Formation Python

Python

2014

Sommaire

Python - Introduction - Histoire

Créateur :

Guido van Rossum, développeur néerlandais de formation scientifique et mathématique. Il a participait au développement du langage ABC engagé par Google en 2005.

FIGURE : Guido van Rossum

La première version publique est sortie en 1991.

Python est devenu un langage pérenne depuis la création de la fondation Python en 2001 et la licence Python. Le but de Python est de fournir un langage simple pour des scientifiques et mathématiciens.

Python - Introduction - Utilisateur

Quelques références d'utilisateurs de Python dans le monde :

- Google - avec leur bot (première version), Google app engine.
- Walt Disney - intégration de script python dans leur chaine de production d'animation
- National Weather Service (NOAA) - Application graphique de surveillance météo
- RedHat - Anaconda - Installeur de la distribution
- AFNIC - Gestionnaire des domaines .fr - Web Service SOAP
- NodeJS - utilisation de Waf

Python - Introduction - Particularités

- Langage interprété
  pas besoin de compiler > plus simple pour scientifiques
  interpréteur présent sur toutes les plateformes (Windows, Linux, ...)
- Haut niveau
  pas de gestion de la mémoire système arithmétique et expression booléenne processus léger
- Typage dynamique typage de la variable en fonction de la donnée insérée type déterminé à l'exécution

Python - Introduction - Particularités

- Programmation Orientée Objet.
  invisible par le débutant, tout est objet en python. développeur expérimenté reste à l'aise (classes, héritage, polymorphisme, etc.)
- La syntaxe "simple"
  prévue pour les personnes non-développeurs comment affichez-vous 2*3 en Java/C# ? indentation forcée

Python - Introduction - Installation

- Installation windows
  Il y a plusieurs version de python, quoi faire ?
  Etape 1 : Télécharger le MSI de Python ici :
  Etape 2 : Installer
  Etape 3 : Tester la Python console (C :\ Python27\ python.exe)

Python - Introduction - Installation

- Installation GNU/Linux
  Il y a plusieurs version de python, quoi faire ?
  Déjà installé dans les distributions classique :
  RedHat, Debian, Ubuntu
  - Installer depuis les sources
    # wget
    http ://www.python.org/ftp/python/2.7.3/Python-2.7.3rc1.tar.bz2
    # tar xvjf Python-2.7.3rc1.tar.bz2
    # cd Python-2.7.3rc1
    # ./configure
    # make
    possibilité d'installer :
    # make install

Python - Introduction - Prise en main

- mode interactif
  - Pratique pour faire des essais
  - Essayez

```
1  print "bonjour"
2  print "bonjour" * 3
```

- mode script
  - Édition auparavant dans un éditeur, puis lancement via l'interpréteur python
  - Pour lancer vos projets

Python - Utiliser un l'IDE Eclipse

- Pourquoi utiliser un IDE ?
  - Auto complétion
  - Coloration syntaxique
  - Debugging (pas à pas, espions)
  - Templates de classes
- Exemple d'installation de PyDev
  - Étape 1 : Télécharger Eclipse Java (pas Java EE)
  - Étape 2 : Dézippper puis exécuter eclipse.exe
  - Étape 3 : Dans Help > Install new Software rajouter l'URL de PyDev : http ://pydev.org/updates
  - Étape 4 : Créer un "module" Python (équivalent Projet)

Python - Variables

Les variables existent dès leur première affectation et sont détruites lorsqu'elles ne sont plus accessibles. Nous n'avons pas besoin de définir le type, l'expression littérale est utilisée pour créer la variable. Exemple de création de variables :

```
1 var = 5          # print var
2 var = 6.8          # print var
3 var = "Hello World"     # print var
4 var = [ 1, "Hello", 5.6 ]  # print var
5 var = ( 'Lundi', 'Mardi', 'Mercredi' )      # print var
6 var = { 'jour': 1, 'nuit':0, 'matin':0.5, 'soir':0.5}  # print
      var
```

Nommage des variables : lettre a-zA-Z, ou souligné "_"
chiffre 0-9 (pas au début)
sensible à la casse

Python - Variables

Affichage de variables :

```
1 print var
2 print var, var2, var3
```

Affichage formaté :

```
1 print "%type" % var    # Syntaxe de l'écriture formatée
2 print "%s %s" % (var, var2)
3 var = "Python"
4 print " Je suis en formation %s aujourd'hui. " % var
```

Types :
d : décimal - o : octal - x : hexadécimal - c : caractère - s : chaîne de caractère - f : flottant

Python - Variables

- Comme nous avons un typage dynamique. Python possède une fonction pour connaître le type d'un objet. Cette fonction est ***type()***.
- Nous pouvons aussi connaître les variables accessibles localement via la fonction ***locals()*** .
- Et celle accessible globalement avec ***globals()***
- Une des fonction les plus utilisée en python est la fonction ***help()***. Exemple : help(str)

Python - Types et opérations

Création d'un type numérique :

```
1 var = int(5)       # print var
2 var = 5            # print var
3 var = float(5.6)     # print var
4 var = 5.6          # print var
5 var = long(5)        # print var
6 var = 5L           # print var
7 var = complex(10, 2) # print var
8 var = 10+2j        # print var
```

Python - Types et opérations

Écriture littéral des types numériques un entier peut s'écrire en :

- binaire :

```
1 var = 0b01010101    # print var
```

- en octal :

```
1 var = 0755 # print var
2 var = 0o755  # print var
```

- en hexadécimal :

```
1 var = 0x41 # print var
```

- un float avec l'exposant :

```
1 var = 3.14e-10 # print var
```

Python - Types et opérations

- Opérations sur types numériques :

x+y	addition
x-y	soustraction
x*y	multiplication
x/y	division
x // y	division entier
x%y	reste
-x	change le signe
+x	ne change pas x
x ** y	puissance
divmod(x, y)	couple x // y, x % y

- Essayez :

```
1 print 9/4
2 print 9//4
```

Python - Types et opérations

Priorités des opérateurs

- Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations sont effectuées dépend de règles de priorité
- Les parenthèses ont la priorité la plus forte, elles permettent donc de casser les priorités naturelles
- Lorsque 2 opérateurs ont la même priorité (par exemple + et -), ils sont évalués de gauche à droite.

Python - Types et opérations

Opérations binaires sur un entier :

x | y	ou
x & y	et
x ^ y	ou exclusif
x << n	décalage à gauche de n
x >> n	décalage à droite de n
~ x	inversion

Python - Types et opérations

- Exemple d'opérations binaires sur des entiers :

```
>>> a = 5
>>> b = 6
>>> res = a | b
>>> print res
    7
```

- Explication :
  5 en binaire 0b101
  6 en binaire 0b110
  donc 0b101 OU BINAIRE 0b110 donne 0b111 soit 7

Python - Types et opérations

La séquence est un conteneur d'information :

- Chaîne de caractère : str, unicode
- Tableau dynamique : liste
  - Liste de données
  - Manipulation aisée
  - La liste peut stocker des informations de types différents
- Tableau statique : tuple
  Liste en "lecture seule" (non mutable)

Python - Types et opérations

Création de séquences :

- Chaîne de caractères :

```
>>> var = str("une chaine de caractere")
>>> var = "une chaine de caractere"
```

- Chaîne de caractère Unicode :

```
>>> var = u"une chaine de caractère"
```

- Liste : (à définir entre crochets)

```
>>> var = list(("chaine", 'c', 5, 6.6))
>>> var = ["chaine", 'c', 5, 6.6]
```

- Tuple : (à définir entre parenthèses)

```
>>> var = tuple(("Lundi", "Mardi","Mercredi", "..."))
>>> var = ("Lundi", "Mardi", "Mercredi", "...")
```

Python - Chaîne de caractères

Les chaînes de caractères (string)

- Se sont des séquences de caractères
- Comme toutes les séquences, on peut effectuer différentes opérations

Concaténation de chaînes	'bon' + 'jour'
Répétition de chaînes	'bon'*2

- Par défaut en utf-8 (Unicode)
- Les valeurs littérales peuvent être entre guillemets simple 'exemple', double "exemple" ou triple """exemple"""

Python - Manipulation de Chaîne de caractères

- Les éléments des différentes séquences sont indexés à partir de zéro et peuvent être accédés en utilisant les crochets

```
1 nom = 'Dupont'
2 c1, c2, c3 = nom[1], nom[4], nom[-1]
3 # caractère 'u', 'n' et 't'
4 print(nom[0])
5 # affiche 'D'
```

- On utilise « \ » (antislash) pour insérer les caractères spéciaux ou pour écrire une instruction sur deux lignes
- En Python, les chaînes de caractères sont non mutables

Python - Types et opérations

- Opérateurs d'affectation :

Simple	x = 12
Multiple	x=y=z=4
Parallèle	x, y = 4, 12
Augmentée (incrémentation)	+=
Diminuée (décrémentation)	-=
...	...

- Assignation de valeurs consécutives

```
1 (Lundi, Mardi, Mercredi, Jeudi, \
2 Vendredi, Samedi, Dimanche) = [0,1,2,3,4,5,6]
```

Python - Chaîne de caractères

L'affichage se fait par la fonction « print »

- Lorsqu'il ne s'agit pas d'une chaîne de caractères, la conversion se fait avec « str »

```
1 print "Hello world"
2 import math
3 print math.pi
```

- La saisie avec la fonction « input() »(renvoie une valeur selon le type de données saisie)

```
1 saisie = input("tapez quelque chose s'il vous plait ")
```

- Pour une saisie avec un renvoie d'une chaîne de caractère (indépendamment de la donnée en entrée) vaut mieux utiliser « raw_input() »

Python - Chaîne de caractères

- Insertion des contenus de variables dans une chaîne de caractère avec la fonction « format »
  - Remarque : L'utilisation de « format » de cette façon n'est valable que depuis la version 2.7 de Python.

```
1 temp = 15
2 jour = "lundi matin"
3 print "la température de {} sera de {} degrés.".format(jour
      ,temp)
```

```
1 nbr = 158.156658
2 print "nbr = {:08.3f}, nbr = {:06.3e}".format(nbr, nbr)
3 # nbr = 0158.157, nbr = 1.582e+02
```

```
1 n = 15
2 print "n = {:b}, n = {:o}, n = {:d}, n = {:x} ".format(n, n
      , n, n)
3 # n = 1111, n = 17, n = 15, n = f
```

Exercices : Premiers pas en Python

1. Affectez les variables temps et distance par les valeurs 6.892 et 19.7.
   - Calculez et affichez la valeur de la vitesse (m/s).
   - Améliorez l'affichage en imposant deux chiffres après le point décimal.
2. Saisir un nom et un âge en utilisant l'instruction input() puis affichez-les.
   - Refaire la saisie du nom, mais avec l'instruction raw_input() puis affichez le résultat.
   - Enfin, utilisez la " bonne pratique " : recommencez l'exercice en transtypant les saisies effectuées avec l'instruction raw_input()

Python - Types et opérations

Opérations sur les séquences :

x in s	True si x est dans s
x not in s	False si x est dans s
s+t	concaténation de s avec t
s * n, n * s	effectue n copie de s
s [i]	item à la position i dans s, commence à 0
s [i :j]	tranche d'items de i à j
s [i :j :k]	tranche d'items de i à j par pas de k
len(s)	taille de la séquence
min(s)	retourne le plus petit item
max(s)	retourne le plus grand item
s.index(i)	retourne l'index du premier item
s.count(x)	compte le nombre d'items x dans s

Python - Types et opérations : String

Opérations spécifiques des chaînes :

str.capitalize()	str.isspace()
str.lower() str.upper() str.swapcase()	str.strip(char) str.lstrip(char) str.rstrip(char)
str.isupper() str.islower()	str.center(taille, optchar) str.ljust(taille, optchar) str.rjust(taille, optchar)
str.expandtabs(size)	str.replace(old, new, optcount)
str.isalnum() str.isalpha() str.isdigit()	str.partition(sep)
str.join(séquence)	str.split(sep) str.splitlines()

Python - Types et opérations : Liste

- Les listes sont des séquences (ordonnées) au même titre que les chaînes de caractères
- Contrairement aux chaînes, elles peuvent être modifiées (mutable)
- Représentation : séquence entre crochets [ ]

Python - Types et opérations : Liste

Quelques fonctions utiles pour les listes :

list.append(objet)	ajoute un objet à la fin de la liste
list.count(valeur)	compte le nombre d'occurrences d'une valeur
list.extend(séquence)	ajoute une séquence à la liste
list.index(valeur, start, stop)	retourne l'index de la première occurrence (start et stop sont optionnel)
list.insert(index, objet)	ajoute un objet avant l'index
list.pop(index)	supprime et retourne l'objet à l'index (index optionnel)
list.remove(value)	supprime la première occurrence
list.reverse()	reverse la liste
list.sort()	trie la liste

Python - Types et opérations : Tuple

Un **tuple** est assez proche d'une liste mais n'est pas mutable.

