Commissariat Général à l'Energie Atomique
Centre Régional d'Etudes Nucléaires de Kinshasa
C.G.E.A./C.R.E.N-K.

Concepts avancés en Python

Prof. Dr Saint-Jean DJUNGU

Février 2024

Tél : +243 818899608 *email :saintjean.djungu.ahuka@cgea-rdc.org*

Chapitre 1 : Classes et objets

1.1. Introduction à la programmation orientée objet

La *programmation orientée objet* (POO) est un concept de programmation très puissant qui permet de structurer ses programmes d'une manière nouvelle. En POO, on définit un *objet* qui peut contenir des *attributs* ainsi que des *méthodes* qui agissent sur lui-même. Par exemple, on définit un objet *citron* qui contient les attributs *saveur* et *couleur*, ainsi qu'une méthode *presser* permettant d'en extraire le jus.

En Python, on utilise une *classe* pour construire un objet. Dans notre exemple, la classe correspondrait au *moule* utilisé pour construire autant d'objets citrons que nécessaire.

La POO permet de rédiger du code plus compact et mieux réutilisable. L'utilisation de classes évite l'utilisation de variables globales en créant ce qu'on appelle un *espace de noms* propre à chaque objet permettant d'y *encapsuler* des attributs et des méthodes. De plus, la POO amène de nouveaux concepts tels que le *polymorphisme* (capacité à redéfinir le comportement des opérateurs), ou bien encore l'*héritage* (capacité à définir une classe à partir d'une classe préexistante et d'y ajouter de nouvelles fonctionnalités). Tous ces concepts seront définis dans ce chapitre.

1.2. Classes en Python

1.2.1. Définition

Une *classe* est une structure de données particulière ainsi que les opérations permettant de la manipuler.

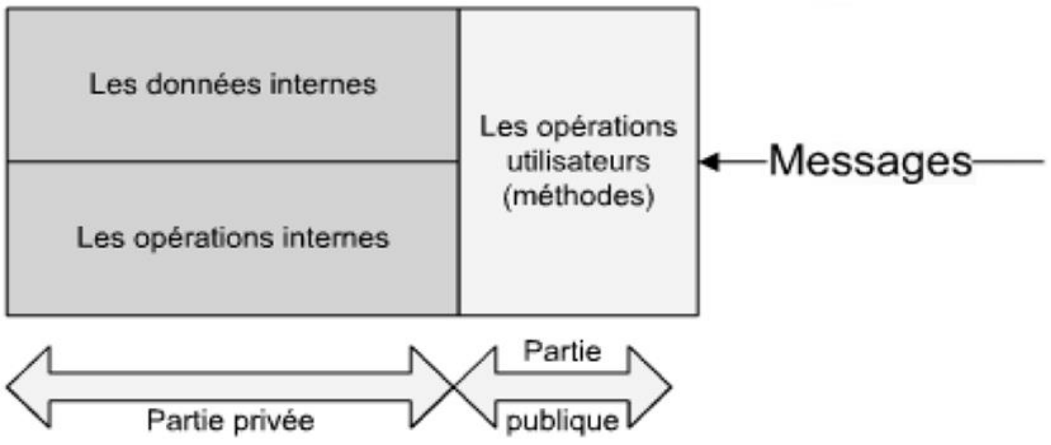


Figure 1.1 : Une classe est une boite noire

Une *classe* est un type permettant de regrouper dans la même structure : les informations (champs, propriétés, attributs) relatives à une entité ; les procédures et fonctions permettant de les manipuler (méthodes). Champs et méthodes constituent les membres de la classe.

Les classes doivent être perçues comme des types évolués, qui peuvent représenter un nombre incroyablement divers de choses : des personnes, des étudiants, des avions, des ordinateurs, des livres, des bibliothèques, des romans, des joueurs, etc., et même des évènements ou des échanges.

Utiliser (ou définir) une nouvelle classe, c'est en quelque sorte utiliser (ou définir) un nouveau type avec des traitements associés. Mais définir un type et les opérations associées n'est pas suffisant : encore faut-il savoir l'utiliser. Pour les entiers, il suffit de définir une variable de type entier et de la manipuler. Pour les classes, il faut construire un objet.

Une *classe* définit donc des *objets* qui sont des *instances* (des représentants) de cette classe. Dans ce chapitre on utilisera les mots *objet* ou *instance* pour désigner la même chose.

Les classes sont les principaux outils de la POO. Ce type de programmation permet de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent, entre eux et avec le monde extérieur. Le premier bénéfice de cette approche de la programmation réside dans le fait que les différents objets utilisés peuvent être construits indépendamment les uns des autres (par exemple par des programmeurs différents) sans qu'il n'y ait de risque d'interférence. Ce résultat est obtenu grâce au concept d'encapsulation : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte « enfermées » dans l'objet. Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des procédures bien définies : l'interface de l'objet.

1.2.2. Constituants d'une classe

Une *classe* est constituée des *attributs* (ou *propriétés*, ou encore des variables associées) et des *méthodes* (ou *opérations*, ou encore *fonctions associées*).

Un *attribut* est une variable interne à la classe et une méthode n'est rien d'autre qu'une fonction associée à une classe.

Un *constructeur* est une méthode spéciale d'une classe. Il est appelé pour construire une instance de classe.

Nota : Python intègre des particularités – pour ne pas dire bizarreries – que l'on ne retrouve pas dans les langages objets populaires (C++, Java, C#). Par exemple, création à la volée d'une propriété sur un objet (instance de classe), possibilité de l'utiliser dans les méthodes (alors qu'elle n'apparaît nulle part dans la définition de la classe).

1.2.3. Construction et utilisation d'une classe en Python

En Python, le mot-clé *class* permet de créer sa propre classe, suivi du nom de cette classe.

Exemple

Prenons pour exemple, la création de la classe nommée *Personne* dans le module désigné *ModulePersonne.py***.**

```
#début définition
class Personne:
   """Classe Personne"""
   #constructeur
   def __init__(self):
            #lister les champs
            self.nom = ""
            self.age = 0
            self.salaire = 0.0
   #fin constructeur
#fin définition
```

Dans cet exemple, *nom*, *age* et *salaire* sont des attributs alors que la méthode *__init__()* est le constructeur qui permet d'initialiser les attributs.

Tous les attributs et toutes les méthodes d'une classe se réfèrent toujours à self qui désigne l'objet lui-même. Attention, les méthodes prennent au moins self comme argument.

1.2.4. Variable de classe

Une *variable de classe* est un champ directement accessible sur la classe, et qui est partagée par toutes les instances de la classe.

Exemple

```
#début définition
class Personne:
   """Classe Personne"""
   #variable de classe
    Compteur = 0
   #constructeur
   def __init__(self):
            #lister les champs
            self.nom = ""
            self.age = 0
            self.salaire = 0.0
            Personne.compeur += 1
   #fin constructeur
#fin définition
```

1.3. Gestion des objets

Considérons la classe Personne définie à la section 1.2. Nous avons ci-dessous, un exemple d'un programme qui permet d'instancier la classe Personne dans le programme intitulé *instancePersonne.py*.

```
#appel du module
import ModulePersonne as MP
#instanciation
p = MP.Personne()
#affectation aux champs
p.nom = input("Nom : ")
p.age = int(input("Age : "))
p.salaire = float(input("Salaire : "))
#affichage
print(p.nom,", ",p.age,", ",p.salaire)
```

Exécution

Nom : Nonge

Age : 25

Salaire : 2450

Nonge , 25 , 2450.0

1.4. Classes complètes

a. Exemple

Prenons cette fois-ci, la classe Personne ayant en plus du constructeur, les méthodes *saisie* et *affichage*. Le module *ModulePersonne2* se présente comme suit :

```
#début définition
class Personne:
    """Classe Personne"""
    #constructeur
    def __init__(self):
            #lister les champs
            self.nom = ""
            self.age = 0
            self.salaire = 0.0
    #fin constructeur
    #saisie des infos
    def saisie(self):
            self.nom = input("Nom : ")
            self.age = int(input("Age : "))
            self.salaire = float(input("Salaire : "))
    #fin saisie
    #affichage des infos
    def affichage(self):
             print("Son nom est ", self.nom)
             print("Son âge : ", self.age)
             print("Son salaire : ", self.salaire)
    #fin affichage
#fin définition
```

Le programme qui permet d'utiliser la classe *Personne* devient très simple et se présente comme suit :

```
#appel du module
import ModulePersonne2 as MP
#instanciation
p = MP.Personne()
#saisie
p.saisie()
#méthode affichage
p.affichage()
```

Exécution :

```
Nom : Anael
Age : 5
Salaire : 300
Son nom est  Anael
Son âge :  5
Son salaire :  300.0
```

b. Méthodes paramétrées

Rajouter dans la classe *Personne* la méthode *retraite()* qui calcule le nombre d'années avant l'âge limite de la retraite d'un employé. Le programme devient :

```
#début définition
class Personne:
    """Classe Personne"""
    #constructeur
    def __init__(self):
        #lister les champs
        self.nom = ""
```

```
            self.age = 0
            self.salaire = 0.0
        #fin constructeur
        #saisie des infos
        def saisie(self):
            self.nom = input("Nom : ")
            self.age = int(input("Age : "))
            self.salaire = float(input("Salaire : "))
        #fin saisie
        #affichage des infos
        def affichage(self):
            print("Son nom est ", self.nom)
            print("Son âge : ", self.age)
            print("Son salaire : ", self.salaire)
        #fin affichage
        #reste avant retraite
        def retraite(self, limite):
            reste = limite - self.age
            if (reste < 0):
                print("Vous etes a la retraite")
            else:
                print("Il vous reste %s années" % (reste))
            #fin retraite
    #fin définition
```

