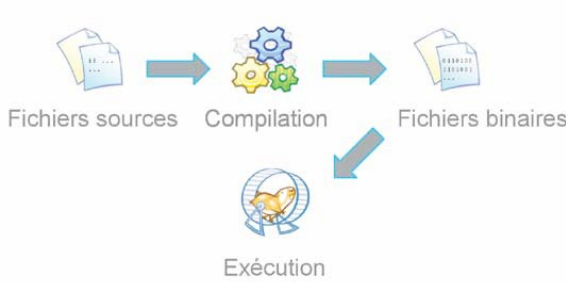
Sommaire

Le Python

Python est un langage interprété, c'est à-dire que son code n'est pas compilé pour une architecture spécifique. Cela lui permet d'être multi-plateforme : vous écrirez le même code, que vous souhaitiez exécuter votre programme sous Linux, Windows ou Mac OS X.

Langage compilé

Dans le cas d'un langage compilé, le code du programme, appelé " source " ou " code source ", sera stocké dans des fichiers textes, puis passera par une étape de compilation : le code source sera transformé en instructions directement comprises par la machine sous la forme d'un fichier binaire exécutable. L'étape de compilation sera plus ou moins longue en fonction de l'architecture et de la longueur du programme, mais elle produira toujours un code adapté à la machine sur laquelle elle aura été exécutée.



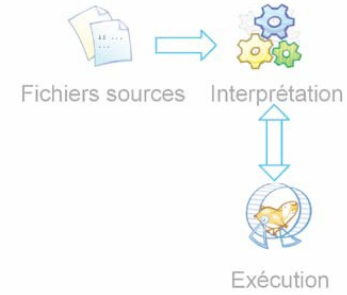
Langage compilé

La figure illustre les différentes étapes d'exécution d'un code issu d'un langage compilé. Le C et le C++ sont des exemples de langages compilés : ils sont très performants, mais le développement est plus long qu'avec un langage interprété.

Langage interprété

Dans le cas d'un langage interprété, le code source ne sera pas compilé mais sera interprété au fur et à mesure de son exécution. Cette méthode permet de s'affranchir de l'étape de compilation et ne provoque pas de crash (bien sûr des erreurs sont toujours possibles...). Par contre, suivant le code, on pourra remarquer des performances moins bonnes qu'avec un langage compilé.

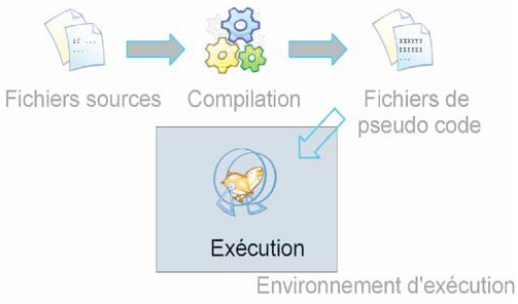
La figure illustre les étapes de l'exécution d'un code issu d'un langage interprété. Perl est un exemple de langage interprété.



Langage interprété

Langage semi-interprété

Dans le cas d'un langage semi-interprété, on va passer par une étape de compilation qui ne produira pas un code binaire mais un code intermédiaire, souvent appelé byte code, qui sera interprété dans un environnement spécifique : une machine virtuelle. Le langage modèle pour cette méthode est le Java. La figure illustre les étapes de l'exécution d'un code issu d'un langage semi-interprété.



Langage semi-interprété

Python appartient en fait à cette sous-catégorie des langages interprétés. Contrairement à Java où l'étape de compilation est distincte de l'exécution, avec Python la compilation en byte code et l'interprétation sont effectuées à la volée. Vous pouvez retrouver le byte code des fichiers sources Python : le fichier source porte l'extension .py et après exécution, vous pourrez voir qu'un fichier de même nom mais portant l'extension .pyc est apparu : c'est le fichier de byte code.

Logiciel Python

Le logiciel Python est constitué d'une interface sommaire (GUI). Il existe un nombre important d'interfaces plus jolies et plus ergonomiques, parmi celles-ci nous utiliserons de préférence EduPython. Il est important de savoir que deux branches de Python cohabitent, la version 2.7… et la branche 3… La version de l'interface Edupython est indépendante de celle de Python.

Démarrage

- Interface basique

Téléchargement à l'adresse https://www.python.org/downloads/ après installation lancer l'IDE (Python GUI)