- Représentation : séquence entre parenthèses ( )
- Exemple (attention au tuple singleton) :

```
1 empty = ()  # tuple vide
2 single = ('vert', ) # tuple singleton
3 point = (3,54)  # tuple avec 2 éléments
```

Python - Types et opérations : Set

Un **set** est une collection non ordonnée et sans éléments redondants (ensemble d'éléments)

- Les éléments servent eux même de clef
- On utilise la fonction set() qui prend une séquence en argument
- Opérateurs ensemblistes

Python - Types et opérations : Set

Quelques fonctions utiles pour les set :

len(s1)	Renvoie la taille de l'ensemble s1
s1.issubset(s2)	Indique si s2 est un sous-ensemble de l'ensemble s1
s1.issuperset(s2)	L'inverse de issubset()
s1.add(ele)	Rajoute un élément à l'ensemble
s1.remove(ele)	Supprime un élément de l'ensemble
s1.pop()	Supprime et renvoie un élément aléatoire de l'ensemble
s1.clear()	Supprime tous les éléments de l'ensemble
s1.union()	Crée un nouveau set issue de l'union de ceux en arguments
s1.intersection()	Fait l'intersection entre les sets

Exercices : Les listes

1. Définir la liste : maliste =[17, 38, 10, 25, 72], puis effectuez les actions suivantes :

   - triez et affichez la liste ;
   - ajoutez l'élément 12 à la liste et affichez la liste ;
   - renversez et affichez la liste ;
   - affichez l'indice de l'élément 17 ;
   - enlevez l'élément 38 et affichez la liste ;
   - affichez la sous-liste du 2éme au 3éme élément ;
   - affichez la sous-liste du début au 2éme élément ;
   - affichez la sous-liste du 3éme élément à la fin de la liste ;
   - affichez le dernier élément en utilisant un indexage négatif ;
   - créez une copie de votre liste dans une nouvelle variable.
   - ajoutez à votre nouvelle liste l'élément 42, que remarquez-vous ?

Python - Types et opérations : Dictionnaire

- Le dictionnaire est une liste d'une paire d'information (ou table de hachage) : C'est une collection (non ordonnée) de paires (clef/valeur).
- Tout comme dans le dictionnaire papier où il y a une liaison entre le mot et sa définition.
- La clef d'un dictionnaire est unique, la valeur non.
- Représentation : séquence entre accolades { }
- On peut obtenir les valeurs par clef par contre, il n'y a pas de notion de position dans un dictionnaire

Python - Types et opérations : Dictionnaire

Il existe plusieurs méthodes de création : Création d'un dictionnaire vide :

```
>>> d = dict()
>>> d = {}
```

Création d'un dictionnaire avec de pairs

```
>>> d = dict( one=1, two=2 )
```

Création d'un dictionnaire avec une liste de liste de clef, valeur

```
>>> d = dict( [ ['one', 1] , [ 'two', 2 ] ] )
```

Création direct d'un dictionnaire

```
>>> d = { 'one':1 , 'two':2 }
```

Python - Types et opérations : Dictionnaire

Les dictionnaires sont des objets pouvant évoluer dans le temps, nous avons diverses opérations pour cela :

dict.clear()	vide le dictionnaire
dict.copy()	créer une copie du dictionnaire
dict.get(clef [, defaut])	retourne la valeur de la clef ou défaut.
dict.has_key (clef)	retourne True si la clef est dans le dictionnaire, sinon False
dict.items()	retourne une liste de tuple (clef, valeur)
dict.keys()	retourne la liste des clefs
dict.values()	retourne la liste des valeurs
dict.pop(clef)	supprime la paire et retourne la valeur
d[clef] = value	ajoute ou modifie une paire

Python - Structure de contrôle

Un programme est une suite de N instructions s'exécutant les unes après les autres. Ce flux d'instructions est contrôlable afin de pouvoir appeler telle ou telle instruction en fonction des besoins de l'utilisateur.

C'est de la programmation structurée.

Nous avons besoin de deux notions pour structurer un programme. Le bloc d'instruction et l'entête de bloc d'instruction.
Deux familles d'entête de bloc d'instruction :

- les alternatives, exécution en fonction de condition réussie. (if, else, ...)
- les boucles, exécution a plusieurs reprises d'un bloc. (while, for)

Un bloc d'instruction est précédé d'un entête de bloc qui fini toujours par " **:** " (deux points).
Un bloc d'instruction est formé par l'indentation du code.

Python - Structure conditionnelle

Structure alternative (conditionnelle)

- Les mots-clés sont : if / elif / else
- Exemple :

```
1 a = input("Donnez-moi une valeur pour 'a', s.v.p ")
2 if a < 0:
3   print "négatif"
4 elif a == 0:
5   print "nul"
6 else:
7   print "positif"
```

Python - Structure conditionnelle

Pour avoir une condition de continuation, nous utilisons généralement des opérateurs de contrôle.

Deux familles :

- les opérateurs logiques
- les opérateurs de comparaison

Python - Structure conditionnelle

Il y a 3 opérateurs logiques : or, and et not.

Les opérateurs logiques or et and sont dit paresseux. Car ils n'évaluent pas la seconde expression dans le cas ou la première valide la condition.

- OU (X or Y)
  - Si X est vrai, alors l'expression vaut X et Y n'est pas évalué. Sinon, l'expression vaut Y
- ET (X and Y)
  - Si X est faux, alors l'expression vaut X et Y n'est pas évalué. Sinon, l'expression vaut Y
- Non (not X)
  - L'expression est évaluée à la valeur booléenne (True ou False) opposée de X

Python - Structure conditionnelle

Exemple 1 :

VRAI or quelque chose ⇒ sera toujours vrai (True),
FAUX and quelque chose ⇒ sera toujours faux (False)

Exemple 2 :

Table de vérité pour le « or » :

a	b	a or b
0	0	0
0	1	1
1	0	1
1	1	1

Exemple 3 :

Utilisation de l'expression ternaire avec le and et le or :

```
1 var = 3
2 res = var > 3 and "Ok" or "Nok"
```

Python - Structure conditionnelle

En Python, toute variable peut renvoyer un booléen utilisable dans une condition

- Types simples :
  - int : si ma variable vaut 0, à l'analyse de if (variable) , variable vaut false. Si variable vaut autre chose que 0, alors variable vaut true.
  - char : si ma variable est une chaine vide, la condition renvoi False. Si ma variable contient un élément, elle vaut True.
- Types complexes :
  une méthode spéciale sera appelée (que nous devrons surcharger) pour choisir si nous voulons renvoyer true ou false.

L'opérateur logique retourne l'expression qui valide la condition.

Python - Structure conditionnelle

Les types prédéfinis : le booléen (bool)

- Valeurs possibles : vrai (True) ou faux (False)
- Tous les types peuvent être convertis en booléen

bool	Faux = False
int	Faux = 0
float	Faux = 0.0
string	Faux = ""
tuple	Faux = ( )
list	Faux = [ ]
dict	Faux = { }

Python - Structure conditionnelle

Les opérateurs de comparaisons retourne Vrai ou Faux suite à la comparaisons de deux objets.

A<B	A est inférieur à B
A>B	A est supérieur à B
A <= B	A est inférieur ou égale à B
A >= B	A est supérieur ou égale à B
A == B	A est strictement égale à B
A != B ou A <> B	A est strictement différent de B
A is (not) B	A (n') est (pas) le même objet que B

Les opérateurs peuvent être enchaînes.
Exemple : A < B < C, A < B > C

Exercices : Contrôle du flux d'instructions

1. Saisir d'un flottant. S'il est positif ou nul, affichez sa racine, sinon affichez un message d'erreur.

2. Saisir deux nombres, comparez-les pour trouver le plus petit et affichez le résultat.

3. Refaire l'exercice en utilisant l'instruction ternaire suivante :
   res = a if <condition> else b

Exercices : Contrôle du flux d'instructions

4. On désire sécuriser une enceinte pressurisée. On se fixe une pression seuil et un volume seuil : pSeuil = 3.5, vSeuil = 7.41. On demande de saisir la pression et le volume courant de l'enceinte et d'écrire un script qui simule le comportement suivant :
   - si le volume et la pression sont supérieurs aux seuils : arrêt immédiat.
   - si seul la pression est supérieure à la pression seuil : demander d'augmenter le volume
   - si seul le volume est supérieur au volume seuil : demander de diminuer le volume de l'enceinte ;
   - sinon déclarer que "tout va bien".

Python - Structure itérative

Structure de boucles

Structure permettant la répétition de bloc.

Deux structures possibles :

- **while** - exécute le bloc d'instructions tant que la condition de continuation est vérifiée
- **for** - exécute le bloc d'instructions pour les éléments d'une séquence (liste, chaîné de caractères, ...) ou un objet itérable

Python - Structure itérative

Les boucles "**while**".

- Syntaxe :

```
1 while 'condition de continuation':
2   bloc d'instruction
```

- Exemple :

```
1 i=0
2 while i < 5:
3   i=i+1
4   print i
```

Python - Structure itérative

Les boucles "**for**".

- Syntaxe :

```
1 for elem in séquence:
2   bloc d'instruction
```

- Exemple :

```
1 for i in range(10):
2   print "Bonjour " + str(i)
```

Python - Structure itérative

Il est possible d'interrompre l'exécution d'un bloc d'instruction d'une boucle.

- **break** - permet d'interrompre le déroulement de la boucle et de la quitter
- **continue** - permet d'interrompre le déroulement de la boucle et de passer à l'itération suivante

Python - Structure de contrôle

La fonction **range()**.

