file esista già vuol dire che non si deve ricominciare da capo, quindi si

carica l’intero contenuto del file in una lista, chiamata **alllinks**. In questa lista ogni elemento è un link, perché il file è stato diviso riga per riga (e quando lo scriveremo, metteremo un link in ogni riga). Definiamo anche una variabile che faccia da contatore, per sapere quante richieste di post siano state fatte, e una che stabilisca se stiamo ripristinando un download interrotto o se dobbiamo ricominciare da capo.

Richiedere i post della pagina al server di Facebook

Siamo al momento della raccolta vera e propria dei post della pagina. Siccome dobbiamo fare tante richieste una dopo l’altra, utilizziamo un ciclo. Il ciclo **while** andrà avanti finché la variabile **active** sarà True. Ne consegue che per fermare il ciclo, se necessario, non dovremo fare altro che porre tale variabile uguale a False.

Il link viene costruito unendo il dominio di Facebook, la parte iniziale del link, l’id della pagina, e la parte finale. Sarà quindi qualcosa del tipo **https://it-it.facebook.com/pages_reaction_units/more/?page_id=286796408016028&cursor={“card_id”:”videos”,”has_next_page”:true}
&surface=www_pages_home&unit_count=300&referrer&dpr=1&__user=0&__a=1**, come si può notare ci sono tutti i vari pezzi che abbiamo costruito finora. Se provate ad aprire questo indirizzo col browser vi accorgerete che fornisce una serie di informazioni, tra cui l’html dei vari post che sono stati richiesti (cioè gli ultimi 300 post della pagina). Inseriamo il link appena costruito nel file che li deve memorizzare, così se lo script dovesse bloccarsi mentre cercare di recuperare i post sapremo di dover ricominciare da questo preciso link, e non doverli rifare tutti da capo. Usando la

modalità di accesso al file “a” eseguiamo un “append”, cioè inseriamo direttamente questo link alla fine del file, in una nuova riga, senza bisogno di preoccuparci di quali altri link ci fossero prima (non dobbiamo quindi aprire il file, leggerlo, aggiungere il nuovo link, e poi salvarlo). È un risparmio di risorse importante.

Sempre utilizzando la funzione geturl possiamo recuperare anche con il nostro script tutta la risposta del server di Facebook. Siccome ci interessa soltanto la parte con i vari post che abbiamo richiesto, la estraiamo e la memorizziamo nella variabile **postshtml**. Il codice HTML dei vari post va un po’ ripulito: Facebook usa molti caratteri che non sono UTF-8 per gestire le emoticon, in genere sono utf-16. Però per il nostro scopo sono fastidiosi, le emoticon non ci interessano affatto e l’elaborazione dei testi è molto più facile con l’utf-8. Quindi ci assicuriamo di tradurre tutti i caratteri in UTF-8, togliendo anche l’escape dei caratteri speciali. Facebook, infatti, decide che alcuni caratteri sono particolari e li presenta con al loro notazione Unicode, una cosa del tipo \u0001. Questo è molto scomodo per noi, quindi forziamo la trasformazione in caratteri leggibili. A questo punto potrebbero essere rimasti dei simboli che UTF-8 non è in grado di gestire, perché si tratta delle famigerate emoticon UTF-16. Si riconoscono perché il codice Unicode è compreso tra \uD800 e \uDFFF. Siccome non ci interessano, usiamo una semplice espressione regolare per cancellarli, sostituendoli con la stringa vuota “”. Ora abbiamo finalmente l’intero codice HTML dei post, pulito e pronto per essere letto e interpretato. Siccome ogni post di Facebook è contrassegnato da un orario nel formato Unix Time (uno standard di internet), possiamo spezzare il contenuto dell’HTML nei singoli post dividendo proprio in base a **‘data-utime’**, che è la stringa che Facebook usa per indicare l’orario di un post.

In questo momento, la lista **postsarray** contiene i vari post:

