# *Module* `myNetwork`

Le module `myNetwork` permet de gérer la carte Wifi de votre ESP32. Voici un descriptif de quelques fonctions définies par ce module :

## A. *Création d'un point d'accès*

- `myNetwork_create(ssid,passwd=None,authmode=None)`
  Cette fonction passe la carte Wifi en mode "*master*" afin de transformer votre ESP32 en point d'accès : le nom de votre réseau sera la valeur de la variable `ssid`.
  Voici le comportement de `myNetwork_create` en fonction des paramètres d'appels :
  - Si l'argument `passwd` n'est pas fourni lors de l'appel, le réseau créé sera ouvert *(sans mot de passe)*.
  - Si l'argument `authmode` n'est pas fourni *(mais avec un mot de passe)*, le réseau créé sera sécurisé *(par défaut, WPA2-PSK)*. Voici les différentes valeurs possibles pour cet argument :
    - `1` : WEP
    - `2` : WPA-PSK
    - `3` : WPA2-PSK
    - `4` : WPA/WPA2-PSK

**Indications :**
- Sur tous les exercices de ChingInfo, la fonction `myNetwork_create()` est appelée avec le SSID `mySSID`. Cette valeur produira une erreur et obligera l'élève à personnaliser le nom de son réseau *(chaque élève doit avoir son propre réseau)*.
- sur certains ordinateurs Windows, il est préférable de modifier le paramètre `authmode` pour une meilleure connexion.

- `myNetwork_getIp()` : dans le cas où votre carte a créé un point d'accès, cette fonction retourne l'adresse IP de votre carte sous la forme d'une chaîne de caractères *(`vvv.xxx.yyy.zzz`)*

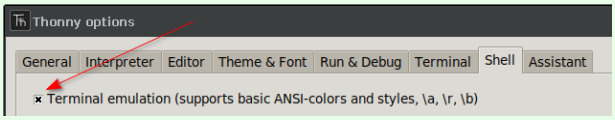
## B. *Connexion à un réseau existant*

- `myNetwork_connect(ssid,passwd)` : cette fonction passe la carte Wifi en mode "*slave*" afin de transformer votre ESP32 en une station.
  Votre carte va se connecter au réseau Wifi "`ssid`" en utilisant le mot de passe "`passwd`".

## C. *Quelques informations en plus*

Le code source de ce module est disponible :
1292-myNetwork.py

Afin d'afficher correctement les informations fournies en console par ce module, activez l'affichage des couleurs dans la console Thonny :




Voici un code illustrant quelques fonctionnalités de ce module :

```
from myNetwork import *

mode = 0
if mode == 0:
    print("Connexion à un réseau existant")
    myNetwork_connect("myHome2","abcabcabc");
elif mode == 1:
    print("Création du point d'accès \"myWifi1\"
ouvert (sans mot de passe)")
    myNetwork_create("myWifi1");
elif mode == 2:
    print("Création du point d'accès \"myWifi2\"
avec mot de passe")
    myNetwork_create("myWifi2","01234567");
```

1292-exemple.py

# *Module* myTcp

Le module myTcp va nous permettre d'implémenter facilement un serveur TCP sur notre carte ESP32.

Avec le module myNetwork, nous créerons ou nous nous connecterons à un réseau Wifi afin de tester notre serveur TCP.

Voici les différentes fonctions composant ce module :

- myTcp_start(p) : cette fonction va créer un serveur TCP qui écoutera sur le port p.
- myTcp_listen() : cette fonction portera le serveur en écoute et dès qu'une connexion au serveur sera initiée, cette fonction renverra la requête sous forme d'une chaîne de caractères.
- myTcp_listenBytes() : même fonctionnalité que la fonction précédente, mais la requête sera renvoyée sous la forme d'un tableau d'octets.
- myTcp_send(ch) : cette fonction ne peut être utilisée qu'après l'exécution de la fonction myTcp_listen() *(ou* myTcp_listeBytes() *)*.
  Elle enverra au client, ayant initié la connexion, la réponse ch qui est donnée sous la forme d'une chaîne de caractères.
- myTcp_sendBytes(b) : de même que la fonction précédente, mais par contre sera renvoyé à l'utilisateur le tableau d'octets défini par l'argument b.
- myTcp_sendFichier(file) : de même, mais cette méthode renverra le contenu du fichier dont le chemin est pFile.
- myTcp_close() : cette fonction ferme la connexion actuelle. Elle doit forcément s'exécuter après la fonction myTcp_listen() *(ou* myTcp_listenBytes*)*.

**Remarque :** ce serveur a été créé le plus simplement possible et à certaines limitations.
Par exemple, il ne peut recevoir des requêtes dépassant 1024 octets. Pour modifier ce comportement, vous pouvez modifier le fichier source de ce module.