- Elle génère une liste contenant toutes les valeurs de l'intervalle définie.
- Plusieurs méthodes d'appel :

```
>>> # Toutes les valeurs jusqu'à 10 (exclu)
... print range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> # Toutes les valeurs entre 5 (inclu) et 10 (exclu)
... print range(5, 10)
[5, 6, 7, 8, 9]
>>> # Toutes les valeurs de 0 (inclu) à 10 (exclu)
... # avec un pas de 3
... print range(0, 10, 3)
[0, 3, 6, 9]
```

Exercices : Les boucles

1. Initialisez deux entiers : a = 0 et b = 10.
   - Écrire une boucle affichant et incrémentant la valeur de a tant qu'elle reste inférieure à celle de b.
   - Écrire une autre boucle décrémentant la valeur de b et affichant sa valeur si elle est impaire. Boucler tant que b n'est pas nul.
2. Écrire une saisie filtrée d'un entier dans l'intervalle 1 à 10, bornes comprises. Affichez la saisie.
3. Affichez chaque caractère d'une chaîne en utilisant une boucle for.
4. Affichez chaque élément d'une liste en utilisant une boucle for.

Exercices : Les boucles

5. Affichez les entiers de 0 à 15 non compris, de trois en trois, en utilisant une boucle for et l'instruction **range()**.

6. Utilisez l'instruction **break** pour interrompre une boucle for d'affichage des entiers de 1 à 10 compris, lorsque la variable de boucle vaut 5.

7. Utilisez l'instruction **continue** pour modifier une boucle for d'affichage de tous entiers de 1 à 10 compris, sauf lorsque la variable de boucle vaut 5.

Exercices : Les boucles

8. Écrivez un programme qui compte de 1 à 15, affiche chaque nombre puis compte à rebours par pas de deux (2) jusqu'à 1, en affichant à nouveau chaque nombre.

9. Écrivez un programme qui lit interactivement des lignes saisies par l'utilisateur puis les affiche de manière inversée. Le programme devrait s'arrêter si l'utilisateur saisit le mot "quitter".

10. Chaque terme des séries de Fibonacci sont formés par l'addition des deux termes précédents. Écrivez un programme affichant les premiers 20 termes de ces séries.

Python - Résumé

```
1  rangelist = range(10)
2  print rangelist
3  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
4  for number in rangelist:
5    # test si le nombre est dans le tuple
6    if number in (3, 4, 7, 9):
7      # "Break" termine la boucle sans aller dans le else
8      break
9    else:
10     # nous allons à la prochaine iteration
11     continue
12
13 if rangelist[1] == 2:
14   print "The second item (lists are 0-based) is 2"
15 elif rangelist[1] == 3:
16   print "The second item (lists are 0-based) is 3"
17 else:
18   print "Dunno"
19
20 while rangelist[1] == 1:
21   pass
```

Manipulations de listes (1/3)

- On peut construire une liste en transformant une liste existante

```
1 liste = [4, 2, 7, 1]
2 transformation = [ elem **2 for elem in liste ]
3 # construit la liste [16, 4, 49, 1]
```

- Les éléments de la liste sont pris un par un et placés dans une variable temporaire,
- Ensuite on l'utilise pour effectuer un traitement.

Manipulations de listes (2/3)

- Liste avec une condition

```
liste = [4, 2, 7, 1]
transformation = [ elem**2 for elem in liste if elem > 3 ]
# construit la liste [16, 49]
```

- A partir d'un dictionnaire

```
dictionnaire = { 'un' : 1, 'deux' : 2, 'trois' : 3 }
[(x, y**3) for x, y in dictionnaire.items()]
# construit la liste [('un', 1), ('trois', 27),('deux', 8)]
```

Manipulations de listes (3/3)

- « Slicing » sur les listes

```
1 # Insertion
2 couleur = [ "rouge", "bleu" ]
3 couleur[1:1] = [ "vert" ]
4 # Suppression/Remplacement
5 couleur[1:2] = ["violet", "rose", "jaune"]
```

- Les indices négatifs sont utilisés pour lire la liste depuis la fin
- Le troisième indice correspond au « pas » :
  - [début : fin : pas]

Python - Fonctions

- Décomposent une tâche complexe en plusieurs tâches plus simples
- Évitent la duplication de code (factorisation du code)
- Permettent la réutilisation du code

Python - Fonctions

Syntaxe :

- Mot clef « **def** » est suivi d'un nom et d'une liste de paramètres entre parenthèses,
- Ensemble d'instructions, dans un bloc indenté,
- A la fin du bloc indenté, le mot clef « **return** » pour retourner une variable.

```
1 def nomDeLaFonction(arg1, arg2, ...):
2   # instructions
3   return résultat
```

Python - Fonctions

- Comme le typage est dynamique, les paramètres comme la valeur de retour ne sont pas typés.
- Pas de surcharge des fonctions.
- Appels récursifs
- Une fonction Python peut retourner plusieurs éléments :

```
1 def perimetreSurfaceCercle(r):
2   per = 2*math.pi*r
3   surf = math.pi*r*r
4   return per, surf
5
6 # Appel
7 p ,s = perimetreSurfaceCercle(3)
8 # p = 18.85, s = 28.27
```

Python - Fonctions

- Arguments sont optionnels et nommés

Exemple :

```
1 def fonction(a, b = 10, c = 1):
2   return a*b + c
3
4 # Appel
5 fonction(7) # 71
6 fonction(7, 2)  # 15
7 fonction(7, c = 0)  # 70
8 fonction(b = 5, a = 7)  # 36
```

Python - Fonctions

- Nombre d'arguments arbitraire (passage d'un tuple)

Exemple :

```
1 def somme(*args):
2   result = 0
3   for nbr in args:
4     result += nbr
5   return result
```

Python - Fonctions

- Passage d'un dictionnaire en argument (keyword)

Exemple :

```
1  def fonDic(**kargs):
2    for i in kargs.keys():
3      kargs[i]*=2
4    return kargs
5
6  # Appel
7  print fonDic(a = 15, b = 12)
8  # {'a' : 30, 'b' : 24}
9  dic = {'d' : 34, 'g' : 63, 'c' : 531}
10 print fonDic(**dic)
11 # {'d' : 68, 'g' : 126, 'c' : 1062}
```

Python - Fonctions

- Une fonction peut être passée en paramètre à une autre fonction

Exemple :

```
1  def tab(fonction, inf, sup, pas):
2    for i in range(inf, sup, pas):
3      y = fonction(i)
4      print("f({}) = {}".format(i, y))
5
6  def maFon(x):
7    return 2*x + 3
8
9  #Appel
10 tab(maFon, -4, 4, 2)
11 # f(-4) = -5
12 # ...
13 # f(2) = 7
```

Python - Fonctions

- Une variable affectée dans une fonction a une portée locale, elle est donc détruite à la fin de la fonction, contrairement à une variable globale
- A l'intérieur des fonctions, on peut accéder aux variables globales, mais seulement en lecture
- Pour modifier une variable globale depuis une fonction, il faut utiliser le mot clef « **global** »

Python - Fonctions

- Lambda expression :
  - Fonction anonyme généralement courte
  - Permet de générer une famille de fonctions
  - Peut s'appliquer sur des séquences avec la fonction **map**.

```
1  # déclaration d'une expression
2  carre = lambda x: x*x
3  somme = lambda x, y: x+y
4  # appel d'une expression
5  print carre(4)
6  print somme(12,6)
7  c = lambda x,y,z : (x*y)/z+(z*y)/x
8  print c(5, 6, 7)
9  12
10 print 5*c(6,7,9)/c(9,8,7)
11 4
```

Exercices : Les fonctions

1. Écrire une procédure table qui simule une table de multiplication.
   Cette procédure prend quatre paramètres : base (qui correspond à la constante de multiplication), debut, fin et inc (qui correspond au pas d'avancement des nombres).
   La procédure table doit afficher la table des base , de debut à fin, de inc en inc.
   Tester la procédure par un appel dans le programme principal.

2. Écrire une fonction "compteurMots" ayant un argument (une chaîne de caractères) et qui renvoie un dictionnaire qui contient la fréquence de tous les mots de la chaîne entrée.

Exercices : Les fonctions

3. Implémentez une pile LIFO avec une liste. Vous devez définir 3 fonctions :
   - pile : qui construit une pile à partir d'une suite de variables passées en paramètre ;
   - empile : empile un élément en « haut » de la pile ;
   - dépile : dépile un élément du « haut » de la pile.

Notion de générateur

- Le mot clef « **yield** » lorsqu'il est utilisé dans une fonction permet de la transformer en générateur
- L'appel de cette fonction retourne un objet de type « **generator** » utilisable avec une boucle « **for** »
- A chaque appel, la fonction effectue un traitement jusqu'au mot clé « **yield** », puis retourne l'expression avant de continuer
- Exemple : La fonction **xrange(nombre)** repose sur la notion de générateur pour le traitement des données.

Python - Fonctions

- **Exemple** :

```
1 import sys
2 def gen_fibonacci(max = sys.maxsize):
3   a, b = 0, 1
4   while a < max:
5     a, b = b, a + b
6     yield a
7
8 for n in gen_fibonacci(1000):
9   print n
```

Exercices : Les fonctions

4. Écrire un programme qui affiche n fois les couleur d'un feux de signalisation (rouge, orange, vert).
   Utilisez une méthode contenant un « Yield » pour développer votre programme.

Python - Fonctions

- Résumé :

```
# idem à def f(x): return x + 1
functionvar = lambda x: x + 1
>>> print functionvar(1)
2
def passing_example(a_list, an_int=2, a_string="A default
    string"):
  a_list.append("A new item")
  an_int = 4
  return a_list, an_int, a_string
>>> my_list = [1, 2, 3]
>>> my_int = 10
>>> print passing_example(my_list, my_int)
([1, 2, 3, 'A new item'], 4, "A default string")
>>> my_list
[1, 2, 3, 'A new item']
>>> my_int
10
```

Python - Fonctions

- On peut utiliser les listes et surtout les dictionnaires pour remplacer une suite de if-elif-else.

```
if choix == 0:
  fonctionA()
elif choix == 1:
  fonctionB()
elif choix == 2:
  fonctionC()
elif ...etc...
```

```
dico = {"0" : fonctionA, "1" : fonctionB, "2": fonctionC,
       etc...}
dico[choix]()
```

```
[fonctionA, fonctionB, fonctionC, ...etc...][choix]()
```

Exercices : Les fonctions

5. De la même manière que l'exercice précédent sur la pile LIFO, implémentez une file FIFO avec une liste.
   Essayez d'ajouter un menu de manipulation de la file.

   Remarque : N'utilisez que des procédures sans arguments et une liste en variable globale.

Modularité

- Le code source Python peut être séparé dans différents fichiers (appelés modules)
- Syntaxe pour accéder au contenu d'un module

```
1 import math    # Importe le module "math"
2 print math.pi    # Utilisation du module "math"
3 from math import pi, sqrt
4 # import que les fonction 'pi' et 'sqrt'
5 print pi, sqrt
6 from math import *  # Importe tout le contenu de "math"
7 print pi
```

- Cependant, il y a un risque avec la méthode from d'avoir une collision de nom. Le cas échéant, c'est le dernier import qui prime.

Modularité

- La variable «**sys.path** » contient la liste des répertoires depuis lesquels l'interpréteur python charge des modules (liste modifiable)

```
1 import sys
2 print sys.path
```

- Un script python peut contenir des définition ainsi que du code exécutable. Lors d'un import de ce module la partie exécutable, s'exécutera qu'une seule fois.
- On peut utiliser une instruction particulière pour distinguer les deux parties quand on fait un import, et donc importer que la partie concernant les définitions. Cette instruction est la suivante :

```
1 if __name__ == "__main__":
```

Sommaire

Approche de programmation

Deux type de méthodes de programmation :

- Fonctionnelle,
- Orientée objet

1. Programmation fonctionnelle : c'est une approche
   - Hiérarchique descendante,
   - Qui dissocie le problème de la représentation de données, du problème du traitement de ses données.