in realtà, il primo elemento della lista non contiene post, perché ha tutto l'HTML precedente. Comunque, possiamo scorrere la lista e individuare i post banalmente cercando il loro timestamp, cioè l'orario della pubblicazione. È facile da identificare, perché come dicevamo ogni post viene preceduto da una span (elemento HTML) che contiene una dicitura di questo tipo: **data-utime=\"1531306802\" data-shorten=\"1\" class=\"_5ptz\">**. Siccome noi stiamo dividendo l'HTML a ogni "data-utime", è ovvio che ogni post inizierà con**=\"1531306802\" data-shorten=\"1\"…**, e quindi l'orario in formato Unix sarà il primo numero tra virgolette (nell'esempio è **1531306802**). Per essere sicuri di nona vere problemi, usiamo una RegEx per cancellare dal timestamp ottenuto qualsiasi cosa non sia un numero, e convertiamo il risultato in un **int**, cioè un numero intero. Nel caso non sia possibile risalire a questo numero, come per il primo elemento della lista che non è un vero post, consideriamo il numero pari a zero. Poi, usando datetime, possiamo convertire questo timestamp in una data facilmente leggibile, nel formato anno-mese-giorno ore:minuti:secondi. La data viene quindi aggiunta alla lista **timearray**, che abbiamo appositamente creato. Ciò significa che per ogni elemento di **postsarray**, cioè ogni post della pagina Facebook, abbiamo un corrispondente elemento di **timearray**, cioè la data della pubblicazione del post stesso.

Tutto il testo (se c'è) del post numero **i** si trova dentro all'elemento **postsarray[i]**, ma è ovviamente circondato da un sacco di altri pezzi di HTML che non ci servono. Per estrapolare soltanto il testo dei post basta prelevare tutto ciò che si trova all'interno dei paragrafi (che nella risposta di Facebook sono i tag

<\/p>). Bisogna ricordare che nello scrivere l'espressione regolare per trovare i tag il carattere \ ha bisogno di una sequenza di escape lunga, e va scritto come \\\\. La funzione **finditer** crea l'array **indexes**, che contiene tutte le posizioni in cui si trovano i vari paragrafi: un post di Facebook può

infatti essere diviso in tanti paragrafi, e noi li vogliamo tutti. Ciascun elemento di **indexes**, contiene in realtà due informazioni: la prima (cioè **0**) è l'inizio del paragrafo, e la seconda (cioè **1**) è la fine del paragrafo. Usando il classico sistema di slicing delle stringhe di Python, si può banalmente estrarre il testo di ogni paragrafo semplicemente partendo dal carattere iniziale e finale (quindi **postsarray[i][start:end]**, perché la stringa è **postsarray[i]**). Alla fine del ciclo for che legge tutti i vari **indexes**, avremo la variabile **thispost** che contiene tutti i vari paragrafi uniti, senza gli altri tag inutili.

Possiamo assegnare tutto il testo del paragrafo all'elemento stesso da cui eravamo partiti, così lo avremo ripulito. Prima, però, togliamo i tag che ancora esistono. Per esempio, il grassetto viene realizzato con i tag , quindi noi cancelliamo tutto cioè che si trova tra i simboli **<** e **>**. E cancelliamo anche gli slash non necessari. Quindi, **gatti\/cani** diventa **gatti/cani**. Alla fine presentiamo l'array sul terminale, così è più facile fare il debug e capire se qualcosa non vada bene. Lo scraping è pur sempre legato a qualcosa di molto casuale, e può capitare che in situazioni particolari qualcosa improvvisamente non funzioni.

Scoprire gli ID dei prossimi post da scaricare

Ora abbiamo ricostruito la lista **postsarray**, che contiene tutti i post presenti nell'attuale risposta di Facebook. Dobbiamo ancora capire come costruire la prossima richiesta, per ottenere una nuova risposta.

Le varie richieste che vengono inviate sono qualcosa del tipo **https://it-it.facebook.com/pages_reaction_units/more/?page_id=**

286796408016028&cursor={“timeline_cursor”:”timeline_unit:1:000 00000001528624041: 04611686018427387904:09223372036854775793: 04611686018427387904″,”timeline_section_cursor”:{}, “has_next_page”:true}&surface=www_pages_home &unit_count=8&dpr=1&__user=0&__a=1. Se ci si fa caso, è praticamente identica alla prima richiesta, con due differenze fondamentali: l’argomento **cursor** contiene i riferimenti della timeline di Facebook che indica da dove iniziano i post da scaricare. E poi c’è la **unit_count** che è limitata a **8**, quindi si possono scaricare al massimo 8 post per volta. Siccome lo stesso Facebook ha bisogno di sapere quali siano i riferimenti della timeline della pagina da scaricare, è ovvio che nella attuale risposta (quella che abbiamo appena ricevuto) ci debbano essere. E infatti ci sono, si possono trovare proprio nella forma dell’url con i due argomenti **cursor** e **unit_count**, quindi possiamo ottenerli cercando questi pezzi dell’URL dentro la stringa **newhtml** (che contiene l’ultima risposta che abbiamo ottenuto da Facebook). Siccome la prima parte dell’url è sempre la stessa, non dobbiamo fare altro che modificare la parte finale includendo i riferimenti della timeline appena ottenuti nella variabile **lending**. In questo modo, al prossimo ciclo verrà di nuovo costruito il **link**, ma usando questo **lending** come parte finale, e si potrà fare la nuova richiesta a Facebook per i successivi 8 post della pagina. Ovviamente, il testo della timeline estrapolato va un po’ pulito, con le funzioni che avevamo già visto per la rimozione dei caratteri Unicode inutili e per la decodifica dell’url. Se non riusciamo a trovare un url per i prossimi post, vuol dire che sono finiti, quindi dobbiamo interrompere il ciclo impostando la variabile **active** come falsa.