Programme d'utilisation

```
    #appel du module
    import ModulePersonne3 as MP
    #instanciation
    p = MP.Personne()
    #saisie
```

```
p.saisie()
#méthode affichage
p.affichage()
#reste avant retaite
p.retraite(65)
```

Exécution

```
Nom : Nael
Age : 38
Salaire : 2100
Son nom est  Nael
Son âge :  38
Son salaire :  2100.0
Il vous reste 27 années
```

c. Gestion d'une collection d'objets

Considérons cette fois-ci un programme Python que permet d'assurer la gestion d'un ensemble des personnes.

```
#appel du module
import ModulePersonne2 as MP
#liste vide
liste = []
#nombre de personnes ?
n = int(input("Nb de pers : "))
#saisie liste
for i in range(0,n):
        a = MP.Personne()
        a.saisie()
        liste.append(a)
#affichage
print("*** début affichage ")
```

```
for p in liste:
        print("-------")
        p.affichage()
```

Exécution

```
Nb de pers : 2
Nom : Josh
Age : 18
Salaire : 5240
Nom : Dana
Age : 15
Salaire : 3670
*** début affichage
-------
Son nom est  Josh
Son âge :  18
Son salaire :  5240.0
-------
Son nom est  Dana
Son âge :  15
Son salaire :  3670.0
```

d. Accès indicé et modification des objets

Voyons maintenant la possibilité d'accéder à une information de l'ensemble des personnes et de la modifier. On peut, par exemple, multiplier le salaire de quelqu'un de la collection par 2.

```
#appel du module
import ModulePersonne2 as MP
#liste vide
liste = []
#nb. de pers ?
n = int(input("Nb de pers : "))
```

```
#saisie liste
for i in range(0,n):
    a = MP.Personne()
    a.saisie()
    liste.append(a)
#accès par numéro
numero = int(input("N° ind. à traiter :"))
if (numero < len(liste)):
    b = liste[numero]
    b.salaire = b.salaire * 2
    #affichage de nouveau
    print("xxx début affichage 2")
    for p in liste:
        print("-------")
        p.affichage()
else:
    print("indice non valable")
```

Exécution

```
Nb de pers : 2
Nom : Josh
Age : 18
Salaire : 5240
Nom : Dana
Age : 15
Salaire : 3670
N° ind. à traiter :1
xxx début affichage 2
-------
Son nom est  Josh
Son âge :  18
Son salaire :  5240.0
```

```
-------
Son nom est  Dana
Son âge :  15
Son salaire :  7340.0
```

Dans cet exemple, c'est le salaire de Dana, deuxième sur la liste et indice 1, qui est finalement multiplié par deux.

1.5. Héritage et polymorphisme

Les classes constituent le principal outil de la programmation orientée objet, qui est considérée de nos jours comme la technique de programmation la plus performante. L'un des principaux atouts de ce type de programmation réside dans le fait que l'on peut toujours se servir d'une classe préexistante pour en créer une nouvelle, qui héritera toutes ses propriétés mais pourra modifier certaines d'entre elles et/ou y ajouter les siennes propres. Le procédé s'appelle *dérivation*. Il permet de créer toute une hiérarchie de classes allant du général au particulier.

1.5.1. Héritage

L'*héritage* peut évoquer la capacité qu'ont nos parents à nous transmettre certains traits physiques ou de caractère (ne dit-on pas, j'ai hérité ceci ou cela de ma mère ou de mon père ?). Qu'en est-il en programmation ?

a. Définition

En programmation, l'*héritage* est la capacité d'une classe d'hériter des propriétés d'une classe préexistante. On parle de *classe mère* et de *classe fille*. En Python, l'héritage peut être multiple lorsqu'une classe fille hérite de plusieurs classes mères.

b. Exemple

Nous pouvons par exemple définir une classe Personne, qui contienne un ensemble de caractéristiques propres. À partir de cette classe parente, nous pouvons dériver une ou plusieurs classes filles, comme : une classe *Employe*, une classe *Homme*, une classe *Femme*, etc., qui hériteront toutes les caractéristiques de la classe Personne, en y ajoutant leurs spécificités.

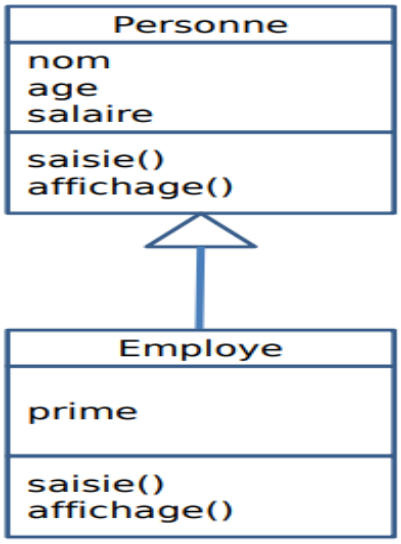


Figure 1.2 : Héritage

Déclaration en Python

class Employe(Personne):

...

Sur base de la figure 1.2, la classe Employe hérite de tous les attributs (*nom*, *age*, *salaire*) et toutes les méthodes (*saisie(), affichage()*) de la classe Personne, mais possède en plus l'attribut *prime*. On va procéder comme suit dans la sous-classe Employé :

Personne.attribut = ...

Personne.nomMéthode() : ...

1.5.2. Polymorphisme

Si une méthode de la classe fille possède le même nom que celle de la classe mère, c'est la première qui prend la priorité. Dans ce cas, on dit que la méthode est redéfinie ou surcharge des méthodes.

Exemple : ModulePersonne.py

```
#début définition de la classe mère
class Personne:
    """Classe Personne"""
    #constructeur
    def __init__(self):
```

```
        #lister les champs
        self.nom = ""
        self.age = 0
        self.salaire = 0.0
    #fin constructeur
    #saisie des infos
    def saisie(self):
        self.nom = input("Nom : ")
        self.age = int(input("Age : "))
        self.salaire = float(input("Salaire : "))
    #fin saisie
    #affichage des infos
    def affichage(self):
        print("Son nom est ", self.nom)
        print("Son âge : ", self.age)
        print("Son salaire : ", self.salaire)
    #fin affichage
#fin définition
#classe Employé, sous-classe de la classe Personne
class Employe(Personne):
    #constructeur
    def __init__(self):
        Personne.__init__(self)
        self.prime = 0.0
    #fin constructeur
    #saisie
    def saisie(self):
        Personne.saisie(self)
        self.prime = float(input("Prime : "))
    #fin saisie
    #affichage
    def affichage(self):
```

```
            Personne.affichage(self)
            print("Sa prime : ", self.prime)
        #fin affichage
    #fin classe Employé
```

Instanciation : principalEmploye.py

```
    #appel du module
    import ModulePersonne as MP
    #instanciation
    e = MP.Employe()
    #saisie
    e.saisie()
    #affichage
    print(">> Affichage")
    e.affichage()
```

Exécution

```
Nom : Kevin
Age : 35
Salaire : 2500
Prime : 480
>> Affichage
Son nom est  Kevin
Son âge :  35
Son salaire :  2500.0
Sa prime :  480.0
```

1.5.3. Liste polymorphe

Prenons maintenant un exemple un peu plus complexe en faisant combiner les notions de module, liste, classe, sous-classe et polymorphisme.

```
#appel du module
import ModulePersonne as MP
#liste vide
liste = []
#nb. de pers ?
n = int(input("Nb de pers : "))
#instanciation
for i in range(0,n):
    code = input("1 Personne, 2 Employé : ")
    if (code == "1"):
        m = MP.Personne()
    else:
        m = MP.Employe()
    liste.append(m)
#saisie liste
print("*** début saisie")
for p in liste:
    print("-------")
    p.saisie()
#affichage
print(">>> début affichage")
for p in liste:
    print("-------")
    p.affichage()
```

Exécution

```
Nb de pers : 2
1 Personne, 2 Employé : 1
1 Personne, 2 Employé : 2
*** début saisie
-------
Nom : Coel
```

Age : 26

Salaire : 1000

Nom : Anael

Age : 37

Salaire : 2500

Prime : 500

>>> début affichage

Son nom est Coel

Son âge : 26

Son salaire : 1000.0

Son nom est Anael

Son âge : 37

Son salaire : 2500.0

Sa prime : 500.0

1.6. Exercices

Exercice 1.1

Une boulangerie est ouverte de 7 heures à 13 heures et de 16 heures à 20 heures, sauf le lundi après-midi et le mardi toute la journée. On suppose que l'heure h est un entier entre 0 et 23. Le jour j code 0 pour lundi, 1 pour mardi, etc.

Ecrire un programme Python orienté objet qui demande le jour et l'heure, puis affiche si la boulangerie est ouverte.

Exercice 2.2

Ecrire un programme Python orienté objet qui aide un utilisateur d'internet à calculer le prix de la connexion téléphonique. Le programme demandera à l'utilisateur d'entrer l'heure à laquelle il s'est connecté et l'heure à laquelle il s'est déconnecté. Vous pouvez faire les simplifications suivantes:

- Il n'est pas nécessaire d'indiquer les minutes. On calcule donc avec des heures entières.

- L'heure du début de la connexion est toujours inférieure à l'heure de la fin de la connexion.

Cela veut donc dire que l'utilisateur n'est jamais connecté avant minuit jusqu'à après minuit et que la durée maximale d'une connexion est de 23 heures.

Les tarifs de connexion sont les suivants:

- Tarif 1 de 7h à 17h : 1 dollar/heure
- Tarif 2 de 0h à 7h et de 17h à 24h : 0.50 dollar/heure

Modulariser le programme sous forme de méthodes auxiliaires.