Approche de programmation

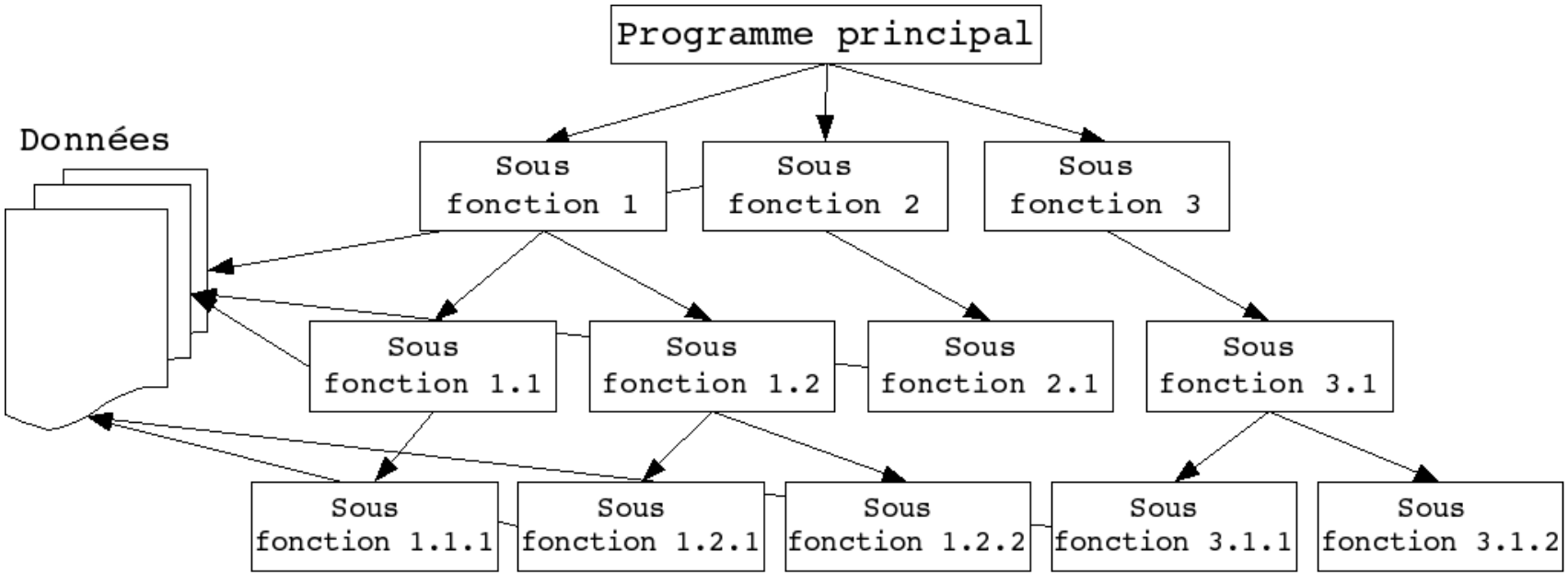


FIGURE : Représentation graphique d'une approche fonctionnelle

Approche de programmation

Limites de l'approche fonctionnelle :

- Le découpage impose la hiérarchie
  $\Rightarrow$ le couplage fonction/hiérarchie est statique

- Conséquence : l'évolution nécessite des modifications lourdes.

Approche de programmation

2. La programmation orientée objet (POO) :

   - C'est un type de programmation qui a pour avantage de posséder une meilleure organisation des programmes (spécialement les plus gros programme), elle contribue à la fiabilité des logiciels et elle facilite la réutilisation de code existant.
   - la POO consiste en la définition et l'assemblage de briques logicielles appelées «**objets** » afin de résoudre une problématique donnée.
   - Un objet est une entité composée de données caractérisant cet objet, et de traitements s'effectuant sur ces données.

La Programmation Orientée Objet (POO)

- Considère le logiciel comme une collection d'objets dissociés, identifiés et possédant des caractéristiques
- Une caractéristique est soit :
  - Un attribut (i.e. une donnée caractérisant l'état de l'objet),
  - Une entité comportementale de l'objet (i.e. une fonction).
- Les objets communiquent entre eux par appel de fonctions (méthodes), envoi dynamique de messages.
  - Cette approche rapproche les données et leurs traitements associés au sein d'un unique objet.
- Avantage : l'évolution ne remet en cause que l'aspect dynamique sans remettre en cause les objets.

Intérêt de la POO

- Permet de modéliser la réalité facilement
- L'encapsulation du code
- Modularité du code
- Composants réutilisables
- L'héritage permet de créer facilement de nouveaux objets de plus en plus spécifiques

La Programmation Orientée Objet (POO)

- La POO introduit de nouveaux concepts, en particulier ceux d'encapsulation, de classe, d'héritage et de polymorphisme.

- Principe d'encapsulation des données :
  - Pour accéder à l'état d'un objet (à ses données), il faut passer par des fonctions (méthodes ou opérations),
  - L'état d'un objet est caché en son sein ⇒ pour changer l'état d'un objet il faut lui envoyer un message (appeler une fonction).

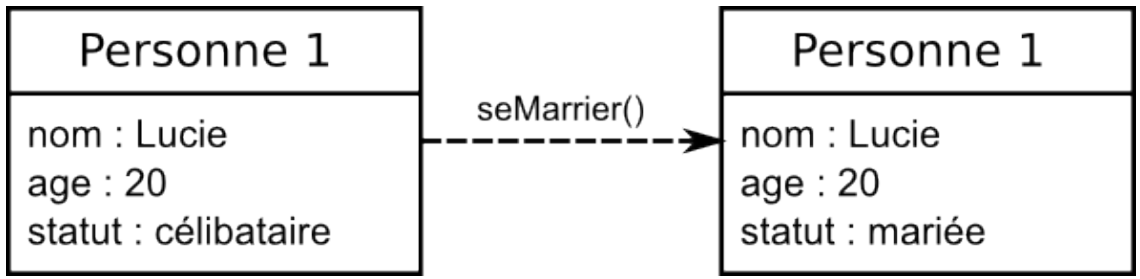


FIGURE : Modification des données d'un objet

La Programmation Orientée Objet (POO)

- Un objet informatique ?

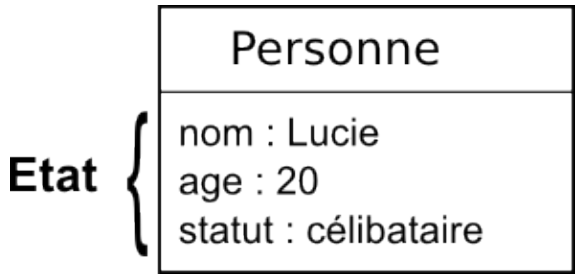


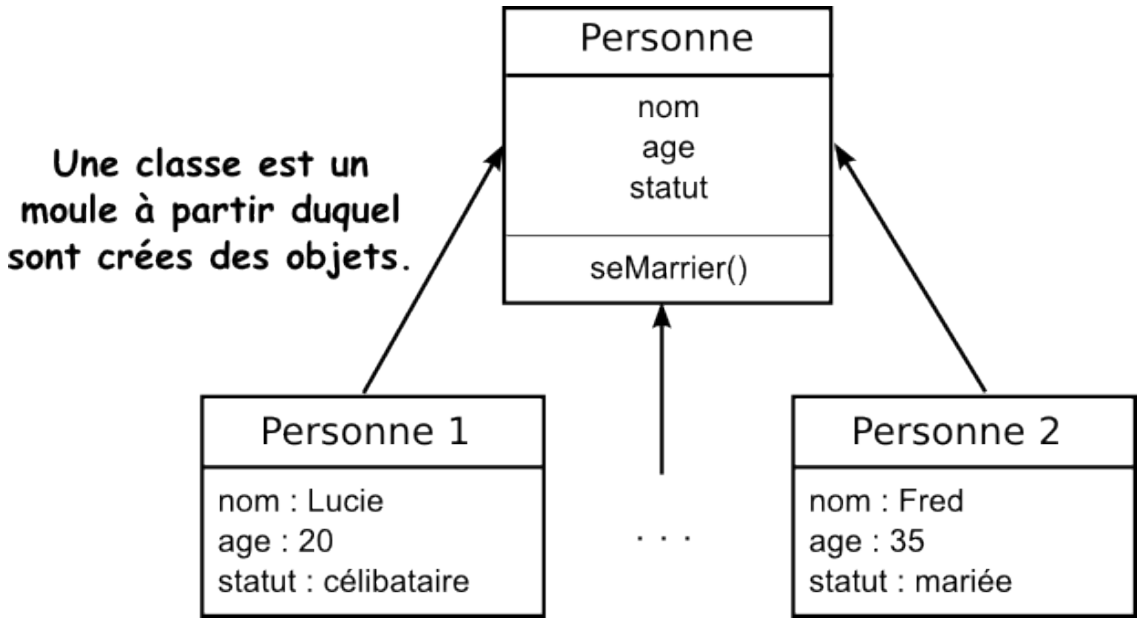
FIGURE : Objet = Etat + Comportement + Identité

- État : valeurs instantanées des attributs (données) d'un objet,
- Comportement : opérations possibles sur un objet, (appel de méthodes ou messages envoyés par d'autres objets),
- Identité : chaque objet à une existence propre (il occupe une place mémoire qui lui est propre) ; on les différencie par leur noms (noms de variable).

La Programmation Orientée Objet (POO)

Notion de **Classe** :

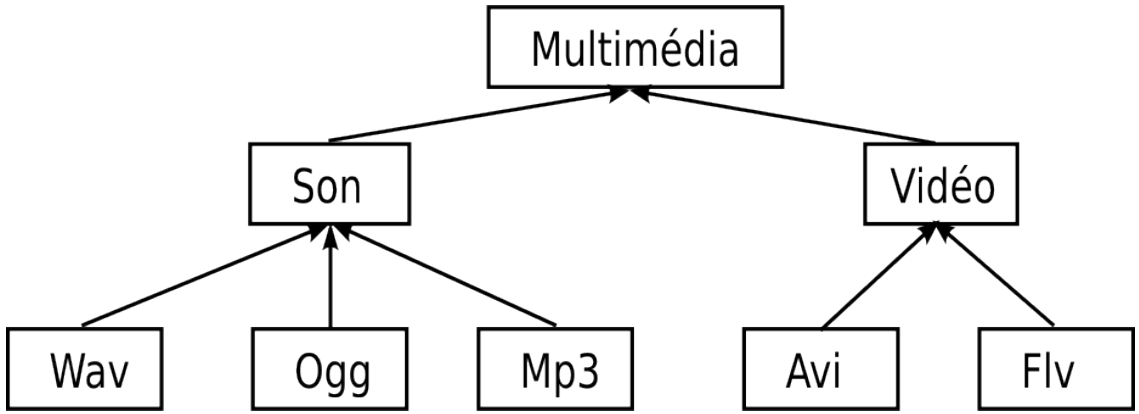
- Un système complexe a un grand nombre d'objets. Pour réduire cette complexité, on regroupe des objets en classes.
- une classe est la description d'un ensemble d'objets ayant une structure de données commune (attributs) et disposant des mêmes méthodes.



