Formation Raspberry Pi – Python
Régulation de température d'un chauffage électrique via Internet

Maquette de démo : 1 raspberry pi (connecté au réseau local) + 1 carte thermomètre/thermostat DS1631 avec relais + une ampoule 24/40W (en guise de chauffage électrique)

Sommaire

Préambule

1. Le thermomètre/thermostat Dallas DS1631

2. Connexion du DS1631 au GPIO du Raspberry pi

3. Configuration du Raspberry pi pour l'utilisation du bus i2c

4. Logiciels nécessaires

a) Python 3
modules standards `sys`, `time`, `datetime`, `tkinter`, `sqlite3`, `threading` et `configparser`
b) Modules externes de python à installer
`smbus`, `DS1631`, `matplotlib`, `flask`
c) ipython3 (an enhanced Interactive Python)
d) Geany (IDE Python)
e) DB Browser for SQLite
f) Firefox

Travail à réaliser

Partie A – Application locale

A.1. Lecture de la température

A.1.1. Avec un interpréteur python
A.1.2. Premier script en mode console
A.1.3. En mode graphique avec le module tkinter

A.2. Script de lecture périodique de la température

A.2.1. Script en mode console
A.2.2. Script en mode graphique

A.3. Lecture et écriture des températures de consigne du thermostat

A.3.1. Avec un interpréteur python
A.3.2. Script en mode console
A.3.3. Script en mode graphique

A.4. Base de données SQLite

A.4.1. Premiers pas avec SQLite
A.4.2. Sauvegarde périodique des températures dans une base de données

A.5. Enregistrement périodique du fichier image de la courbe de température

A.6. Graphe déroulant de la température

Partie B – Application web

B.1. Qu'est-ce qu'un serveur web ?

B.2. Introduction au framework web flask : Hello world !

B.3. Site web « Régulation de température » : squelette du site

B.4. Page d'accueil

B.4.1. Page d'accueil « statique »
B.4.2. Page d'accueil dynamique avec AJAX
B.4.2.1. Une page web au format JSON
B.4.2.2. Actualisation de la température toutes les secondes dans la page d'accueil
B.4.2.3. Qu'est-ce qu'AJAX (Asynchronous JavaScript and XML) ?
B.4.2.4. Explication du code du template *accueil_ajax.html*
B.4.2.5. L'outil réseau de Firefox

B.5. Les pages du thermostat

B.6. Page Historique des températures (tableau dynamique)

B.7. Page Historique des températures (tableau et graphe dynamiques) avec mise en œuvre d'un thread

B.7.1. Rappel sur les threads
B.7.2. Mise en œuvre dans la page Historique des températures

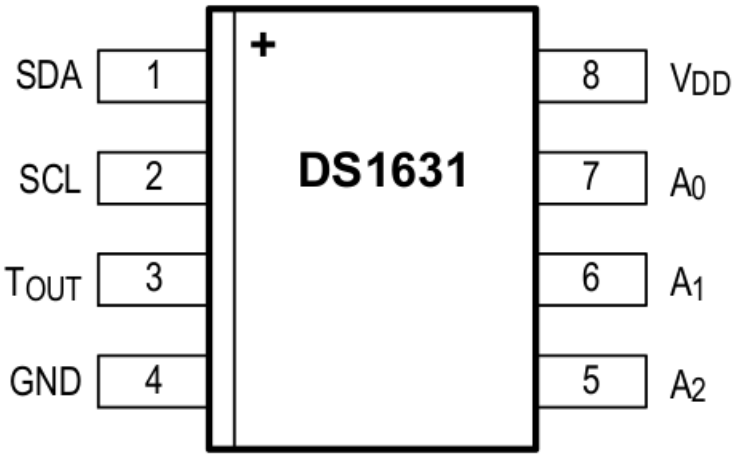
B.8. Version multi-capteurs

B.8.1. Généralisation à plusieurs capteurs
B.8.2. Petite amélioration
B.8.3. Fichier de configuration
B.8.4. Consommation électrique

B.9. Complément pour les électroniciens : analyse des trames i2c

1. Le thermomètre/thermostat Dallas DS1631

Il s’agit d’un circuit intégré 8 broches :



Le thermomètre/thermostat DS1631 (Dallas Semiconductor) fait parti de la famille des capteurs "intelligents" : sur la même puce, il y a un capteur de température classique associé à une électronique d'interface (convertisseur analogique - numérique, contrôleur avec son jeu d'instructions, EEPROM, port série synchrone : bus I2C).

- Fonction thermomètre

Le DS1631 est un **thermomètre numérique** : plage de mesure - 55,0 °C à + 125,0° C avec une résolution que l'on peut choisir de 9 bits (0,5 °C) à 12 bits (0,0625 °C).

La température est fournie sous la forme d'un nombre binaire en complément à deux.

La durée de la conversion dépend de la résolution : 93,75 ms avec 9 bits, 750 ms avec 12 bits.

Le DS1631 s'interface avec un bus I2C (broches SDA et SCL), en configuration esclave :

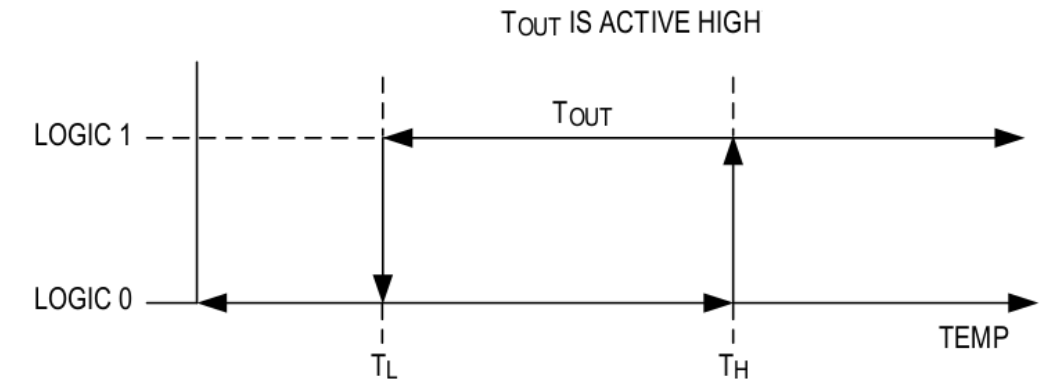
- L'adresse I2C (7 bits) du DS1631 est : 1 0 0 1 A2 A1 A0
    - A2, A1 et A0 correspondent aux niveaux logiques appliqués à ces 3 entrées
    - On peut donc connecter jusqu'à 8 boîtiers DS1631 sur un bus I2C (plage d’adresses **0x48** à **0x4F**)

La précision de la mesure (à ne pas confondre avec la résolution) est de 0,5 °C dans la plage 0 °C à +70 °C.

- Fonction thermostat

Le DS1631 intègre également un **thermostat** avec 2 températures de consigne configurables (TH et TL).
Ces deux températures de consigne sont stockées dans une mémoire permanente de type EEPROM.

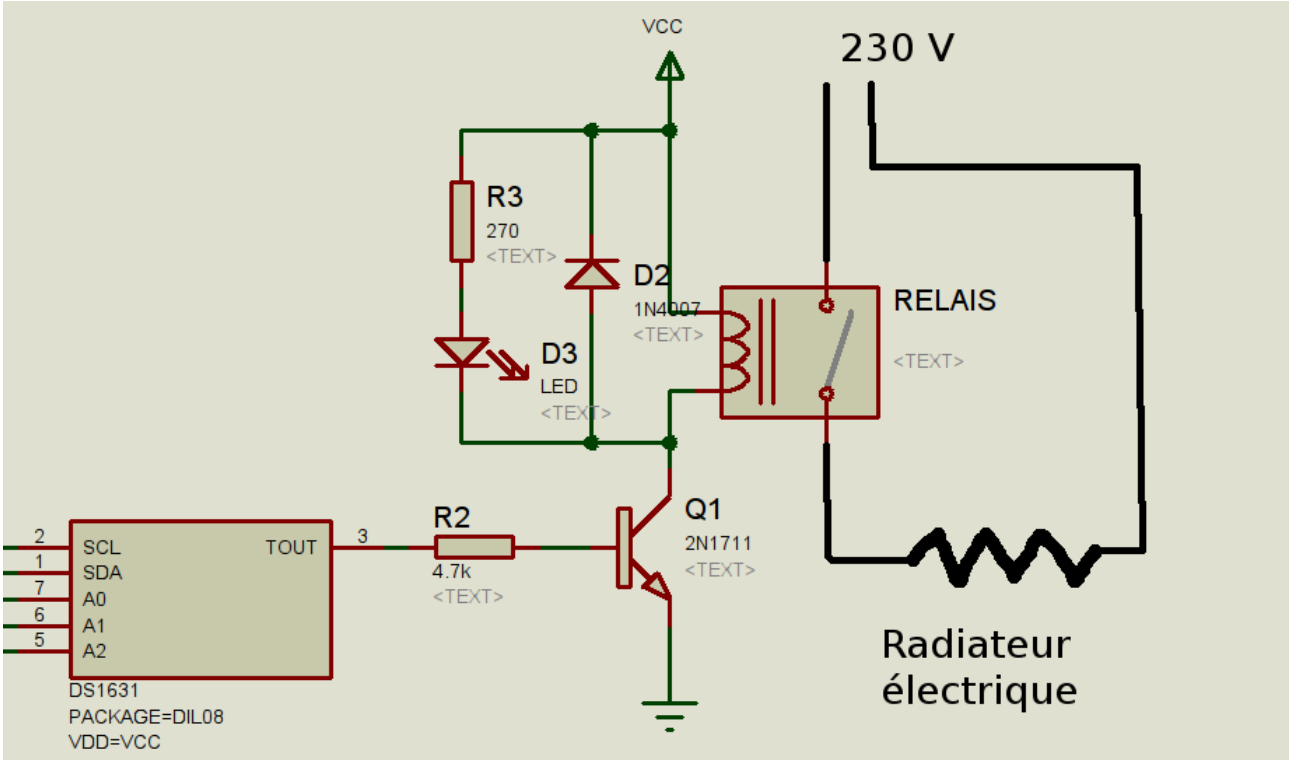
Le niveau de la broche de sortie Tout suit le cycle d’hystérésis suivant :



En configuration « active low », le niveau logique est inversé.

Application pratique : régulation de température d'un chauffage électrique

La broche Tout sert d'entrée au système de commande des radiateurs électriques :



Si Tout = 1 alors le radiateur est sous tension.
Si Tout = 0 alors le radiateur est hors tension.
Pour une régulation de température, avec un relais normalement ouvert, il faut donc une configuration « active low » :

T > TH : Tout = 0
T < TL : Tout = 1

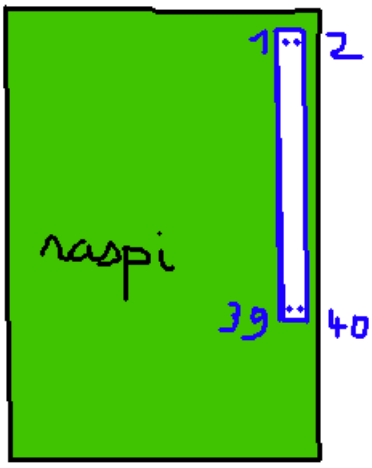
L'état du radiateur est visualisé par la LED.