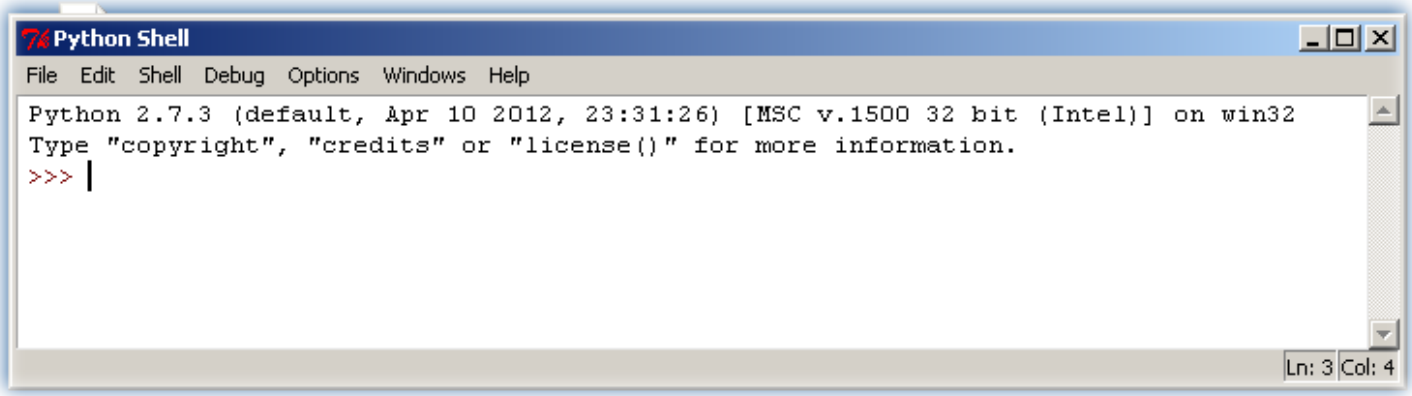
Un **environnement de développement intégré** (EDI ou IDE en anglais pour integrated development environment) est un programme regroupant un ensemble d'outils pour le développement de logiciels.

une **interface graphique** (GUI en anglais pour graphical user interface) est un dispositif de dialogue homme-machine,

- EduPython.

Une installation comprenant la base avec une interface complète peut être téléchargée à l'adresse http://download.tuxfamily.org/edupython/Setup_EP27.exe

L'interface « Python Shell » (Ligne de commande). Fenêtre « Console Python » dans EduPython



Les trois >>> représentent le **Prompt** (signal d'invite à entrer une commande).

Présentation en ligne de commande

Expérimentation

Utilisons Python comme une calculette

Entrer : (compléter avec le résultat).

>>> 5+3 [Touche entrée] A la fin de chaque saisie il faut valider avec la [Touche entrée]

>>> 5-7

*>>> 3*7*

*>>> 5+3*2*

```
>>> (5+3)*2
```

Nous constatons que les parenthèses jouent leur rôle.

```
>>> 20 / 3

>>> 20.0 / 3                Point décimal

>>> 20 / 3.0
```

Nous constatons que la division avec des entiers donne un résultat entier.

```
>>> 2 ** 3

>>> 4 ** 2
```

Nous constatons que la puissance s'écrit **.

Notions de variable.

En programmation la notion de variable permet de manipuler facilement des nombres, des chaînes de caractères,...

La notion de variable est liées à la notion de type d'objets.

Les types de variables.

Le type entier : Tout nombres entiers.

Le type réel : (ou virgule flottante) Tout réel.

Le type chaîne de caractères) : Tout assemblage de caractères (autorisés).

Le type liste : Liste d'éléments.

Nommage des variables.

Le nom des variables doit être explicite, succinct (penser à la reprise d'un programme après plusieurs mois).

Il ne doit comporter aucun caractères exotiques (seulement des lettres des chiffres et éventuellement le symbole _ pas de ponctuation pas d'accents).

Il doit commencer par une lettre.

Il ne doit pas correspondre à un nom de fonction du langage :

Mots réservés :

and	as	assert	break	class	continue	def
del	elif	else	except	False	finally	for
from	global	if	import	in	is	lambda
None	nonlocal	not	or	pass	raise	return
True	try	while	with	yield		

Manipulation des variables .

Affectation d'une valeur :

Pour affecter une valeur à une variable :

```
>>> temperature = 20
```

```
>>> message = " Il fait actuellement : "
>>> message, temperature
```

Noter les apostrophes en début et fin de la chaîne de caractères, la fonction " print " est plus appropriée pour gérer l'affichage.

```
>>> print message ; temperature

>>> print message , temperature
```

La virgule empêche le saut de ligne après une commande d'affichage " print ".

```
>>> temperature = temperature + 1
>>> print message , temperature
```

Alors que :

```
>>> message = message + 1
```

est interdit car les types ne sont pas compatibles. (message est du type chaîne de caractère, on ne peut pas lui ajouter une valeur numérique)

Éventuellement (si s'est nécessaire) :

```
>>> message = message + " + "
>>> print message , temperature
```

Attention :

```
>>> msg = " Le père Noël est passé "
>>> msg

>>> print msg
```

Il est préférable d'utiliser la fonction " print ".

Affectations multiples.

```
>>> a=b=5
>>> a

>>> b

>>> c , d = 3 , 2
>>> c

>>> d
```

Création d'un programme

A ce niveau du cours, il est plus confortable d'écrire les lignes de code dans un fichier (Voir Annexe A) afin de

pouvoir modifier le code et le relancer sans le ressaisir.

