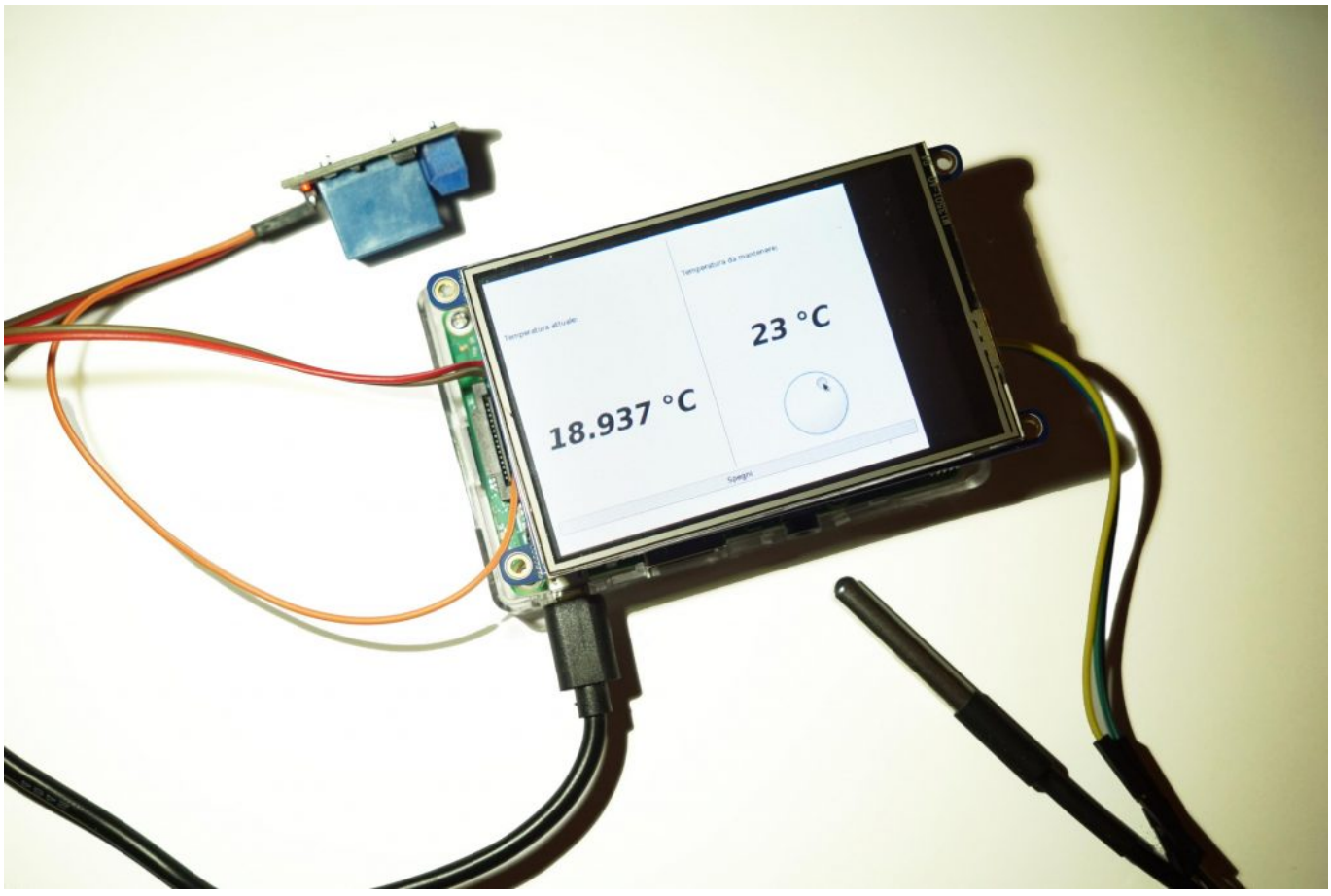
# Un termostato touchscreen con RaspberryPi

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

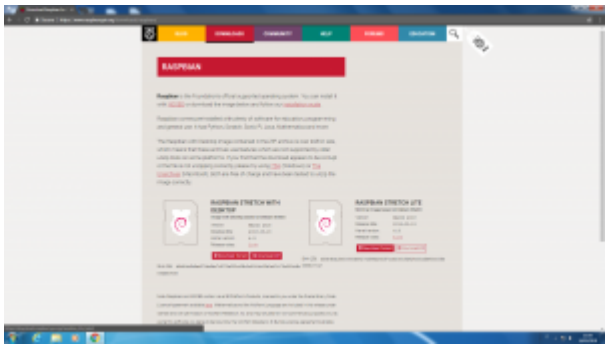
Raspberry Pi Zero W, con lo stesso schermo, bisogna solo aspettare che venga pubblicata la versione Buster di Raspbian per questo dispositivo (prevista tra qualche mese).

Table of Contents



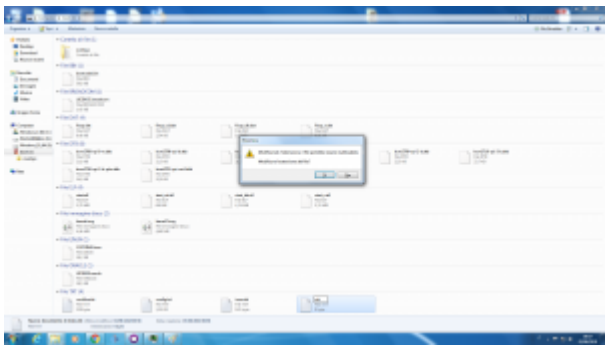
# Preparare la scheda sd

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.