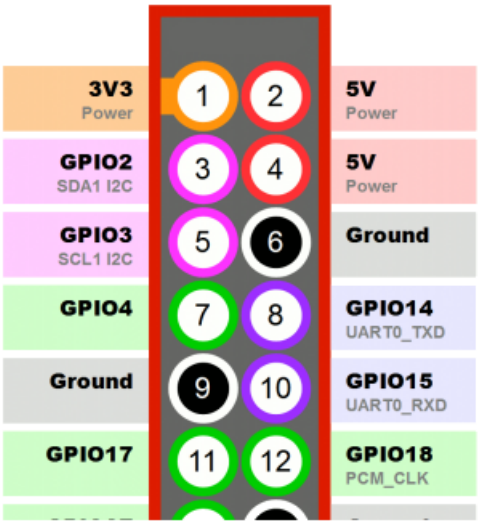
Bon à savoir : sur le site web de maxim-dallas, vous pouvez commander des échantillons gratuits de DS1631, DS1621, DS18B20 etc.

2. Connexion du DS1631 au GPIO du Raspberry Pi

- Le bus i2c du Raspberry pi

Caractéristiques électriques 3,3 V
Résistances de pull-up (1,8 kΩ)
Fréquence de l'horloge 100 kHz





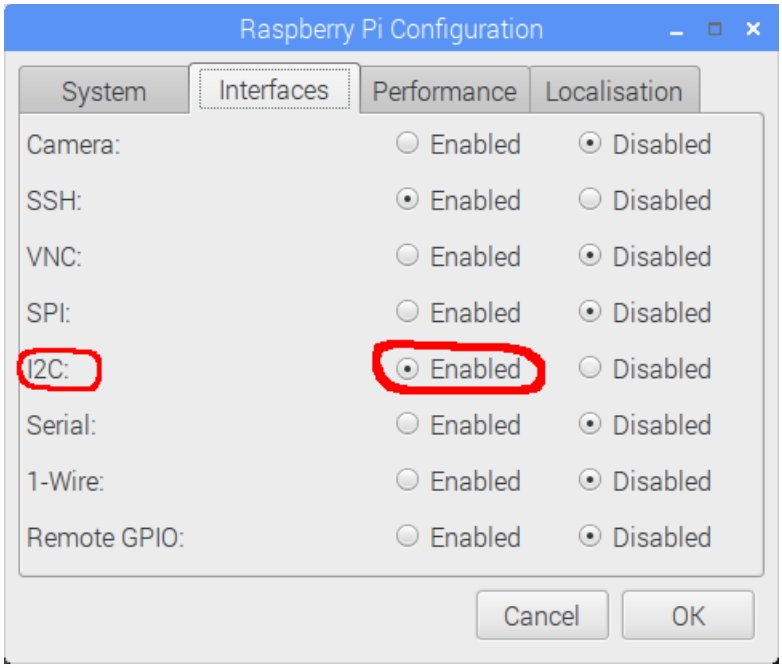
Il y a 4 fils à relier entre le GPIO et le DS1631 : [A FAIRE, hors tension SVP]

- 2 fils pour le bus i2c : SDA et SCL
- 2 fils d'alimentation : GND (0 V) et VDD (+5 V ou +3,3 V)

3. Configuration du Raspberry pi pour l'utilisation du bus i2c

Maintenant que la connexion électrique a été réalisée, on peut démarrer le Raspberry pi.

Avec la distribution Linux **Raspbian Jessie** :
Menu → Préférences → Configuration du Raspberry Pi → Interfaces [A FAIRE]



Le paquet **i2c-tools** de Linux fournit des utilitaires de communication avec le bus i2c
Installation :
Dans un terminal Linux :
`$ sudo apt install i2c-tools` [déjà fait]

`$ sudo i2cdetect -y 1` [A faire]

```
pi@raspberrypi:~$ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- 48 -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Point test :
Si tout se passe bien, vous devez voir apparaître l'adresse i2c du DS1631 (entre 0x48 et 0x4F suivant la configuration des broches A0, A1 et A2).

4. Logiciels nécessaires

Cible : Raspberry pi + OS **Raspbian Jessie**

a) Python 3 [rien à faire]

Dans toute la suite, nous utiliserons **Python dans sa version 3**.

Par défaut, la distribution Raspbian Jessie inclut la version 3.4 de Python.

Nous aurons besoin des modules standards `sys`, `time`, `datetime`, `tkinter`, `sqlite3`, `threading` et `configparser`.

b) Modules externes de Python 3 à installer [déjà fait]

Module `smbus` : communication avec le bus i2c du Raspberry pi
Module `DS1631` : c'est un module que j'ai écrit et qui permet de gérer la communication entre le Raspberry pi et les capteurs de la famille Dallas DS1631. Ce module s'appuie sur le module `smbus`.
Module `matplotlib` : courbes et animations en 2D et 3D
Module `flask` : framework web

```
sudo apt install python3-smbus
sudo apt install python3-matplotlib
sudo apt install python3-flask
```

On peut aussi passer par `pip`, le gestionnaire de modules de python (notamment pour faire les mises à jour).

```
sudo apt install python3-pip
```

```
sudo pip3 install -U pip
(ou : sudo python3 -m pip install -U pip)
```

```
sudo pip3 install -U DS1631
sudo pip3 install -U flask
sudo pip3 install -U matplotlib
```

c) ipython3 (an enhanced Interactive Python)

Il s'agit d'un célèbre interpréteur Python aux fonctionnalités avancées.

Installation :
`sudo apt install python3-ipython` [déjà fait]

d) Geany (IDE Python)

Il s'agit d'un IDE Python, similaire au Notepad++ de Windows.

Installation :
`sudo apt-get install geany` [déjà fait]

e) DB Browser for SQLite (SQLiteBrowser)

Une application de gestion et d'administration de base de données SQLite.
Pour ceux qui connaissent MySQL, c'est l'équivalent de PhpMyadmin ou de MySQL Workbench.

Installation :
`sudo apt-get install sqlitebrowser` [déjà fait]

f) Firefox

Le célèbre navigateur web de la fondation Mozilla.

Installation :
`sudo apt-get install firefox-esr` [déjà fait]

Travail à réaliser

Sur le bureau du Raspberry pi, commencer par créer un dossier */travail* (s'il existe déjà, vider son contenu).

Partie A – Application locale

A.1. Lecture de la température

A.1.1. Avec un interpréteur python

Nous allons mettre en œuvre le module *DS1631*.
Une aide sur le module est fournie par l'outil *pydoc* (the Python documentation tool) :
Dans un terminal Linux :

```
$ pydoc3 -p 9000
```

Puis ouvrir Firefox avec l'URL :
http://localhost:9000
http://localhost:9000/get?key=DS1631

Dans un terminal Linux, lancer l'interpréteur *ipython3* :

```
$ ipython3

In [1]: import DS1631

In [2]: ic1 = DS1631.DS1631(1, 0x48)  # avec la bonne adresse i2c

In [3]: ic1.set_conversion_mode("continuous")

In [4]: ic1.set_resolution(9)

In [5]: ic1.start_convert()

In [6]: print(ic1.get_temperature())

In [7]: ic1.set_resolution(12)

In [8]: print(ic1.get_temperature())

In [9]: print(ic1.get_temperature())
```

A tester !

Bravo, vous venez de faire de la programmation orientée objet.
Un peu de vocabulaire :

```
ic1 = DS1631.DS1631(1, 0x48)
```

`ic1` est une instance de la classe `DS1631` du module *DS1631*.

On dit aussi que `ic1` est un objet de la classe `DS1631`.

Une classe possède des fonctions que l'on appelle **méthodes** et des données que l'on appelle **attributs**.

```
In [10]: ic1.get_temperature?    # pour obtenir des informations sur la méthode
```

La méthode `get_temperature()` retourne la température sous la forme d'un nombre à virgule flottante (type `float`), que vous avez parfaitement le droit de stocker dans une variable :

```
In [11]: t1 = ic1.get_temperature()
In [12]: print(t1)
```

Si vous avez un deuxième DS1631 sur votre bus i2c, il suffit d'instancier un nouvel objet de la classe DS1631 :

```
In [13]: ic2 = DS1631.DS1631(1, 0x4F)  # avec une adresse différente
```

Autrement, vous disposez d'un nombre illimité de DS1631 virtuels (classe `DS1631virtualdevice` du module `DS1631`) :

```
In [14]: ic3 = DS1631.DS1631virtualdevice(initial_temperature=22)  # à tester
```

Pour des raisons pratiques, vous pouvez indexer vos différents DS1631 dans une liste (une liste d'objets) :

```
In [15]: capteurs = [ic1, ic3]     # type list, entre crochets
```

`capteurs[0].get_temperature()`
aura alors la même signification que :
`ic1.get_temperature()`

Pour lire la température de tous nos capteurs, une boucle `for` s'impose :

```
In [16]: for capteur in capteurs :
    print(capteur.get_temperature())   # avec l'indentation
```

Pour les électroniciens curieux, vous pouvez éditer avec Geany le code source du module `DS1631`, constitué du seul fichier *DS1631.py* (une copie est disponible dans le dossier */ressources*, ou sur le web *https://pypi.org/project/DS1631*).
Par la suite, ce fichier pourra vous servir de modèle pour créer vos propres classes de communication avec le bus i2c.
On pourra éviter de réinventer la roue en faisant au préalable une recherche sur pypi *https://pypi.org*

Remarque : le module `DS1631` est écrit en « pur » Python.
Mais il faut savoir qu'un module Python intègre souvent des extensions écrites en langage C.
C'est par exemple le cas du module `ds18b20pi` (capteur de température Dallas DS18B20, sur un bus 1-Wire) que vous pouvez utiliser sur votre Raspberry pi (*https://pypi.org/project/ds18b20pi*).

A.1.2. Premier script en mode console

- **IDE Python**

Nous faisons le choix d'utiliser Geany.

Ouvrir l'éditeur Geany

Éditer → Préférences
Éditeur → Indentation :
Largeur : 4
Type : Espaces
Fichier → Encodage : Unicode (UTF-8)
Valider
Outils → Gestionnaire de plugin
Diviser la fenêtre à activer

Fichier → Nouveau
Document → Définir le type de fichier → Fichier source Python
Document → Indentation automatique à activer
Construire → Définir les commandes de construction :
Commandes d'exécution : Execute python3 "%f"

Fichier → Enregistrer sous :
dossier */home/pi/Desktop/travail*
avec le nom ***lecture_temperature.py***

Script à réaliser

Le script doit lire et afficher la température de votre thermomètre (supposé unique).
Par exemple :