Se per qualche motivo non è stato possibile recuperare i post e le date dei post, le due apposite liste vengono inizializzate come vuote, così il programma non si bloccherà.

È arrivato il momento di salvare il risultato in un file.

L’elenco dei post scaricati durante questo ciclo è nella lista **postsarray**, possiamo trasformarla in un testo da salvare nel file prendendo i vari elementi della lista e aggiungendoli alla variabile **postsfile**, uno in ogni riga (**\n** indica un invio a capo riga). Se si desidera che il file sia un CSV, il testo del post viene preceduto dalla data di pubblicazione del post, che si trova nella lista timearray. La data e il testo del post sono separati da una tabulazione, cioè **\t**, perché se utilizzassimo altri simboli come virgola e punto virgola il risultato sarebbe inaffidabile: un post di Facebook può facilmente contenere della punteggiatura, ma non una tabulazione.

Ora che il testo da scrivere nel file è tutto nella variabile **postsfile**, dobbiamo capire se sia necessario creare il file da capo oppure no. Se questa è la prima iterazione del ciclo, e non si sta eseguendo il ripristino di un download interrotto precedentemente, bisogna scrivere il testo nel file, dunque sovrascrivendo qualsiasi cosa ci fosse (se il file esisteva già). Altrimenti, bisogna soltanto aggiungere l’attuale testo a ciò che era già stato scaricato in precedenza, usando per la modalità di scrittura append (cioè **a**) che avevamo già visto.

Ovviamente, alla fine del ciclo si incrementa di 1 il contatore **timelineiter**, che ha per l’appunto la funzione di farci sapere quante iterazioni sta facendo il programma alla ricerca di altri post da scaricare. Inoltre, prima di concludere le operazioni del ciclo, e ricominciarlo da capo, attendiamo un certo numero di secondi. Questo è importante perché se riprendessimo immediatamente a fare richieste a Facebook, il server potrebbe accorgersi che stiamo insistendo troppo e magari bloccare la connessione.

Il blocco principale dello script

Terminata la funzione per lo scraping di Facebook, ricomincia il blocco principale dello script. Per sicurezza, controlliamo che in questo momento sia stata specificamente richiesta, dall'interprete Python, la funzione main. Questo avviene soltanto se lo script è stato lanciato direttamente dal terminale, e non se è stato importato in un altro script. Questo controllo facilita un eventuale utilizzo del nostro script come libreria per altri programmi.

Ora, si definisce la pagina Facebook da cui si parte, che può essere fornita dall'utente come primo argomento dello script. Il secondo argomento, se esiste, deve rappresentare la cartella in cui si vuole ottenere il file di output, e se non è specificato si suppone che sia la cartella attuale (cioè **./**, presumibilmente la stessa in cui si trova anche lo script). Il terzo argomento, se esiste, può essere la parola "CSV": in questo caso, significa che l'utente vuole ottenere il CSV invece del TXT. Le varie informazioni vengono passate alla funzione di scraping per cominciare il download dei vari post. L'utilizzo è quindi molto semplice, e richiede praticamente soltanto l'indirizzo completo della pagina che si vuole scaricare, così come ogni utente può leggerlo in un browser web. Per esempio, si può lanciare lo script col comando **python3 scrapefb.py https://it-it.facebook.com/chiesapastafarianaitaliana/ ./ CSV** in modo da ottenere nella cartella corrente un CSV con data e testo di tutti i post della pagina della Chiesa Pastafariana Italiana, oppure si può usare il comando **python3.exe scrapefb.py https://it-it.facebook.com/chiesapastafarianaitaliana/ C:\Temp** per ottenere il TXT nella cartella **C:\Temp**.

Il codice completo

Potete trovare il codice completo dello script all'indirizzo https://gist.github.com/zorbaproject/c1f8fff28cd0becea3a0fb6d0badd159. Per utilizzarlo è necessario avere Python3 installato sul proprio sistema, ed è stato provato sia su GNU/Linux che su Windows.