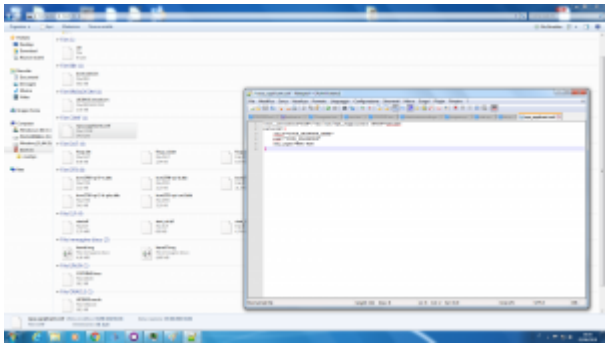
Ottenuta l'immagine del sistema operativo, la si scrive su una scheda microSD con il programma Win32Disk Imager:


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):



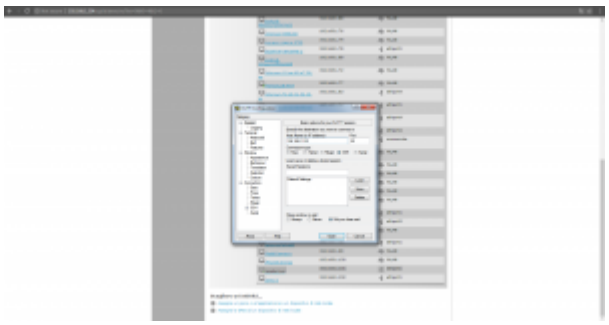
Sempre nella scheda SD si può creare il file di testo **wpa_supplicant.conf**, inserendo un testo del tipo

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.

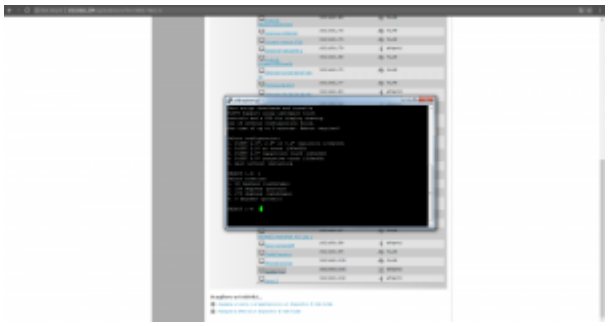
L’indirizzo Il programma che simula un terminale remoto SSH su Windows si chiama Putty, basta indicare l’indirizzo e premere **Open**. Quando viene richiesto il login e la password, basta indicare rispettivamente **pi** e **raspberry**. Per chi non lo sapesse, la password non compare mentre le si scrive, come da tradizione dei sistemi Unix.



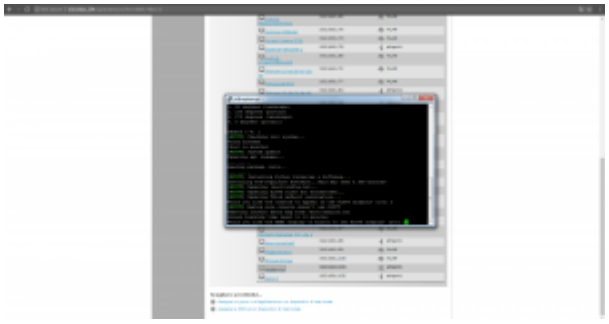
Se il login è corretto si accede al terminale del Raspberry:

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.



