Chapitre 4
Le Raspberry Pi en console avec urwid

1. Introduction

`urwid` est une bibliothèque assez récente dans le paysage UNIX. Elle repose essentiellement sur une autre bibliothèque qui est relativement ancienne, la bibliothèque `curses`. À l'origine de `curses`, on retrouve les premiers développeurs informatiques. À l'époque, la console était le seul moyen d'interagir avec la machine. Ainsi, les premiers développeurs ont très rapidement dû faire face à la nécessité d'assembler un jeu de fonctions pour dessiner des courbes et écrire du texte afin de créer des interfaces utilisateurs. La bibliothèque `curses` était née et les premières interfaces graphiques avec. Cependant, l'apprentissage de cette bibliothèque est déroutant et relativement complexe. De ce fait, ce chapitre se concentrera sur l'étude de `urwid`, une bibliothèque beaucoup plus simple à prendre en main et avec laquelle le développeur peut rapidement créer des interfaces console riches. Dernier point important : contrairement à la bibliothèque `curses`, qui est livrée avec la bibliothèque standard de Python, la bibliothèque `urwid` doit être installée via le gestionnaire de paquets `pip`.

2. urwid, les fondamentaux

La bibliothèque `urwid` s'articule principalement autour de l'utilisation de widgets. Un widget est simplement une classe Python avec des attributs pré-conçus : un champ texte, une liste déroulante, un espace vide, etc. Dans le jargon `urwid` cependant, un widget est en charge d'afficher du texte, de représenter un bouton ou de servir de conteneur pour encapsuler d'autres widgets. Comme avec d'autres frameworks graphiques, la construction d'une application avec `urwid` s'effectue en imbriquant successivement des widgets les uns dans les autres.

La bibliothèque `urwid` n'est malheureusement pas parmi les bibliothèques standards Python. Vous devez l'installer via l'outil `pip3` :

```
pi@raspberrypi:~ $ sudo pip3 install urwid
```

Voyons les éléments nécessaires à l'écriture d'une application avec `urwid`. Ouvrez une console Python et tapez :

```
>>> import urwid
>>> texte = urwid.Text("Bonjour")
>>> filler = urwid.Filler(texte)
>>> main = urwid.MainLoop(filler)
>>> main.run()
```

Le message "Bonjour" apparaît alors au milieu de l'écran. Le positionnement par défaut des composants d'une application s'effectue toujours de la gauche vers la droite. Un autre aspect à prendre en compte est la mise en forme des éléments dans la console. Pour mieux comprendre, ouvrez un terminal et agrandissez celui-ci au maximum. Imaginez deux axes X et Y. L'axe X est horizontal et commence au coin gauche supérieur. L'axe Y est vertical et commence au même point. Ils représentent respectivement l'axe des abscisses et l'axe des ordonnées :

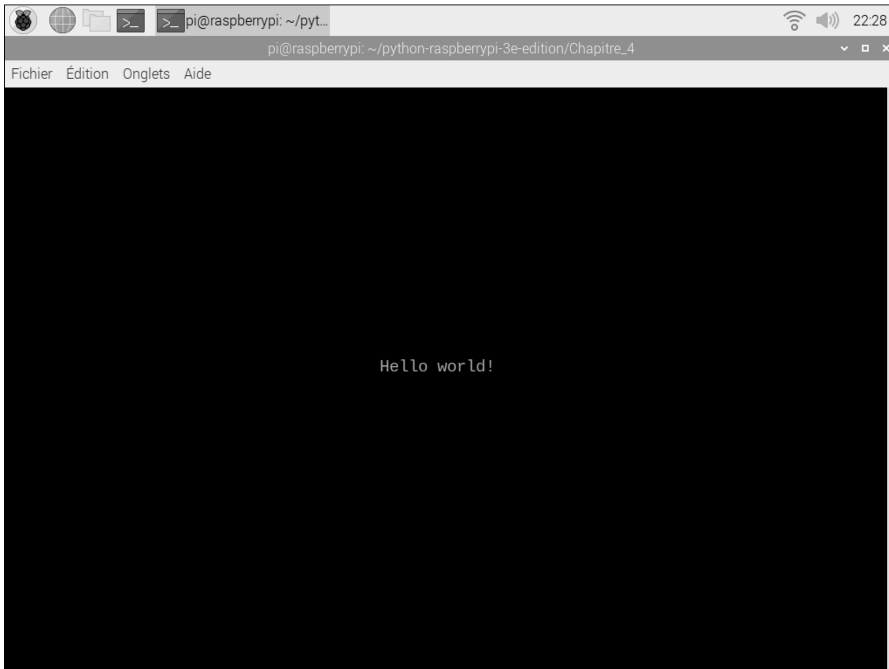
© Editions ENI - All rights reserved

La disposition de ces axes s'inspire de la bibliothèque `curses` sur laquelle `urwid` repose en grande partie.