IBM e il quantum computing per tutti

Due anni fa è avvenuto, un po' in sordina, uno di quegli eventi che potrà avere ripercussioni a lungo termine sullo sviluppo dell'informatica: IBM ha aperto l'accesso, tramite cloud computing, al proprio calcolatore quantistico. Si tratta di qualcosa di cui tutti, anche i meno esperti, hanno sentito parlare, ma spesso si fa molta confusione sull'argomento e non si riescono a comprendere appieno le implicazioni di questa rivoluzione. Naturalmente, per capirle dobbiamo prima capire la differenza tra un computer classico ed uno quantistico. Prima di cominciare, però, specifichiamo un punto importante: la rivoluzione non è ancora iniziata, la si sta soltanto preparando, per ora. Con l'attuale computer quantistico di IBM non si può fare molto, e le normali operazioni di programmazione sembrano inutilmente complicate. Questo perché la potenza di calcolo è ancora troppo bassa. Naturalmente, il problema dei computer quantistici è che man mano che la loro potenza (o, se volete, il numero di qubit, lo spiegheremo più

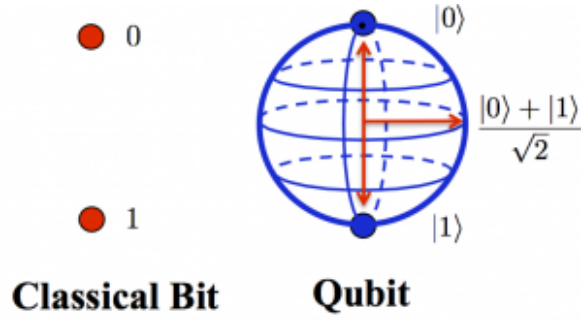
avanti) aumenta diventano molto più difficili e costosi da costruire. Ed è per questo motivo che IBM permette l'accesso a tutti i programmatori tramite il cloud: in questo modo potrà contare su una squadra di alfa-tester volontari, e potrà cercare di migliorare l'efficienza del calcolatore scoprendo i problemi che salteranno fuori durante i vari test. Insomma, i calcolatori quantistici per ora non sono utili. Ma tra qualche anno potrebbero esplodere con la loro potenza, e potrebbero consentirci di eseguire algoritmi che oggi con un computer classico non si possono proprio realizzare.

Le macchine di Turing

Un computer è fondamentalmente una macchina di Turing. Una macchina di Turing è un modello matematico che rappresenta ciò che un calcolatore programmabile può fare. Ovviamente, ciò che un calcolatore può fare nel mondo reale è dettato dalle leggi della fisica classica. La macchina di Turing nasce per rispondere ad una domanda: esiste sempre un metodo attraverso cui un qualsiasi enunciato matematico possa essere stabilito come vero o falso? In altre parole, è possibile sviluppare un algoritmo con cui capire se una qualunque affermazione sia vera o falsa?

La risposta è no, non è possibile sviluppare un algoritmo generale che stabilisca la veridicità di una qualsiasi affermazione, e questo perché in realtà il concetto di "algoritmo" non è ben definito. Tuttavia, Alan Turing propose comunque un modello matematico (la sua famosa "macchina") con cui verificare, per ogni singola affermazione, se sia possibile arrivare a stabilirla come vera o falsa. Con una macchina di Turing infatti è possibile sviluppare algoritmi per ridurre una affermazione ai suoi componenti di base, cercando di verificarla, e vi sono due opzioni: o la macchina ad un certo punto termina l'algoritmo e fornisce una risposta, oppure l'algoritmo andrà avanti all'infinito (in una sorta di

loop continuo, chiamato **halting**) senza mai fornire una risposta. Oggi si usa proprio la macchina di Turing per definire il concetto di “algoritmo”.



Mentre un bit ha due soli possibili valori, un qubit ha una probabilità di essere più vicino a 1 o a 0, quindi è un qualsiasi numero compreso tra 0 e 1

Una macchina di Turing utilizza come input-output un nastro su cui legge e scrive un valore alla volta, in una precisa cella del nastro, ed in ogni istante di tempo **t(n)** la macchina avrà un preciso stato **s(n)** che è il risultato dell’elaborazione. Ovviamente la macchina può eseguire alcune operazioni fondamentali: spostarsi di una cella avanti, spostarsi di una cella indietro, scrivere un simbolo nella casella attuale, cancellare il simbolo presente nella casella attuale, e fermarsi.