Exemples d'exécution:

Début de la connexion : 5
Fin de la connexion : 7
Vous avez été connecté 2 heures pour 1 dollar

Début de la connexion : 6
Fin de la connexion : 8
Vous avez été connecté 2 heures pour 1.50 dollar

Début de la connexion : 8
Fin de la connexion : 10
Vous avez été connecté 2 heures pour 2 dollars

Début de la connexion : 0
Fin de la connexion : 23
Vous avez été connecté 23 heures pour 16.50 dollars

Début de la connexion : 10
Fin de la connexion : 5
Bizarre, le début est après la fin ...

Début de la connexion : 10

Fin de la connexion : 10

Bizarre, vous n'avez pas été connecté du tout ...

Exercice 1.3

Ecrire un programme Python orienté objet récursif et itératif qui permet de calculer la factorielle d'un nombre.

Exercice 1.4

Écrire une classe nommée Carac, permettant de conserver un caractère, Elle disposera:

- d'un constructeur à un paramètre fournissant le caractère voulu ;
- d'un constructeur sans paramètre qui attribuera par défaut la valeur « espace » au caractère ;
- d'une méthode nommée estVoyelle fournissant la valeur vrai lorsque le caractère concerné est une voyelle et la valeur faux dans le cas contraire.

Écrire un petit programme Python utilisant cette classe.

Exercice 1.5

Écrire une classe Rectangle disposant :

- de trois constructeurs : le premier sans paramètre créera un rectangle dont les deux dimensions sont égales à 1 ; le second à un paramètre sera utilisé à la fois pour les deux dimensions, considérées comme égales ; le troisième à deux paramètres correspondant aux deux dimensions du rectangle. Les dimensions seront de type réel ;
- d'une méthode périmètre fournissant en résultat le périmètre du rectangle ;
- d'une méthode surface fournissant en résultat la surface du rectangle ;
- d'une méthode agrandit disposant d'un paramètre de type réel correspondant à la valeur par laquelle il faut multiplier les dimensions du rectangle.

Écrire un petit programme Python d'utilisation.

Exercice 1.6

On se propose d'établir les résultats d'examen d'un ensemble d'étudiants. Chaque étudiant sera représenté par un objet de type Etudiant, comportant obligatoirement les champs suivants :

- le nom de l'étudiant (type *chaine*),
- son admissibilité à l'examen, sous forme d'une valeur d'un type énuméré comportant les valeurs suivantes : N (non admis), P (passable), AB (*Assez bien)*, B (*Bien*), TB (*Très bien).*

Les noms des étudiants sont contenus dans un fichier. On demande de définir convenablement la classe Etudiant et d'écrire un programme principal qui :

- pour chaque étudiant, lit dans le fichier 3 notes d'examen, en calcule la moyenne et renseigne convenablement le champ d'admissibilité, suivant les règles usuelles :
  - moyenne < 10 : Non admis
  - 10 <= moyenne <12 : Passable
  - 12 <= moyenne <14 : Assez bien
  - 14 <= moyenne <16 : Bien
  - 16 <= moyenne : Très bien
- affiche l'ensemble des résultats en fournissant en clair la mention obtenue.

Chapitre 2 : Interfaces graphiques

1.1. Introduction

Jusqu'à ce niveau, tous les exemples de code que nous avons réalisés fonctionnent exclusivement en mode caractères. Les informations sont affichées dans une console et également saisies à partir de celle-ci. La simplicité de ce mode de fonctionnement est un atout indéniable pour l'apprentissage d'un langage. Par contre, la plupart des utilisateurs de vos futures applications s'attendent certainement à avoir une interface un petit peu moins austère qu'un écran en mode caractères. Nous allons donc étudier dans ce chapitre comment fonctionnent les interfaces graphiques avec Python.

Une interface graphique utilisateur (en anglais *Graphical User Interface GUI*), est une façade visuelle du programme qui le lie avec l'extérieur et qui va donc faciliter la tâche de l'utilisateur. Ce type d'interface homme-machine s'oppose à la notion de ligne de commande où la majorité de l'interaction entre l'utilisateur et l'ordinateur se fait au clavier, sans visualisation élaborée, dans un terminal ou dans une fenêtre de terminal en mode texte.

Un programme basé sur une interface graphique, est piloté par les événements de l'utilisateur (clic sur un bouton, choix dans une liste, saisie d'un texte, etc.). On parle de la *programmation événementielle*, car le programme est piloté par les événements (actions) de l'utilisateur.

Si vous ne le saviez pas encore, apprenez dès à présent que le domaine des interfaces graphiques est extrêmement complexe. Chaque système d'exploitation peut en effet proposer plusieurs « bibliothèques » de fonctions graphiques de base, auxquelles viennent fréquemment s'ajouter de nombreux compléments, plus ou moins spécifiques de langages de programmation particuliers. Tous ces composants sont généralement présentés comme des classes d'objets, dont il vous faudra étudier les attributs et les méthodes.

Avec Python, la bibliothèque graphique la plus utilisée jusqu'à présent est la bibliothèque *tkinter*, qui est une adaptation de la bibliothèque *Tk* développée à l'origine pour le langage *Tcl*. Plusieurs autres bibliothèques graphiques fort intéressantes ont été proposées pour Python : *wxPython*, *pyQT*, *pyGTK*, etc.

Dans le cadre de ce cours, nous nous limiterons cependant à *tkinter*, dont il existe fort heureusement des versions similaires (et gratuites) pour les plates-formes Linux, Windows et Mac OS.

1.2. Premiers pas avec tkinter

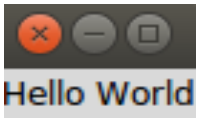
Pour la suite des explications, nous supposerons bien évidemment que le module *tkinter* a déjà été installé sur votre système. Pour pouvoir en utiliser les fonctionnalités dans un script Python, il faut que l'une des premières lignes de ce script contienne l'instruction d'importation:

```
from tkinter import *
```

Comme toujours sous Python, il n'est même pas nécessaire d'écrire un script. Vous pouvez faire un grand nombre d'expériences directement à la ligne de commande, en ayant simplement lancé Python en mode interactif. Dans l'exemple qui suit, nous allons créer une fenêtre très simple, et y ajouter deux widgets typiques : un bout de texte (ou label) et un bouton (ou button).

```
from tkinter import *
fen = Tk()
tex = Label(fen, text='Hello World')
tex.pack()
fen.mainloop()
```

Une fenêtre comme celle-ci devrait apparaître :



Examinons à présent plus en détail chacune des lignes de commandes exécutées :

1. Comme cela a déjà été expliqué précédemment, il est aisé de construire différents modules Python, qui contiendront des scripts, des définitions de fonctions, des classes d'objets, etc. On peut alors importer tout ou partie de ces modules dans n'importe quel programme, ou même dans l'interpréteur fonctionnant en mode interactif (c'est-à-dire directement à la ligne de commande). C'est ce que nous faisons à la première ligne de notre exemple : *from tkinter import ** consiste à importer toutes les classes contenues dans le module *tkinter*.

2. À la deuxième ligne de notre exemple : *fen = Tk()*, nous utilisons l'une des classes du module *tkinter*, la classe *Tk()*, et nous en créons une instance (autre terme désignant un objet spécifique), à savoir la fenêtre *fen*.

3. À la troisième ligne : *tex = Label(fen, text='Hello World')*, nous créons un autre objet, cette fois à partir de la classe Label(). Comme son nom l'indique, cette classe définit toutes sortes d'étiquettes (ou de libellés). En fait, il s'agit tout simplement de fragments de texte quelconques, utilisables pour afficher des informations et des messages divers à l'intérieur d'une fenêtre.

Quels arguments avons-nous donc fournis pour cette instanciation ?

- Le premier argument transmis (*fen*), indique que le nouveau widget que nous sommes en train de créer sera contenu dans un autre widget préexistant, que nous désignons donc ici comme son « maître » : l'objet *fen* est le widget maître de l'objet *tex*. On pourra dire aussi que l'objet *tex* est un widget esclave de l'objet *fen*.
- Le deuxième argument sert à préciser la forme exacte que doit prendre notre objet. C'est en effet une option de création qui fournie sous la forme d'une chaîne de caractères le texte de l'étiquette.

4. À la quatrième ligne de notre exemple : *tex.pack()*, nous activons une méthode associée à l'objet *tex* : la méthode *pack()*. La méthode *pack()* fait partie d'un ensemble de méthodes qui sont applicables non seulement aux widgets de la classe *Label()*, mais aussi à la plupart des autres widgets *tkinter*, et qui agissent sur leur disposition géométrique dans la fenêtre.

5. La septième ligne : *fen.mainloop()* est très importante, parce que c'est elle qui provoque le démarrage du réceptionnaire d'événements associé à la fenêtre. Cette instruction est nécessaire pour que notre application soit « à l'affût » des clics de souris, des pressions exercées sur les touches du clavier, etc. C'est donc cette instruction qui « la met en marche », en quelque sorte.

Comme son nom l'indique (*mainloop*), il s'agit d'une méthode de l'objet *fen*, qui active une boucle de programme, laquelle « tournera » en permanence en tâche de fond, dans l'attente de messages émis par le système d'exploitation de l'ordinateur. Celui-ci interroge en effet sans cesse son environnement, notamment au niveau des périphériques d'entrée (souris, clavier, etc.). Lorsqu'un événement quelconque est détecté, divers messages décrivant cet événement sont expédiés aux programmes qui souhaitent en être avertis.