Test ou exécution conditionnelle

Test simple. IF THEN

```
a = 150
if (a > 100):
    print("a dépasse la centaine")
```

Tester avec a=80

Test double. IF THEN ELSE

```
a = 20
if (a > 100):
    print("a dépasse la centaine")
else:
    print("a ne dépasse pas cent")
```

Tests combinés. IF THEN ELIF

```
a = 0
if a > 0 :
    print("a est positif")
elif a < 0 :
    print("a est négatif")
else:
    print("a est nul")
```

Tester avec a=2 et a=-2

Opérateurs de comparaison

La condition évaluée après l'instruction if peut contenir les opérateurs de comparaison suivants :

x == y	x est égal à y
x != y	x est différent de y
x > y	x est plus grand que y
x < y	x est plus petit que y
x >= y	x est plus grand que, ou égal à y

x <= y	x est plus petit que, ou égal à y

Répétitions en boucle

l'instruction while " Tant que "

L'une des tâches que les machines font le mieux est la répétition sans erreur de tâches identiques.
Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus fondamentales : la boucle construite à partir de l'instruction while.
Entrer les commandes ci-dessous : (vous pouvez écrire les commentaires derrière le caractère #)

```
a=0
while(a<7):          #(n'oubliez pas le double point !)
    a=a+1            #(n'oubliez pas l'indentation !)
    print(a)
```

Réessayer avec une virgule derrière « print(a) ».

```
a=0
while(a<7):          #(n'oubliez pas le double point !)
    a=a+1            #(n'oubliez pas l'indentation !)
    print(a),
```

L'instruction For " Pour "

Contrairement à la boucle " While " le nombre de répétition est connu a l'entrée dans la boucle. Celui-ci ne doit pas dépendre d'une variable modifiée dans la boucle.

```
for a in range (1,11) :
    print (a),
```

```
debut=3
fin=15
for a in range (debut,fin) :
    print (a),
```

Les chaînes de caractères

Démonstration syntaxe

```
chaine1='Bonjour'
print (chaine1)

chaine1= "Bonjour "
print (chaine1)

chaine1='Aujourd'hui bonjour'
print (chaine1)


chaine1= "Aujourd'hui bonjour "
print (chaine1)

chaine1='Aujourd\'hui bonjour'          #Le caractère \ est un caractère d'échappement
print (chaine1)

chaine1='Aujourd\'hui\nbonjour'         #Le couple \ et n fait un retour à la ligne
print (chaine1)
```

La concaténation

```
print (" Bonjour  "+ "Monsieur ")

chaine1= "Bonjour "
chaine2= "Monsieur "
chaine3=chaine1+chaine2
print (chaine3)
```

Accès indexé

```
chaine3= "Bonjour Monsieur "
print (chaine3[3])
print (chaine3[0:2]
print (chaine3[7:9]
```

On considère la chaîne comme un tableau dont le premier indice est 0.

Longueur d'une chaine

```
chaine3= "Bonjour Monsieur "
print (len(chaine3))
```

Notion de fonction

Nous avons déjà vu quelques fonctions : print(), len(), range()... ce sont des fonctions prédéfinies.
Nous avons aussi la possibilité de créer nos propres fonctions !

Intérêt des fonctions

Selon l'angle il y a plusieurs intérêt à créer des fonctions :

Permet de réduire le volume de code

Lorsqu'une partie de code se retrouve à plusieurs endroits, nous pouvons le transformer en une fonction qui sera écrite une seule fois.

Permet de simplifier la lecture du code

En effet un simple appel à une fonction permet d'appréhender plus clairement le programme.
A condition de bien documenter le code (commentaire et nom de la fonction explicite)

Permet de faciliter la mise au point et la maintenance du code

Une fonction peut être retravaillée, améliorée sans pour autant perturber (en principe) le bon fonctionnement du programme.

Permet de coder plus rapidement

Une fonction bien conçue et bien documentée peut être réutilisée dans d'autres programmes.