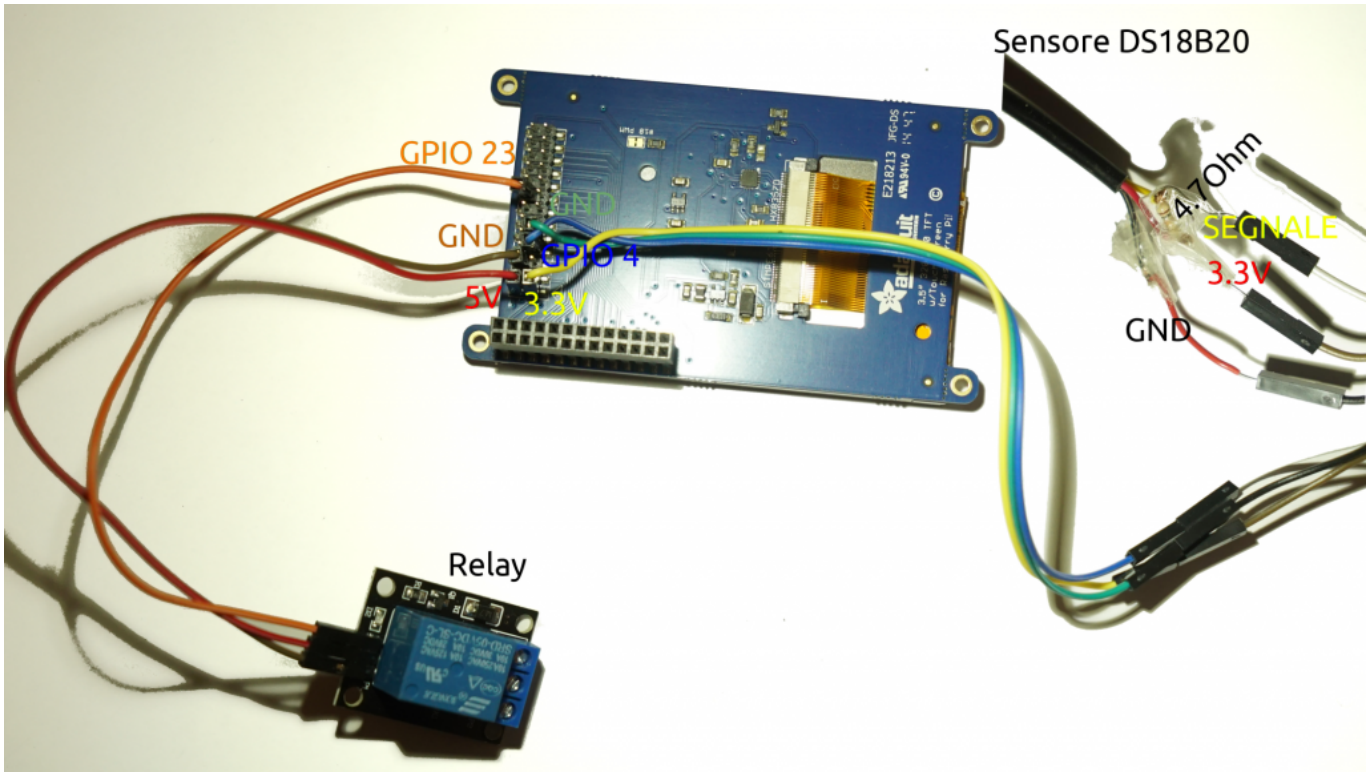
**Un appunto (temporaneo) su Raspbian Buster**
Se state usando l'immagine che abbiamo fornito all'inizio dell'articolo, basata su Raspbian Stretch, vi sarete accorti che lo script di Adafruit non funziona. Questo perché al momento il pacchetto tslib, necessario per lo script, non è disponibile per Buster, 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

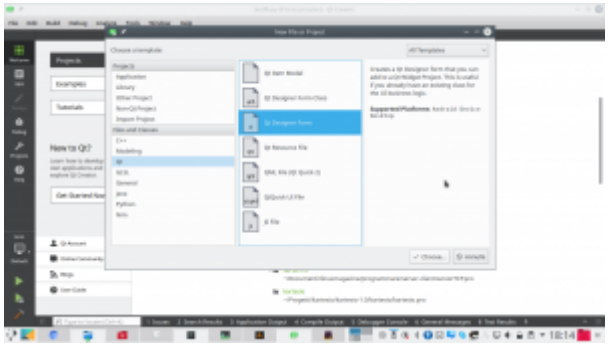
Possiamo verificare il funzionamento del relay e del termometro con un programma molto semplice da eseguire sul terminale:

Per provare il programma basta dare i seguenti comandi:

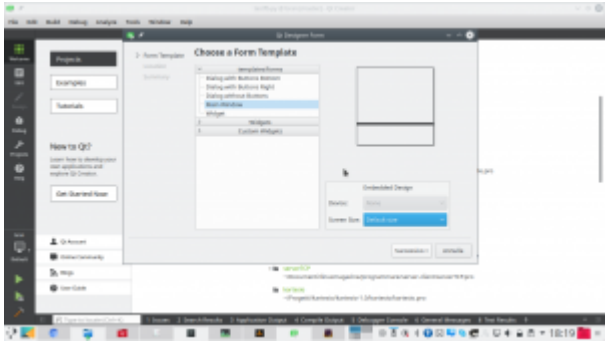
Che cosa fa questo programma? Prima di tutto si assicura che siano caricati i moduli necessari per accedere ai pin GPIO e al sensore di temperatura. Poi esegue un ciclo infinito accendendo il relay, leggendo la temperatura attuale, aspettando 5 secondi, spegnendo il relay, leggendo ancora la temperatura,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

# L’interfaccia grafica

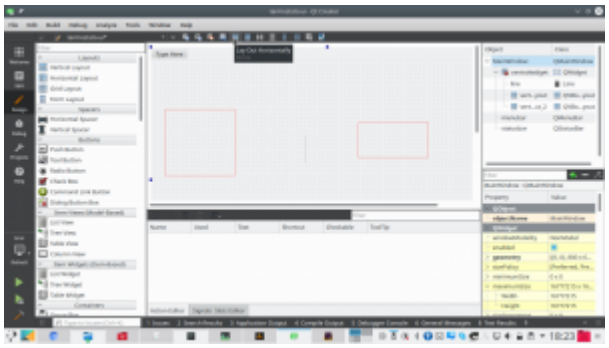
L’interfaccia grafica del nostro termostato è disponibile, assieme al resto del codice, nel repository Git: https://codice-sorgente.it/cgit/termostato-raspberry.git/tree/termostato.ui. Tuttavia, per chi volesse disegnarla da se, ecco i passi fondamentali. Prima di tutto si apre l’IDE QtCreator, creando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



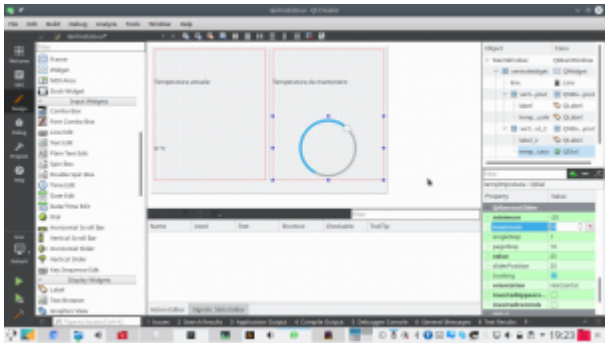
Il template da utilizzare è Main Window, perché quella che andiamo a realizzare è la classica finestra principale di un programma. Per il resto, la procedura guidata chiede solo dove salvare il file che verrà creato.