```
+20.5 °C
Enter to exit...
```

Pour définir l'encodage du fichier (1ère ligne du script, ça commence par # mais ce n'est pas un commentaire) :

```
# -*- coding: utf-8 -*-

# à compléter

# fin du programme
input("Enter to exit...")
```

Pour exécuter le script :
Construire → Exécuter avec Python 3 (ou touche F5)

A propos des erreurs d'indentation

Si vous mélangez tabulation et espaces pour indenter votre code, vous risquez d'obtenir une erreur :

```
IndentationError: unexpected indent
```

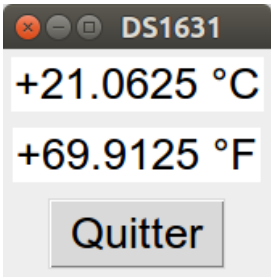
Une solution :
Document → Remplacer les tabulations pour des espaces

A.1.3. En mode graphique avec le module `tkinter`

Le module `tkinter` permet de réaliser des interfaces graphiques de manière relativement simple.

Dans le dossier */ressources/partieA/A13*, tester le script ***lecture_temperature_GUI.py***

Compléter le script ***lecture_temperature_fahrenheit_GUI_ACOMPLETER.py*** de manière à obtenir le résultat suivant :



A.2. Scripts de lecture périodique de la température

A.2.1. Script en mode console

Dans le dossier */home/pi/Desktop/travail*, créer le script ***lecture_temperature_en_boucle.py***

Ce script doit lire la température toutes les 0,75 seconde (durée de conversion pour une résolution de 12 bits).

On pourrait utiliser la fonction `sleep` du module `time` pour faire une pause :

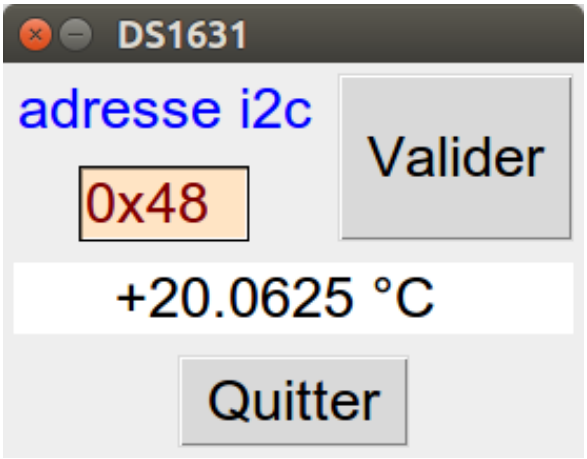
```
import time
time.sleep(0.75)
```

Exemple de rendu :

```
CTRL+C to skip
wait...
+19.9375 °C
wait...
+19.9375 °C
wait...
+20.0 °C
wait...
+20.0 °C
```

A.2.2. Script en mode graphique avec le module `tkinter`

Dans le dossier */ressources/partieA/A22*, tester le script ***lecture_temperature_en_boucle_GUI.py***



La temporisation est réalisée avec la méthode `after()` de `tkinter` (ligne 79).

Ligne 40, vous pouvez modifier la résolution, et observer l'influence sur la durée de conversion :
`ic1.set_resolution(9)  # 9 à 12 bits`

A.3. Lecture et écriture des températures de consigne du thermostat

A.3.1. Avec un interpréteur Python

Tester les commandes suivantes.
Observer l'état de la LED qui visualise l'état du radiateur.

```
$ ipython3

In [1]: import DS1631

In [2]: ic1 = DS1631.DS1631(1, 0x48)

In [3]: ic1.set_tout_polarity("active-low")  # ou "active-high"

In [4]: ic1.set_conversion_mode("continuous")

In [5]: ic1.set_resolution(12)

In [6]: ic1.start_convert()

In [7]: print(ic1.get_thigh())

In [8]: print(ic1.get_tlow())

In [9]: ic1.set_thigh(25.5)

In [10]: ic1.set_tlow(23.5)

In [11]: print(ic1.get_thigh())

In [12]: print(ic1.get_tlow())
```

A.3.2. Script en mode console

Dans le dossier */home/pi/Desktop/travail*, créer le script ***modification_thermostat.py***

Ce script doit permettre la modification des températures de consigne du thermostat.

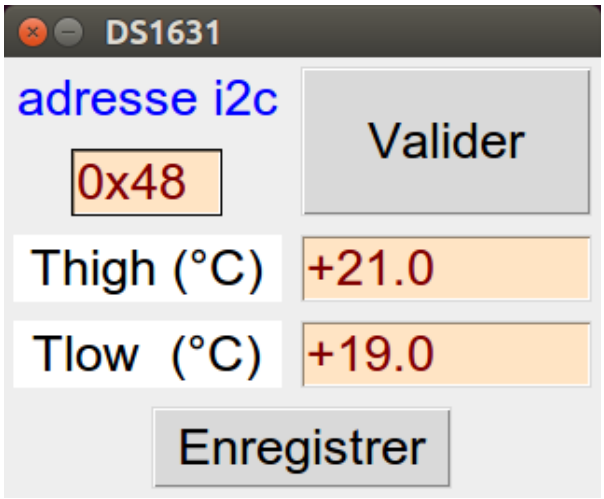
Exemple :

```
Lecture des températures de consigne
Thigh : +21.0 °C
Tlow  : +19.0 °C

Saisie des nouvelles températures de consigne
Th (°C) ? 25
Tl (°C) ? 23
Enter to exit...
```

Une correction est disponible dans le dossier */ressources/partieA/A32*.

A.3.3. Script en mode graphique

Dans le dossier */ressources/partieA/A33*, tester le script ***modification_thermostat_GUI.py***



Si nécessaire, modifier la ligne 42 :
`ic1.set_tout_polarity("active-low")`

A.4. Base de données SQLite

Lire des températures c'est bien, les enregistrer c'est mieux.
Nous n'allons pas utiliser un simple fichier texte (au format csv par exemple), mais le moteur de base de données SQLite.

SQLite fait parti de la famille des bases de données relationnelles (MySQL, MariaDB, PostgreSQL…).

Contrairement à ses consœurs, SQLite n'utilise pas le concept client/serveur.

Une base de données SQLite est intégralement contenue dans un fichier unique.

L'accès à la base de données se fait avec des requêtes en langage SQL.

La plupart des langages de programmation possèdent une librairie dédiée : en Python, c'est le module standard `sqlite3`.

A.4.1. Premiers pas avec SQLite

Nous allons créer une base de données qui contient la table suivante :

Table **amis**

id	prenom	telephone
1	Guy	0618759875
2	Fabrice	0702198651
3	Wilfrid	0689887204

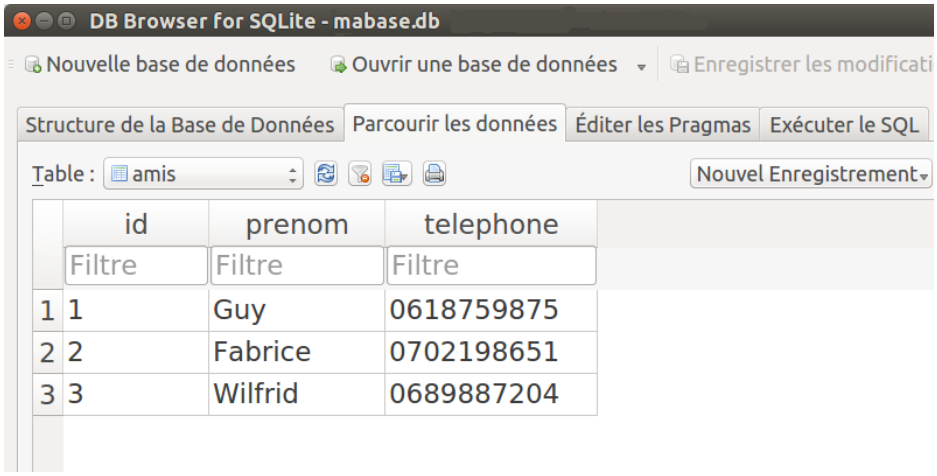
Cette table possède 3 champs : **id**, **prenom**, **telephone**
id est ici une « clé primaire » : c'est un champ particulier qui sert à identifier de façon unique un enregistrement (= ligne, = entrée).

Ouvrons l'interpréteur ipython3 :

```
$ ipython3

In [1]: cd /home/pi/Desktop/travail

In [2]: import sqlite3

In [3]: conn = sqlite3.connect("mabase.db", isolation_level=None)

In [4]: c = conn.cursor()

In [5]: c.execute('''
CREATE TABLE IF NOT EXISTS amis
(id INTEGER PRIMARY KEY AUTOINCREMENT,
prenom TEXT,
telephone TEXT)''')

In [6]: t = ('Guy', '0618759875',)    # type tuple, parenthèses et virgules

In [7]: c.execute('INSERT INTO amis (prenom, telephone) VALUES (?, ?)', t)

In [8]: t = ('Fabrice', '0702198651',)

In [9]: c.execute('INSERT INTO amis (prenom, telephone) VALUES (?, ?)', t)

In [10]: t = ('Wilfrid', '0689887204',)

In [11]: c.execute('INSERT INTO amis (prenom, telephone) VALUES (?, ?)', t)
```

Vous constaterez qu'un fichier *mabase.db* a été créé dans le répertoire */home/pi/Desktop/travail*
Il s'agit d'un fichier binaire qu'il est inutile et inopportun de manipuler directement.

Par contre, c'est le rôle du logiciel **DB Browser for SQLite (SQLiteBrowser)** :
Fichier → Ouvrir une base de données



Continuons :
Fabrice a changé de numéro de téléphone : 0485623780

```
In [12]: t = ('0485623780', 2,)

In [13]: c.execute('UPDATE amis SET telephone=? WHERE id=?', t)
```

Wilfrid habite maintenant en zone blanche :

```
In [14]: t=(3,)

In [15]: c.execute('DELETE FROM amis WHERE id=?', t)
```

Pour afficher le contenu complet de la table, de manière pythonesque :

```
In [16]: for row in c.execute('SELECT * FROM amis'):
    print(row)
```

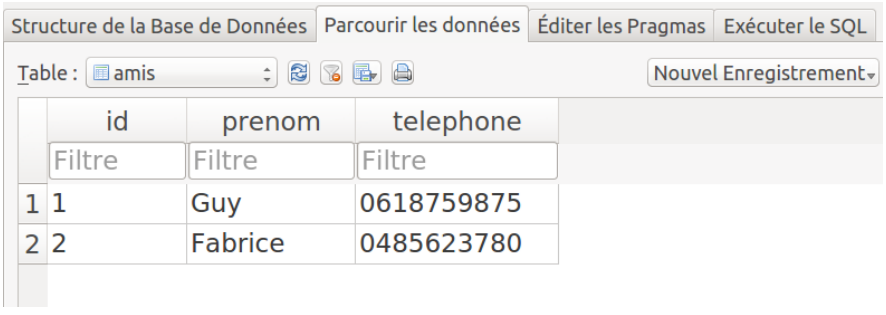
Ce qui doit donner :