Principe

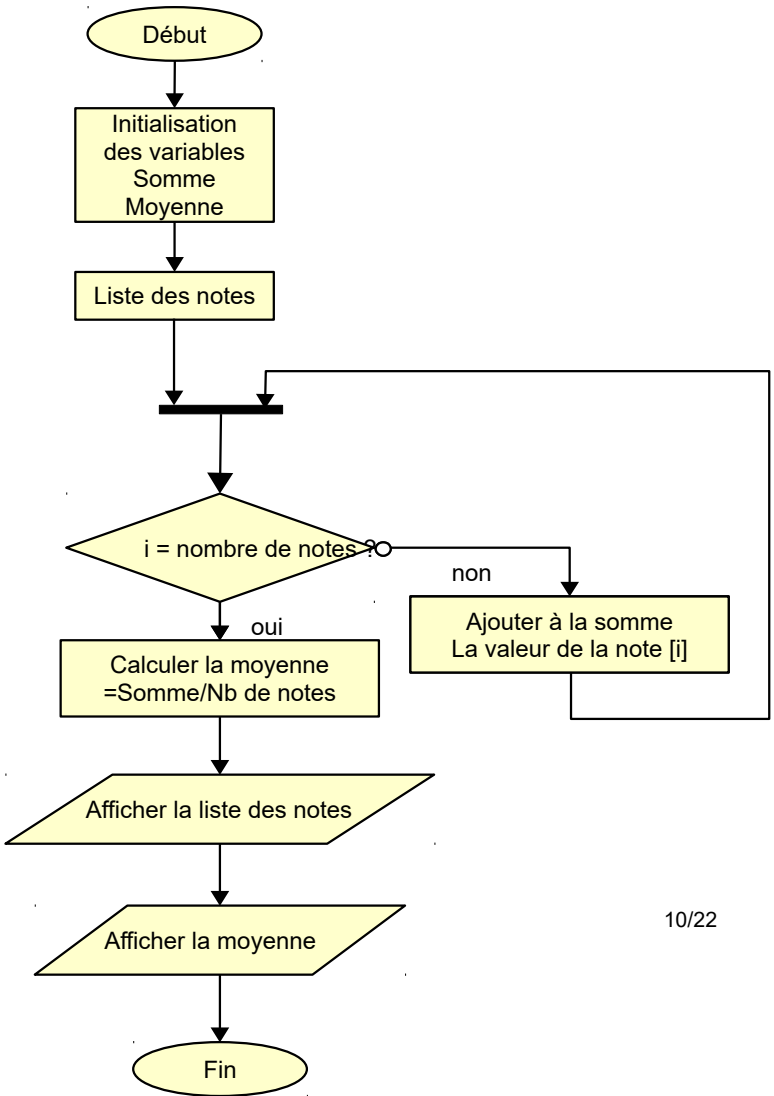
La fonction

Exemple calcul de moyenne:
Programme sans fonction

```
Somme=0
Moyenne=0
ListeNotes=[12,15.2,13,8,9,12,9.3,10]

for i in range (0,len(ListeNotes)) :
    Somme=Somme+ListeNotes[i]

Moyenne=Somme/len(ListeNotes)
```



```
print (" Notes " ),(ListeNotes)
print (" Moyenne " ),(Moyenne)
```

Les variables Somme et Moyenne sont du type " float " (virgule flottante).
La variable ListeNotes est du type " list " (liste).
La variable de compteur de boucle i est du type " int " (entier).
La boucle for se charge d'incrémenter (augmenter de 1) automatiquement la variable i.

- Programme avec définition d'une fonction

```
def Moy(ListeNotes):                    #Définition de la fonction Nom et Paramètres
    Somme=0
    Moyenne=0
    for i in range(0,len(ListeNotes)):
        Somme=Somme+ListeNotes[i]
    Moyenne=Somme/len(ListeNotes)

    print ("Notes" ),(ListeNotes)
    print ("Moyenne "),(Moyenne)
    return                              #Fin de la définition de la fonction

ListeNotes_1=[12,15.2,13,8,9,12,9.3,10]
ListeNotes_2=[8,12.5,14.3,13.5,18,9.5,11,9.5,10,19]
Moy(ListeNotes_1)                       #Appel de la fonction avec paramètre
Moy(ListeNotes_2)                       #Appel de la fonction avec paramètre
Moy(ListeNotes_1+ListeNotes_2)          #Appel de la fonction avec paramètre
```

- Nous pouvons remarquer que la fonction " sum(ListeNotes) " aurait pu être utilisée. Il est intéressant de bien regarder la liste des fonctions existantes (http://docs.python.org/2/library/functions.html) afin de ne pas recréer des éléments existants.

```
def Moy(ListeNotes):    #Définition de la fonction Nom et Paramètres
    Moyenne=0
    Moyenne=sum(ListeNotes)/len(ListeNotes)

    print ("Notes" ),(ListeNotes)
    print ("Moyenne "),(Moyenne)
    return          #Fin de la définition de la fonction
```

- La fonction peut aussi retourner une valeur (return).

```
def Moy(ListeNotes):    #Définition de la fonction Nom et Paramètres
    Moyenne=0
    Moyenne=sum(ListeNotes)/len(ListeNotes)
    return (Moyenne)        #Fin de la définition de la fonction avec envoi d'un paramètre

ListeNotes_1=[12,15.2,13,8,9,12,9.3,10]
ListeNotes_2=[8,12.5,14.3,13.5,18,9.5,11,9.5,10,19]
print (ListeNotes_1)
print(Moy(ListeNotes_1))            #Appel de la fonction avec paramètre
print (ListeNotes_2)
print(Moy(ListeNotes_2))            #Appel de la fonction avec paramètre
```

Programmation orientée objet (POO)

Concept

Prenons comme exemple le dé de jeu.