La Programmation Orientée Objet (POO)

Notion d'**Héritage** :

- L'héritage est un mécanisme de transmission des caractéristiques d'une classe (ses attributs et méthodes) vers une sous-classe.
- Une classe peut être spécialisée en d'autres classes, afin d'y ajouter des caractéristiques spécifiques ou d'en adapter certaines.
- L'héritage évite la duplication et encourage la réutilisation.



La Programmation Orientée Objet (POO)

Notion de **Polymorphisme** :

Avec l'héritage vient le concept de Polymorphisme :

- c'est la possibilité de redéfinir certaines des méthodes héritées de sa classe de base. Le polymorphisme permet ainsi de traiter de la même manière des objets de types différents, pour peu qu'ils soient issus de classes dérivées d'une même classe de base,
- Le polymorphisme permet aux méthodes d'avoir plus de spécialisation.

La modélisation et la POO

Le développement en POO se fait en plusieurs temps :

1. **La conception** du diagramme de classes (UML, ...), indépendante du langage
   - quelles classes ? quels liens entre les classes ?
2. **L'implémentation**, spécifique au langage
   - coder, tester et documenter les classes

Avantages : architecturer et factoriser le code, gain de temps, maintenance...

La modélisation avec UML

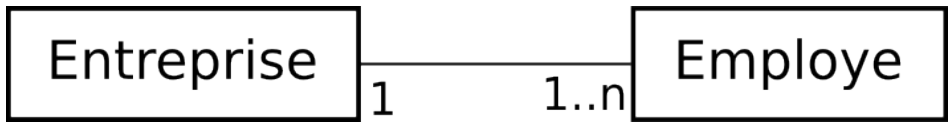
- **Unified Modeling Language** (UML) est un langage graphique qui permet de représenter et de communiquer les divers aspects d'un système d'information.
  - Aux graphiques sont bien sûr associés des textes qui expliquent leur contenu.
- UML comporte ainsi plusieurs types de diagrammes représentant autant de vues distinctes pour représenter des concepts particuliers du système d'information.
- Avantages : architecturer et factoriser le code, gain de temps, maintenance...

La modélisation avec UML

- UML peut être utilisé pour :
  - Représenter les limites d'un système et ses principales fonctions à l'aide de scénarios et d'acteurs,
  - Illustrer le déroulement des scénarios à l'aide de diagrammes d'interactions,
  - Représenter la structure statique du système avec des classes
  - Modéliser le comportement des objets à l'aide de diagrammes d'états
  - Révéler l'architecture d'implantation physique avec des diagrammes de déploiements
  - ...

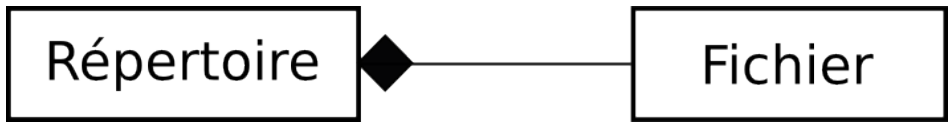
UML - Diagramme de classe

- Association entre deux classes : trait plein + cardinalité
  - Une entreprise a plusieurs employés (1..n)
  - Un client commande un produit
  - Cardinalité :
    - ★ * : plusieurs
    - ★ n-m : entre ’n’ et ’m’ (n<m)
    - ★ n : exactement ’n’



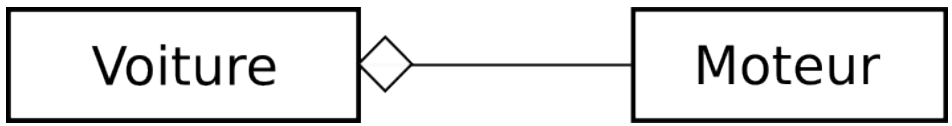
UML - Diagramme de classe

- Composition : trait plein + losange sombre du cote du conteneur
  - Un répertoire contient un fichier ;
  - Lorsqu'on instancie le fichier on instancie (ou on l'attribut à) un répertoire au préalable ;
  - Lorsqu'on détruit le répertoire, on détruit le fichier en cascade.



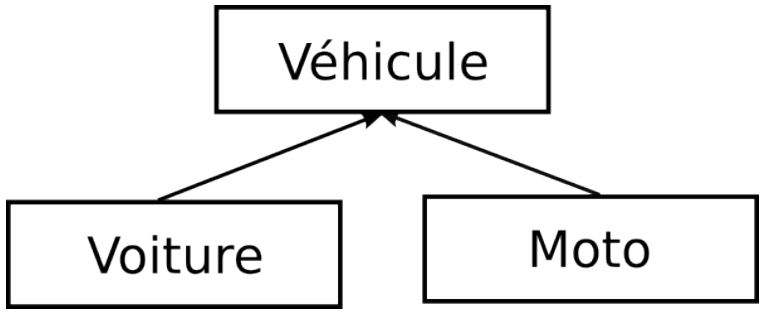
UML - Diagramme de classe

- Agrégation : trait plein + losange clair du cote du conteneur
  - Une voiture contient un moteur ;
  - Lorsqu'on détruit la voiture, on ne détruit pas forcement avec le moteur.



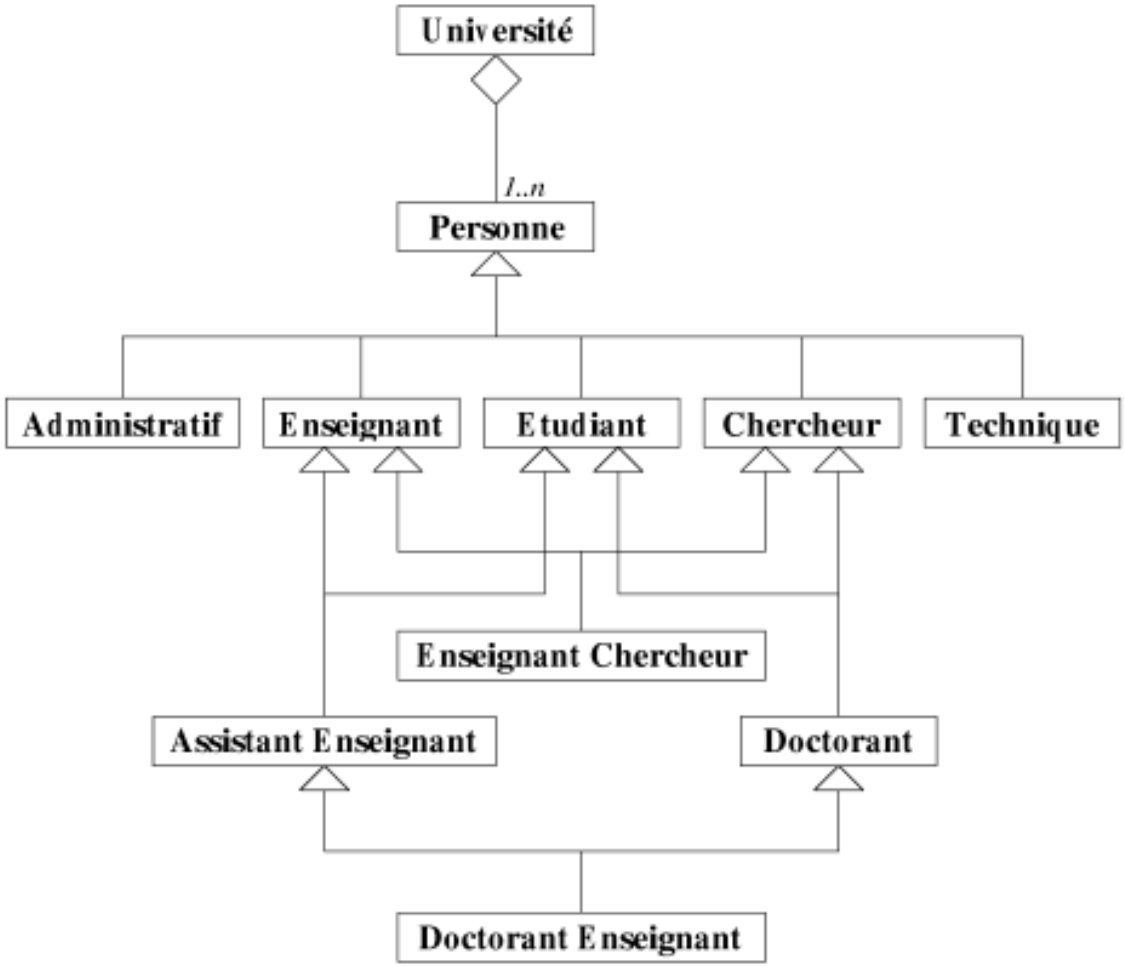
UML - Diagramme de classe

- Héritage : trait + triangle du cote de la classe la plus générique
  - Une voiture est un type de véhicule
  - Un ’mp3’ est un type de fichier qui est un type de fichier ’Son’



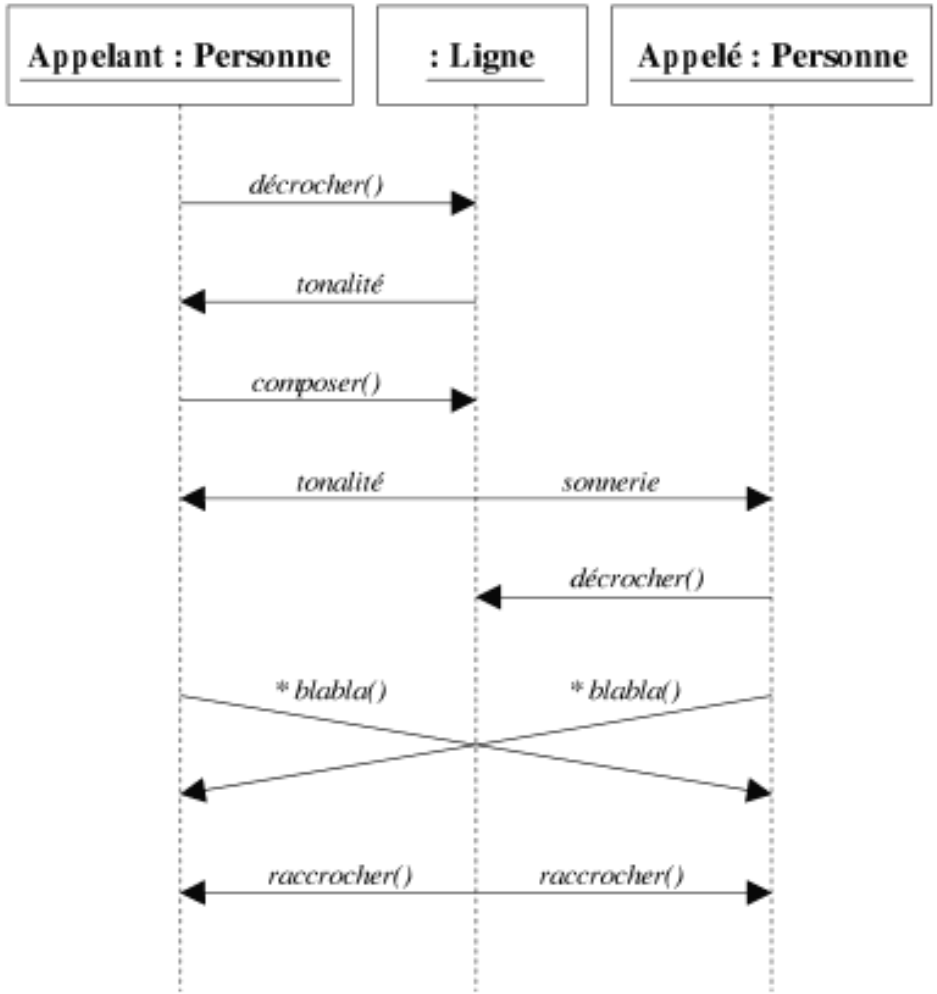
Exemple de diagrammes UML

- Diagramme de classes :



Exemple de diagrammes UML

- Diagramme de séquence :



Python - Classe (1/6)

- La déclaration d'une classe se fait avec le mot clef « **class** » suivi d'un nom et de 2 points (:)

```
1 class Personne:
2  pass
```

- L'instanciation se fait simplement avec le nom de la classe sans opérateurs supplémentaires

```
1 monInstancePersonne = Personne()
```

Python - Classe (2/6)

On peut distinguer deux sortes de données dans une classe :

- Variables d'instances : qui appartiennent aux instances d'objets. On y accède en utilisant une référence sur l'objet (en utilisant le mot-clé « **self** »)
- Variables de classes : appartiennent à la classe. On y accède en préfixant avec le nom de la classe

Remarque :

Toujours penser à initialiser tous les attributs dans le constructeur, sinon ils n'existent pas !