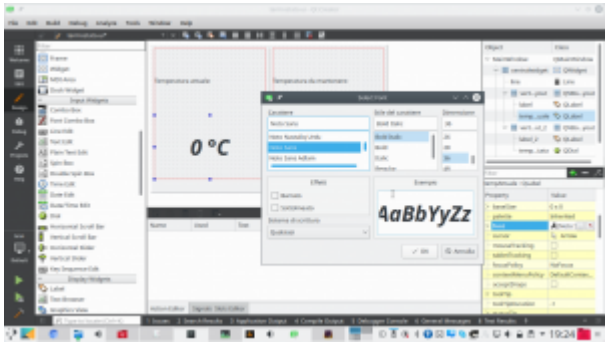
Quando si disegna una finestra, la prima cosa da fare è dividere il contenuto in layout. Considerando il nostro progetto, possiamo aggiungere due oggetti **Vertical Layout**, affiancandoli. Poi, con la barra degli strumenti in alto, impostiamo il form con un **Layout Horizontally**. I due layout verticali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



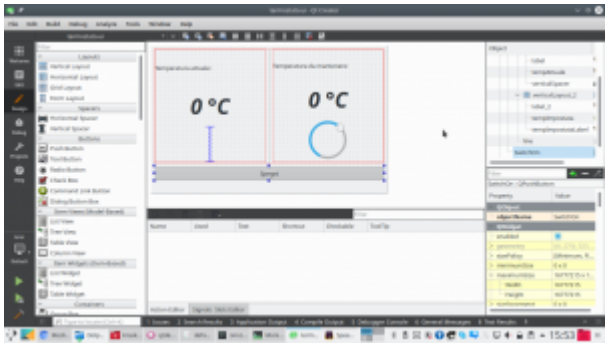
Inseriamo un paio di **label**: in particolare, una dovrà essere chiamata **tempAttuale**, e un'altra **tempImpostataLabel**. Queste etichette conterranno rispettivamente il valore della temperatura attualmente registrata dal termometro e quello che si è deciso di raggiungere (cioè la temperatura da 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").

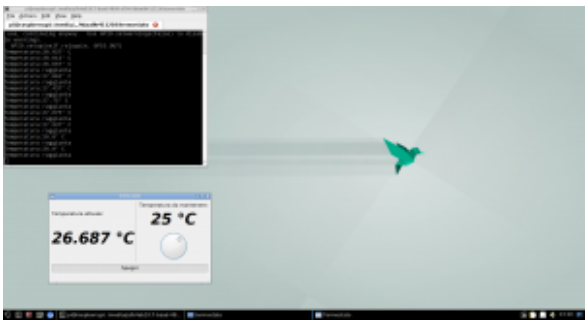
La funzione **dostuff** è quella che si occupa effettivamente di creare il thread dedicato al controllo del relay. Prima di tutto, si controlla che il thread non sia già stato avviato, usando il flag **alreadyOn** che avevamo creato all'inizio del codice di questa classe. Se il thread non è già attivo, 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

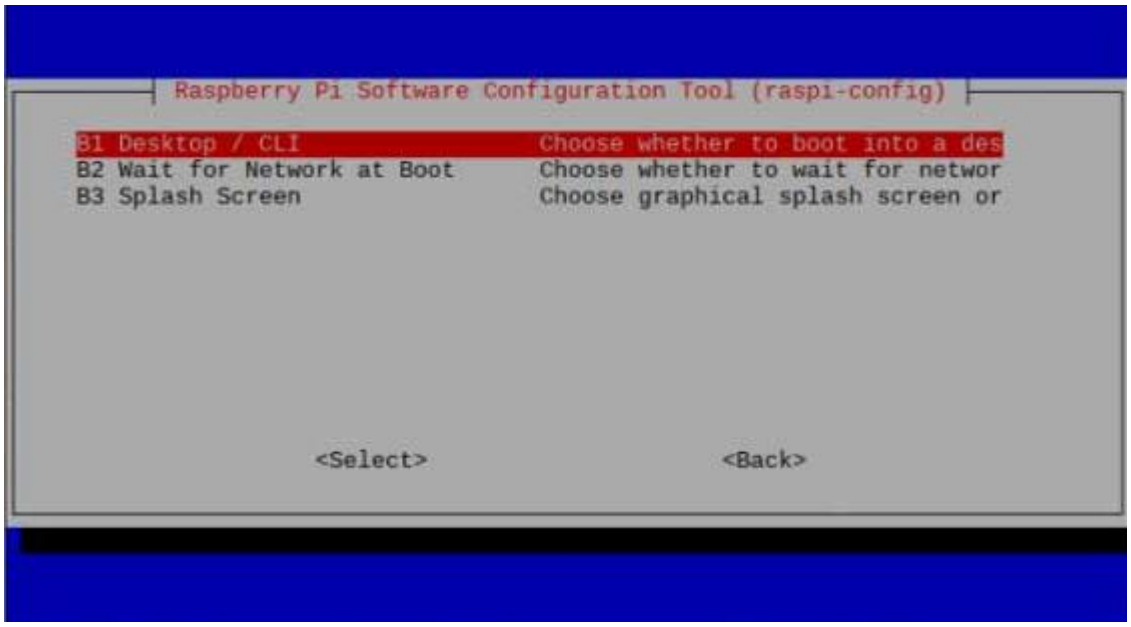
# Avvio automatico

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.