Sans plus tarder, voici l'exemple classique du "Hello world" (*Chapitre_4/urwid_1.py*) :

```
1 #!/usr/bin/env python3
2 import urwid
3
4
5 def sortie(touche):
6     if touche in ("q", "Q"):
7         raise urwid.ExitMainLoop()
8
9
10 def main():
11     texte = urwid.Text("Hello world!")
12     padding = urwid.Padding(texte, align="center", width="pack")
13     interface = urwid.Filler(padding, valign="middle")
14     main = urwid.MainLoop(interface, unhandled_input=sortie)
15     main.run()
16
17
18 if __name__ == "__main__":
19     main()
```

Le résultat de l'exécution de ce script est le suivant :



Cet exemple s'articule autour de quatre widgets, qui sont respectivement :

- `Text`
- `Padding`
- `Filler`
- `MainLoop`

Le widget `Text` prend en paramètre une chaîne de caractères et l'affiche. Celui-ci s'imbrique ensuite dans deux autres widgets : `Padding` et `Filler`. Ces widgets remplissent la zone d'affichage en prenant soin de placer les éléments qu'ils contiennent au centre de cette zone, notamment en indiquant les alignements horizontal et vertical via les attributs `align` et `valign`. Enfin, l'élément final est passé au widget `MainLoop`. Parmi ces quatre widgets, `MainLoop` est le plus important, car il permet de démarrer la boucle événementielle `urwid`. Notez que ce widget accepte une fonction pour capturer les touches appuyées au clavier. Dans cet exemple, les touches **q** et **Q** ferment le programme.

Ce bref tour d'horizon pose les bases nécessaires pour mieux appréhender la bibliothèque `urwid`.

3. Projet #1 : une horloge en console

Voyons comment construire une application légèrement plus complexe, à savoir une horloge géante affichée dans la console (*Chapitre_4/urwid_2.py*) :

```
1 #!/usr/bin/env python3
2 import urwid
3 import time
```

La logique du programme est encapsulée au sein de la classe Horloge afin d'en réduire la complexité. L'initialisation de la classe configure l'horloge, définit la palette de couleurs, instancie la boucle événementielle et attache une alarme de 1 seconde à une fonction :

```
6 class Horloge:
7     def __init__(self):
8         self.configurer_horloge()
9         self.palette = [("horloge", "dark blue", "")]
10         self.boucle = urwid.MainLoop(
11             self.horloge,
12             palette=self.palette,
13             unhandled_input=self.quitter)
14         self.boucle.set_alarm_in(1, self.actualiser)
```

© Editions ENI - All rights reserved

Nous étudierons plus tard l'utilité d'une alarme. Mais en premier lieu, examinons la configuration de l'horloge :

```
21     def configurer_horloge(self):
22         text = time.strftime("%H:%M:%S")
23         font = urwid.font.HalfBlock7x7Font()
24         self.texte_horloge = urwid.BigText(text, font)
25         self.horloge = urwid.Filler(
26             urwid.AttrMap(
27                 urwid.Padding(
28                     self.texte_horloge,
29                     "center",
30                     width="clip"
31                 ), "horloge"
32             ), "middle"
33         )
```

La logique du programme consiste à imbriquer des widgets les uns dans les autres. Cependant, deux widgets retiennent notre attention : `BigText` et `HalfBlock7x7Font`. `BigText`, comme son nom l'indique, est un widget utilisé pour afficher un texte large en utilisant une police adéquate. La palette de couleurs à la ligne 9 indique qu'un élément ayant la propriété `horloge` est de couleur bleu foncé. Enfin, la construction de l'horloge est une suite d'appels aux widgets `Filler`, `AttrMap` et `Padding`. À cet instant, l'horloge est pratiquement opérationnelle. Il reste à gérer l'actualisation du temps :