```
(1, 'Guy', '0618759875')
(2, 'Fabrice', '0485623780')
```

C'est fini pour l'instant, on pense à fermer la connexion à la base :

```
In [17]: conn.close()
```

Constatons les modifications avec **DB Browser for SQLite** :
Touche F5 pour rafraîchir les données de la table :



A.4.2. Sauvegarde périodique des températures dans une base de données

Dans le dossier */ressources/partieA/A42* , tester le script ***sauvegarde_temperature.py***

Vous pouvez choisir la période de mise à jour de la base de données :
```
period = 5.0
```

et le nom du fichier de la base de données :
```
db_filename = "db/data002.db"
```

La base contient une table **ic1** avec 4 champs :

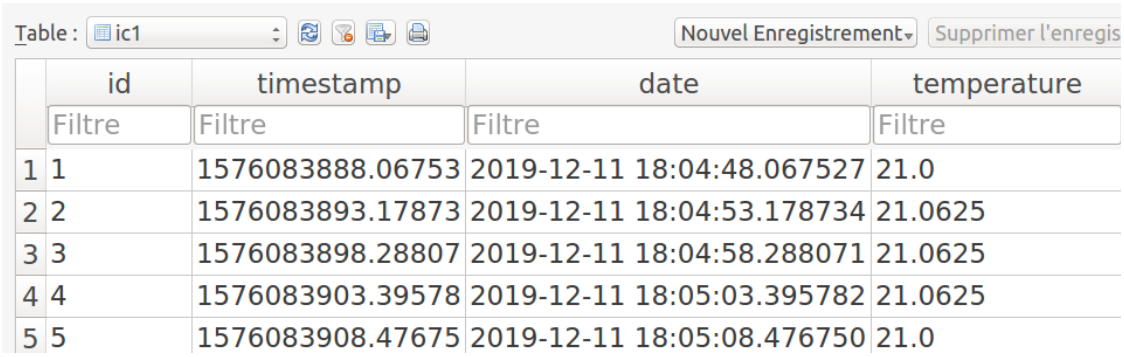


Table : ic1 — Nouvel Enregistrement — Supprimer l'enregis

	id	timestamp	date	temperature
	Filtre	Filtre	Filtre	Filtre
1	1	1576083888.06753	2019-12-11 18:04:48.067527	21.0
2	2	1576083893.17873	2019-12-11 18:04:53.178734	21.0625
3	3	1576083898.28807	2019-12-11 18:04:58.288071	21.0625
4	4	1576083903.39578	2019-12-11 18:05:03.395782	21.0625
5	5	1576083908.47675	2019-12-11 18:05:08.476750	21.0

Il y a deux champs qui contiennent la même information **timestamp** et **date**.
Évidemment, un seul champ aurait dû suffire...

A.5. Enregistrement périodique du fichier image de la courbe de température

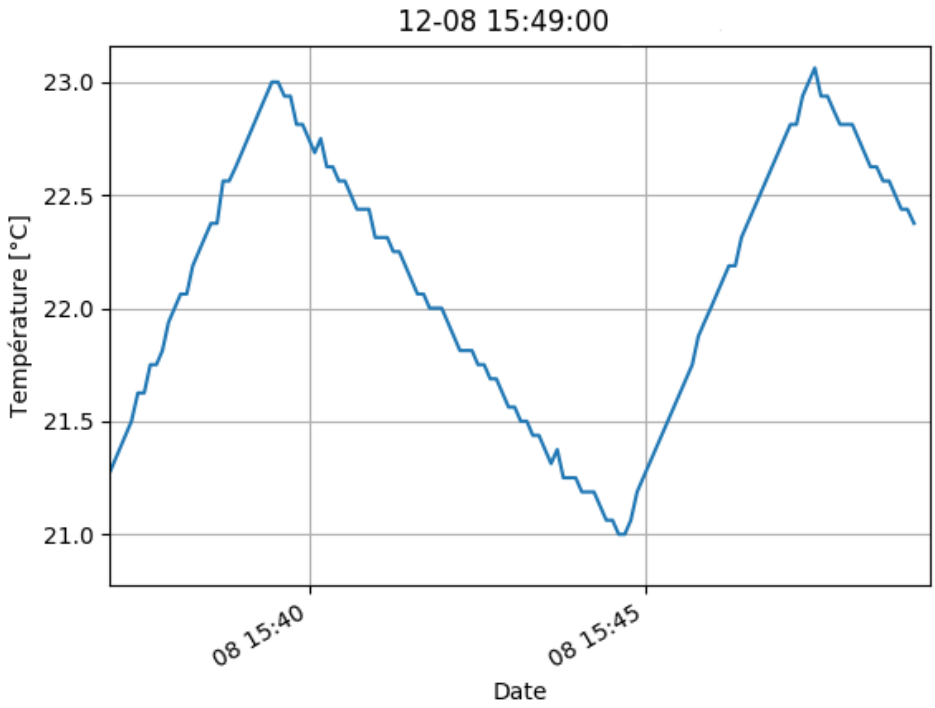
Dans le dossier */ressources/partieA/A5* , tester le script ***sauvegarde_temperature_et_image.py***

Ce script complète le script ***sauvegarde_temperature.py*** avec la sauvegarde périodique du fichier image de la courbe de température.
Ce n'est pas forcement très utile pour l'instant, mais cela le deviendra dans la partie B (application web).

L'image est créée avec le module `matplotlib`.
`matplotlib` est un gros module (dans le bon sens du terme) et comme vous pouvez le voir, le code n'est pas forcement simple pour un néophyte.
Paramètres à tester :

```
# nom du fichier du graphe
graphe_filename = "image/data03.png"  # ou format svg

# largeur de l'axe du temps du graphe (en secondes)
DELTAT = 720.0
```

Par exemple :



A.6. Graphe déroulant de la température

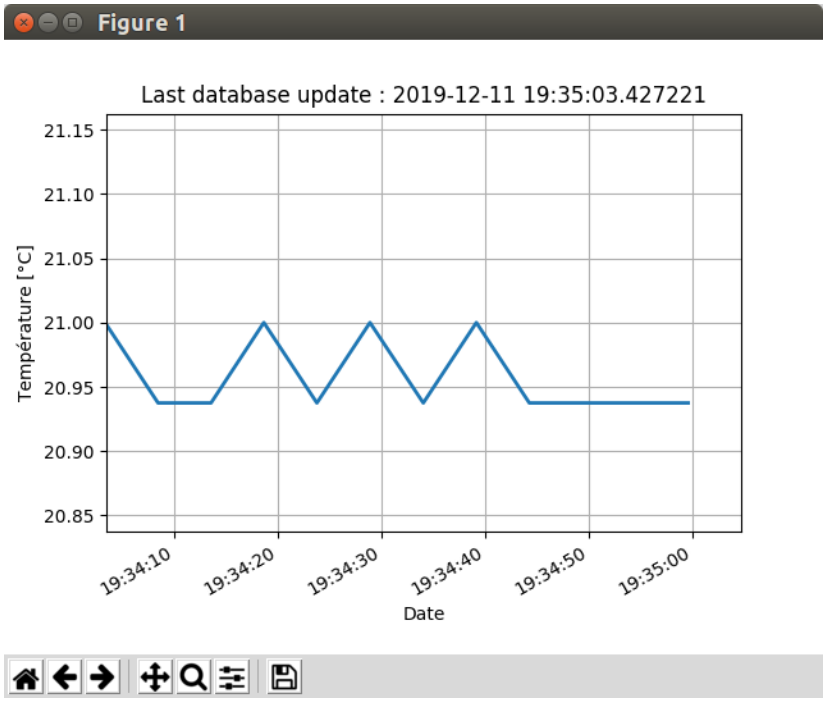
Le graphe déroulant est créé avec le module `matplotlib.animation`

Dans le dossier */ressources/partieA/A6*, commencer par lancer le script ***sauvegarde_temperature.py***
Dans un second temps, lancer le script *graphe_deroulant_temperature.py*

Le script *sauvegarde_temperature.py* (vu dans la partie A.4.2) lit périodiquement la température du DS1631 et l'enregistre avec la date dans la base de données.

Le script *graphe_deroulant_temperature.py* ne fait que consulter périodiquement la base de données pour alimenter les données de l'animation.
Si pour une raison ou une autre, la base de données n'est plus actualisée (typiquement si le script ***sauvegarde_temperature.py*** n'est pas lancé***)***, vous aurez droit à l'avertissement « No new data. Please run database engine ! ».

Paramètres :
```
# période de mise à jour de la base de données et du graphe déroulant
period = 5.0  # en secondes

# nom du fichier de la base de données
db_filename = "db/data002.db"

# largeur de l'axe du temps du graphe déroulant (en secondes)
DELTAT = 60.0
```



Au niveau du code, pour créer l'animation, vous remarquerez la présence de l'instruction `yield` caractéristique de ce que l'on appelle en Python les **générateurs**.

Un **générateur** est une pythonnerie super utile mais je ne suis toujours pas parvenu à en comprendre parfaitement le mécanisme et encore moins à pouvoir l'expliquer simplement.

Quand vous aurez du temps et de l'aspirine, voici un excellent site sur Python en général et sur les générateurs en particulier : http://sametmax.com/comment-utiliser-yield-et-les-generateurs-en-python

Autrement, sans réellement savoir ce qu'est un générateur, leur mise en œuvre est plutôt intuitive. Finalement, c'est peut-être ça le plus important.

Partie B – Application web

Dans la partie A, tout se passe localement (sur le seul Raspberry pi).
Ici, on veut avoir accès aux températures et aux thermostats depuis n'importe où dans le monde.
On s'appuyant sur le protocole HTTP du réseau Internet, nous allons mettre en œuvre un serveur web qui tourne sur le Raspberry pi.
Ce serveur web aura la particularité d'avoir accès aux ressources matérielles du Raspberry pi (en particulier au bus i2c où sont connectés les composants DS1631).

B.1. Qu'est-ce qu'un serveur web ?

Un serveur web est une application qui communique avec le protocole HTTP d'Internet.
Un client web (généralement un navigateur web) envoie une requête HTTP au serveur web.

Exemple de requête HTTP :

```
GET https://fr.wikipedia.org
```

Le serveur web renvoie alors au client la ressource située à l'URL `https://fr.wikipedia.org`

Ici, la ressource est une page web, c'est-à-dire un document qui contient du code HTML.
Le type « MIME » est « `text/html` ».

Mais cela peut aussi être :

- une image (type MIME `image/jpeg`, `image/png`, etc.)
- des données, par exemple au format JSON : `application/json`
  `{"ville":"valence","temperature":6.9}`
- un document pdf (`application/pdf`)
- du son, une vidéo, un document LibreOffice, etc.

Notez bien que c'est toujours le client qui demande, et le serveur qui répond.

B.2. Introduction au framework web `flask` : Hello world !

Le serveur web le plus utilisé dans le monde est Apache, et on peut sans problème le faire fonctionner sur un Raspberry pi.
Mais restons dans l'univers Python, où plusieurs alternatives se présentent à nous :
`Django`, `CherryPy`, `Bottle`, `Flask`, etc.

Pour faire de gros sites web avec plein de trafic, Django s'impose. C'est lourd et efficace.

Pour un petit site avec un apprentissage rapide de l'environnement de développement, une solution comme `flask` est parfaite.
Il faut évidemment connaître Python mais il est surtout question de web : vous devez avoir les connaissances de base en HTTP, HTML, CSS et JavaScript.

- **Page Hello world !**

3 fichiers sont nécessaires, avec l'arborescence suivante :