---

# Leggere, modificare, e scrivere i PDF

Tutti hanno bisogno di realizzare o modificare documenti in formato PDF: è tipicamente una delle funzioni più richieste all'interno di una applicazione qualsiasi. Soprattutto per quanto riguarda il desktop, mercato in cui i clienti principali sono aziende e pubblica amministrazione, che ovviamente utilizzano i computer per produrre documenti digitali proprio in formato PDF. Questo perché il Portable Document Format inventato da Adobe nel 1993, e le cui specifiche sono open source e libere da qualsiasi royalty, è lo standard universale ormai accettato da qualsiasi sistema operativo e su qualsiasi dispositivo per la trasmissione di documenti. Proprio perché il formato è utilizzabile gratuitamente da chiunque in lettura e scrittura, è stato inserito in praticamente qualsiasi programma ed è così conosciuto dal grande pubblico. Ormai chiunque sa cosa sia un PDF, e qualsiasi utente vorrà poter archiviare informazioni in questo formato. È quindi fondamentale essere in grado di scrivere programmi che possano lavorare con i PDF, altrimenti si resterà sempre un passo indietro. Il problema è che il formato PDF è abbastanza complicato da gestire, ed è quindi decisamente poco pratico realizzare un proprio sistema per leggere e scrivere questi file. Bisogna basarsi su delle

apposite librerie, e ne esistono varie, anche se purtroppo spesso non sono ben documentate come l'importanza dell'argomento richiederebbe, e chi si avvicina al tema rischia di non sapere da dove iniziare. Per questo motivo, abbiamo deciso di presentarvi un metodo per leggere e uno per creare PDF multipagina, con le principali caratteristiche dei PDF/A.

Come programma di esempio, abbiamo realizzato una interfaccia grafica per gli OCR Tesseract e Cuneiform, capace di funzionare sia su Windows che su GNU/Linux e MacOSX. Per motivi di spazio e di pertinenza, non presenteremo tutto il codice del programma ma soltanto le parti relative alla manipolazione dei PDF. Trovate comunque il link all'intero codice sorgente alla fine dell'articolo.

Le librerie Qt, sulle quali si basa non soltanto l'interfaccia grafica multipiattaforma del nostro programma di esempio, ma anche lo strumento di scrittura dei PDF, sono rilasciate con due licenze libere e una commerciale. Le due licenze libere sono GNU GPL e GNU LGPL: in entrambe i casi sono completamente gratuite, la differenza è che la prima richiede la pubblicazione dei programmi basati sulle Qt con la stessa licenza (quindi si deve fornire il codice sorgente), mentre la LGPL permette di utilizzare le librerie pur non distribuendo il codice sorgente del proprio programma. L'opzione GPL è valida per tutte le applicazioni che verranno rilasciate come software libero da programmatori amatoriali, mentre la LGPL è più indicata per aziende che non vogliono rilasciare il programma come free software. La licenza commerciale serve solo nel caso si voglia modificare il codice sorgente delle librerie Qt stesse senza pubblicare il codice delle modifiche.

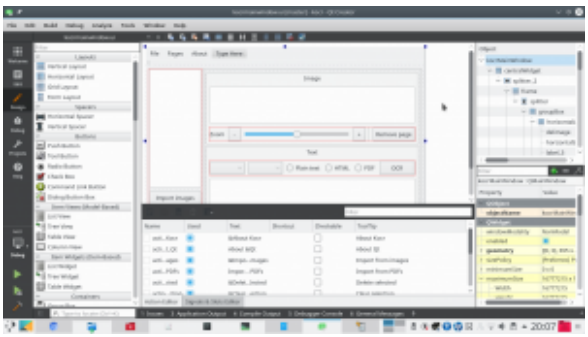
Il programma è scritto in C++ con le librerie multipiattaforma Qt, delle quali ci serviremo per scrivere i PDF usando le funzioni della classe QPDFWriter. Per la lettura dei PDF, invece, utilizzeremo la libreria libera e open source Poppler, che si integra perfettamente con le librerie Qt.

# Come funziona il formato PDF?

Il formato PDF è uno standard ufficiale dal 2007, declinato in una serie di sottoformati: A,X,E,H,UA, a seconda dei vari utilizzi che se ne vogliono fare. Quello che si segue solitamente è il PDF/A, progettato per l'archiviazione dei documenti anche a lungo termine: è pensato per integrare tutti i componenti necessari. Prima che si stabilisse questo standard, infatti, i PDF non erano davvero adatti a conservare e trasmettere documenti, perché mancavano spesso alcuni componenti fondamentali. Per esempio, se un PDF veniva visualizzato su un computer nel quale non erano installati i font con cui sul PC originale era stato scritto il testo, tutta l'impaginazione saltava. Ora, invece, i font possono essere integrati, assieme ad eventuali altri oggetti, così è possibile visualizzare correttamente un PDF/A su qualsiasi dispositivo, a prescindere dal suo sistema operativo. Questo significa che ogni PDF moderno è di fatto un po' più grande di quanto lo sarebbe stato un PDF degli anni '90, perché porta al suo interno i vari font, ma questo non è un problema considerando che il costo dello spazio dei dischi rigidi diminuisce continuamente e un paio di kilobyte in più in un file non si notano nemmeno.