Un modello così rigido, che esegue operazioni elementari per tutto il tempo che si ritiene necessario, può di fatto svolgere qualsiasi tipo di calcolo concepibile. Tuttavia, così come Godel aveva dimostrato che alcuni teoremi non si possono dimostrare senza cadere in un ciclo infinito, anche con la macchina di Turing non è detto che arrivi ad un risultato, perché alcuni calcoli non si possono portare a termine senza cadere in un processo infinito (in altre parole, il tempo necessario per la soluzione sarebbe infinito, e nel mondo

reale nessuno di noi può aspettare fino all'infinito per avere una risposta). Per fare un esempio banale, se ciò che vogliamo fare è semplicemente ottenere il risultato di 3*4, è ovvio che basta scomporre il problema nei suoi termini fondamentali, ovvero un ciclo ripetuto 4 volte in cui si somma +3. Il ciclo richiede un numero finito di passaggi, quindi la moltiplicazione 3*4 è una operazione che può essere svolta con una macchina di Turing senza cadere nell'**halting**. Se volessimo moltiplicare due numeri enormi servirebbe un ciclo con molti più passaggi, ma sarebbero comunque un numero finito, quindi anche se potrebbe essere necessario un tempo molto lungo, prima o poi il ciclo finirebbe e la macchina produrrebbe un risultato.

Dalla fisica classica alla fisica quantistica

Ora, abbiamo detto che per la macchina di Turing valgono le leggi della fisica classica, e ci riferiamo in particolare ai principi della termodinamica. Il primo principio stabilisce quali eventi siano possibili e quali no, il secondo principio stabilisce quali eventi siano probabili e quali no. Il primo principio, espresso come definizione dell'energia interna di un sistema (U):

$dU=Q-L$

ci dice che l'energia interna di un sistema chiuso non si crea e non si distrugge, ma si trasforma ed il suo valore rimane dunque costante. Infatti, in un sistema isolato Q=0 (il calore) quindi dU=-L, ovvero l'energia interna può trasformarsi in lavoro e viceversa senza però sparire magicamente. Il secondo principio, che viene espresso con l'equazione

$dS/dt>=0$

ci dice che ogni evento si muove sempre nella direzione che fa aumentare l'entropia e mai nell'altra. Al massimo può rimanere costante, nelle rare situazioni reversibili. Infatti la derivata dell'entropia (S) in funzione del tempo (t) è sempre maggiore o uguale a zero. In altre parole, l'enunciato del secondo principio della termodinamica si può riassumere con la frase "riscaldando un acquario si ottiene una zuppa di pesce, ma raffreddando una zuppa di pesce non si ottiene un acquario". Insomma, la maggioranza degli eventi termodinamici non è reversibile, almeno nel mondo reale, e questo vale anche per la macchina di Turing.

Una macchina di Turing quantistica, invece, si baserebbe sulle leggi fisiche della meccanica quantistica, e quindi le sue operazioni potrebbero essere reversibili. Il vantaggio ovvio è che se le operazioni sono reversibili di fatto non è possibile che la macchina quantistica cada nell'**halting** della macchine di Turing classiche. Basandosi proprio sulla meccanica quantistica, ci saranno una serie di differenze con la macchina di Turing classica:

- un bit quantistico, chiamato **qubit**, non può essere letto con assoluta precisione. In genere, si fa soltanto una previsione probabilistica del fatto che sia più vicino a 0 o a 1
- la lettura di un qubit è una operazione che lo danneggia modificandone lo stato, quindi non è più possibile leggerlo nuovamente
- date le due affermazioni precedenti, è ovvio che non si possano conoscere con precisione le condizioni iniziali e nemmeno i risultati
- un qubit può essere spostato da un punto all'altro con precisione assoluta, a condizione che l'originale venga distrutto: è il teletrasporto quantistico
- i qubit non possono essere scritti direttamente, ma si possono scrivere sfruttando l'entanglement, cioè la correlazione quantistica

- un qubit può essere 0 od 1, ma può anche essere una combinazione di questi due stati (un po' 0, un po' 1, come il gatto di Schroedinger), quindi può di fatto contenere molte più informazioni di un normale bit

Apparentemente, queste caratteristiche rendono la macchina inutile, ma è solo il punto di vista di una persona abituata a ragionare nel modo tradizionale. In realtà, una macchina di Turing quantistica è imprecisa durante il momento di input e quello di output, ma è infinitamente precisa durante i passaggi intermedi.

Per chi non ha studiato le basi della fisica, uno sviluppatore ha scritto una guida che riassume alcuni dei concetti fondamentali, soprattutto per quanto riguarda le porte logiche, fondamentali per manipolare i qubit.