2.3. Widget Tkinter

Pour créer un logiciel graphique vous devez ajouter dans une fenêtre des éléments graphiques que l'on nomme *widget*. Ce *widget* peut être tout aussi bien une liste déroulante que du texte. En gros, il existe 15 classes de base pour les widgets tkinter :

Widget	**Description**
Button	Un bouton classique, à utiliser pour provoquer l'exécution d'une commande quelconque
Canvas	Un espace pour disposer divers éléments graphiques. Ce widget peut être utilisé pour dessiner, créer des éditeurs graphiques, et aussi pour implémenter des widgets personnalisés
Checkbutton	Une case à cocher qui peut prendre deux états distincts (la case est cochée ou non). Un clic sur ce widget provoque le changement d'état
Entry	Un champ d'entrée, dans lequel l'utilisateur du programme pourra insérer un texte quelconque à partir du clavier
Frame	Une surface rectangulaire dans la fenêtre, où l'on peut disposer d'autres widgets. Cette surface peut être colorée. Elle peut aussi être décorée d'une bordure
Label	Un texte (ou libellé) quelconque (éventuellement une image)
Listbox	Une liste de choix proposés à l'utilisateur, généralement présentés dans une sorte de boîte. On peut également configurer la Listbox de telle manière qu'elle se comporte comme une série de « boutons radio » ou de cases à cocher
Menu	Un menu. Ce peut être un menu déroulant attaché à la barre de titre, ou bien un menu « pop up » apparaissant n'importe où à la suite d'un clic
Menubutton	Un bouton-menu, à utiliser pour implémenter des menus déroulants
Message	Permet d'afficher un texte. Ce widget est une variante du widget. Label, qui permet d'adapter automatiquement le texte affiché à une certaine taille ou à un certain rapport largeur/hauteur
Radiobutton	Représente (par un point noir dans un petit cercle) une des valeurs d'une variable qui peut en posséder plusieurs. Cliquer sur un bouton radio donne la valeur correspondante à la variable, et « vide » tous les autres boutons radio associés à la même variable
Scale	Vous permet de faire varier de manière très visuelle la valeur d'une variable, en déplaçant un curseur le long d'une règle
Scrollbar	Ascenseur ou barre de défilement que vous pouvez utiliser en association avec les autres widgets : Canvas, Entry, Listbox, Text
Text	Affichage de texte formaté. Permet aussi à l'utilisateur d'éditer le texte affiché. Des images peuvent également être insérées
Toplevel	Une fenêtre affichée séparément, au premier plan

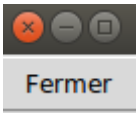
Tableau 2.1 : Classes des widgets tkinter

Les classes de widgets intègrent chacune un grand nombre de méthodes. On peut aussi leur associer (lier) des événements, comme nous venons de le voir dans les sections précédentes. Tous ces widgets peuvent être positionnés dans les fenêtres à l'aide de trois méthodes différentes : la méthode *pack()*, la méthode *grid()* et la méthode *place()*.

2.2.1. Boutons

Les *boutons* permettent de proposer une action à l'utilisateur. Dans l'exemple ci-dessous, on lui propose de fermer la fenêtre.

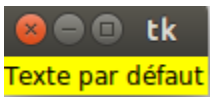
```
from tkinter import *
fen = Tk()
# bouton de sortie
bouton=Button(fen, text="Fermer", command=fen.quit)
bouton.pack()
fen.mainloop()
```



2.2.2. Labels

Les *labels* sont des espaces prévus pour écrire du texte. Les labels servent souvent à décrire un widget comme un input.

```
from tkinter import *
fen = Tk()
# label
label = Label(fen, text="Texte par défaut", bg="yellow")
label.pack()
fen.mainloop()
```



Il est possible d'indiquer une valeur de couleur par son nom en anglais: "*white*", "*black*", "*red*", "*yellow*", etc. ou par son code hexadécimale: #000000, #00FFFF, etc.

background (ou *bg*) : couleur de fond du widget

foreground (ou *fg*) : couleur de premier plan du widget

activebackground : couleur de fond du widget lorsque celui-ci est actif

activeForeground : couleur de premier plan du widget lorsque le widget est actif

disableForeground : couleur de premier plan du widget lorsque le widget est désactivé

highlightbackground : couleur de fond de la région de surbrillance lorsque le widget a le focus

highlightcolor : couleur de premier plan de la région de surbrillance lorsque le widget a le focus

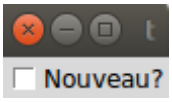
selectbackground : couleur de fond pour les éléments sélectionnés

selectforeground : couleur de premier plan pour les éléments sélectionnés

2.2.3. Case à cocher

Les *checkbox* proposent à l'utilisateur de cocher une option.

```
from tkinter import *
fen = Tk()
# case à cocher
bouton = Checkbutton(fen, text="Nouveau?")
bouton.pack()
fen.mainloop()
```



2.2.4. Boutons radio

Les boutons radio sont des cases à cocher qui sont dans un groupe et dans ce groupe seul un élément peut être sélectionné.

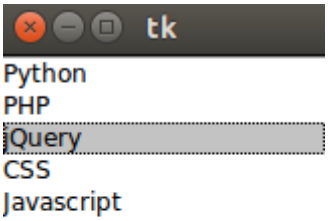
```
from tkinter import *
fen = Tk()
# Boutons radio
value = StringVar()
bouton1 = Radiobutton(fen, text="Oui", variable=value, value=1)
bouton2 = Radiobutton(fen, text="Non", variable=value, value=2)
bouton3 = Radiobutton(fen, text="Peu être", variable=value, value=3)
```

```
bouton1.pack()
bouton2.pack()
bouton3.pack()
fen.mainloop()
```

2.2.5. Listes

Les listes permettent de récupérer une valeur sélectionnée par l'utilisateur.

```
from tkinter import *
fen = Tk()
# liste
liste = Listbox(fen)
liste.insert(1, "Python")
liste.insert(2, "PHP")
liste.insert(3, "jQuery")
liste.insert(4, "CSS")
liste.insert(5, "Javascript")
liste.pack()
fen.mainloop()
```



2.2.6. Canvas

Un *canvas* (toile, tableau en français) est un espace dans lequel vous pouvez dessiner ou écrire ce que vous voulez :

```
from tkinter import *
fen = Tk()
# canvas
canvas = Canvas(fen, width=150, height=120, background='yellow')
```

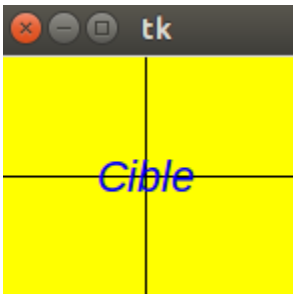
ligne1 = canvas.create_line(75, 0, 75, 120)

ligne2 = canvas.create_line(0, 60, 150, 60)

txt = canvas.create_text(75, 60, text="Cible", font="Arial 16 italic", fill="blue")

canvas.pack()

fen.mainloop()



Vous pouvez créer d'autres éléments :

- *create_arc()* : arc de cercle
- *create_bitmap()* : bitmap
- *create_image()* : image
- *create_line()* : ligne
- *create_oval()* : ovale
- *create_polygon()* : polygone
- *create_rectanle()* : rectangle
- *create_text()* : texte
- *create_window()* : fenêtre

Si vous voulez changer les coordonnées d'un élément crée dans le canevas, vous pouvez utiliser la méthode *coords*.

- *canvas.coords(element, x0, y0, x1, y1)*

Pour supprimer un élément vous pouvez utiliser la méthode *delete*

- *canvas.delete(element)*

Vous pouvez trouver d'autres méthodes utiles en exécutant l'instruction suivante :

- *print dir(Canvas())*

2.2.6. Scale

Le widget *scale* permet de récupérer une valeur numérique via un scroll.

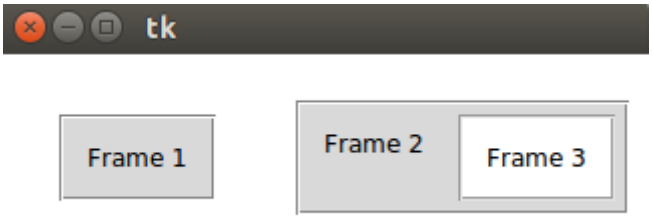

```
from tkinter import *
fen = Tk()
value = DoubleVar()
scale = Scale(fen, variable=value)
scale.pack()
fen.mainloop()
```

2.2.7. Frames

Les frames (cadres) sont des conteneurs qui permettent de séparer des éléments.

```
from tkinter import *
fen = Tk()
fen['bg']='white'

# frame 1
Frame1 = Frame(fen, borderwidth=2, relief=GROOVE)
Frame1.pack(side=LEFT, padx=30, pady=30)

# frame 2
Frame2 = Frame(fen, borderwidth=2, relief=GROOVE)
Frame2.pack(side=LEFT, padx=10, pady=10)

# frame 3 dans frame 2
Frame3 = Frame(Frame2, bg="white", borderwidth=2, relief=GROOVE)
Frame3.pack(side=RIGHT, padx=5, pady=5)

# Ajout de labels
```

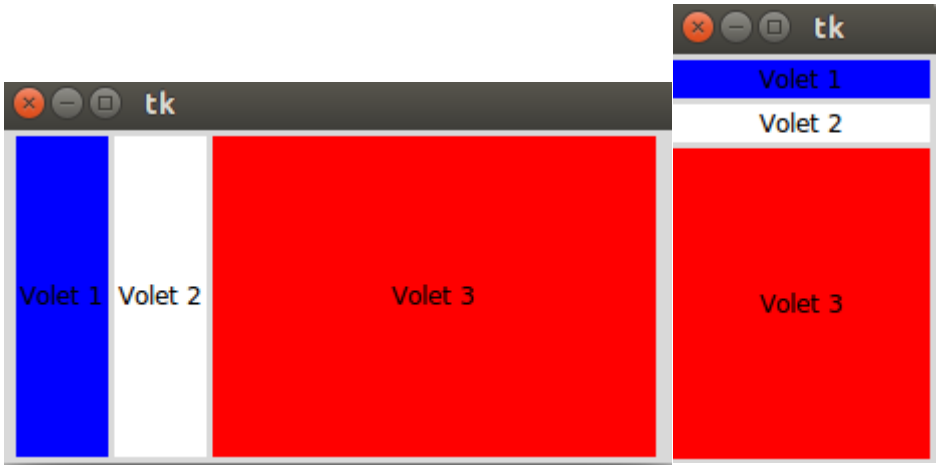
```
Label(Frame1, text="Frame 1").pack(padx=10, pady=10)
Label(Frame2, text="Frame 2").pack(padx=10, pady=10)
Label(Frame3, text="Frame 3",bg="white").pack(padx=10, pady=10)
fen.mainloop()
```



2.2.8. PanedWindow

Le *panedwindow* est un conteneur qui peut contenir autant de panneaux que nécessaire disposé horizontalement ou verticalement.