Il formato PDF nasce da un formato precedente che è tutt'ora in uso e che si chiama PostScript. PostScript è di fatto un linguaggio di programmazione che permette di descrivere delle pagine: i file PS sono dei semplici file di testo che contengono una serie di istruzioni per il disegno di una pagina, con le sue immagini e il testo. Si tratta di un

linguaggio che va interpretato, quindi la sua elaborazione richiede una buona quantità di risorse e di tempo. Un file PDF, invece, è di fatto una sorta di PS già interpretato, il che permette di risparmiare tempo. Per fare un esempio, in un file PS si troveranno molte condizioni “if” e cicli “loop”, e si tratta di istruzioni che consumano molte risorse quando vanno interpretate. Nei PDF, invece, viene direttamente inserito il risultato dei vari cicli, così da risparmiare tempo durante la visualizzazione. Quello che è importante capire è che il formato PDF è progettato per la stampa, è pensato per essere facilmente visualizzato e stampato allo stesso modo su qualsiasi dispositivo. Insomma, una funzione di sola lettura. Non è affatto progettato per permettere la continua modifica dei file. Ciò non significa che sia proibito, i file PDF possono ovviamente essere modificati come qualsiasi altro file, ma la modifica può essere molto complicata da fare in certi casi proprio perché le informazioni vengono memorizzate puntando a massimizzare l’efficienza della lettura, non della scrittura o della modifica. Per esempio, i testi vengono memorizzati una riga alla volta, e non in blocchi di paragrafi o colonne, come invece risulterebbe comodo per modificarli successivamente. Un’altra differenza importante è che nei PDF ogni pagina è un elemento a se stante, mentre nei PostScript le pagine sono legate e condividono alcune caratteristiche (come le dimensioni).



Utilizzando l’IDE gratuito QtCreator è molto facile anche disegnare

l'interfaccia grafica multipiattaforma

# Includere Poppler

Cominciamo subito col nostro programma di esempio. Le librerie necessarie possono essere incluse nell'intestazione del codice come da prassi del C++. Quelle che servono per la gestione dei PDF sono le seguenti:

Per scrivere i PDF utilizzeremo infatti la libreria QpdfWriter, che si trova nella stessa cartella di tutte le altre librerie Qt, e che quindi viene trovata in automatico dall'IDE. Per la lettura dei PDF, invece, useremo Poppler, che va installata a parte. Qui le cose cambiano un po', perché mentre in GNU/Linux esiste un percorso standard nel quale installare le librerie, e quindi si può facilmente trovare poppler nella cartella **poppler/qt5/**, su Windows questo non esiste. Quindi, sfruttando gli **ifdefine** forniti dalle librerie Qt, possiamo distinguere la posizione dei file che contengono la libreria Poppler a seconda del fatto che il sistema sia Windows (**Q_OS_WIN**) o GNU/Linux (**Q_OS_LINUX**). La posizione delle librerie per Windows potrà essere stabilita nel file di progetto, che vedremo più avanti. Possiamo ora cominciare a vedere il codice: non lo vedremo tutto, solo le parti fondamentali per la gestione dei PDF.

Entro breve, le librerie Qt integreranno direttamente una classe per la lettura dei PDF, chiamata QPDFDocument, senza quindi la necessità di usare Poppler. Al momento tale classe non è ancora considerata stabile, quindi abbiamo deciso di presentare questo articolo basandoci ancora su Poppler. Quando

il rilascio di QtPdf sarà ufficiale, la presenteremo in nuovo articolo.

La prima funzione che implementiamo dovrà permettere l'importazione dei PDF. Infatti, vogliamo permettere agli utenti di importare dei PDF scansionati, in modo da poter eseguire su di essi l'OCR e ricavare il testo.

Chiamando la funzione **getOpenFileNames** di **QFileDialog** si visualizza una finestra standard per consentire all'utente la selezione di più file, il cui percorso completo viene inserito in una lista di stringhe che chiamiamo **files**.

Possiamo anche creare una MessageBox per chiede conferma all'utente, così se dovesse avere scelto i file per sbaglio potrà annullare il procedimento prima di cominciare a lavorare sui file (operazione che può richiedere del tempo).

Banalmente, se il pulsante premuto dall'utente è **Cancel**, allora interrompiamo la funzione. Altrimenti, con un semplice ciclo for scorriamo tutti gli elementi della lista di file, passandoli uno alla volta a un funzione che si occuperà di estrarre le pagine dal PDF e aggiungerle alla lista delle pagine su cui lavorare.

# Aprire un PDF in lettura

Abbiamo chiamato la funzione che opera effettivamente l'estrazione delle pagine da un PDF, **addpdftolist**.

Questa funzione comincia controllando che il file che ha ricevuto come argomento **pdfin** sia esistente (**QfileInfo.exists** controlla che il file esista e non sia vuoto).

Ora abbiamo bisogno di una cartella temporanea, nella quale inserire tutte le immagini che estrarremo dalle pagine del PDF. La libreria **QTemporaryDir** si occupa proprio di creare una cartella temporanea a prescindere dal sistema operativo. Possiamo memorizzare il percorso di tale cartella in una stringa che chiamiamo **tmpdir**. Dobbiamo anche specificare che la cartella non va sottoposta all'auto rimozione, altrimenti il programma cancellerà la cartella automaticamente al termine di questa funzione, mentre noi ne avremo ancora bisogno in altre funzioni. La cancellazione di tale cartella potrà essere fatta manualmente alla chiusura definitiva del programma.