On peut décrire le dé :

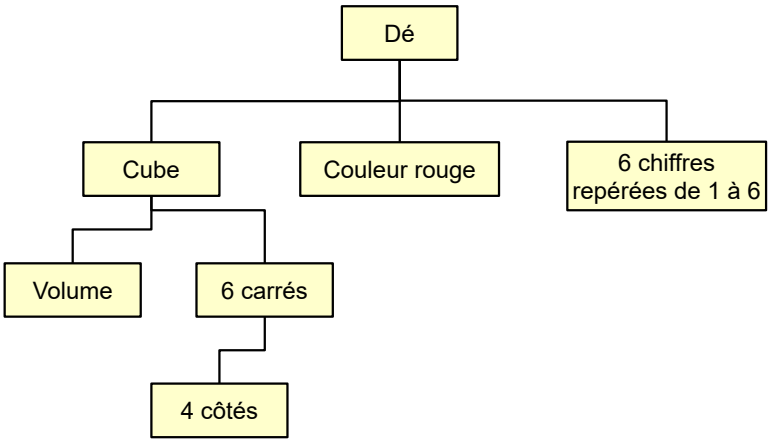
- est un **Cube**
- est de couleur rouge
- peut être lancé pour renvoyer un nombre entre 1 et 6

On peut décrire le cube :

- est un volume géométrique à 3 dimensions
- est constitué de 6 **carrés**

On peut décrire le carré :

- est une figure géométrique à 4 côtés.



Et nous pourrions continuer précisant le terme dimension mais dans notre cas ce n'est pas utile. Nous pouvons ainsi établir le schémas suivant : le dé hérite des caractéristiques du cube (c'est un cube). Mais on ne peut pas dire que le cube hérite des caractéristiques du carré. En effet, on indique bien qu'il est constitué de mais pas qu'il est et c'est la toute la différence.

Nous dirons donc que :

- l'objet cube implémente l'objet surface carré
- l'objet dé hérite de l'objet cube

Objet, Méthode, Classe, Attribut

Objet

Une chaîne de caractère peut être considérée comme un objet.

Méthode

Nous pouvons établir une liste d'opérations standards, appelés méthodes, applicables à un objet donné.

Lancer le Dé pour obtenir un chiffre de 1 à 6 est assimilable à une méthode.

Méthodes applicables à une chaîne de caractères " Chaine " :

- mettre la chaîne en minuscule - Chaine.lower()
- mettre la chaîne en majuscule – Chaine.upper()
- mettre la première lettre de la chaîne en majuscule – Chaine.capitalize()
- etc...

Classe

Il semble évident que ces méthodes ne sont applicables qu'aux chaînes de caractères.

Les objets concernés par ces méthodes appartiennent tous à la classe " str " (string).

On peut s'en rendre compte en affichant le type d'une chaîne {print (type(Chaine))}.

Lorsque l'on crée un objet dans une classe nous pouvons lui appliquer toutes les méthodes de cette classe.

Une méthode est aussi considérée comme un objet dans sa classe.

Créons une classe nouvelle

```
class De:            #définition de notre classe dé
    """ Classe définissant un dé caractérisé par :
    -son nombre de faces
    -sa couleur """
    def __init__(self):     #notre méthode constructeur
        """constructeur"""
        self.couleur="Rouge "
        self.face=6

CubeAPoint=De()                 #On associe l'objet CubeAPoint à la classe De()
print CubeAPoint.couleur
CubeAPoint.couleur="bleu"       #change la valeur de l'attribut couleur pour CubeAPoint
print CubeAPoint.couleur
```

- Définition de la classe. Mot clé " class ", nom de la classe terminé par " : "
- Un commentaire expliquant la classe entre " " "
- Définition du constructeur de la classe. Mot clé " def ", nom du constructeur " __init__ " invariable en python, un paramètre nommé " self ".
- Un commentaire
- Instanciation de l'attribut (couleur dans notre cas). On crée une variable " self.couleur " et on lui donne la valeur " rouge ".

Remarque

L'attribut " .couleur " sera toujours rouge par défaut pour tous les objets de la classe " De ".

Créons un constructeur avec des paramètres :

```
    def __init__(self, couleur, face):      #notre méthode constructeur
        """constructeur"""
        self.couleur=couleur
        self.face=face

CubeAPoint=De("vert",6)       #CubeAPoint associé à la classe De avec les attributs vert et 6
print CubeAPoint.couleur
CubeAPoint.couleur="bleu"     #change la valeur de l'attribut couleur pour CubeAPoint
print CubeAPoint.couleur
print CubeAPoint.face
```

La définition du constructeur intègre deux paramètres " couleur " et " face ".

Créons une méthode dans une classe

Notre classe comprend deux attributs, ajoutons lui une méthode :

Nous allons ajouter une méthode pour tirer avec notre dé un nombre aléatoire compris entre 1 et le nombre de l'attribut " .face ".

Pour obtenir un nombre aléatoire, nous allons utiliser un module externe (classe : random) en ajoutant au début de notre programme " import random ".

Une méthode nous permet d'exécuter dans la classe une action comme vu précédemment (mettre une chaîne en minuscule). Nous allons donc utiliser une fonction, on l'appellera " tirage_de " pour définir l'attribut " .point ".