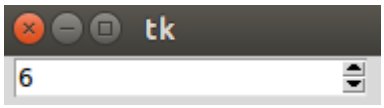
```
from tkinter import *
fen = Tk()
p = PanedWindow(fen, orient=HORIZONTAL)
p.pack(side=TOP, expand=Y, fill=BOTH, pady=2, padx=2)
p.add(Label(p, text='Volet 1', background='blue', anchor=CENTER))
p.add(Label(p, text='Volet 2', background='white', anchor=CENTER) )
p.add(Label(p, text='Volet 3', background='red', anchor=CENTER) )
p.pack()
fen.mainloop()
```



2.2.9. Spinbox

La *spinbox* propose à l'utilisateur de choisir un nombre

```
from tkinter import *
fen = Tk()
s = Spinbox(fen, from_=0, to=10)
s.pack()
fen.mainloop()
```



2.2.10. LabelFrame

Le *labelframe* est un cadre avec un label.

```
from tkinter import *
fen = Tk()
l = LabelFrame(fen, text="Titre de la frame", padx=20, pady=20)
l.pack(fill="both", expand="yes")
Label(l, text="A l'intérieure de la frame").pack()
fen.mainloop()
```

2.2.11. Alertes

Pour pouvoir utiliser les alertes de votre os, vous pouvez importer le module **tkMessageBox** (Python 2).

```
form tkMessageBox import *
```

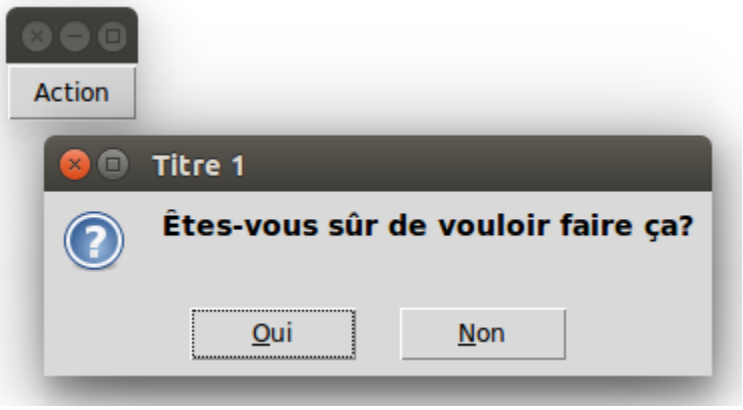
Pour python 3 :

```
form tkinter.messagebox import *
```

Exemple d'utilisation :

```
from tkinter import *
from tkinter.messagebox import *
fen = Tk()
def callback():
    if askyesno('Titre 1', 'Êtes-vous sûr de vouloir faire ça?'):
        showwarning('Titre 2', 'Tant pis...')
```

```
    else:
        showinfo('Titre 3', 'Vous avez peur!')
        showerror("Titre 4", "Aha")
Button(text='Action', command=callback).pack()
fen.mainloop()
```



Voici les alertes possibles :

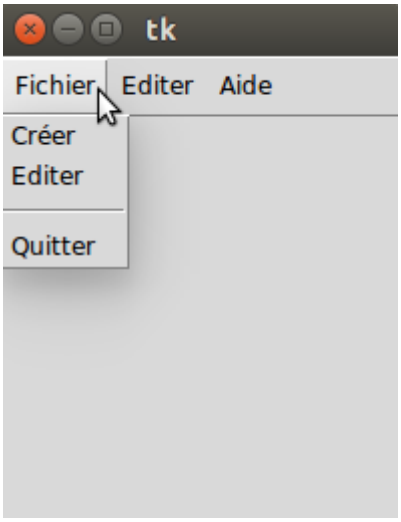
- *showinfo()*
- *showwarning()*
- *showerror()*
- *askquestion()*
- *askokcancel()*
- *askyesno()*
- *askretrycancel()*

2.2.12. Barre de menu

Il est possible de créer une barre de menu comme-ceci :

```
from tkinter import *
fen = Tk()
def alert():
    showinfo("alerte", "Bravo!")
menubar = Menu(fen)


menu1 = Menu(menubar, tearoff=0)
menu1.add_command(label="Créer", command=alert)
menu1.add_command(label="Editer", command=alert)
```

```
menu1.add_separator()
menu1.add_command(label="Quitter", command=fen.quit)
menubar.add_cascade(label="Fichier", menu=menu1)

menu2 = Menu(menubar, tearoff=0)
menu2.add_command(label="Couper", command=alert)
menu2.add_command(label="Copier", command=alert)
menu2.add_command(label="Coller", command=alert)
menubar.add_cascade(label="Editer", menu=menu2)
menu3 = Menu(menubar, tearoff=0)
menu3.add_command(label="A propos", command=alert)
menubar.add_cascade(label="Aide", menu=menu3)

fen.config(menu=menubar)
fen.mainloop()
```



2.3. Actions diverses

2.3.1. Placer des widgets

Il est possible de placer les widgets à l'aide du paramètre *side* :

side=TOP : haut

side=LEFT : gauche

side=BOTTOM : bas

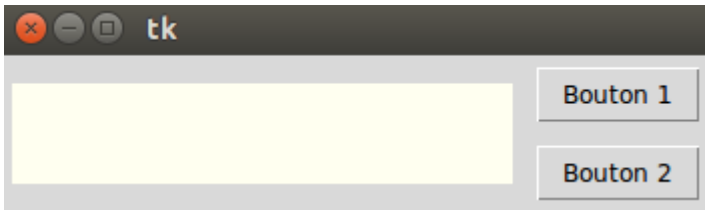
side=RIGHT : droite

Exemple :

```
from tkinter import *
fen = Tk()
Canvas(fen, width=250, height=100, bg='ivory').pack(side=TOP, padx=5, pady=5)
Button(fen, text ='Bouton 1').pack(side=LEFT, padx=5, pady=5)
Button(fen, text ='Bouton 2').pack(side=RIGHT, padx=5, pady=5)
fen.mainloop()
```

Autre exemple :

```
from tkinter import *
fen = Tk()
Canvas(fen, width=250, height=50, bg='ivory').pack(side=LEFT, padx=5, pady=5)
Button(fen, text ='Bouton 1').pack(side=TOP, padx=5, pady=5)
Button(fen, text ='Bouton 2').pack(side=BOTTOM, padx=5, pady=5)
fen.mainloop()
```



2.3.2. Unités et options

a. Unités de dimensions

Si vous indiquez une dimension à travers un *integer*, l'unité utilisée sera les "*pixels*". Il est cependant possible de changer l'unité :

i : pouces

m : millimètre

c : centimètre

b. Options de dimensions

height : Hauteur du widget

width : Largeur du widget

padx, pady : Espace supplémentaire autour du widget. X pour horizontal et V pour vertical

borderwidth: Taille de la bordure

highlightthickness : Largeur du rectangle lorsque le widget a le focus

selectborderwidth : Largeur de la bordure tridimensionnelle autour du widget sélectionné

wraplength : Nombre de ligne maximum pour les widgets en mode « word wrapping »

2.4. Autres spécificités

2.4.1. Curseur

Vous pouvez changer l'apparence de votre curseur:

Button(fenetre, text ="arrow", relief=RAISED, cursor="arrow").pack()

Button(fenetre, text ="circle", relief=RAISED, cursor="circle").pack()

Button(fenetre, text ="clock", relief=RAISED, cursor="clock").pack()

Button(fenetre, text ="cross", relief=RAISED, cursor="cross").pack()

Button(fenetre, text ="dotbox", relief=RAISED, cursor="dotbox").pack()

Button(fenetre, text ="exchange", relief=RAISED, cursor="exchange").pack()

Button(fenetre, text ="fleur", relief=RAISED, cursor="fleur").pack()

Button(fenetre, text ="heart", relief=RAISED, cursor="heart").pack()

Button(fenetre, text ="man", relief=RAISED, cursor="man").pack()

```
Button(fenetre, text ="mouse", relief=RAISED, cursor="mouse").pack()
Button(fenetre, text ="pirate", relief=RAISED, cursor="pirate").pack()
Button(fenetre, text ="plus", relief=RAISED, cursor="plus").pack()
Button(fenetre, text ="shuttle", relief=RAISED, cursor="shuttle").pack()
Button(fenetre, text ="sizing", relief=RAISED, cursor="sizing").pack()
Button(fenetre, text ="spider", relief=RAISED, cursor="spider").pack()
Button(fenetre, text ="spraycan", relief=RAISED, cursor="spraycan").pack()
Button(fenetre, text ="star", relief=RAISED, cursor="star").pack()
Button(fenetre, text ="target", relief=RAISED, cursor="target").pack()
Button(fenetre, text ="tcross", relief=RAISED, cursor="tcross").pack()
Button(fenetre, text ="trek", relief=RAISED, cursor="trek").pack()
Button(fenetre, text ="watch", relief=RAISED, cursor="watch").pack()
```