Python - Classe (3/6)

Les méthodes :

- Fonctions placées à l'intérieur d'une classe
- Le premier paramètre est toujours une référence d'instance (par convention, nommé « **self** ») et sera automatiquement renseigné lors de l'appel
- Méthodes statiques : précéder la définition de la méthode par : « **@staticmethod** »

Python - Classe (4/6)

- Constructeur
  - Méthode particulière exécutée automatiquement lors de l'instanciation d'un nouvel objet
  - Doit s'appeler « **__init__** »
  - Comme toute méthode, son premier paramètre est une référence sur l'objet (" self ")
- Destructeur
  - Doit s'appeler « **__del__** »

Python - Classe (5/6)

- Exemple :

```
1  class Personne:
2    nbr = 0
3    def __init__(self, nom, prenom):
4      self.m_nom = nom
5      self.m_prenom = prenom
6      Personne.nbr += 1
7
8    def __del__(self):
9      Personne.nbr -= 1
10     print("Bye")
11
12 #utilisation
13 var1 = Personne("Dupont", "Lise")
14 print var1.m_nom,  var1.m_prenom  # Dupont Lise
15 var1.m_prenom = "Jasmine"
16 print var1.m_prenom   # Jasmine
```

Python - Classe (6/6)

- Protection d'accès des attributs et méthodes
  - « we're all consenting adults here »
  - C'est la citation qui résume la philosophie du Python concernant la protection d'accès des attributs et méthodes
  - Il n'y a pas de réelle protection, seulement une convention : lorsque le nom d'un attribut ou d'une méthode commence par 2 « underscores », alors c'est un élément privé

Python - Classe : property

- Exemple :

```
1  class Personne:
2      ...
3      def __init__(self, nom, prenom):
4          self.m_nom = nom
5          self.__prenom = prenom
6          Personne.nbr += 1
7
8      def get_prenom(self):
9          return self.__prenom
10
11     def set_prenom(self, value):
12         self.__prenom = value
13
14     def del_prenom(self):
15         del self.__prenom
16     ...
17     prenom = property(get_prenom, set_prenom, del_prenom, "
           prenom's docstring")
```

Python - Classe : property

- Exemple :

```
1 #utilisation
2 var1 = Personne("Dupont", "Lise")
3 print var1.m_nom,  var1.prenom      # Dupont Lise
4 var1.prenom = "Julie"
5 print var1.prenom
6 print var1.m_nom,  var1.prenom
```

Python - Classe : property

- Exemple :

```
1  class Personne:
2      ...
3      @property # readonly
4      def prenom(self):
5          return self.__prenom
6
7      @prenom.setter
8      def prenom(self, value):
9          self.__prenom = value
10
11     @prenom.deleter
12     def prenom(self):
13         del self.__prenom
14     ...
```

Python - Classe : property

- Exemple :

```
1 #utilisation
2 var1 = Personne("Dupont", "Lise")
3 print var1.m_nom,  var1.prenom      # Dupont Lise
4 var1.prenom = "Julie"
5 print var1.m_nom,  var1.prenom
```

Les Méthodes spéciales

Les méthodes spéciales :

- Commencent et terminent par 2 « underscores »
- Permettent de surcharger les opérateurs, définir le constructeur et le destructeur, ...

__init__	Appelée juste après la création d'un objet (Constructeur)
__del__	Appelée juste avant la destruction d'un objet (Destructeur)
__str__	Appelée par la fonction de conversion de type « str » et la fonction d'affichage « print »
__lt__	x<y
__le__	x <= y
__eq__	x == y

Les Méthodes spéciales

__ne__	x != y ou x <> y
__gt__	x>y
__ge__	x >= y
__add__	Appelée pour sommer 2 objets
__neg__	Appelée pour obtenir l'opposé d'un objet
__sub__	Appelée pour la soustraction de 2 objets
__mul__	Appelée pour la multiplication de 2 objets
__div__	Appelée pour la division de 2 objets
__getitem__	Appelée pour accéder à « objet[clef] »
__setitem__	Appelée pour modifier « objet[clef] »
...	...

L'héritage (1/2)

- Syntaxe pour définir une classe qui hérite d'une autre classe

```
1 class Client(Personne):
2   pass
```

- La classe fille (Client) peut appeler une méthode ou un attribut de la classe mère (Personne) en préfixant avec le nom du parent

- Python supporte l'héritage multiple

```
1 class Client(Personne, Humain):
2   pass
```

L'héritage (2/2)

- Pour étendre le comportement d'un ancêtre, la méthode du descendant doit appeler la méthode de l'ancêtre explicitement

```
1 class Client(Personne):
2   def __init__(self, nom, prenom, identifiant)
3     Personne.__init__(self, nom, prenom)
4     self.m_identifiant = identifiant
```

- Toutes les classes héritent de « object » directement ou indirectement

Exemple d'héritage

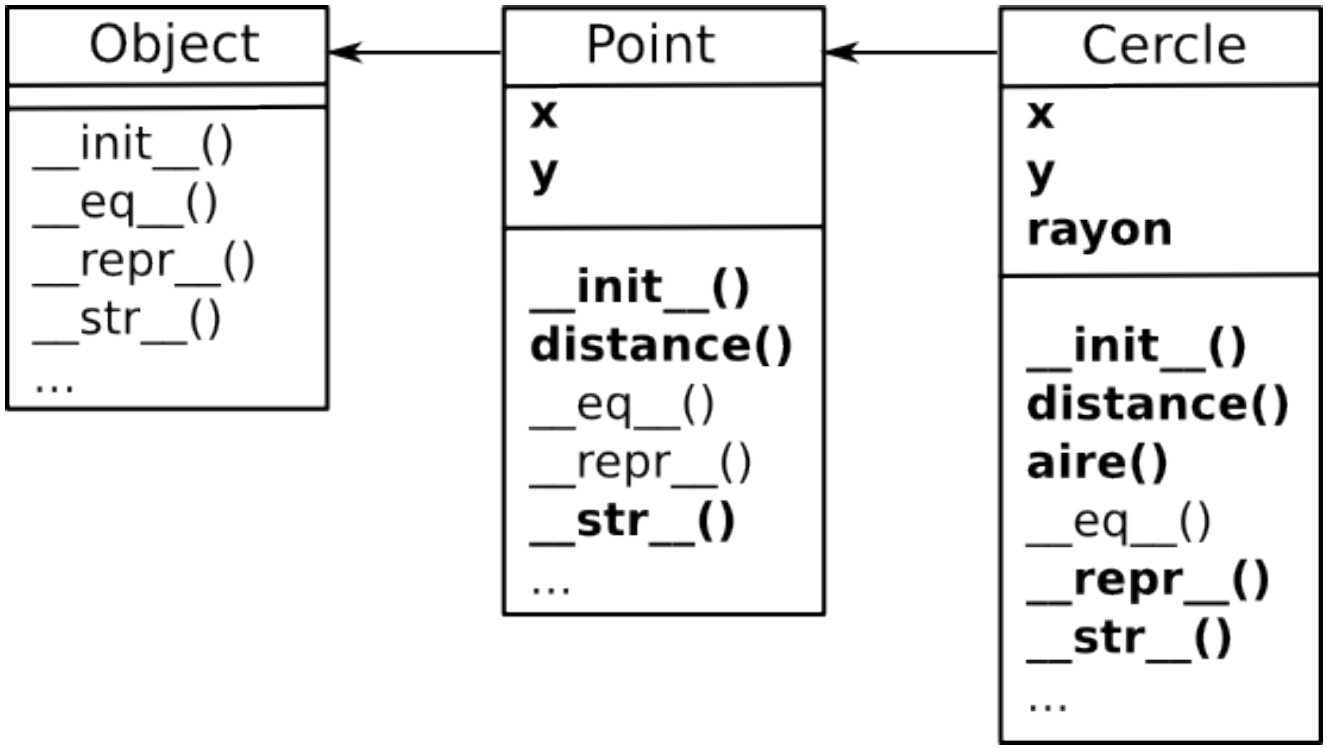


FIGURE : Exemple : La classe cercle hérite de la classe point

Le polymorphisme (1/2)

- Le Python a la particularité de permettre de profiter du polymorphisme sans héritage
- Le « duck typing » :
  - Citation : « If it looks like a duck and quacks like a duck, it must be a duck »
  - Traduction : «si ça ressemble à un canard et que ça fait le bruit d'un canard, c'est sans doute un canard»

Le polymorphisme (2/2)

- Principe du « duck typing »
  - Pour pouvoir utiliser plusieurs objets de la même façon, il suffit que ces objets possèdent tous, les méthodes nécessaires

```
1 def maFonction(a, b, c):
2   return (a + b)*c
3 #Appel
4 maFonction(2, 5, 3) # 21
5 maFonction("abra", "cad", 2) # "abracadabracad"
6 maFonction([2, 4], ["a"], 3)
7 # [2, 4, "a", 2, 4, "a", 2, 4, "a"]
```

La documentation

- Le Python permet d'écrire des commentaires au début des modules, classes, fonctions, ...
- On y accède avec l'attribut « __doc__ »

```
1 """Commentaire de module"""
2 import math
3 class Cercle:
4   """Commentaire de classe"""
5   def __init__(self, rayon):
6     """Commentaire de méthode"""
7     self.rayon = rayon
```

L'introspection (1/2)

- La fonction « dir() » prend en paramètre un objet et retourne une liste qui contient tous les nom utilisés (méthodes, variables, ...)
- La liste « obj.__dict__ » contient les attributs modifiés
- Cette fonction permet donc l'introspection

L'introspection (2/2)

- La méthode « getattr(objet, méthode, valeur par défaut) » permet d'obtenir une référence sur une méthode de l'objet
- « obj.__class__.__name__ » : permet d'obtenir le nom de la classe
- « obj.__class__.__bases__ » : permet d'obtenir les parents de la classe

Exercices : Programmation Orientée Objet

1. Définir une classe "MaClasse" possédant les attributs suivants :
   - attributs de classes : x = 42 et y = x + 7
   - une méthode "affiche" contenant un attribut d'instance z = 42 et qui affiche les valeurs des variable y et z.

   Dans le programme principal, instanciez un objet de la classe "MaClasse" puis invoquez la méthode "affiche".

2. Définir une classe "Vecteur2D" avec un constructeur fournissant les coordonnées par défaut d'un vecteur. (Exemple : x = 0, y = 0) Dans le programme principale, instanciez un "Vecteur2D" sans paramètres, un "Vecteur2D" avec ses deux paramètres et affichez-les.