Notre programme devient :

```
import random                          # import de la classe " random "
class De:                              # définition de notre classe dé
    """ Classe définissant un dé caractérisé par :
son nombre de faces
sa couleur """
    def __init__(self, couleur, face):     # notre méthode constructeur
        """constructeur"""
        self.couleur=couleur
        self.face=face
        self.point=""
    def lance_de(self):                    # méthode utilisateur ne pas mettre le double " _ "
        self.point=random.randint(1, self.face) # nombre aléatoire compris entre 1 et le nb. de
faces

CubeAPoint=De("vert",6)        # l'objet CubeAPoint de la classe De est vert et a 6 faces
for i in range(0,10):
    CubeAPoint.lance_de()      # lancé du dé (méthode)
    print (CubeAPoint.point),  # affichage du nombre de points (attribut point)
```

Attribut

Dans notre exemple la couleur du Dé est un attribut " .couleur " le chiffre obtenu avec la méthode " lance_de " est un autre attribut " .point ".

Résumé

Toute variable et fonction peuvent être considérés comme un objet. Un objet peut être une agrégation d'objets (l'objet roue est composé de l'objet jante et l'objet pneu).

Pour un même " type " d'objet (on parlera de " classe "), je peux avoir plusieurs objets de même fonction mais d'apparence différente (la classe voiture comprend les voitures de couleur rouge et les voitures de couleur noire).

Un attribut définit l' " apparence " d'un objet, c'est tout ce qui va permettre de le distinguer d'un autre objet de la même classe.

Une méthode décrit le " fonctionnement " d'un objet, par un ensemble de fonctions qui peuvent lui être appliquées (la méthode démarrer est une méthode de la classe voiture).

La voiture est la classe fille de la classe véhicule. La classe véhicule est la classe mère de la classe voiture (la voiture hérite des attributs de la classe véhicule)

Accès aux fichiers textes

La lecture ou l'écriture dans un fichier peut être vue comme la lecture dans un livre ou l'écriture dans un cahier.

Plusieurs étapes sont nécessaires :

1. Choisir le livre ou le cahier, l'ouvrir et déterminer si l'on va lire, écrire ou ajouter des informations.
2. Lire le livre ou écrire dans le cahier.
3. Refermer le livre ou le cahier.

Ces étapes se retrouvent dans la manipulation de fichiers en Python.

Ouverture du fichier

Pour ouvrir un fichier il va falloir être en possession de deux informations : son nom et la méthode d'accès que l'on souhaite employer. Pour les fichiers texte, il existe trois méthodes d'accès :

- r (pour " read ") : ouverture en lecture.
- w (pour " write ") : ouverture en écriture.

Si le fichier existe, il sera " écrasé " (détruit, puis remplacé par un nouveau fichier vide), sinon il sera créé.

- a (pour " append ") : ouverture en ajout.

Si le fichier existe et qu'il contient des données, les anciennes données seront conservées et les nouvelles données seront ajoutées à la fin. Si le fichier n'existe pas, il sera créé et les données seront ajoutées comme avec une ouverture en écriture classique.

```
fic = open('data.txt', 'r')
```

On associe un objet " fic " au fichier ouvert " data.txt " avec la méthode lecture " r ".

Lecture intégrale du fichier

Il est possible de lire tout le fichier en une seule commande et de traiter les données par la suite. Cette méthode s'applique préférentiellement aux petits fichiers, car la consommation mémoire sera proportionnelle à la taille du fichier lu.

On peut obtenir le contenu du fichier dans une chaîne de caractères à l'aide de la méthode " read() " :

```
texte = fic.read()
```

On récupère le texte brut dans la chaîne " texte ".

On peut obtenir le contenu du fichier dans une liste, chaque élément de la liste correspond à une ligne avec la méthode " readlines " :

```
texte = fic.readlines()
```

Lecture par tranches de caractères

Lorsque la fonction " read() " est utilisée en lui passant en paramètre un entier n, elle ne renvoie que les n caractères lus depuis le pointeur de fichier. On peut ainsi lire un fichier par tranches de n caractères, ce qui peut être très utile suivant les formats (par exemple, si chaque ligne représente des colonnes d'un nombre de caractères fixé).

```
texte = fic.read(10)
```

La chaîne texte contient les 10 caractères à partir du pointeur (le pointeur s'est déplacé de 10 caractères).