```
├── webapp1.py
├── static
│   └── style.css
├── templates
│   └── accueil1.html
```

(disponible dans le dossier */ressources/PartieB/B2*)

`style.css` est un fichier CSS pour la mise en forme du site web.

`accueil1.html` est un « template », un patron de page HTML.
Le moteur de template utilisé est `Jinja2 (`module inclus dans le package `flask)` :

```
<!DOCTYPE html>
    <head>
        <title>{{ title }}</title>
        <meta charset="UTF-8">
        <link rel="stylesheet"
              href="{{ url_for('static', filename='style.css') }}">
    </head>

    <body>
        <div id="global">
            <h1>{{ title }}</h1>
            <p>{{ texte }}</p>
        </div>
    </body>
</html>
```

Enfin le script python principal *webapp1.py* qui utilise le package `flask` et qui fait le lien avec le template *accueil1.html* :

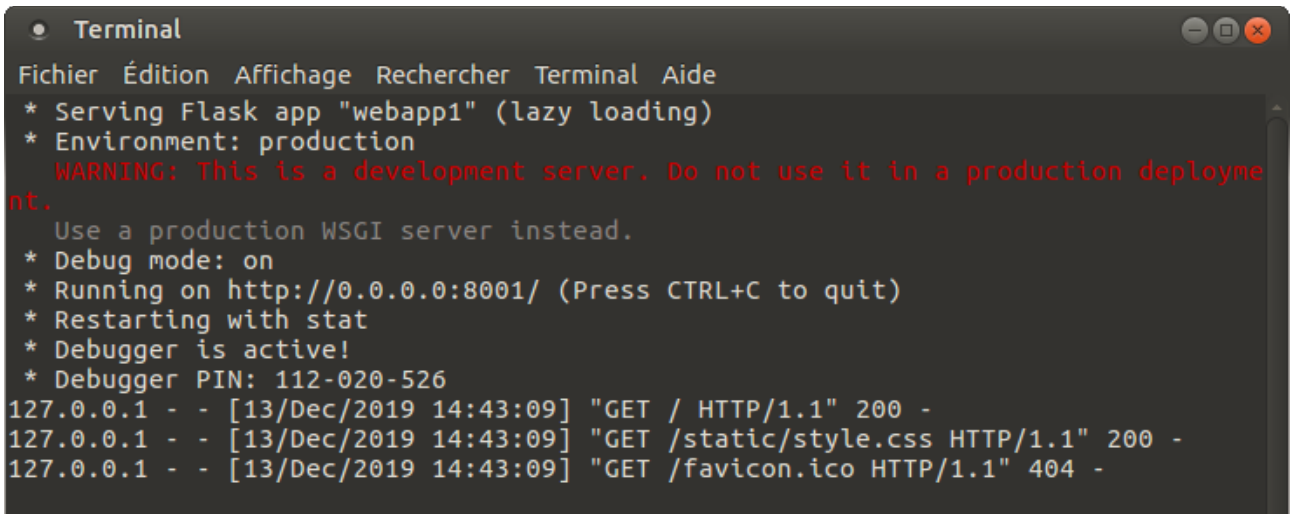
```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    templateData = {'title': "Page d'accueil", 'texte': 'Hello world !'}
    return render_template('accueil1.html', **templateData)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8001)
```

Dans Geany, ouvrir et exécuter le script *webapp1.py*

Remarque :

En cas de problème, passez par le terminal linux :

```
$ cd /home/pi/Desktop/ressources/partieB/B2_tuto_flask
$ export FLASK_APP=webapp1.py
$ python3 -m flask run --host=0.0.0.0 --port=8001
```



`flask` lance son propre serveur web !

Dans Firefox, saisir l'URL *http://localhost:8001*

Ce serveur web est également accessible depuis les autres machines du réseau local (à vérifier !).

Remarques sur le code :

```
templateData = {'title': "Page d'accueil", 'texte': 'Hello world !'}
```

`templateData` est un dictionnaire (type `dict` de Python) : c'est une sorte de tableau associatif avec une structure clés / valeurs `{clef1: valeur1, clef2: valeur2, etc.}`

```
render_template('accueil1.html', **templateData)
```

Dans la fonction `render_template()`, les deux étoiles `**` signifient que l'on passe un dictionnaire en argument (avec une seule étoile `*`, ce serait un tuple).

```
@app.route('/')
def index():
    ...
```

`app.route('/')` est précédé par le symbole `@`
C'est une pythonnerie qui s'appelle un **décorateur**.

Il s'agit simplement d'une règle d'écriture.
Le code :

```
@app.route('/')
def index():
      ...
```

est ici équivalent à :

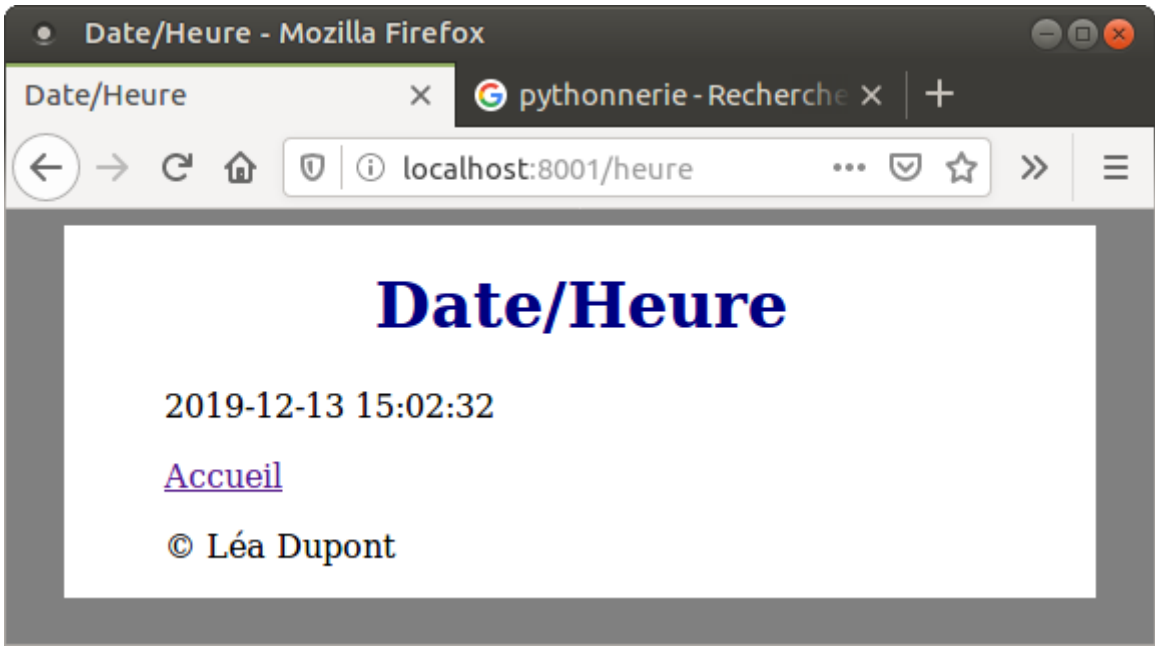
```
def index():
    ...
app.add_url_rule('/', 'index', index)
```

En définitive, nous avons une fonction qui prend en argument une fonction et qui retourne une fonction : une fonction décorée quoi !

- **On ajoute une deuxième page qui donne la date et l'heure courante**

Tester le script *webapp2.py*
Dans le code, observer le mécanisme de création de la deuxième page.

A faire : sur les deux pages, ajouter en bas de page un copyright avec le nom de l'auteur.
Le nom de l'auteur doit être contenu dans une variable Python, information commune aux deux pages :



B.3. Site web « Régulation de température » : squelette du site

A tester : *web03_squelette.py*
Le site est constitué de 5 pages (avec 5 templates) dont 1 formulaire :

http://localhost:8001/modifier_thermostat_accueil

La méthode GET est utilisée pour transmettre les informations du formulaire :
http://localhost:8001/enregistrer_thermostat?Thigh=19&Tlow=18

Étudier le mécanisme de gestion de la méthode GET dans `flask` :