I qubit possono essere implementati misurando la proprietà di spin dei nuclei degli atomi di alcune molecole, oppure la polarizzazione dei fotoni (anche i fotografi dilettanti sanno che la luce, composta da fotoni, può essere polarizzata usando appositi filtri). In realtà, pensandoci bene, ci si può accorgere che una macchina di Turing quantistica non è altro che una macchina di Turing il cui nastro di lettura/scrittura contiene valori casuali. Qual è il vantaggio della casualità? Semplice: la possibilità di fare tentativi a caso per cercare la soluzione di un problema che non si riesce ad affrontare con un algoritmo tradizionale. Naturalmente, aiutando un po' il caso con un algoritmo.

Il vantaggio della casualità

“vera”

Nei casi degli algoritmi più semplici, questa idea è solo una complicazione, ma in algoritmi molto complessi e lenti la macchina quantistica può fornire un risultato accettabile in tempi molto ridotti. Possiamo fare un paragone con il metodo Monte Carlo, un metodo per ottenere un risultato approssimato sfruttando tentativi casuali. Facciamo un esempio: immaginiamo di avere una sagoma disegnata su un muro e di voler misurare la sua area. Se la sagoma è regolare, come un cerchio, è facile: basta usare l’apposito algoritmo (per esempio “raggio al quadrato per pi greco”). Ma se la sagoma è molto più complicata, come la silhouette di un albero, sviluppare un apposito algoritmo diventa estremamente complicato e lento. Il metodo Monte Carlo ci suggerisce di prendere un fucile mitragliatore e cominciare a sparare a caso contro il muro: dopo un po’ di tempo non dovremo fare altro che contare il numero di proiettili che sono finiti dentro alla sagoma e moltiplicare tale numero per la superficie di un singolo proiettile (facile da calcolare sulla base del calibro). Otterremo una buona approssimazione della superficie della sagoma: la qualità dell’approssimazione dipende dal numero di proiettili abbiamo sparato col fucile ma, comunque vada, il tempo complessivo per ottenere il valore della superficie è decisamente poco rispetto all’uso di un algoritmo metodico. Un metodo simile che si usa molto nell’informatica moderna è rappresentato dagli algoritmi genetici, i quali hanno però tre svantaggi: uno è che deve essere possibile sviluppare una funzione di fitness, cosa non sempre possibile (per esempio non si può fare nel brute force di una password), e l’altro è che comunque verrebbero eseguiti su una macchina che si comporta in modo classico, quindi il sistema sarebbe comunque poco efficiente (pur offrendo qualche vantaggio in termini di tempistica). Inoltre, è praticamente impossibile ottenere delle mutazioni davvero casuali nei codici genetici che si utilizzano per cercare una soluzione: nei computer classici la

casualità non esiste, è solo una illusione prodotta da qualche algoritmo che a sua volta non è casuale. I qubit risolvono questi problemi, visto che sono rappresentati da oggetti fisici che sono naturalmente casuali.

Scomposizione in fattori primi

Ovviamente, uno degli utilizzi più interessanti di una macchina di Turing quantistica è l'esecuzione di un **algoritmo di fattorizzazione di Shor**. Come si impara a scuola, fattorizzare un numero (cioè scomporlo nei fattori primi) è una operazione che richiede molto tempo, sempre di più man mano che il numero diventa grande. Si tratta di una cosa molto importante perché l'RSA, la crittografia che al momento protegge qualsiasi tipo di comunicazione (dai messaggi privati alle transazioni finanziarie) si basa proprio sui numeri primi e la sua sicurezza è dovuta al fatto che con un normale computer, ovvero una macchina di Turing classica, scomporre in fattori primi un numero sia una operazione talmente lenta che sarebbero necessari degli anni per riuscirci. Ma sfruttando un calcolatore quantistico e l'algoritmo di fattorizzazione di Shor, questa operazione diventa "facile" e la si può svolgere in un tempo che si calcola con un polinomio, dunque è un tempo "finito" invece di "infinito" e solitamente abbastanza breve. Con questo algoritmo, si potrebbero calcolare le chiavi private di cifratura di tutti i messaggi più riservati, e l'intera sicurezza delle telecomunicazioni crollerebbe. Al momento, con gli attuali computer quantistici, non è possibile applicare l'algoritmo di Shor in modo generale, ma sono già state realizzate alcuni versioni semplificate dell'algoritmo adattate per specifici casi. Se i computer quantistici diventassero più potenti ed affidabili, diventerebbe possibile sfruttare l'algoritmo su qualsiasi chiave crittografica e far crollare la sicurezza di internet. Il mondo, comunque, non

tornerebbe all’età della pietra: sempre usando i computer quantistici sarebbe possibile utilizzare un sistema di crittografia a “blocco monouso” con distribuzione quantistica della chiave crittografica, l’unico metodo di cifratura che sarebbe davvero inviolabile. Nel senso che affinché si possa forzare la crittografia con chiave quantistica le leggi della fisica dovrebbero essere sbagliate (e non lo sono).