Exercices : Programmation Orientée Objet

3. Enrichissez la classe "Vecteur2D" précédente en lui ajoutant une méthode d'affichage et une méthode de surcharge d'addition de deux vecteurs.

   Dans le programme principal, instanciez deux Vecteur2D, affichez-les et affichez leur somme.

Exercices : Programmation Orientée Objet

4. Définir une classe Rectangle avec un constructeur donnant des valeurs (longueur et largeur) par défaut et un attribut nom = "rectangle", une méthode d'affichage et une méthode surface renvoyant la surface d'une instance.

   Définir une classe Carre héritant de Rectangle et qui surcharge l'attribut d'instance : nom = "carré".

   Dans le programme principal, instanciez un Rectangle et un Carre et affichez-les.

Exercices : Programmation Orientée Objet

1. Définissez une classe Personne tel que :
   - attribut d'instance : nom, prénom
2. Définissez une classe Client qui hérite de Personne :
   - attribut d'instance : identifiant
3. Définissez une classe CompteBancaire() qui permette d'instancier des objets tels que compte1, compte2, etc.
   - Le constructeur de cette classe initialisera trois attributs d'instance, client, solde et typeCompte. Ce dernier a une valeur par défaut qui est "user"
   - La variable d'instance type peut avoir la valeur " user " ou alors " superUser "

Exercices : Programmation Orientée Objet

- Quatre autres méthodes seront définies :
  - credit (somme) permettra d'ajouter une certaine somme au solde ;
  - debit (somme) permettra de retirer une certaine somme du solde ;
  - solde () permettra d afficher le nom du titulaire et le solde de son compte ;
  - emprunt(somme) permettra au client d'emprunter cette somme selon les cas suivants :
    - ★ le client possède encore de l'argent dans son compte
    - ★ le solde du client est vide ou négatif :
      si le client est de type « user » il ne pourra pas emprunter
      si le client est de type « superUser » il pourra emprunter

Implémentez également la méthode qui permettra d'afficher tout le descriptif du client.

Exceptions (1/9)

Principe de gestion des erreurs (exceptions) :

- On délimite un bloc d'instructions à exécuter
- Si il n'y a pas d'erreur
  - Le code se déroule normalement
- Si une erreur se produit pendant l'exécution
  - L'exécution est interrompue et le programme recherche le code (de gestion d'erreurs) capable de prendre en charge ce type d'erreur
  - Ensuite, l'exécution reprend normalement
  - Les erreurs non gérées stoppent le programme

Exceptions (2/9)

- Syntaxe pour gérer les exceptions :

```
1 while 1:
2   try:
3     saisie = int(input("Saisir un nombre:"))
4     break
5   except ValueError:
6     print("Saisie invalide...")
7
8 print("La valeur saisie est: " + str(saisie))
```

- Si le code après « try » soulève une exception de type « ValueError », elle peut être gérée

Exceptions (3/9)

Un gestionnaire d'exceptions peut gérer plusieurs types d'erreurs

- Soit en les cumulant

```
except (RuntimeError, TypeError, NameError):
  ...
```

- Soit en ayant plusieurs clauses « except »

```
except ZeroDivisionError:
  print("Division par zéro")
except :
  print("Autre type d'erreur")
```

Exceptions (4/9)

- Autre syntaxe possible :

```
while 1:
  ...
  except ValueError:
    print("Saisie invalide...")
  else:
    print "  fonctionne. "

print("La valeur saisie est: " + str(saisie))
```

Exceptions (5/9)

- Pour avoir du code toujours exécuté quel que soit le déroulement du code à tester, il faut utiliser la clause « finally »

```
try:
  # Instructions
except Exception:
  # Gestion de l'erreur de type Exception
finally:
  # Code toujours exécuté
```

Exceptions (6/9)

Déclencher une exception

- Pour déclencher une exception, il faut utiliser le mot clef « raise »

```
1 raise NameError
```

- Pour créer des objets utilisables pour déclencher une exception, il faut obligatoirement qu'ils dérivent de la classe de base « Exception »
- Le mot clef « raise » permet aussi de relancer une exception

Exceptions (7/9)

- Une exception peut avoir des arguments

```
1 raise NameError("toto", "pouet")
```

- On peut récupérer une référence sur l'instance d'une exception et accéder à ses arguments

```
1 except NameError as instErr:
2   print(instErr.args) # ('toto', 'pouet')
3   x, y = instErr.args # x = 'toto', y = 'pouet'
```

Exceptions (8/9)

Exemple d'exceptions :

ZeroDivisionError	Erreur de division par zéro
NameError	Erreur de nom
TypeError	Erreur de type
KeyboardInterrupt	Interruption de l'utilisateur (Ctrl-C)
ValueError	Utilisé lorsqu'une fonction reçoit un argument avec le bon type mais pas une bonne valeur
MemoryError	Manque de mémoire
ImportError	Échec de l'import d'un module
...	...

Exceptions (9/9)

- Définir ses propre exceptions

```
1 class MonError(Exception):
2   def __init__(self, value):
3     self.value = value
4   def __str__(self):
5     return repr(self.value)
```

Compréhension des packages

- Un package est un répertoire dans lequel nous avons un ou plusieurs modules.
- Le package contient au minimum un fichier nommé __init__.py
- Le nom du répertoire donne le nom du package :
  Prenons un package matelli
  - import matelli me permet d'utiliser toutes les classes de tous les modules de matelli
  - var1 = matelli.module1.Classe1()
  - var2 = matelli.module2.Classe43()

  Avec le from matelli import module1 me permet d'utiliser toutes les classes du module "module1" uniquement de mon package
  - var1 = module1.Classe1()

Compréhension des packages

- Bien qu'indispensable, le fichier __init__.py peut être un fichier vide.
- Dans le cas contraire, son code sera exécuté.
- from package import * , que ce passe t'il ?
  ⇒ Juste l'exécution de __init__.py
- Une variable spéciale permet d'inclure les modules présents dans le package.
- __all__ = ["module1", "module2"]

Compréhension des packages

- Un package est généralement fourni avec un installeur.
- L'installeur est un script Python qui par convention se nomme : "setup.py"
- Son contenu ressemble à ceci

```
1 from distutils.core import setup
2 setup(
3 name='Nom du Package',
4 version='1.0',
5 description='Package pour faire ...',
6 author='Nom Prenom',
7 author_email='name@example.com',
8 packages=['Nom du Package'],
9 )
```

Compréhension des packages

- L'installeur nous aide pour l'installation du package sur la plateforme où sera exécuté notre script (développement, production)
- Si nous sommes administrateur : Le package peut être utilisable par tous les utilisateurs et scripts python du système.
  - python setup.py install
- Si nous sommes utilisateur : Le package est n'utilisable que pour nous et nos scripts.
  - python setup.py install −−user

Compréhension des packages

- Cas particulier, l'installation ne peut être effectuée :
  - Mettre le package dans un chemin particulier
  - Modifier dans le script la variable sys.path
  - Mettre le package dans le même répertoire que le script python.

Compréhension des packages

- L'installeur permet de créer une archive (tar.gz, zip) pour la distribution de notre package.
  - python setup.py sdist
- L'archive se trouve dans le répertoire dist/

Paquetages disponibles (1/2)

- Un module contenant d'autres modules (dits sous-paquetages)
  - Interface graphique
  - Calculs scientifiques
  - Appel à d'autres langages
  - Traitement de fichiers XML
  - Programmation réseaux
- http ://pypi.python.org/pypi

Paquetages disponibles (2/2)

- Quelques packages :

Chaines de caractères	re
Type de données	date, calendrier, Queue (fifo, lifo), copy
Mathématique et Nombre	math, random, opérateur du language
Accès au fichier et répertoire	stat, tempfile, linecache
Persistance de données	shelve, sqlite
Compression de données	gzip, bz2, zipfile, tarfile
Format de fichier	csv, ConfigParser, robotparser
Cryptage	hashlib : sha1 sha512 md5
Système d'exploitation	os, getopt, time, theading
Réseau	socket, ssl, signal
Données Internet	json, base64
Fichiers à balise	htmllib, xml.dom, xml.sax
Protocole Internet	ftplib, smtplib, poplib, httplib, xmlrpc

Sommaire

Python - Modules - Fichier

- Pour manipuler les répertoires Python fournie plusieurs fonctions :
- Exemple :
  - Création de répertoire :

```
1 import os
2 os.mkdir( "chemin du répertoire" )
3 os.mkdir( "chemin du répertoire" , mode )   # pour Unix
```

  - Supprimer un fichier ou un répertoire :

```
1 import os
2 os.remove( "chemin du fichier " ) # pour les fichiers
3 os.removedirs( "chemin du répertoire" )
```

Python - Modules - Fichier

- Comment renommer des fichiers ou un répertoire ?
- Deux fonctions sont disponibles

```
1 import os
2 os.rename("c:/tmp/fichier", "c:/tmp/fichier.old")
3
4
5 import os
6 os.renames("c:/tmp/fichier", "c:/newdir/fichier.new")
```

renames créé les répertoires s'ils n'existent pas.

Python - Modules - Fichier

- Manipuler les différentes parties d'un nom de fichier ?

```
1  import os.path
2  # connaitre le répertoire d'un chemin.
3  os.path.dirname("c:\\Python27\\python.exe" )
4  'c:\\Python27'
5  # récupérer le nom du fichier
6  os.path.basename( "c:\\Python27\\python.exe" )
7  'python.exe'
8  # avoir un tuple des deux
9  os.path.split( "c:\\Python27\\python.exe" )
10 ('c:\\Python27', 'python.exe')
11 # avoir l'extension du fichier
12 os.path.splitext("c:\\Python27\\python.exe" )
13 ('c:\\Python27\\python', '.exe' )
```

Python - Modules - Fichier

- Est-ce un fichier ou un répertoire ?

```
1 import os.path
2 os.path.isfile( "c:\\Python27\\python.exe" )
3 os.path.isdir( "c:\\Python27" )
```

- Est-ce que le fichier ou le répertoire existe ?

```
1 os.path.exists( FileOrDir )
```

Python - Types et opérations - Fichier

- La gestion de fichiers est une des bases de Python, c'est pour cela que le "fichier" est un type de base dans Python au même titre que : int, char, ...
- Pour cela, Python fournit des méthodes simples de gestion de fichiers.
- Comment faire pour lister un répertoire ?
- Deux possibilités :

```
1 import os
2 os.listdir("c:/Python27/")
3
4 import glob
5 glob.glob("c:/Python27/*")
```

Python - Modules - Fichier

- Module shutil
- Fonctions utiles :

shutil.move()	Déplace ou renomme un fichier
shutil.copy()	Copie un fichier dans un répertoire

- Pour modifier les droits : os.chmod()
- glob.glob() permet l'utilisation des métacaractères (wildcards)

Python - Modules - Fichier

- Python fournit une fonction pour l'ouverture des fichiers.

```
1 >>> f = open(filename, 'mode')
```

  - filename : doit être le chemin du fichier
  - mode : est le mode d'ouverture du fichier
    Les différents modes sont :
    - ★ **r** : pour lire
    - ★ **w** : pour écrire
    - ★ **a** : pour écrire à la fin du fichier
    - ★ **b** : ouvre le fichier en mode binaire
    - ★ vous pouvez utiliser le "**+**" avec le **r**, le **w** ou le **a**. Cela permet l'ouverture du fichier en lecture/ écriture