```
from flask import request
...
@app.route(..., methods=['GET'])
...
request.method
...
request.args[]
```

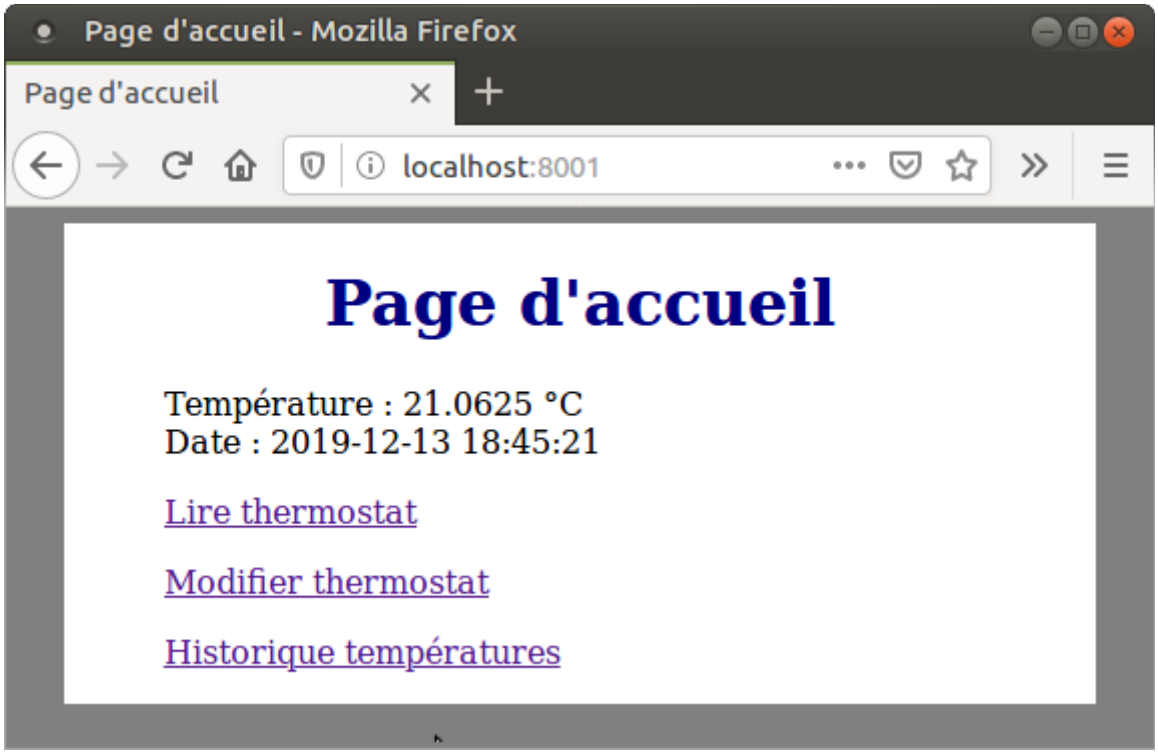
B.4. Page d'accueil

Nous allons implémenter l'affichage de la température du thermomètre DS1631.

B.4.1. Page d'accueil « statique »

A tester : *web041.py*

Par rapport à *web03_squelette.py* :
on importe le module `DS1631`,
on crée une instance de la classe `DS1631`,
on lit la température avec la méthode `get_temperature()`
et on affiche la date courante et la température dans la page d'accueil :



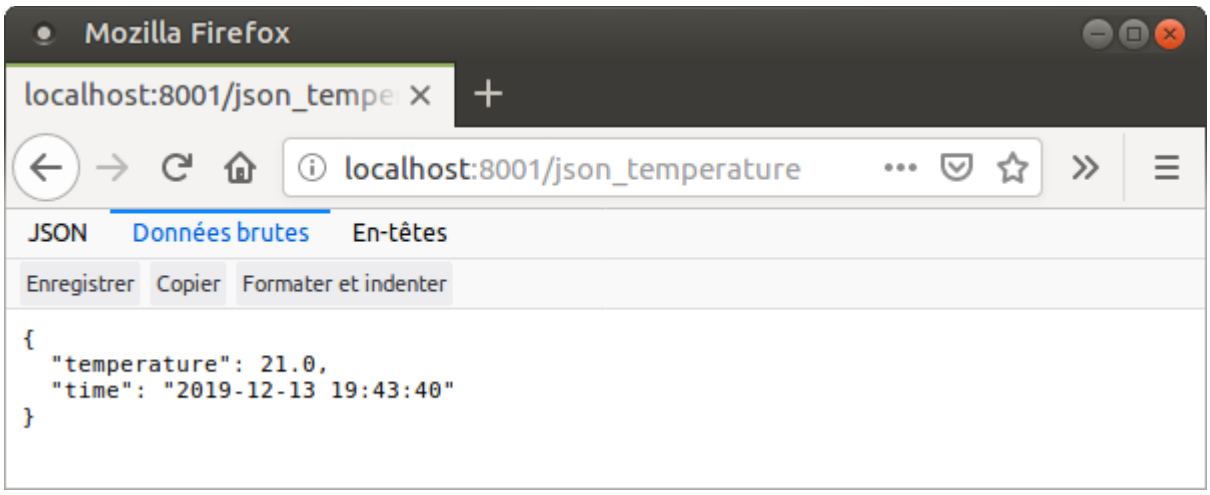
Inconvénient : pour actualiser la température, il faut recharger la page, ce qui n'est pas très pratique.

B.4.2. Page d'accueil dynamique avec AJAX

Une solution est d'utiliser la technique AJAX pour mettre à jour la page régulièrement.

B.4.2.1. Une page web au format JSON

A tester : *web0421.py* et en particulier la nouvelle page « cachée » :
http://localhost:8001/json_temperature



Cette page n'a pas vocation à être consultée directement.
Elle fait partie du mécanisme AJAX.

Par rapport à *web041.py* :
on importe le module `jsonify` de `flask` :
`from flask import jsonify`
on crée une nouvelle page de type MIME `application/json`
`@app.route('/json_temperature')`
qui retourne la température et la date courante.

B.4.2.2. Actualisation de la température toutes les secondes dans la page d'accueil

A tester : *web0422.py*

Une grosse modification dans le template *accueil.html* (que j'ai renommé *accueil_ajax.html*) avec du code JavaScript.

B.4.2.3. Qu'est-ce qu'AJAX (Asynchronous JavaScript and XML) ?

L'architecture informatique AJAX permet de construire des applications Web et des sites web dynamiques interactifs sur le poste client en se servant de différentes technologies ajoutées aux navigateurs web entre 1995 et 2005.

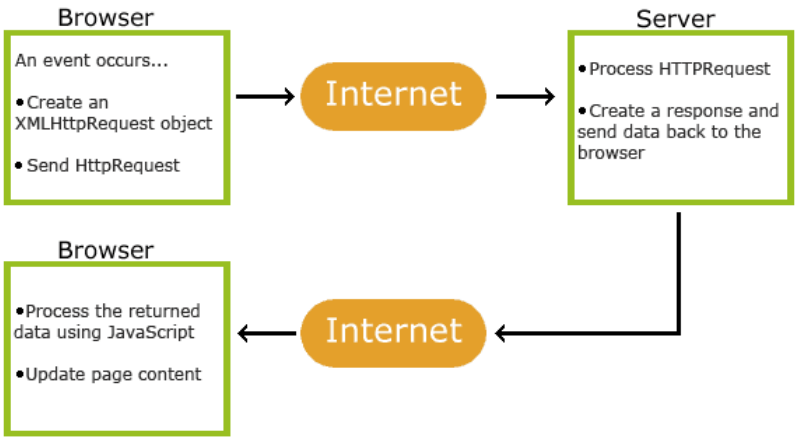
AJAX combine JavaScript, CSS, JSON, XML, le DOM et surtout l'objet **XMLHttpRequest** afin d'améliorer maniabilité et confort d'utilisation des applications internet riches (RIA) :

- le DOM (Document Object Model) et JavaScript permettent de modifier l'information présentée dans le navigateur en respectant sa structure ;
- l'objet XMLHttpRequest sert au dialogue asynchrone avec le serveur Web ;
- XML ou JSON structure les informations transmises entre serveur Web et navigateur.

Les applications AJAX fonctionnent sur tous les navigateurs Web courants (Google Chrome, Safari, Mozilla Firefox, Internet Explorer, Opera, etc.).

Principe

Avec AJAX, le dialogue entre le navigateur et le serveur se déroule la plupart du temps de la manière suivante : un programme écrit en langage de programmation JavaScript, incorporé dans une page web, est exécuté par le navigateur.
Celui-ci envoie en arrière-plan des demandes au serveur Web, puis modifie le contenu de la page actuellement affichée par le navigateur Web en fonction du résultat reçu du serveur, évitant ainsi la transmission et l'affichage d'une nouvelle page complète.



XMLHttpRequest

Le XMLHttpRequest (souvent abrégé XHR) est un objet de programmation, utilisé dans les programmes en langage JavaScript pour assurer la communication entre le navigateur et un serveur Web.
Il est utilisé pour la communication asynchrone : envoyer les requêtes HTTP vers le serveur et déclencher des opérations lors de la réception de réponses de celui-ci.

JSON

JavaScript Object Notation (JSON) est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Il permet de représenter de l'information structurée comme le permet XML par exemple.
Bien qu'utilisant une notation JavaScript, JSON est indépendant du langage de programmation.
Il sert à faire communiquer des applications dans un environnement hétérogène.
Il est notamment utilisé comme langage de transport de données par AJAX et les services Web.
Le type MIME *application/json* est utilisé pour le transmettre par le protocole HTTP.

[Sources : Wikipedia & www.w3schools.com]

B.4.2.4. Explication du code du template *accueil_ajax.html*

Ouvrir le contenu du fichier.

La page HTML contient du code JavaScript dans la balise `<script>`.

Une fois la page reçue du serveur, le code JavaScript est exécuté par le navigateur (donc côté client).

Ligne 36 :

La fonction `loadDoc()` de JavaScript est exécutée :

Toutes les 1000 ms, une requête HTTP :
`GET /json_temperature`
est lancée en arrière fond.

Les données JSON sont ensuite reçues, décodées et affichées dynamiquement dans la page d'accueil.

B.4.2.5. L'outil réseau de Firefox

Pour comprendre le mécanisme sous-jacent, nous allons utiliser un outil de Firefox :
Firefox → Menu → Développement Web → Réseau : XHR (XMLHttpRequest)

Observer les échanges entre le client et le serveur et le contenu des données transmises.

B.5. Les pages du thermostat

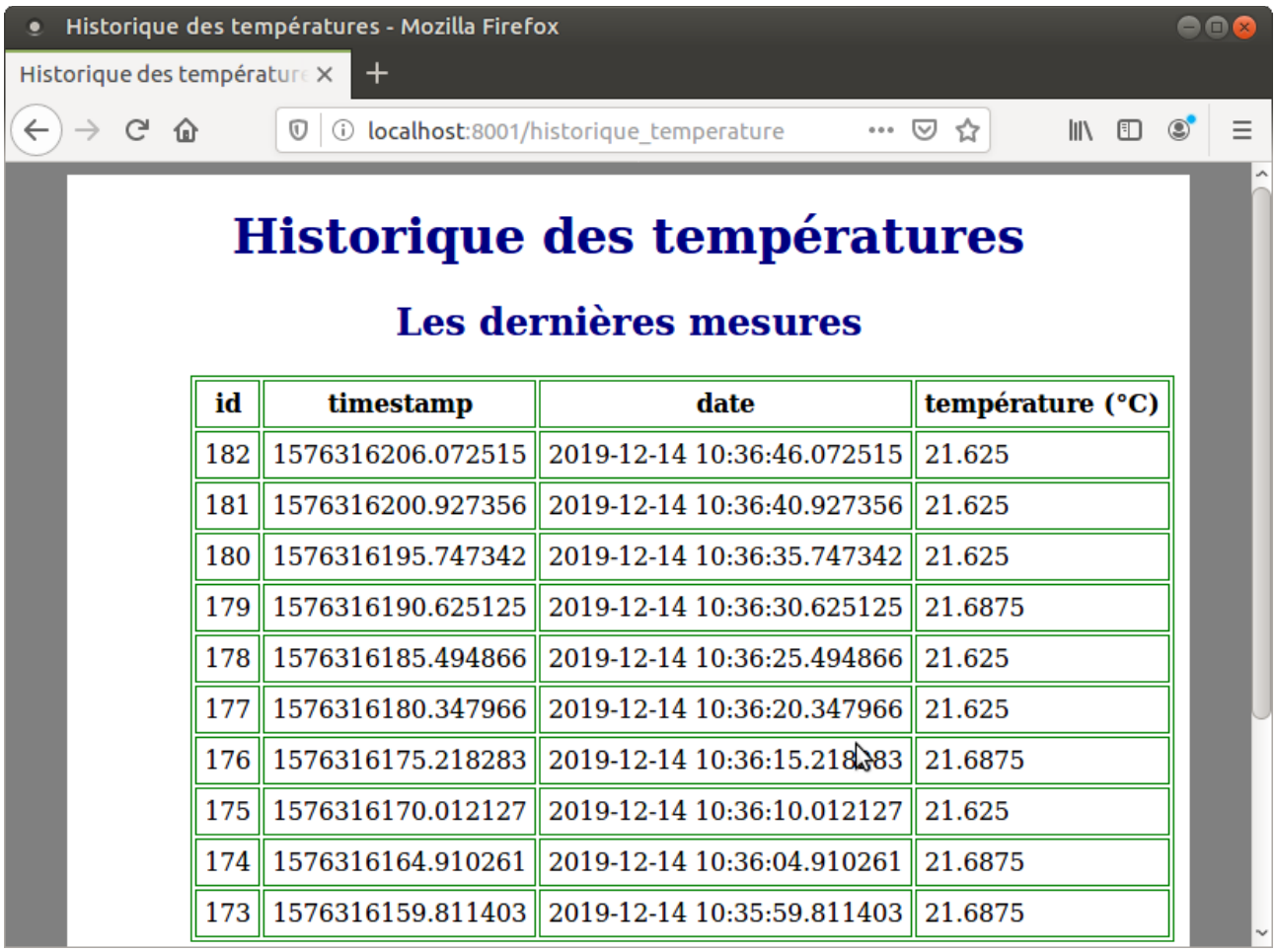
A tester : *web05.py*

Le code `flask` des 3 pages prend désormais en compte le thermostat DS1631.
Notez qu'un gros effort a été fait pour contrôler l'intégrité des informations transmises par le formulaire.