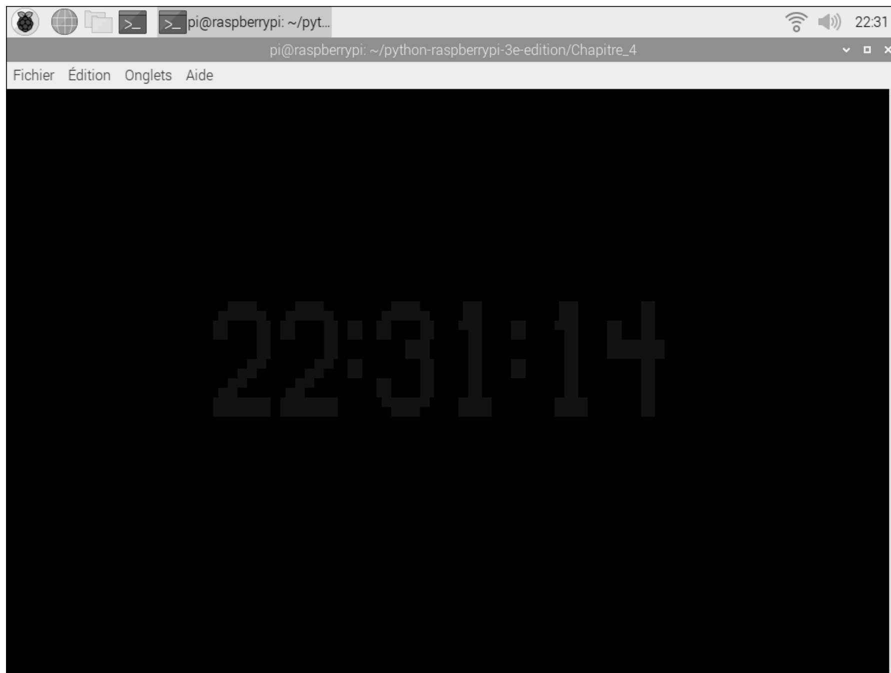
```
16     def actualiser(self, boucle=None, data=None):
17         self.configurer_horloge()
18         boucle.widget = self.horloge
19         boucle.set_alarm_in(1, self.actualiser)
```

La fonction `actualiser` est attachée à une alarme appelée lorsque la seconde est écoulée, à la manière d'un compte à rebours. L'astuce de la mise à jour réside ensuite à reconstruire l'horloge, ce qui incrémentera le temps de 1 seconde, et à attacher à nouveau la fonction `actualiser` à une alarme de 1 seconde. Ainsi, l'horloge est mise à jour automatiquement à chaque seconde !

Enfin, le programme se lance lorsque la fonction `demarrer` de notre classe est appelée :

```
35     def demarrer(self):
36         self.boucle.run()
37
38     def quitter(self, touche):
39         if touche in ("q", "Q"):
40             raise urwid.ExitMainLoop()
41
42
43 if __name__ == "__main__":
44     horloge = Horloge()
45     horloge.demarrer()
```

Et voici le résultat en image :



Vous pouvez maintenant lire l'heure depuis la ligne de commande !

4. Projet #2 : un navigateur de fichiers en console

Pour continuer l'étude de la bibliothèque, voici un programme légèrement plus complexe, à savoir un navigateur de fichiers (*Chapitre_4/urwid_3.py*) :

```
1 #!/usr/bin/env python3
2 import urwid
3 import os
```

© Editions ENI - All rights reserved

Comme dans l'exemple précédent, toute la logique de création du navigateur de fichiers est encapsulée dans une classe appelée `NavigateurFichiers`. À sa création, cette classe initialise un certain nombre de variables et procède à un listing des fichiers du répertoire courant :

```
6 class NavigateurFichiers(urwid.WidgetPlaceholder):
7     def __init__(self, chemin):
8         self.chemin_courant = chemin
9         self.chemin_prec = None
10         self.liste_fichiers = sorted(os.listdir(self.chemin_courant))
11         if self.chemin_courant != "/":
12             self.liste_fichiers.insert(0, "..")
13         super(NavigateurFichiers, self).__init__(self.creer_contenu())
```

La variable `liste_fichiers` est stockée comme membre de l'instance afin d'être réutilisée plus tard. Le code prend aussi soin d'ajouter le chemin `..` en début de liste pour que l'utilisateur puisse remonter d'un répertoire lors de la navigation. Voyons en détail la fonction `creer_contenu` :

```
30     def creer_contenu(self):
31         contenu = []
32         for fichier in self.liste_fichiers:
33             bouton = urwid.Button(fichier)
34             if os.path.isdir(os.path.join(self.chemin_courant, fichier)):
35                 urwid.connect_signal(
36                     bouton, "click", self.maj_contenu, fichier
37                 )
38                 contenu.append(urwid.AttrMap(
39                     bouton, "rep", focus_map="rep_focus")
40                 )
41             else:
42                 contenu.append(urwid.AttrMap(
43                     bouton, None, focus_map="fichier"))
44         return urwid.AttrMap(
45             urwid.LineBox(
46                 urwid.ListBox(
47                     urwid.SimpleFocusListWalker(contenu)),
48                 self.chemin_courant), "contenu"
49         )
```

Cette fonction parcourt la liste des éléments du répertoire courant. Lorsqu'il s'agit d'un répertoire, un bouton est créé et, lorsqu'il est sélectionné, appelle la fonction `maj_contenu`. Si l'élément est un fichier, un simple bouton est créé sans aucune fonction particulière attachée. Ces boutons sont à chaque fois ajoutés dans une liste : `contenu`. La fin de l'exécution de la fonction retourne alors quatre widgets imbriqués les uns dans les autres :

- `AttrMap`
- `LineBox`
- `ListBox`
- `SimpleFocusListWalker`