Siamo finalmente pronti per leggere il PDF. Basta creare un oggetto di tipo **Poppler::Document**, usando la funzione load che permette per l'appunto la lettura di un file PDF. Se il PDF non conteneva un documento valido, conviene terminare la funzione con l'istruzione return per evitare problemi.

Il documento potrebbe avere più pagine, quindi utilizziamo un ciclo for per leggerle tutte una alla volta.

Ogni pagina può essere estratta usando un oggetto **Poppler::Page**, e con l'apposita funzione **page** di un documento. Se la pagina è invalida, il ciclo si ferma.

La pagina può poi essere renderizzata in una immagine, rappresentata dall'oggetto **QImage**, secondo una carta risoluzione orizzontale e verticale (che di solito coincidono, ma non sempre).


Poppler permette di leggere tutti i componenti di un PDF, annotazioni incluse

Nel nostro programma, consideriamo tale risoluzione pari a 300 dpi, e l'abbiamo inserita in una apposita variabile all'inizio del programma chiamata per l'appunto **dpi**.

Per salvare l'immagine della pagina basta chiamare la funzione **save** dell'immagine. Tuttavia, prima dobbiamo decidere il nome del file: sarà ovviamente composto dal percorso della cartella temporanea più il nome **tmppage** seguito dal numero progressivo della pagina e l'estensione **tiff**. Il numero della pagina viene scritto con 4 cifre, giustificando con lo 0. Quindi, la pagina 1 sarà **tmppage0001**, mentre la pagina 23 sarà **tmppage0023**, e la 145 sarà **tmppage0145**. In questo modo siamo sicuri di non confondere mai l'ordine delle pagine.

È bene ricordarsi di eliminare l'oggetto pagina e il documento, per liberare la memoria (lavorando ad alte risoluzioni è facile che venga richiesta molta RAM per svolgere queste operazioni).

Avremmo potuto inserire direttamente ogni immagine estratta nell'elenco delle immagini che vogliamo passare all'OCR, ma possiamo anche semplicemente leggere il contenuto della cartella temporanea cercando tutti i file che contiene e, scorrendoli uno ad uno, passare il loro percorso alla funzione **addimagestolist**. È infatti questa la funzione che si occuperà di inserire le singole immagini nella lista. Chiamando la

funzione soltanto dopo l'estrazione delle pagine, le immagini appariranno nell'interfaccia grafica tutte assieme e l'utente capirà che la procedura è terminata.

La funzione in questione è molto semplice: viene creato un nuovo elemento del qlistwidget (l'oggetto che nell'interfaccia grafica del nostro programma funge da elenco delle pagine). All'elemento viene assegnata una icona, che proviene dal file stesso e che quindi costituirà la sua anteprima. L'elemento viene infine aggiunto all'oggetto presente nell'interfaccia grafica (**ui**).

**Il file di progetto**

Per consentire al compilatore di trovare la libreria Poppler basta inserire nel file di progetto (.pro) le seguenti righe:

In questo modo il compilatore saprà che su Windows i file .h si troveranno nella cartella **include/poppler-qt5** del codice sorgente, mentre la libreria compilata sarà nella cartella **lib**.

# Fusione dei PDF

Dopo avere eseguito l'OCR sulle varie pagine, si ottengono da Tesseract tanti PDF quante sono per l'appunto le pagine del documento. Ciò significa che dovremo riunirle manualmente, fondendo assieme tutti i vari file in un unico PDF. Per farlo, prima di tutto decidiamo il nome di un file temporaneo nel quale riunire tutti i PDF:

Lo facciamo sfruttando lo stesso meccanismo che abbiamo usato per la cartella temporanea, ma con la libreria **QTemporaryFile**. Ovviamente, il file dovrà avere estensione **pdf**, e il suo nome è contenuto nella variabile **tmpfilename**.

Per scrivere sul PDF temporaneo, basta creare un nuovo oggetto di tipo QpdfWriter associato al file e un oggetto **Qpainter** associato al **pdfWriter**. Il QPainter è il disegnatore che si occuperà di, per l'appunto, disegnare il contenuto del PDF secondo le nostre indicazioni.

Le varie pagine, cioè i pdf da riunire, si trovano nella stringa **allpages** separati dal simbolo **|**. Con un semplice ciclo for possiamo prendere un pdf alla volta, inserendo il suo nome nella stringa **inp**.

Se quello su cui stiamo lavorando non è il primo dei file da unire (quindi il contatore delle pagine **i** è maggiore di 0), allora possiamo inserire una interruzione di pagina nel PDF finale con la funzione **newPage**. Questo ci permette di unire i vari file dedicando una nuova pagina a ciascuno.

Possiamo quindi aprire il pdf usando, come già visto, **Poppler::Document**. Con un ciclo for scorriamo le varie pagine: ciascuno dei file da unire dovrebbe contenere una sola pagina, ma è comunque più prudente usare un ciclo per non correre rischi.

# Le varie TextBox

Ora dobbiamo estrarre il testo della pagina, cioè il testo che Tesseract ha inserito grazie alla funzione di OCR.