B.6. Page Historique des températures (tableau dynamique)

Dans le dossier */ressources/partieB/B6*, commencer par lancer le script ***sauvegarde_temperature.py***
Dans un second temps, lancer le script *web06.py*

On obtient le tableau des 10 dernières mesures stockées dans la base de données, avec mise à jour automatique :

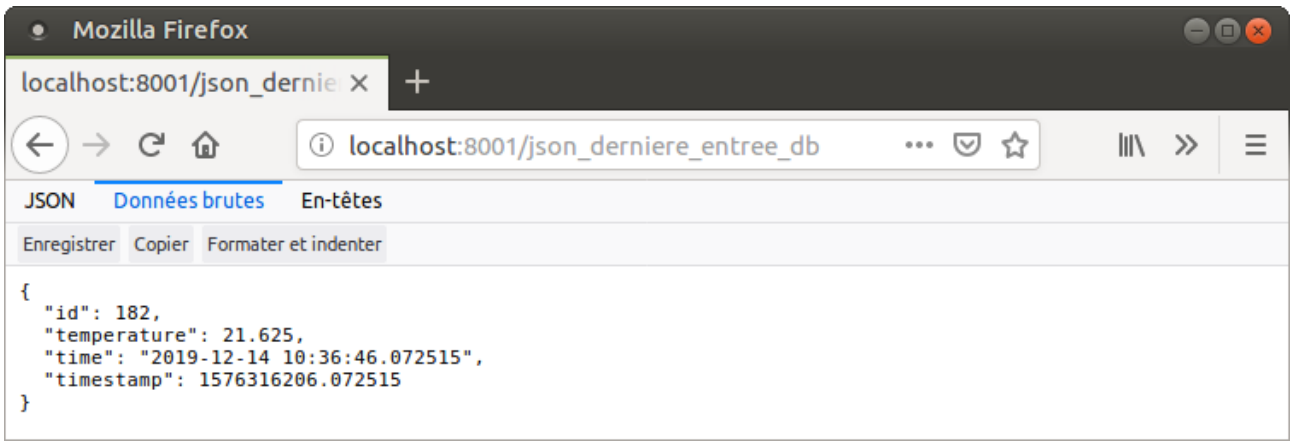


id	timestamp	date	température (°C)
182	1576316206.072515	2019-12-14 10:36:46.072515	21.625
181	1576316200.927356	2019-12-14 10:36:40.927356	21.625
180	1576316195.747342	2019-12-14 10:36:35.747342	21.625
179	1576316190.625125	2019-12-14 10:36:30.625125	21.6875
178	1576316185.494866	2019-12-14 10:36:25.494866	21.625
177	1576316180.347966	2019-12-14 10:36:20.347966	21.625
176	1576316175.218283	2019-12-14 10:36:15.218283	21.6875
175	1576316170.012127	2019-12-14 10:36:10.012127	21.625
174	1576316164.910261	2019-12-14 10:36:04.910261	21.6875
173	1576316159.811403	2019-12-14 10:35:59.811403	21.6875

Pour cela, on utilise la technique AJAX (voir le code du template *historique_temperature_ajax.html*) :

Une requête HTTP est régulièrement lancée pour récupérer la dernière entrée de la base de données :

http://localhost:8001/json_derniere_entree_db



Le tableau est ensuite mis à jour dynamiquement avec du JavaScript.

B.7. Page Historique des températures (tableau et graphe dynamiques) avec mise en œuvre d'un thread

Dans la partie précédente, il faut lancer indépendamment deux programmes :
- le script *sauvegarde_temperature.py*
- puis l'application web `flask`

Nous aimerions faire la même chose mais en une seule étape, en exécutant uniquement l'application web.
Pour cela, il faut que l'application web intègre le script *sauvegarde_temperature.py* sous la forme d'un thread.

B.7.1. Rappel sur les threads

Un thread est ce qu'on appelle au niveau du système d'exploitation un « processus léger ».
Le module `threading.Thread` de Python permet leur mise en œuvre.

Voici 3 exemples à tester :
- Le programme principal : *exemple_thread_A.py*
- Le programme principal et un thread : *exemple_thread_B.py*
- Le programme principal et un deuxième thread : *exemple_thread_C.py*

B.7.2. Mise en œuvre dans la page Historique des températures

Lancer le script *web072.py* (vérifier qu'il n'y a pas d'autres scripts actifs).

Quelques explications :

La classe `DS1631` du module `db_figure_update_engine` est une réécriture sous la forme d'un thread du script *sauvegarde_temperature_et_image.py* vu en A.5

Dans l'application *web072.py*, on importe ce module :
```
import db_figure_update_engine
```

On crée une instance de la classe `DS1631.DS1631` :
```
ic1 = DS1631.DS1631(1, 0x4f)
```

puis on lance le thread :
```
thread = db_figure_update_engine.DS1631(
ds1631object=ic1,
db_file_name="db/data5003.db",
graphe_filename="static/data5003.png",
period=5, deltat=120)
```

Remarquez qu'**on passe l'objet `ic1` en argument**.
On faisant ainsi, le thread a accès directement au circuit intégré DS1631.

Le thread met à jour la base de données ainsi qu'une image */static/data5003.png*

Dans le template *historique_temperature_ajax2.html*, il suffit d'intégrer cette image à la page web avec une balise <*img*>, et de la recharger toutes les 5 secondes avec une commande JavaScript.
C'est de cette manière que l'on obtient le graphe dynamique de l'historique des températures.

N.B. Il serait beaucoup plus efficace, notamment en terme de bande passante, de créer un graphe en JavaScript (avec une librairie graphique comme chart.js) et d'utiliser la technique AJAX pour l'actualiser à chaque nouvelle mise à jour de la base de données.
On déchargerait ainsi le serveur web du Raspberry pi de l'écriture périodique d'une image d'environ 50 ko sur sa carte SD, et on limiterait la basse passante à quelques centaines d'octets (la dernière température dans la base de données au format JSON) au lieu des 50 ko pour le transfert de l'image.

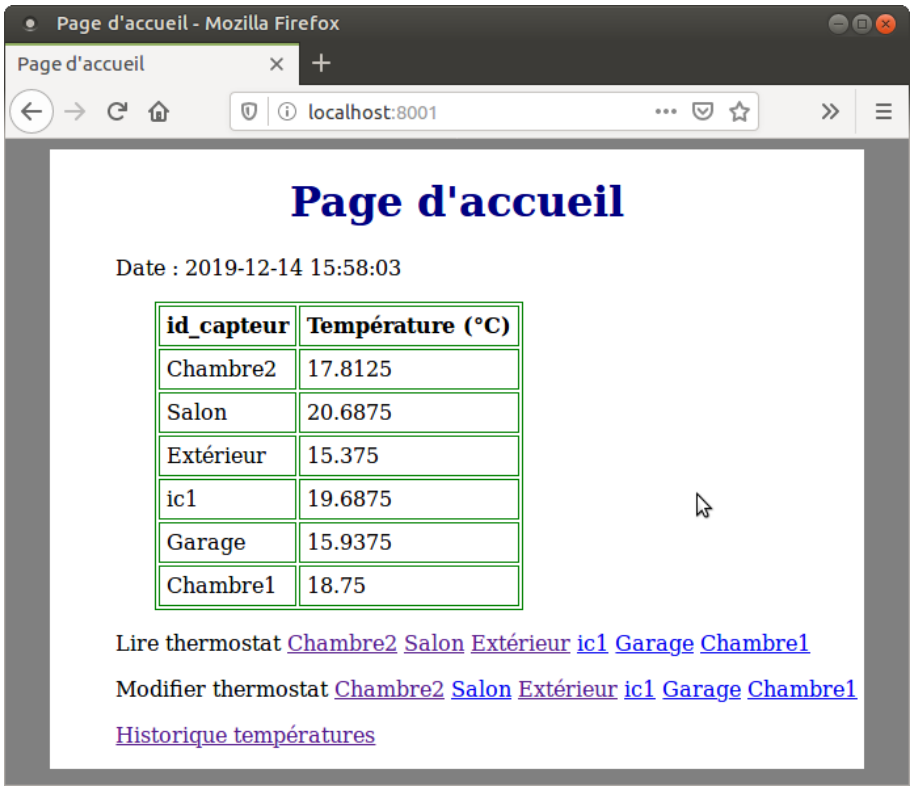
B.8. Version multi-capteurs

B.8.1. Généralisation à plusieurs capteurs

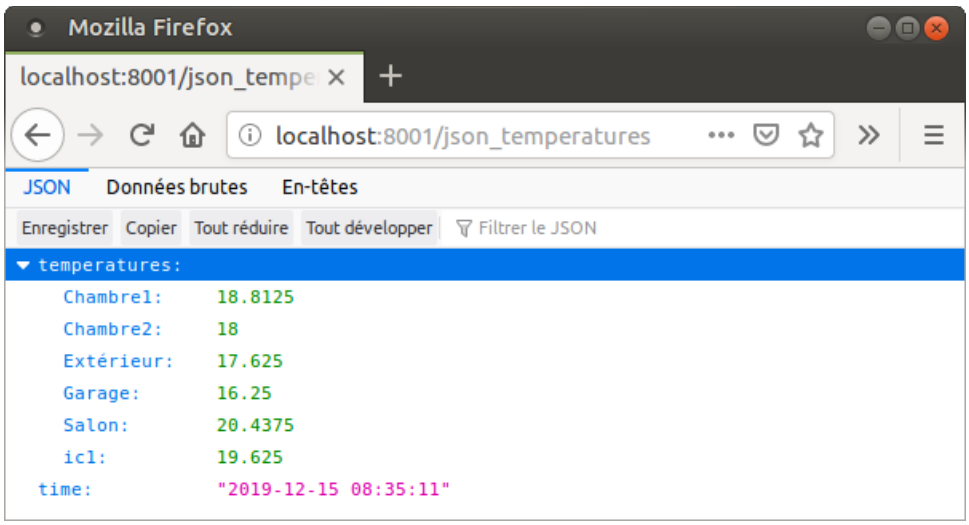
Tout a été fait jusqu'à présent avec un seul circuit DS1631.
Il est temps de généraliser à plusieurs circuits DS1631 (jusqu'à 8 sur le bus i2c) que l'on peut compléter avec un nombre quelconques de capteurs virtuels.

Lancer le script *web081_multicapteurs.py*

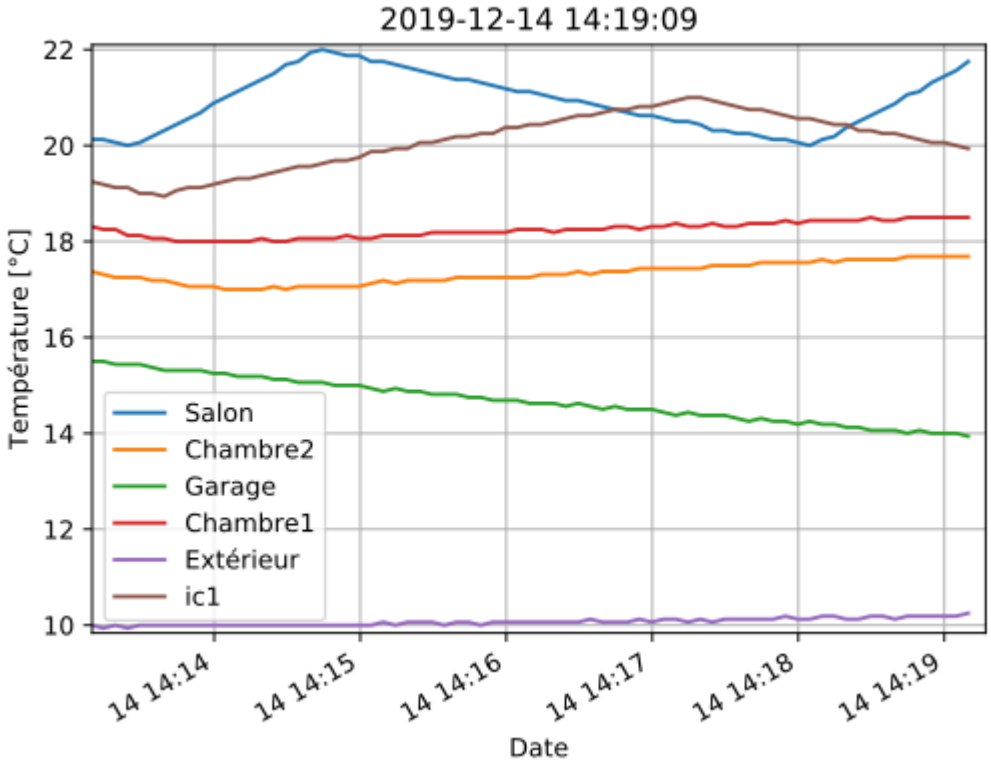
Voici la nouvelle page d'accueil :



avec en arrière-plan :



L'historique des températures :



Je vous laisse le soin d'étudier le nouveau code.

B.8.2. Petite amélioration [à faire]

Dans la page d'accueil, on voudrait que seuls les liens des thermostats connectés effectivement à un radiateur, soient visibles (donc pas de liens vers « Garage » ou « Extérieur »).
Modifier le script *web082_multicapteurs.py* et le template de la page d'accueil pour que ceci se fasse automatiquement.

N.B.
Pour chaque objet, la puissance est donnée par l'attribut `P`.
Pour les DS1631 virtuels, la puissance du radiateur associé au thermostat est un argument du constructeur :