Le code source de ce module est disponible :
1046-myTcp.py

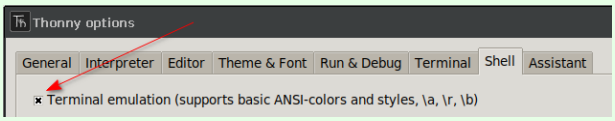
Ci-dessous, l'exemple d'un serveur TCP faisant office de miniserverWeb et répondant à la requête des navigateurs sur le port 80 et affichant toujours la même page affichant la phrase "Hello Word!".

```
from myNetwork import *
myNetwork_create("myWifi");

from myTcp import *
myTcp_start(80)


reponse=b"""HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

<html><head></head>
<body>Hello Word!
<img src="a.jpg">
<img src="b.jpg"></body>
</html>"""

while True:
    print("* Début de l'écoute")
    a = myTcp_listenBytes()
    print("* Début de communication")
    myTcp_sendBytes(reponse)
    print("* Réponse")
    myTcp_close()
    print("* Fermeture de la connexion")
```

1046-miniServeurWeb.py

**Remarque :** Chrome par défaut ouvre 3 connexions TCP pour anticiper le chargement des différentes ressources affichant une page *(image, script JavaScript, feuille de styles, ...)*.
Ainsi, j'ai préféré inclure dans la page Web deux images, pour que les réponses de Chrome reçoivent une réponse et ne créé par un timeout d'une connexion TCP qui afficherait des messages inopportun en console.

Afin d'afficher correctement les informations fournies en console par ce module, activez l'affichage des couleurs dans la console Thonny :



# *Module* myHttp

Le module "myHttp" permet de créer facilement un serveur HTTP sur notre carte ESP32. Il s'appuie sur les modules :

- 1292-myNetwork.py : ce module permet à votre ESP32 de créer ou de se connecter à un réseau Wifi.
- 1046-myTcp.py : ce module permet de créer un serveur TCP

**Attention :** ce module permet de créer un site statique et non pas dynamique. Il ne prend pas en charge la transmission de données par la méthode `POST` et nous allons voir que la gestion de la méthode `GET` est assez limitée.

Le fonctionnement de nos serveurs HTTP est basé sur une simple correspondance entre :

- "*nom du fichier demandé*" par la requête du client
- avec le "*contenu*" envoyer dans la réponse du serveur.

## A. *Gestion du serveur*

- `myHttp_init()` : cette commande crée le serveur TCP qui écoutera sur le port 80.
  À partir de ce moment, on pourra utiliser les fonctions de "*réponse du serveur*" afin de configurer son futur comportement.
- `myHttp_start()` : cette commande active le serveur HTTP.
  À partir de l'exécution de cette commande, le serveur ne peut plus être modifié et toutes commandes suivant cette fonction ne sera pas exécuté.

## B. *Réponse du serveur*

Le but d'un serveur HTTP est de fournir une réponse à la requête du client. Le comportement choisi pour ce serveur est de définir le comportement de notre serveur en fonction du fichier demandé.

Voici les fonctions permettant de modifier le comportement de notre serveur :

- `myHttp_addReponse(fichier,varHeader,varBody)` : cette commande permet de modifier le comportement de votre serveur Web.
  Dans le cas où le navigateur client veut visiter la page `fichier` alors une page de la forme ci-dessous sera retournée au client :

```
<html><head>varHeader</head>
<body>varBody</body></html>
```

- `myHttp_addGpio_out(fichier,portPin,value)` : si la page "`fichier`" est demandée alors le pin `portPin`, digital et en mode sortie, sera affecté de la valeur `value` *(0 ou 1)*.
  La chaîne "`ok`" sera renvoyée au client.
- `myHttp_addGpio_in(fichier,portPin)` : si la page "`fichier`" est demandée alors la valeur du pin `portPin`, analogique et en mode entré, sera renvoyée au client.

Nous pouvons configurer la réponse de notre serveur de trois façons :

- Il peut récupérer un fichier de son espace de stockage et renvoyer son contenu au serveur.
- Il peut répondre une chaîne de caractères particulière en fonction d'un fichier particulier demandé.
- Il peut lire l'état d'un port GPIO ou modifier son état.

En supposant que le client envoie une requête pour le fichier `filename`, voici, dans l'ordre, comment est conditionnée la réponse de notre serveur :

- Si `filename` définie une image *(extension png, jpg, jpeg, gif)*, :
  - le serveur cherchera cette image à partir de la racine de ses fichiers et renverra son contenu *(binaire)* au client.
  - Si l'image n'existe pas, l'erreur HTTP 404 sera renvoyée.
- Si `filename` a été ajouté par la fonction `myHttp_addReponse()`, le contenu associé est renvoyé au serveur.
- Si `filename` a été ajouté par la fonction `myHttp_addGpio_out`, alors le changement de statut du port défini sera fait.
  La réponse "`ok`" sera renvoyée au client.
- Si `filename` a été ajouté par la fonction `myHttp_addGpio_in()`, la valeur du port GPIO définie par cette fonction sera renvoyé au client.
- Si aucune des situations précédentes se présente, le serveur recherche , à partir de sa racine, le fichier `filename` :
  - si le fichier existe son contenu sera renvoyé au client
  - sinon l'erreur HTTP 404 sera renvoyé au client.

**Exemple :** le script ci-dessous va créer un mini siteWeb sur votre ESP32 :

1293-monsite.py

Voici quelques éléments de son comportement :

- la page principale *(`index.html`)* est définie par la commande :
  `myHttp_addReponse("index.html",...,...)`
- les images *(`sete_a.jpg`, `sete_b.jpg`)*, la feuille de styles *(`messtyles.css`)*, la page *(`autrePage.html`)* sont directement lus à partir de la mémoire de l'ESP32. Ce sont des fichiers qui sont déjà présents dans le firmware personnalisé.

Afin d'afficher correctement les informations fournies en console par ce module, activez l'affichage des couleurs dans la console Thonny :