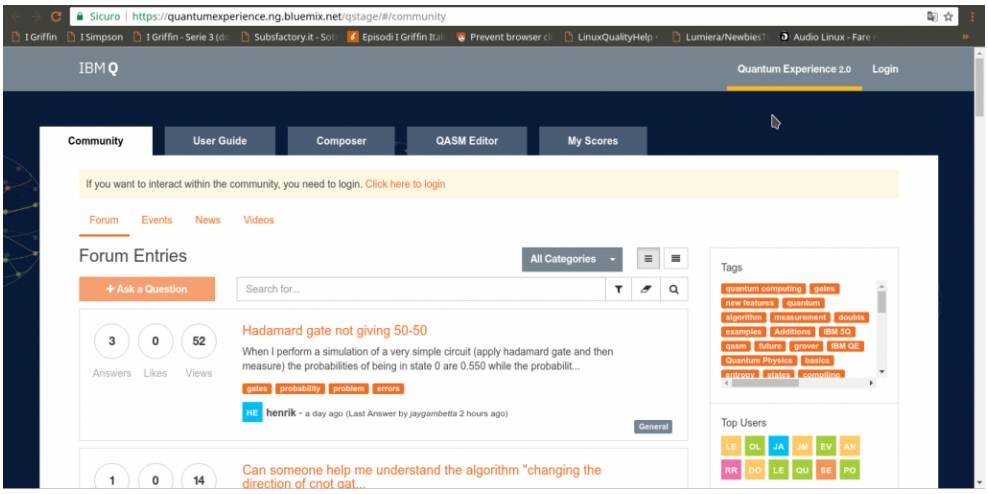
Il quantum computer IBM è gratuito?

Al momento in cui scriviamo, l’accesso al calcolatore quantistico di IBM è gratuito. Quando ci si registra, si ottengono una decina di “units”, cioè dei crediti virtuali. L’esecuzione di un programma costa intorno alle 3 units. Quando le proprie units sono terminate, dopo 24 ore IBM ricarica automaticamente le units iniziali, così si può continuare a sperimentare.

Un esempio pratico: OpenQASM

Il computer quantistico di IBM ha 5 qubit, ma per la maggioranza degli algoritmi sviluppati finora se ne usano soltanto due. Una operazione che può essere interessante provare è la trasformata di Fourier: per chi non la conoscesse, si tratta di una trasformazione che permette di “semplificare” una funzione algebrica riducendone il rumore e rimuovendo i dati non interessanti. Siccome la sua implementazione in un calcolatore classico può essere lunga, è stato sviluppato un algoritmo chiamato FFT, cioè trasformata di Fourier veloce. Per implementare una trasformata di Fourier nel computer quantistico IBM si usa il suo linguaggio nativo OpenQASM:

La sintassi del linguaggio è una via di mezzo tra assembly e C: **qreg** crea un registro di bit quantistici (4, nell'esempio) mentre **creg** crea un registro di bit classici (sempre 4). Poi si possono applicare dei **gate**, ovvero delle porte logiche ai qubit. Per esempio, applicando il gate **X** ai qubit 0 e 2, il registro q diventerà 1010, perché questa porta logica non fa altro che invertire il valore di un qubit (che di default è sempre 0). Il comando **barrier** impedisce che avvengano trasformazioni nei qubit. Altri tipi di porte logiche sono **H** e **CU1**. La prima serve a produrre una variazione casuale nella probabilità di un qubit, cioè nella probabilità che esso sia più vicino ad essere 0 oppure 1. Eseguendo il gate H su tutti i qubit, ci si assicura che i loro stati siano davvero casuali. La seconda, invece, è una delle porte logiche fornite da IBM (ce ne sono una dozzina), che permette di eseguire i passaggi per la serie di Fourier. Infine, il comando **measure** traduce i qubit in bit riempiendo il registro classico **c**, i cui valori possono quindi essere letti senza problemi.