Lecture ligne à ligne

Enfin, la méthode " readline() " (sans s) permet de lire un fichier ligne à ligne : le pointeur de fichier s'arrêtera dès qu'il rencontrera un caractère \n (fin de ligne).

```
texte = fic.readline()')
```

Écriture dans un fichier

Pour écrire dans un fichier, la commande est très simple : il s'agit de " write() ". Cette méthode prend en paramètre une chaîne de caractères qui sera inscrite dans le fichier :

```
fic.write('Phrase à écrire dans le fichier référencé par l'objet-fichier fic')
```

Fermeture du fichier

Pour fermer un fichier, on utilisera la méthode " close() " :

```
fic.close()
```

Applications fenêtrées

Bibliothèques

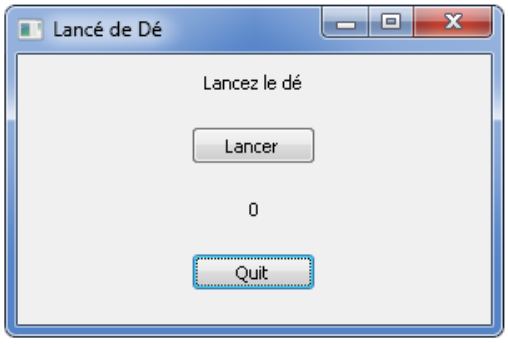
Tkinter

Bibliothèque basique livrée avec python.

WxPython

Bibliothèque à ajouter comme module.

Création d'une interface avec WxPython




```
#!/usr/bin/python
# -*- coding: cp1252 -*-

import wx
import random

class Frame(wx.Frame):
    def __init__(self,parent, title):

        wx.Frame.__init__(self, parent, -1, title, size=(400,200))
        self.panel = wx.Panel(self)
        text=wx.StaticText(self.panel, -1, "Lancez le dé")
        bouton = wx.Button(self.panel, -1, "Quit")
        lancer=wx.Button(self.panel,-1,"Lancer")
```

```
            self.Bind(wx.EVT_BUTTON, self.quitter, bouton)
            self.Bind(wx.EVT_BUTTON, self.tirage_de, lancer)
            self.result_lance=wx.StaticText(self.panel,-1,label="0")
            sizer = wx.BoxSizer(wx.VERTICAL)
            sizer.Add(text, 0, wx.ALL|wx.CENTER, 10)
            sizer.Add(lancer,0,wx.ALL|wx.CENTER,10)
            sizer.Add(self.result_lance,0,wx.ALL|wx.CENTER,10)
            sizer.Add(bouton, 0, wx.ALL|wx.CENTER, 10)
            self.panel.SetSizer(sizer)
            self.panel.Layout()
            self.Centre()

        def quitter(self, evt):
            self.Destroy()

        def tirage_de(self, evt):
            point=random.randint(1, 6)
            self.result_lance.SetLabel(str(point))

class Programme(wx.App):
    def OnInit(self):
        frame = Frame(None, "Lancé de Dé")
        self.SetTopWindow(frame)
        frame.Show(True)
        return True

app = Programme(True)
app.MainLoop()
```

Décomposons

On importe le module " WxPython "

```
import wx
```

Création d'une classe de gestion de l'affichage (fenêtre, texte, bouton, titre...)

```
class Frame(wx.Frame):
        def __init__(self,parent, title):
                wx.Frame.__init__(self, parent, -1, title, size=(400,200))
```

Création d'un objet " panel " de la classe " Frame " méthode " wx.Panel " du module " wx "

```
            self.panel = wx.Panel(self)
```

Création de deux objets " Button " et d'un objet " StaticText " (Boutons " bouton " et " lancer " et texte " text ")

```
            text=wx.StaticText(self.panel, -1, "Lancez le dé")
```

```
bouton = wx.Button(self.panel, -1, "Quit")
```


```
lancer=wx.Button(self.panel,-1,"Lancer")
```

Association de l'évènement clic dur les boutons avec les fonctions " quitter " et " tirage_de ".

```
self.Bind(wx.EVT_BUTTON, self.quitter, bouton)
self.Bind(wx.EVT_BUTTON, self.tirage_de, lancer)
```

Création du " StaticText " résultat du lancé de Dé, on lui défini la chaîne " 0 " par défaut.

```
self.result_lance=wx.StaticText(self.panel,-1,label="0")
```

Format mise en page verticale des objets (StaticText et Button)

```
sizer = wx.BoxSizer(wx.VERTICAL)
```

Organisation des objets de haut en bas

```
sizer.Add(text, 0, wx.ALL|wx.CENTER, 10)
sizer.Add(lancer,0,wx.ALL|wx.CENTER,10)
sizer.Add(self.result_lance,0,wx.ALL|wx.CENTER,10)
sizer.Add(bouton, 0, wx.ALL|wx.CENTER, 10)
```

Mise en place de ces objets

```
self.panel.SetSizer(sizer)
```

Construction de l'objet " panel "

```
self.panel.Layout()
```

Ouverture de la fenêtre au centre de l'écran

```
self.Centre()
```

Création de la fonction " quitter " pour détruire la fenêtre

```
def quitter(self, evt):
    self.Destroy()
```

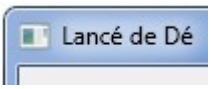
Création de la fonction " tirage_de " pour effectuer un lancé du dé

```
def tirage_de(self, evt):
    point=random.randint(1, 6)
    self.result_lance.SetLabel(str(point))
```

Création d'une classe " wx.App "

```
class Programme(wx.App):
  def OnInit(self):
```

```
frame = Frame(None, "Lancé de Dé")
```




```
self.SetTopWindow(frame)
frame.Show(True)
return True
```

Appel de la classe " Programme " en boucle

```
app = Programme(True)
app.MainLoop()
```

Annexe 1 - Utilisation de fichier programme

Pour enregistrer un programme nous allons le stocker dans un fichier.