Potremmo semplicemente prelevare il testo con la funzione

**text**, ma preferiamo usare **textList**. Infatti, la prima ci fornisce semplicemente tutto il testo della pagina, ma a noi questo non va bene: abbiamo bisogno di avere anche l'esatta posizione, nella pagina, di ogni parola. Per questo esiste **textList**, una lista di **Poppler::TextBox**, dei rettangoli che contengono il testo e hanno una precisa posizione e dimensione.

Con un ulteriore ciclo for possiamo scorrere tutte le **textBox** ottenendo il rettangolo (**QrectF** è un rettangolo con dimensioni float) che le rappresenta usando la funzione **boundingBox**.

Ora c'è un piccolo problema: le dimensioni e la posizione del rettangolo sono state indicate, da Poppler, con il sistema di riferimento della pagina che stiamo leggendo. Invece, il nostro pdfWriter avrà probabilmente un sistema di riferimento diverso, a causa della risoluzione. Possiamo calcolare il rapporto orizzontale e verticale semplicemente dividendo larghezza e altezza della pagina di pdfWriter per quelle della pagina di Poppler.

Adesso possiamo tranquillamente scrivere il testo usando il nuovo rettangolo, che abbiamo appena calcolato, come riferimento. Il testo (attributo **text** della **textBox** attuale) si aggiunge usando la funzione drawText del **painter**.

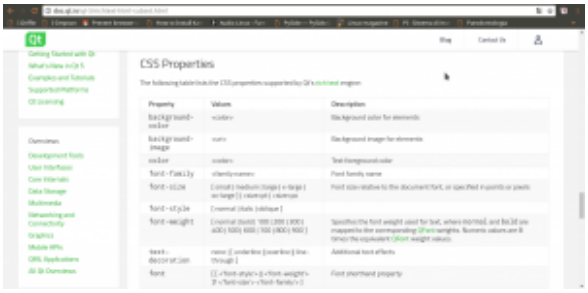
Soltanto dopo avere terminato questo ciclo for, e quindi avere scritto tutti i testi dove necessario, possiamo disegnare sulla pagina l'immagine di sfondo, con la funzione drawPixmap che si usa per inserire in un **painter** una immagine a mappa di pixel (una bitmap qualsiasi). La pixmap è ovviamente ottenuta dall'immagine che preleviamo tramite Poppler usando la già vista funzione **renderToImage**. Inserendo l'immagine dopo il testo, siamo sicuri che sarà visibile soltanto l'immagine, e il testo risulterà invisibile ma ovviamente selezionabile e

ricercabile. In alternativa avremmo anche potuto scegliere il colore “trasparente” per il testo.

Ovviamente, quando abbiamo finito di leggere un file, dobbiamo eliminare il suo oggetto **document** per non occupare troppo spazio. Per quanto riguarda il PDF che stiamo scrivendo, non c’è bisogno di chiudere il file: QpdfWriter lo farà automaticamente appena la funzione termina.

# Scrivere dell’HTML

C’è ancora un ultimo caso da considerare: se invece di Tesseract si vuole utilizzare l’OCR Cuneiform su Windows, purtroppo non si ottiene un PDF e nemmeno un file HOCR (cioè un HTML con la posizione delle varie parole). Si ottiene soltanto un semplice file HTML, che mantiene la formattazione ma non la posizione delle parole.




Un testo formattato può essere inserito in un PDF con QTextDocument usando la formattazione CSS delle pagine HTML (http://doc.qt.io/qt-5/richtext-html-subset.html)

Non è ottimale, ma può comunque essere utile avere un PDF che contenga il testo nella pagina, così lo si può ricercare facilmente. In questo caso, la prima cosa da fare è leggere il file html che si ottiene:

Leggendo il file come semplice testo grazie alle librerie QFile e QTextStream, possiamo inserire tutto il codice nella stringa **hocr**.

Ora, possiamo creare un nuovo documento di testo formattato, usando la libreria QTextDocument. Il contenuto del testo sarà indicato proprio dal codice **html** della stringa hocr, che quindi mantiene la formattazione. Impostiamo anche la larghezza massima del testo pari a quella della pagina di **pdfWriter**.

Come prima, dovremo calcolare la corretta dimensione con cui inserire il testo, per evitare che sia troppo piccolo o troppo grande. Siccome stavolta è solo testo, possiamo calcolare la dimensione del font con cui scriverlo usando una proporzione.

Dopo avere scelto la giusta dimensione del testo affinché riempia tutta la pagina, possiamo inserire il testo nel painter, e quindi nel PDF, usando la funzione drawContents del QTextDocument. Il vantaggio d i questa funzione, rispetto a drawText, è che in questo modo si mantiene la formattazione e l'allineamento standard HTML.

Ovviamente, anche in questo caso si conclude la pagina inserendo sopra al testo l'immagine della pagina stessa, così il testo non sarà visibile, ma comunque ricercabile e selezionabile.

# Il codice sorgente e il binario dell'esempio

Per capire come venga organizzato il codice sorgente, vi conviene controllare quello del nostro programma di esempio. Banalmente, il programma è composto da un file di progetto, un

file **main.cpp** che costituisce la base dell'eseguibile, e due file (uno .h e uno .cpp) per la classe **mainwindow**, che rappresenta l'interfaccia principale del programma. Inoltre, abbiamo inserito due cartelle con il codice sorgente e il codice binario della libreria Poppler per Windows.

Trovate tutto il codice su GitHub assieme a dei pacchetti precompilati per Windows e GNU/Linux: **https://github.com/zorbaproject/kocr/releases**