Sul sito https://quantumexperience.ng.bluemix.net/ è possibile iscriversi usando il proprio account GitHub o Google

Naturalmente, grazie alla casualità, se si esegue l'algoritmo più volte si otterranno risultati diversi. Tuttavia, con questo algoritmo si può approssimare in un tempo molto breve

una trasformata di Fourier per l'array classico 1010. Ovviamente, non si può davvero utilizzare questo algoritmo per analizzare il segnale di un ricevitore radio, non sarebbe affatto pratico. Però in futuro potremmo riuscire ad avere computer quantistici tanto potenti da permetterci di eseguire una trasformata di Fourier, approssimata, in tempi rapidi anche per quantità di dati enormi. Intanto, possiamo provare a giocarci un po' e a inventare nuovi gate, nuove porte logiche per i qubit. In fondo, il motivo per cui IBM ha reso pubblico l'accesso al calcolatore quantistico è che soltanto sperimentando nuovi utilizzi di questo tipo di computer sarà possibile aumentare l'efficienza e soprattutto capire quanto davvero il qubit rivoluzionerà la storia dell'informatica e la vita di tutti i giorni.

Caricare codice QASM con Python

Per caricare un programma sul cloud di IBM ed eseguirlo con il processore quantistico si può installare l'apposita libreria direttamente con il comando:

Poi si può sfruttare il codice di test per provare il caricamento di un codice OpenQASM. Prima di tutto si deve modificare il file **config.py** inserendo il proprio token di autorizzazione personale:

e poi si può avviare il programma. Il suo codice è abbastanza semplice, ne presentiamo i punti salienti:

Il programma comincia importando la libreria di IBM ed il file **config** che contiene il token.

Le varie funzione del programma sono inserite in una apposita

classe, chiamata **TestQX**, per sfruttare la programmazione ad oggetti dalla routine principale.

Una delle prime funzioni è quella che si occupa della verifica del token: viene creato l'oggetto api, dal quale si possono ottenere le credenziali utilizzando con il metodo **_check_credentials()**. La funzione restituisce un valore true se le credenziali sono valide.

Un'altra funzione, sfruttando il metodo **get_last_codes()** permette di verificare se siano già stati caricati dei codici con il proprio token, in modo da poterli eventualmente avviare o poter controllare i risultati. Questa funzione, però, può essere utile più che altro in un utilizzo meno sperimentale di quello che faremo le prime volte che proviamo il calcolatore quantistico.

La funzione **test_api_run_experiment** esemplifica l'esecuzione di un codice OpenQASM sul computer quantistico. Oltre a costruirsi un oggetto per accedere alle API, si deve inserire tutto il codice OpenQASM all'interno di una variabile. Potremmo leggere il codice da un file di testo, ma ovviamente per un semplice test conviene scrivere tutto il codice direttamente nel programma. Ovviamente tutto il codice QASM può essere scritto su una sola riga perché le varie righe possono essere separate con un punto virgola (come nel linguaggio C).

Una opzione da definire è il **device**: può essere di due tipi, **simulator** oppure **real**, a seconda del fatto che il codice debba essere eseguito su un simulatore oppure sul vero calcolatore quantistico.

Il parametro **shot** indica il numero di volto in cui si deve eseguire l'esperimento: di solito sul simulatore si fanno 100

cicli, mentre sul computer quantistico almeno 1000. Il massimo è 8192 esecuzioni del codice, ma si può anche provare ad eseguire una sola volta il codice tanto per vedere se funziona.

Infine, si può semplicemente avviare l'esperimento con il metodo **run_experiment** delle API. Il risultato viene fornito sotto forma di array, e ciò che ci interessa è l'elemento **status**.

Di fatto, quindi, eseguire un esperimento è molto semplice, bastano poche righe di codice con le quali si controllano tutti i parametri dell'esperimento e si può leggere il risultato. Python è quindi una ottima opzione per eseguire esperimenti in modo automatizzato. Se si è alle prime armi, tuttavia, conviene utilizzare l'interfaccia web del sito ufficiale per capire come muoversi nella scrittura del codice OpenQASM.

Programmare il calcolatore quantistico

Per imparare il linguaggio nativo del calcolatore quantistico di IBM si può leggere la guida ufficiale, pubblicata assieme ad alcuni esempi su GitHub. Ovviamente è in lingua inglese, ma si occupa di spiegare in modo schematico tutto quello che serve, se non per scrivere dei programmi, almeno per capire gli esempi. In particolare, a pagina 9 è presente una tabella con i principali comandi del linguaggio. Uno dei metodi migliori per utilizzare davvero il calcolatore quantistico di IBM è sfruttare le API per Python. I programmi "quantistici" non si scrivono in Python, si deve sempre usare il codice nativo OpenQASM, ma Python può essere un modo semplice per lanciare i programmi "quantistici" dal nostro computer.

Il Composer sul sito ufficiale permette di sperimentare con le porte logiche