2.4.2. Relief

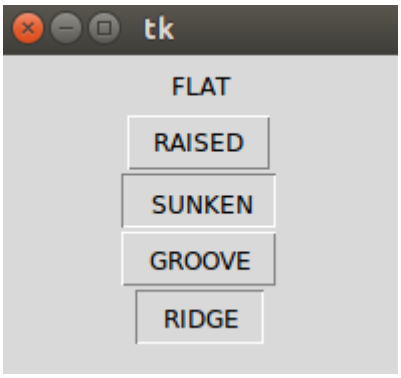
Vous pouvez changer le relief sur vos éléments :

```
FLAT
RAISED
SUNKEN
GROOVE
RIDGE
from tkinter import *
fen = Tk()
b1 = Button(fen, text ="FLAT", relief=FLAT).pack()
b2 = Button(fen, text ="RAISED", relief=RAISED).pack()
b3 = Button(fen, text ="SUNKEN", relief=SUNKEN).pack()
```

b4 = Button(fen, text ="GROOVE", relief=GROOVE).pack()

b5 = Button(fen, text ="RIDGE", relief=RIDGE).pack()

fen.mainloop()



2.4.3. Grille

Il est possible de placer les éléments en raisonnant en grille:

*from tkinter import **

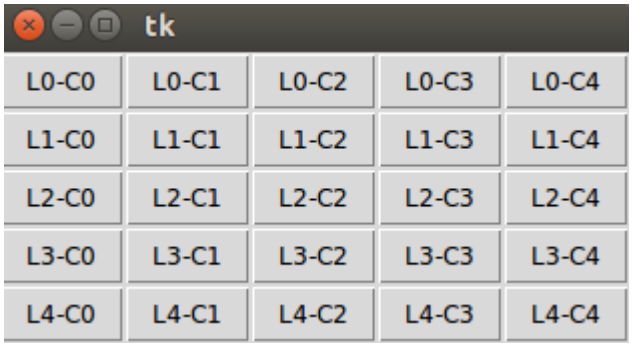
fen = Tk()

for ligne in range(5):

for colonne in range(5):

Button(fen, text='L%s-C%s' % (ligne, colonne), borderwidth=1).grid(row=ligne, column=colonne)

fen.mainloop()



*from tkinter import **

fen = Tk()

Button(fen, text='L1-C1', borderwidth=1).grid(row=1, column=1)

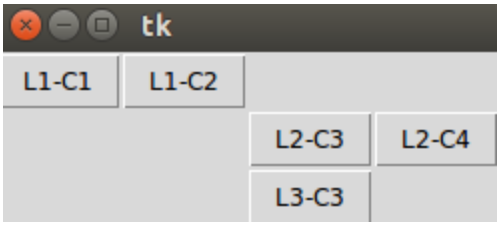
Button(fen, text='L1-C2', borderwidth=1).grid(row=1, column=2)

Button(fen, text='L2-C3', borderwidth=1).grid(row=2, column=3)

Button(fen, text='L2-C4', borderwidth=1).grid(row=2, column=4)

Button(fen, text='L3-C3', borderwidth=1).grid(row=3, column=3)

fen.mainloop()



2.4.4. Intégrer une image

Pour intégrer une image vous pouvez créer un canevas et l'ajouter à l'intérieur comme ceci :

*from tkinter import **

fen = Tk()

photo = PhotoImage(file="DL1A2996.jpg")

canvas = Canvas(fen,width=350, height=200)

canvas.create_image(0, 0, anchor=NW, image=photo)

canvas.pack()

fen.mainloop()

2.4.5. Récupérer la valeur d'un input

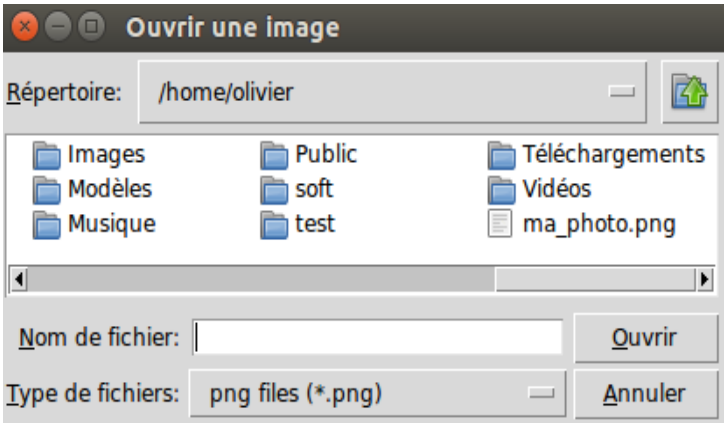
Pour récupérer la valeur d'un input il vous faudra utiliser la méthode *get()* :

```
from tkinter import *
from tkinter import messagebox
fen = Tk()
def recupere():
   messagebox.showinfo("Alerte", entree.get())

value = StringVar()
value.set("Valeur")
entree = Entry(fen, textvariable=value, width=30)
entree.pack()
bouton = Button(fen, text="Valider", command=recupere)
bouton.pack()
fen.mainloop()
```

2.4.6. Récupérer une image et l'afficher

Pour cela, vous devez importer le module suivant :

```
from tkinter import *
from tkinter.filedialog import *
filepath = askopenfilename(title="Ouvrir une image",filetypes=[('png files','.png'),('all files','.*')])
photo = PhotoImage(file=filepath)
canvas = Canvas(fenetre, width=photo.width(), height=photo.height(), bg="yellow")
canvas.create_image(0, 0, anchor=NW, image=photo)
canvas.pack()
fen.mainloop()
```



La fonction *askopenfilename* retourne le chemin du fichier que vous avez choisi avec le nom de celui-ci.

Exemple : */home/olivier/ma_photo.png*

2.5. Evènements

Vous pouvez récupérer les actions utilisateurs à travers les *events* (évènement en français).

Pour chaque widget, vous pouvez *binder* (lier en français) un évènement, par exemple dire lorsque l'utilisateur appuie sur telle touche, faire cela. Voici un exemple qui récupère les touches appuyées par l'utilisateur :

```
from tkinter import *
fen = Tk()
def clavier(event):
    touche = event.keysym
    print(touche)


canvas = Canvas(fen, width=500, height=500)
canvas.focus_set()
canvas.bind("<Key>", clavier)
canvas.pack()
fen.mainloop()
```

On remarque que l'évènement est encadré par des chevrons. D'autres évènements existent:

<Button-1> : Click gauche

<Button-2> : Click milieu

<Button-3> : Click droit

<Double-Button-1> : Double click droit

<Double-Button-2> : Double click gauche

<KeyPress> : Pression sur une touche

<KeyPress-a> : Pression sur la touche **A** (minuscule)

<KeyPress-A> : Pression sur la touche **A** (majuscule)

<Return> : Pression sur la touche entrée

<Escape> : Touche Echap

<Up> : Pression sur la flèche directionnelle haut

<Down> : Pression sur la flèche directionnelle bas

<ButtonRelease> : Lorsque qu'on relache le click

<Motion> : Mouvement de la souris

<B1-Motion> : Mouvement de la souris avec click gauche

<Enter> : Entrée du curseur dans un widget

<Leave> : Sortie du curseur dans un widget

<Configure> : Redimensionnement de la fenêtre

<Map> <Unmap> : Ouverture et iconification de la fenêtre

<MouseWheel> : Utilisation de la roulette

Pour supprimer la liaison de l'évènement, vous pouvez utiliser les méthodes *unbind* ou *unbind_all*.

Voici un exemple où l'on peut bouger un carré avec les touches directionnelles :

```
from tkinter import *
```

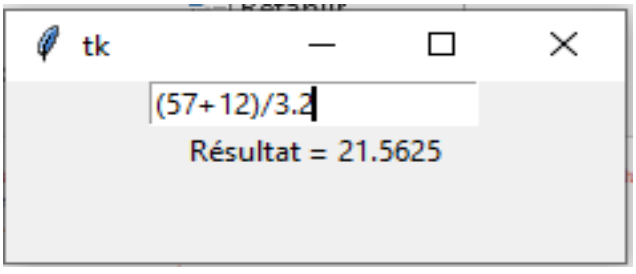
```
fen = Tk()
# fonction appelée lorsque l'utilisateur presse une touche
def clavier(event):
    global coords
    touche = event.keysym
    if touche == "Up":
        coords = (coords[0], coords[1] - 10)
    elif touche == "Down":
        coords = (coords[0], coords[1] + 10)
    elif touche == "Right":
        coords = (coords[0] + 10, coords[1])
    elif touche == "Left":
        coords = (coords[0] -10, coords[1])
    # changement de coordonnées pour le rectangle
    canvas.coords(rectangle, coords[0], coords[1], coords[0]+25, coords[1]+25)

# création du canvas
canvas = Canvas(fen, width=250, height=250, bg="ivory")
# coordonnées initiales
coords = (0, 0)
# création du rectangle
rectangle = canvas.create_rectangle(0,0,25,25,fill="violet")
# ajout du bond sur les touches du clavier
canvas.focus_set()
canvas.bind("<Key>", clavier)
```

```
# création du canvas
canvas.pack()
fen.mainloop()
```

2.6. Application : Calculatrice minimaliste

Bien que très court, le petit script ci-dessous implémente une calculatrice complète, avec laquelle vous pourrez même effectuer des calculs comportant des parenthèses et des fonctions scientifiques. N'y voyez rien d'extraordinaire. Toute cette fonctionnalité n'est qu'une conséquence du fait que vous utilisez un interpréteur plutôt qu'un compilateur pour exécuter vos programmes.

```
#Exemple utilisant la bibliothèque graphique tkinter et le module math
from tkinter import *
from math import *
# définition de l'action à effectuer si l'utilisateur actionne
# la touche "enter" alors qu'il édite le champ d'entrée :
def evaluer(event):
    chaine.configure(text = "Résultat = " + str(eval(entree.get())))

# ----- Programme principal : -----
fen = Tk()
entree = Entry(fen)
entree.bind("<Return>", evaluer)
chaine = Label(fen)
entree.pack()
chaine.pack()
fen.mainloop()
```



Chapitre 3 : Applications web

3.1. Pages web interactives

Le protocole HTTP qui gère la transmission des pages web autorise l'échange de données dans les deux sens. Mais dans le cas de la simple consultation de sites, le transfert d'informations a surtout lieu dans l'un des deux, à savoir du serveur vers le navigateur : des textes, des images, des fichiers divers lui sont expédiés en grand nombre (ce sont les pages consultées) ; en revanche, le navigateur n'envoie guère au serveur que de toutes petites quantités d'information : essentiellement les adresses URL des pages que l'internaute désire consulter.