Sous Windows

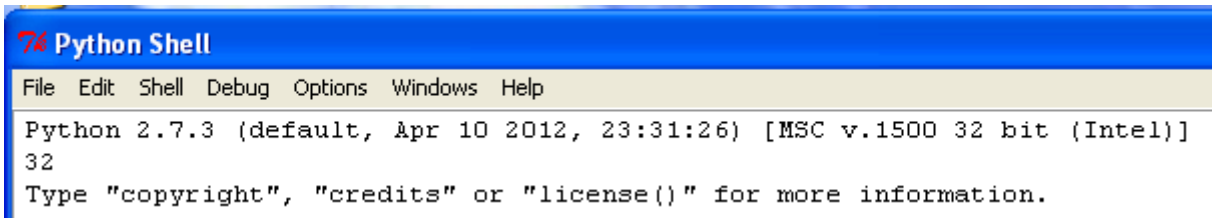
Sous Windows à l'installation de « Python » un « IDLE » (environnement de travail) est installé donc rien à faire.

Sous LINUX

Sous LINUX « Python est généralement fourni avec la distribution donc en tapant « python » on ouvre un environnement sommaire de programmation.

Pour utiliser l'environnement « IDLE » il faut l'installer. (apt-get install idle)

Éditeur IDLE



Sous Windows l'éditeur est accessible directement au lancement de l'interface.

Sous LINUX il faut lancer l'environnement « IDLE ».

Pour créer un fichier programme

- Ouvrir une nouvelle fenêtre

---> File ---> New Window

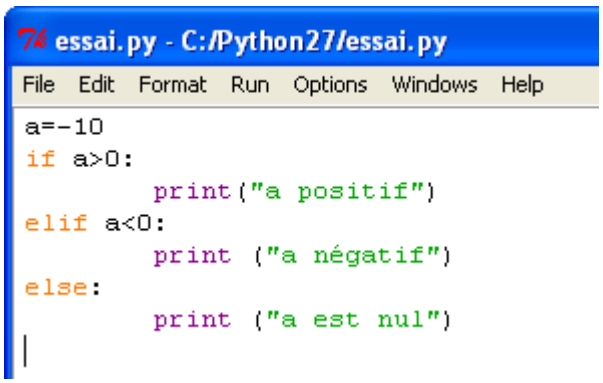
- Une nouvelle fenêtre s'ouvre.
- Entrer le code
- Sauvegarder le fichier avec extension .py

---> File ---> Save

- Lancer l'exécution

---> F5

- Observer le résultats dans la fenêtre « Python Shell »



Annexe 2 – Convertir un programme Python en exécutable

Py2exe

Pour exécuter un programme Python sur un poste sous Windows, il est nécessaire de le convertir en exécutable « .exe ». La bibliothèque py2exe permet de générer sous Windows des exécutables à partir d'un source Python. En plus, il y aura au minimum une dll Python et un zip contenant les modules compilés que vous devrez fournir avec votre programme.

Il vous permettra également de créer toute une distribution où vous pourrez empaqueter tous les fichiers nécessaires pour que votre programme fonctionne correctement

Sous LINUX ce n'est pas nécessaire, généralement les distributions intègrent le Python, donc ce qui suit est inutile.

L' extension Python Py2exe est utilisée sous Windows avec un environnement Python<3.

Installation

- Récupérer la bibliothèque " http://www.py2exe.org " ou
  " http://freefr.dl.sourceforge.net/project/py2exe/py2exe/0.6.9/ " choisir la version compatible avec Python.
- Installer py2exe

Utilisation

Création d'un fichier " setup "

- Sous l'éditeur de texte créer un fichier « setup.py » (enregistrer sans l'exécuter)

```
from distutils.core import setup
import py2exe
setup(console=["nom_du_programme.py"])
```

En ligne de commande " dos "

- Taper la commande suivante " c:/python27/python.exe setup.py py2exe "

2 répertoires sont créés: dist et build - ce dernier ne sert que lors de la construction de la distribution et peut donc être effacé

Dans le répertoire dist, vous récupérerez les fichiers nécessaire a une exécution autonome. Tous ces fichiers ne sont pas forcément utiles. Cela dépendra du contenu de votre fichier .py.

Cxfreeze

Un autre outil permet de réaliser cela de façon plus simple et utilisable avec toute version de Python.

Lien

http://cx-freeze.sourceforge.net

Annexe 3 – Pygame

Lien

http://www.pygame.org/

Installation

Sous Windows lancer l'install

Import du module

Dans l'IDLE taper : " import pygame "