```
ic_garage = DS1631.DS1631virtualdevice(initial_temperature=16, P=0, etc.)
```

La puissance est ensuite accessible avec l'attribut `P`.
Ainsi, `ic_garage.P vaut 0` (donc pas de radiateur).

Pour les DS1631 « réels », l'attribut `P` de la classe `DS1631` n'existe pas, c'est à vous de le créer.
Par exemple : `ic1.P = 700`

B.8.3. Fichier de configuration

On voudrait maintenant éviter d'avoir à manipuler le code du script *webXXX.py* chaque fois que l'on désire changer les paramètres.
Pour cela, nous allons déporter tous les paramètres dans un fichier texte */static/config.ini*
Ce fichier est structuré en section, option et valeur :

```
[section]
option : valeur
```

La lecture et le décodage de ce fichier sera grandement simplifié par l'utilisation du module standard `configparser`.

Lancer le script *web083_multicapteurs.py*

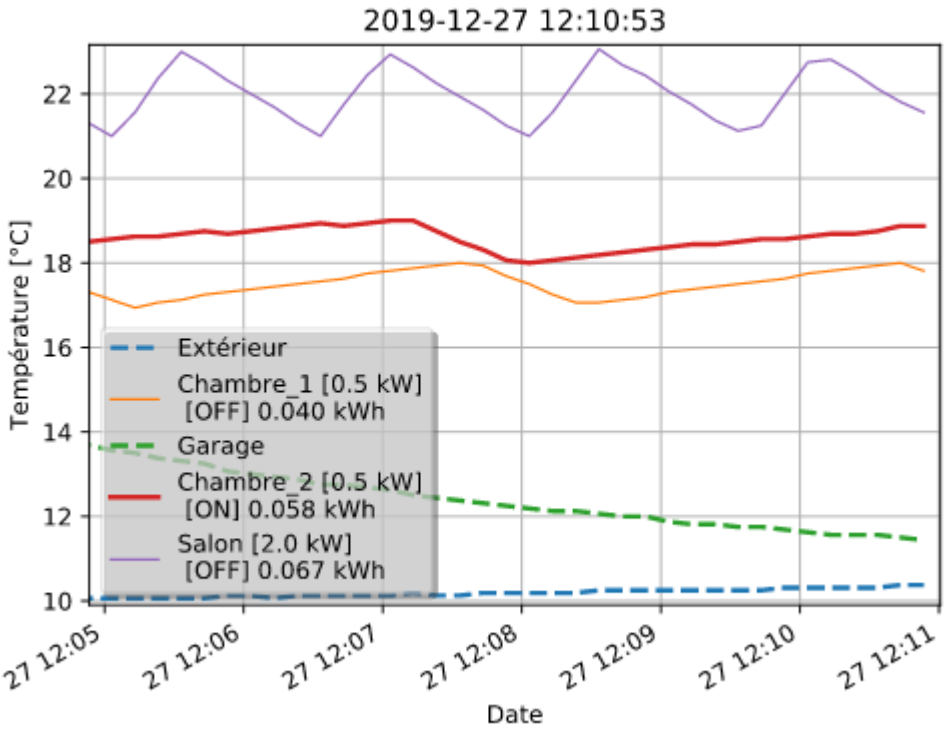
Ajouter un capteur virtuel (chambre_3) dans le fichier de configuration.

B.8.4. Consommation électrique

La consommation électrique est calculée de manière purement logicielle pour chaque radiateur (pas besoin d'un appareil de mesure, si ce n'est pour effectuer un étalonnage).

Lancer le script *web084_multicapteurs_kwh.py*

La page historique ressemble à présent à ceci :



Seul le fichier *db_figure_kwh_update_engine_multicapteurs.py* a été modifié, avec :

- deux nouveaux champs dans la table : etat, kwh
- une estimation de l'état des radiateurs et de la consommation
- l'intégration des informations dans la légende de l'image

N.B. Il n'y a pas moyen de connaître directement le niveau de la broche Tout, et donc l'état du radiateur, car cette possibilité n'a pas été prévue par les concepteurs du DS1631...
Cependant, on peut faire une prédiction à partir des flags THF et TLF du registre de configuration, la précision temporelle des prédictions allant de pair avec la périodicité de consultation des flags.
On peut alors estimer la consommation électrique.

Table : Chambre_2 | Nouvel Enregistrement | Supprimer l'enregistrement

	id	timestamp	date	emperatur	etat	kwh
	Filter	Filter	Filter	Filter	Filter	Filter
28	28	1577444863....	2019-12-27 12:07:43....	18.3125	OFF	0.0333625...
29	29	1577444873....	2019-12-27 12:07:53....	18.0625	OFF	0.0333625...
30	30	1577444883....	2019-12-27 12:08:03....	18.0	ON	0.0347526...
31	31	1577444893....	2019-12-27 12:08:13....	18.0625	ON	0.0361425...
32	32	1577444903....	2019-12-27 12:08:23....	18.125	ON	0.0375327...

B.9. Complément pour les électroniciens : analyse des trames i2c

L'analyseur logique est un clone VKTECH Saleae, que l'on trouve pour 5 €...
Il possède 8 entrées pour signaux logiques, avec une fréquence d'échantillonnage maximale de 24 MHz.

Le logiciel open source *PulseView* permet l'acquisition et le décodage du protocole i2c.
Installation :
`sudo apt install sigrok` [déjà fait]

Brancher l'analyseur logique sur le port USB de votre Raspberry pi.
Ouvrir PulseView dans le terminal Linux :

```
$ pulseview
```

Connect to device :
Choose the driver : *fx2lafw*
Scan for devices :
choisir *Saleae Logic with 8 channels*

- **Exemple de trame i2c générée par la méthode `get_temperature()`**

S = bit Start
AW = Address Write (0x48)
W = bit Write

A = bit Acknowledge
DW = Data Write (0xAA est l'adresse du registre interne READ_TEMPERATURE du DS1631)
A
Sr = bit Start repeat
AR = Address Read (0x48)
R = bit Read
A
DR = Data Read (0x17)
A
DR = Data Read (0xA0)
Nack = bit Not acknowledge
P = Stop

Le registre interne READ_TEMPERATURE à l'adresse 0xAA contient 2 octets en lecture seule : 0x17A0 (au format big endian = octet de poids fort en premier).
(Cf. datasheet du DS1631)

Questions :

a) Quelle est l'adresse i2c du DS1631 ?
b) Que vaut la fréquence de l'horloge ?
c) Quelle est la température ? (Cf. datasheet du DS1631)

Correction :

a) Adresse i2c = 0x48
b) La fréquence de l'horloge est de 100 kHz.
c) Température :

0x17A0 = 0b00010111 10100000 (16 bits en complément à deux)
Le premier bit est 0, le nombre est donc positif.
0x17A0 = 6048 en décimal
6048/256 = 23,625 °C

Vérifions avec la fonction `convert_word_to_temperature()` du module `DS1631` :

```
$ ipython3

In [1]: import DS1631

In [2]: DS1631.convert_word_to_temperature?
Docstring:
word (2 bytes) in little endian format
Return temperature in °C (float)

In [3]: DS1631.convert_word_to_temperature(0xA017)
Out[3]: 23.625
```