Vous savez qu'il existe des sites web où vous êtes invité à fournir vous-même des quantités d'information plus importantes : vos références personnelles pour l'inscription à un club ou la réservation d'une chambre d'hôtel, votre numéro de carte de crédit pour la commande d'un article sur un site de commerce électronique, votre avis ou vos suggestions, etc.

Dans ces cas-là, vous vous doutez bien que l'information transmise doit être prise en charge, du côté du serveur, par un *programme* spécifique. Il faudra donc associer étroitement ce programme au serveur web. Quant aux pages web destinées à accueillir cette information (on les appelle des formulaires), il faudra les doter de divers widgets d'encodage (champs d'entrée, cases à cocher, boîtes de listes, etc.), afin que le navigateur puisse soumettre au serveur une *requête* accompagnée d'arguments. Le serveur pourra alors confier ces arguments au programme de traitement spécifique, et en fonction du résultat de ce traitement, renvoyer une réponse adéquate à l'internaute, sous la forme d'une nouvelle page web.

Il existe différentes manières de réaliser de tels programmes spécifiques, que nous appellerons désormais *applications web*.

L'une des plus répandues à l'heure actuelle consiste à utiliser des pages HTML «enrichies» à l'aide de scripts écrits à l'aide d'un langage spécifique tel que PHP. Ces scripts sont directement insérés dans le code HTML, entre des balises particulières, et ils seront exécutés par le serveur web (Apache, par exemple) à condition que celui-ci soit doté du module interpréteur adéquat. Il est possible de procéder de cette façon avec Python via une forme légèrement modifiée du langage nommée PSP (*Python Server Pages*).

Cette approche présente toutefois l'inconvénient de mêler trop intimement le code de présentation de l'information (le HTML) et le code de manipulation de celle-ci (les fragments de script PHP ou PSP insérés entre balises), ce qui compromet gravement la lisibilité de l'ensemble. Une meilleure approche consiste à écrire des scripts distincts, qui génèrent du code HTML « classique » sous la forme de chaînes de caractères, et de doter le serveur web d'un module approprié pour interpréter ces scripts et renvoyer le code HTML en réponse aux requêtes du navigateur (par exemple *mod_python*, dans le cas d'Apache).

Mais avec Python, nous pouvons pousser ce type de démarche encore plus loin, en développant nous-mêmes un véritable serveur web spécialisé, tout à fait autonome, qui contiendra en un seul logiciel la fonctionnalité particulière souhaitée pour notre application. Il est en effet parfaitement possible de réaliser cela à l'aide de Python, car toutes les bibliothèques nécessaires à la gestion du protocole HTTP sont intégrées au langage. Partant de cette base, de nombreux programmeurs indépendants ont d'ailleurs réalisé et mis à la disposition de la communauté une série d'outils de développement pour faciliter la mise au point de telles applications web spécifiques. Pour la suite de notre étude, nous utiliserons donc l'un d'entre eux. Nous avons choisi *Cherrypy*, car il nous semble particulièrement bien adapté aux objectifs de ce cours.

3.2. Installation et utilisation de Cherrypy

3.2.1. Installation

Notons que *Cherrypy* est une bibliothèque externe à Python. Pour ce faire, son installation se fait via l'outil *pip* dans le terminal du système :

```
py -m pip install cherryp
```

3.2.2. Mise en ligne d'une page web minimaliste

Dans votre répertoire de travail, préparez un petit fichier texte que vous nommerez *tutoriel.conf*, et qui contiendra les lignes suivantes :

```
[global]
server.socket_host = "127.0.0.1"
server.socket_port = 8080
server.thread_pool = 5
tools.sessions.on = True
```

tools.encode.encoding = "Utf-8"

[\annexes]

tools.staticdir.on = True

tools.staticdir.dir = "annexes"

Il s'agit d'un simple fichier de configuration que notre serveur web Cherrypy consultera au démarrage. Notez surtout le numéro de port utilisé (8080 dans notre exemple).

La ligne *server.thread_pool =5* indique le nombre de *threads* que le serveur *Cherrypy* devra ouvrir pour pouvoir traiter en parallèle les requêtes provenant en même temps d'utilisateurs différents.

Remarquez également la ligne concernant l'encodage. Il s'agit de l'encodage que *Cherrypy* devra utiliser dans les pages web produites. Il est possible que certains navigateurs attendent une autre norme que Utf-8 comme encodage par défaut.

Les 3 dernières lignes du fichier indiquent le chemin d'un répertoire où vous placerez les documents « statiques » dont votre site peut avoir besoin (images, feuilles de style, etc.). Veuillez à présent encoder le petit script ci-dessous :

```
import cherrypy
class MonSiteWeb(object): # Classe maîtresse de l'application
    def index(self): # Méthode invoquée comme URL racine (\)
        return "<h1>Bonjour à tous !</h1>"
    index.exposed = True # la méthode doit être 'publiée'
###### Programme principal : ##############
cherrypy.quickstart(MonSiteWeb(), config ="tutoriel.conf")
```

Lancez l'exécution du script. Si tout est en ordre, vous obtenez quelques lignes d'information similaires aux suivantes dans votre terminal. Elles vous confirment que «quelque chose » a démarré, et reste en attente d'événements :

```
[03/Nov/2023:13:11:51] ENGINE Listening for SIGTERM.
[03/Nov/2023:13:11:51] ENGINE Bus STARTING
[03/Nov/2023:13:11:51] ENGINE Started monitor thread 'Autoreloader'.
```

[03/Nov/2023:13:11:51] ENGINE Serving on http://127.0.0.1:8080

[03/Nov/2023:13:11:51] ENGINE Bus STARTED

Vous venez effectivement de mettre en route un serveur web !

Il ne vous reste plus qu'à vérifier qu'il fonctionne bien, à l'aide de votre navigateur préféré. Si vous utilisez ce navigateur sur la même machine que le serveur, dirigez-le vers une adresse telle que http://localhost:8080, *localhost* étant une expression consacrée pour désigner la machine locale (vous pouvez également spécifier celle-ci à l'aide de l'adresse IP conventionnelle : 127.0.0.1), et 8080 le numéro de port choisi dans le fichier de configuration. Vous devriez obtenir la page d'accueil suivante :

Vous pouvez tout aussi bien accéder à cette même page d'accueil depuis une autre machine, en fournissant à son navigateur l'adresse IP ou le nom de votre serveur sur le réseau local, en lieu et place de *localhost*.

Examinons à présent notre script d'un peu plus près. Sa concision est remarquable : seulement 6 lignes effectives !

Après importation du module *cherrypy*, on y définit une nouvelle classe *MonSiteWeb()*. Les objets produits à l'aide de cette classe seront des gestionnaires de requêtes. Leurs méthodes seront invoquées par un dispositif interne à Cherrypy, qui convertira l'adresse URL demandée par le navigateur, en un appel de méthode avec un nom équivalent (nous illustrerons mieux ce mécanisme avec l'exemple suivant). Si l'URL reçue ne comporte aucun nom de page, comme c'est le cas ici, c'est le nom index qui sera recherché par défaut, suivant une convention bien établie sur le Web. C'est pour cette raison que nous avons nommé ainsi notre unique méthode, qui attend donc les requêtes s'adressant à la racine du site.

- Ligne 4 : les méthodes de cette classe vont donc traiter les requêtes provenant du navigateur, et lui renvoyer en réponse des chaînes de caractères contenant du texte rédigé en *HTML*. Pour ce premier exemple, nous avons simplifié au maximum le code *HTML* produit, le résumant à un petit message inséré entre deux balises de titre (*<h1>* et *</h1>*). En toute rigueur, nous aurions dû insérer le tout entre balises *<html>* *</html>* et *<body></body>* afin de réaliser une mise en page correcte. Mais puisque cela peut

déjà fonctionner ainsi, nous attendrons encore un peu avant de montrer nos bonnes manières.

- Ligne 5 : les méthodes destinées à traiter une requête HTTP et à renvoyer en retour une page web, doivent être « publiées » à l'aide d'un attribut *exposed* contenant une valeur « vraie ». Il s'agit là d'un dispositif de sécurité mis en place par *Cherrypy*, qui fait que par défaut, toutes les méthodes que vous écrivez sont protégées vis-à-vis des tentatives d'accès extérieurs indésirables. Les seules méthodes accessibles seront donc celles qui auront été délibérément rendues publiques à l'aide de cet attribut.
- Ligne 7 : la fonction *quickstart()* du module *cherrypy* démarre le serveur proprement dit. Il faut lui fournir en argument la référence de l'objet gestionnaire de requêtes qui sera la racine du site, ainsi que la référence d'un fichier de configuration générale.

3.2.3. Ajout d'une deuxième page

Le même objet gestionnaire peut bien entendu prendre en charge plusieurs pages :

```
import cherrypy
class MonSiteWeb(object):
    def index(self):
    # Renvoi d'une page HTML contenant un lien vers une autre page
    # (laquelle sera produite par une autre méthode du même objet) :
        return '''
            <h2>Veuillez <a href="unMessage">cliquer ici</a>
            pour accéder à une information d'importance cruciale.</h2>
        '''
    index.exposed = True
    def unMessage(self):
        return "<h1>La programmation, c'est génial !</h1>"
    unMessage.exposed = True
cherrypy.quickstart(MonSiteWeb(), config ="tutoriel.conf")
```

Ce script dérive directement du précédent. La page renvoyée par la méthode *index()* contient cette fois une balise-lien dont l'argument est l'URL d'une autre page. Si cette URL est un simple nom, la page correspondante est supposée se trouver dans le répertoire racine du site. Dans la logique de conversion des URL utilisée par *Cherrypy*, cela revient à invoquer une

méthode de l'objet racine possédant un nom équivalent. Dans notre exemple, la page référencée sera donc produite par la méthode *unMessage()*.

3.3. Présentation et traitement d'un formulaire

Les choses vont vraiment commencer à devenir intéressantes avec le script suivant :

```
import cherrypy
class Bienvenue(object):
    def index(self):
    # Formulaire demandant son nom à l'utilisateur :
        return '''
            <form action="salutations" method="GET">
                Bonjour. Quel est votre nom ?
                <input type="text" name="nom" />
                <input type="submit" value="OK" />
            </form>
        '''
     index.exposed = True
     def salutations(self, nom =None):
         if nom: # Accueil de l'utilisateur :
             return "Bonjour, {0}, comment allez-vous ?".format(nom)
    else: # Aucun nom n'a été fourni :
         return 'Veuillez svp fournir votre nom <a href="/">ici</a>.'
    salutations.exposed = True
    cherrypy.quickstart(Bienvenue(), config ="tutoriel.conf")
```

La méthode *index()* de notre objet racine présente cette fois à l'utilisateur une page web contenant un formulaire : le code HTML inclus entre les balises <*form*> et </*form*> peut en effet contenir un ensemble de *widgets* divers, à l'aide desquels l'internaute pourra encoder des informations et exercer une certaine activité : champs de saisie, cases à cocher, boutons radio, boîtes de listes, etc. Pour ce premier exemple, un champ et un bouton suffiront :

Ligne 6 : la balise <*form*> doit contenir deux indications essentielles : l'action à accomplir lorsque le formulaire sera expédié (il s'agit en fait de fournir ici l'URL d'une ressource web capable de réceptionner une requête accompagnée d'arguments), et la méthode (*GET* ou *POST*) à utiliser pour transmettre ces arguments.
La différence entre *GET* et *POST* concerne la façon d'associer les arguments à la requête, à savoir : dans son en-tête (*GET*), ou en annexe (*POST*). Pour Cherrypy, cependant, cette distinction n'a aucune importance. Vous pouvez donc indifféremment utiliser l'une ou l'autre.

La **ligne 8** contient la balise HTML qui définit un champ d'entrée (balise <*input type="text" name="nom" /*>). Son attribut *name* permet d'associer une étiquette à la chaîne de caractères qui sera encodée par l'utilisateur. Lorsque le navigateur transmettra au serveur la requête HTTP correspondante, celle-ci contiendra donc cet argument bien étiqueté. Comme nous l'avons déjà expliqué plus haut, Cherrypy convertira alors cette requête en un appel de méthode classique, dans lequel l'étiquette sera associée à son argument, de la manière habituelle sous Python.

La **ligne 9** définit un widget de type « bouton d'envoi » (balise <*input type="submit" value="OK" /*>). Le texte qui doit apparaître sur le bouton est précisé par l'attribut value.

Les **lignes 14** à **19** définissent la méthode qui réceptionnera la requête, lorsque le formulaire aura été expédié au serveur. Son paramètre *nom* recevra l'argument correspondant, reconnu grâce à son étiquette homonyme. Comme d'habitude sous Python, vous pouvez définir des valeurs par défaut pour chaque paramètre (si un champ du formulaire est laissé vide par l'utilisateur, l'argument correspondant n'est pas transmis). Dans notre exemple, le paramètre *nom* contient par défaut un objet vide : il sera donc très facile de vérifier par programme, si l'utilisateur a effectivement entré un *nom* ou pas.
Le fonctionnement de tous ces mécanismes est finalement très naturel et très simple : les URL invoquées dans les pages web sont converties par *Cherrypy* en appels de méthodes possédant les mêmes noms, auxquelles les arguments sont transmis de manière tout à fait classique.

3.4. Structuration d'un site à pages multiples

Voyons à présent comment structurer notre site web, en établissant une hiérarchie entre les classes d'une manière analogue à celle qui lie les répertoires et sous-répertoires dans un système de fichiers.

Dans le script ci-dessous, soyez particulièrement attentifs à la définition les balises-liens :

```
import cherrypy

class HomePage(object):
    def __init__(self):
        # Les objets gestionnaires de requêtes peuvent instancier eux-mêmes
        # d'autres gestionnaires "esclaves", et ainsi de suite :
        self.maxime = MaximeDuJour()
        self.liens = PageDeLiens()
        # L'instanciation d'objets gestionnaires de requêtes peut bien entendu
        # être effectuée à n'importe quel niveau du programme.

    def index(self):
        return '''
            <h3>Site des adorateurs du Python royal - Page d'accueil.</h3>
            <p>Veuillez visiter nos rubriques géniales :</p>
            <ul>
                <li><a href="/entreNous">Restons entre nous</a></li>
                <li><a href="/maxime/">Une maxime subtile</a></li>
                <li><a href="/liens/utiles">Des liens utiles</a></li>
            </ul>
        '''
    index.exposed = True

    def entreNous(self):
        return '''
            Cette page est produite à la racine du site.<br />
            [<a href="/">Retour</a>]
        '''
    entreNous.exposed =True

class MaximeDuJour(object):
    def index(self):
        return '''
            <h3>Il existe 10 sortes de gens : ceux qui comprennent
            le binaire, et les autres !</h3>
            <p>[<a href="../">Retour</a>]</p>
        '''
    index.exposed = True

class PageDeLiens(object):
    def __init__(self):
        self.extra = LiensSupplementaires()
```

```
    def index(self):
        return '''
            <p>Page racine des liens (sans utilité réelle).</p>
            <p>En fait, les liens <a href="utiles">sont plutôt ici</a></p>
        '''
    index.exposed = True

    def utiles(self):
        # Veuillez noter comment le lien vers les autres pages est défini :
        # on peut procéder de manière ABSOLUE ou RELATIVE.
        return '''
            <p>Quelques liens utiles :</p>
            <ul>
                <li><a href="http://www.cherrypy.org">Site de CherryPy</a></li>
                <li><a href="http://www.python.org">Site de Python</a></li>
            </ul>
            <p>D'autres liens utiles vous sont proposés
            <a href="./extra/"> ici </a>.</p>
            <p>[<a href="../">Retour</a>]</p>
        '''
    utiles.exposed = True

class LiensSupplementaires(object):
    def index(self):
        # Notez le lien relatif pour retourner à la page maîtresse :
        return '''
            <p>Encore quelques autres liens utiles :</p>
            <ul>
                <li><a href="http://pythomium.net">Le site de l'auteur</a></li>
                <li><a href="http://ubuntu-fr.org">Ubuntu : le must</a></li>
            </ul>
            <p>[<a href="../">Retour à la page racine des liens</a>]</p>
        '''
    index.exposed = True

racine = HomePage()
cherrypy.quickstart(racine, config ="tutoriel.conf")
```

Lignes 4 à 10 : la méthode constructeur des objets racine est l'endroit idéal pour instancier d'autres objets « esclaves ». On accédera aux méthodes gestionnaires de requêtes de ceux-ci exactement comme on accède aux sous-répertoires d'un répertoire racine.

Lignes 12 à 22 : la page d'accueil propose des liens vers les autres pages du site. Remarquez la syntaxe utilisée dans les balises-liens, utilisée ici de manière à définir un chemin absolu :

- Les méthodes de l'objet racine sont référencées par un caractère / suivi de leur nom seul. Le caractère / indique que le « chemin » part de la racine du site. Exemple : */entreNous*.
- Les méthodes racines des objets esclaves sont référencées à l'aide d'un simple / suivant le nom de ces autres objets. Exemple : */maxime/*
- Les autres méthodes des objets esclaves sont référencées à l'aide de leur nom inclus dans un chemin complet : Exemple : */liens/utiles*

Lignes 36, 62 et 75 : pour retourner à la racine du niveau précédent, on utilise cette fois un chemin relatif, avec la même syntaxe que celle utilisée pour remonter au répertoire précédent dans une arborescence de fichiers (deux points).

Lignes 41-42 : vous aurez compris qu'on installe ainsi une hiérarchie en forme d'arborescence de fichiers, en instanciant des objets « esclaves » les uns à partir des autres. En suivant cette logique, le chemin absolu complet menant à la méthode *index()* de cette classe devrait être par conséquent */liens/extra/index*.

Tél : +243 818899608 *email :saintjean.djungu.ahuka@cgea-rdc.org*