Python - Modules - Fichier

- Exemple :

```
>>> f = open("/tmp/fichier_test", 'r')
>>> f
<open file '/tmp/fichier_test', mode 'r' at 0x7fe76057cc90>
# Nous pouvons récupérer le mode du fichier
>>> f.mode
'r'
# Nous pouvons aussi récupérer son nom
>>> f.name
'/tmp/fichier_test'
```

Python - Modules - Fichier

- Exemple :

```
1 # La méthode pour connaitre la position du curseur dans le
      fichier est :
2 >>> f.tell()
3 0
4 # La méthode pour déplacer le curseur dans le fichier est :
5 >>> f.seek(-128, 2)
```

- Nous pouvons déplacer ce curseur, ici 128 octets avant la fin
  - 0 ou os.SEEK_SET : position absolue
  - 1 ou os.SEEK_CUR : position courante du curseur
  - 2 ou os.SEEK_END : position de la fin

Python - Modules - Fichier

- Écriture :
  - Ouverture de fichiers :

```
1 fileObj = open("/home/myFile", 'w')
2 # ou:
3 fileObj = open("/home/myFile", 'a')
4 # Ouverture de fichiers binaires
5 fileObj = open("/home/myFile", 'wb')
6 # idem avec 'ab' et 'rb'
```

  - Écriture séquentielle :

```
1 fileObj.write("Bonjour, toto!")
```

  - Penser à toujours fermer les fichiers après manipulation

```
1 fileObj.close()
```

Python - Modules - Fichier

- Lecture séquentielle :

```
1 fileObj = open("/home/myFile", 'r')
2 text = fileObj.read()  # lit l'intégralité du fichier
3 text1 = fileObj.read(10)   # lit les 10 premiers caractères
4 text2 = fileObj.read(15)   # lit les 15 caractères suivants
5 fileObj.close()
```

- Si EOF est atteint read(n) renvoie une chaîne vide ""
- Il est plus judicieux d'utiliser une variable tampon et read(n), plutôt que read()
- Si le fichier n'existe pas l'exception IOError est levée

Python - Modules - Fichier

Fichiers textes :

- La méthode readline()
  - Renvoie une seule ligne à la fois (une chaîne s'arrêtant au caractère "\n")
  - Renvoie une chaîne vide si EOF est atteint
- La méthode readlines()
  - Renvoie une liste avec comme éléments les lignes du fichier (caractère "\n" inclus)

Python - Modules - Fichier

- Le module "pickle" :
  - Il permet d'enregistrer des données avec conservation de leur type :

```
1  import pickle
2  a, b = 12, 25.45
3  f = open("home/myFile", 'w')
4  pickle.dump(a, f)
5  pickle.dump(b, f)
6  f.close()
7  f = open("home/myFile", 'r')
8  i = pickle.load(f)
9  j = pickle.load(f)
10 f.close()
```

Python - Modules - Fichier

- Fichiers et exceptions :
  - Quand une méthode sur un fichier échoue nous avons une exception : IOError
  - La bonne pratique :

```
try:
  open
  try:
    seek / read / write
  finally:
    ...
except:
    ...
```

Exercices : Module fichier

1. Écrire une fonction qui prend un répertoire/chemin en argument et qui affiche sa complète arborescence.

2. Écrivez un script qui combine les contenus de deux fichiers pour en faire un nouveau.
   Le fichier résultant devra contenir une ligne du 1er fichier suivie d'une ligne du 2éme fichier.

Sommaire

Python - Modules - Date

Il existe plusieurs modules dans python :

- time
- calendar
- datetime

time permet la représentation du temps de deux façon :

- Sous forme de secondes
- Sous forme de tuple de 9 entiers :
  year, month, day, hours, minutes, seconds, weekday, day in year, DST (été, hiver)

Des variables permettent de paramétrer : timezone, daylight

Python - Modules - Date

Exemple de représentation du temps avec les deux façon :

- Sous forme de secondes

```
>>> import time
>>> print time.time()
1354887013.48
```

- Sous forme de tuple de 9 entiers :
  year, month, day, hours, minutes, seconds, weekday, day in year, DST (été, hiver)

```
>>> print time.gmtime()
time.struct_time(tm_year=2012, tm_mon=12, tm_mday=7,
    tm_hour=13, tm_min=31, tm_sec=18, tm_wday=4, tm_yday
    =342, tm_isdst=0)
```

Python - Modules - Date

Nous avons des opérations pour créer et manipuler le temps

time()	retour d'un float du temps Unix
clock()	retour d'un float avec le temps d'exécution du programme
sleep()	permet de faire une pause
gmtime()	convertit un temps Unix en un tuple de temps (UTC)
localtime()	retour d'un tuple de temps (Local)
asctime()	convertit un tuple de temps en chaîne de caractère
ctime()	convertit un temps en seconde en chaîne de caractère
mktime()	convertit un tuple de temps en temps Unix (secondes)
strftime()	convertit un tuple de temps en chaine de caractère personnalisé
strptime()	convertit une chaine de caractère personnalisé en tuple de temps
tzset()	change la zone de temps local

Python - Modules - Date

- Exemple de manipulation de date :

```
1 t1 = time.time()
2 time.sleep(30)
3 t2 = time.time()
4
5 print time.ctime(t1)
6 print time.ctime(t2)
7
8 print time.localtime(t1)
9 print time.localtime(t2)
```

Python - Modules - Date

- Il faut définir un "format", le format est une chaîne de caractères avec des directives.

- Les directives sont consultables sur `http://docs.python.org/library/time.html#time.strftime`

Directive	Meaning
%a	Locale's abbreviated weekday name.
%A	Locale's full weekday name.
%b	Locale's abbreviated month name.
%B	Locale's full month name.
%c	Locale's appropriate date and time representation.
%d	Day of the month as a decimal number [01,31].

Python - Modules - Date

%H	Hour (24-hour clock) as a decimal number [00,23].
%I	Hour (12-hour clock) as a decimal number [01,12].
%j	Day of the year as a decimal number [001,366].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%p	Locale's equivalent of either AM or PM.
%S	Second as a decimal number [00,61].
%U	Week number of the year (Sunday as the first day of the week)
%w	Weekday as a decimal number [0(Sunday),6].
%W	Week number of the year (Monday as the first day of the week)
%x	Locale's appropriate date representation.
%X	Locale's appropriate time representation.
%y	Year without century as a decimal number [00,99].
%Y	Year with century as a decimal number.
%Z	Time zone name (no characters if no time zone exists).
%%	A literal '%' character.

Python - Modules - Date

- Exemple d'affichage personnalisé :

```
>>> from time import gmtime, strftime, localtime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Fri, 23 Mar 2012 11:10:17 +0000'

>>> strftime("%a, %d %b %Y %H:%M:%S %Z", localtime())
'Fri, 23 Mar 2012 12:11:56 PMT'

>>> strptime( time.asctime( time.localtime() ) )
time.struct_time(tm_year=2013, tm_mon=4, tm_mday=12,
    tm_hour=14, tm_min=45, tm_sec=17, tm_wday=4, tm_yday
    =102, tm_isdst=-1)
```

Python - Modules - Date

- Calendar est un module fournissant l'affichage d'un calendrier.
- Mais aussi des fonctionnalités simples comme :
  - savoir si l'année est bissextile
  - savoir le nombre d'années bissextiles entre deux années
  - savoir le nombre de jour dans un mois
  - savoir le numéro d'un jour dans une semaine. (lundi=0)
- Exemple :

```
1 import calendar
2 print calendar.weekday(2012, 03, 23)
3 calendar.prmonth(2012, 03) # affichage du mois
4 print calendar.monthrange(2012, 02)
```

Python - Modules - Date

- datetime est le module simplifié de time.
- Il est composé de 4 classes :
  - date : création d'objets (année, mois, jour)
  - time : création d'objets (heures, minute, sec, microsec)
  - datetime : objets (année, mois, jour, heure, minute, sec, microsec)
  - timedelta : objet de différence de temps (jours, sec, microsec)
- Faire une différence entre objets de type date, time ou datetime nous retourne un objet timedelta.

Python - Modules - BDD

- On peut établir une connexion à une Base De Données (BDD) en utilisant le module MySQLdb
- Ce module Python est basé sur l'API C de MySQL
- Il n'est pas disponible directement dans l'installation de Python. Il suffit cependant d'installer le package supplémentaire et d'importer le package MySQLDB
- Installation :
  - Téléchargement de l'installeur en python (Windows)
  - Utilisation des paquets de notre distribution linux (python-mysqldb)

Python - Modules - BDD

- Exemple de connexion à une base de données MySQL :

```
1 import MySQLdb
2
3 conn = MySQLdb.connect (host = "localhost", user = "
      testuser", passwd = "testpass", db = "test")
4 cursor = conn.cursor ()
5 cursor.execute ("SELECT VERSION()")
6 row = cursor.fetchone ()
7 print "server version:", row[0]
8
9 conn.close ()
```

Python - Modules - BDD

- Python fourni le module sqlite3 qui permet de faire des bdd légères .
- Exemple 1 :

```
1  import sqlite3 as lite
2  import sys
3
4  con = None
5  try:
6      con = lite.connect('test.db')
7      cur = con.cursor()
8      cur.execute('SELECT SQLITE_VERSION()')
9      data = cur.fetchone()
10     print "SQLite version: %s" % data
11 except lite.Error, e:
12     print "Error %s:" % e.args[0]
13     sys.exit(1)
14 finally:
15     if con:
16         con.close()
```

Python - Modules - BDD

- Exemple 2 :

```
1  import sqlite3 as lite
2  import sys
3
4  con = lite.connect('test.db')
5  with con:
6      cur = con.cursor()
7      cur.execute('SELECT SQLITE_VERSION()')
8
9      data = cur.fetchone()
10
11     print "SQLite version: %s" % data
```

Python - Modules - Multiprocessing

- Il existe un module qui regroupe tous les concepts de multi-tâches nommé « multiprocessing »

- Afin de pouvoir créer un programme multi-tâche , il suffit de créer un objet « processus » et de l'exécuter.

```
1 from multiprocessing import Process
2
3 def mon_processus( name ):
4   print "Hello", name
5
6 if __name__ == '__main__':
7   p = Process(target=mon_processus, args=('monName',))
8   p.start()
9   p.join()
```

Python - Modules - Multiprocessing

- Avons-nous bien fait un nouveau processus ?
- Pour le vérifier, nous pouvons afficher notre PID et celui de notre parent.

```
1  def info( info ):
2    print info, ">>> ", os.getpid()
3    print info, ">>> parent >>> ", os.getppid()
4
5  def mon_processus( name ):
6    info( "processus["+name+"]" )
7    print "Hello", name
8
9  if __name__ == '__main__':
10   info( "main" )
11   p = Process(target=mon_processus, args=('monName',))
12   p.start()
13   p.join()
```

Python - Modules - Multiprocessing

- Nous pouvons alors créer autant de processus que nécessaire. (dans les limites du système d'exploitation)

```
1  def info( info ):
2    print info, ">>>", os.getpid()
3    print info, ">>> parent >>>", os.getppid()
4  def mon_processus( name ):
5    info( "processus["+name+"]" )
6    print "Hello", name
7  if __name__ == '__main__':
8    info( "main" )
9    names = ["gwenael", "nicolas", "francois", "paul"]
10   for name in names:
11     p = Process(target=mon_processus, args=(name,))
12     p.start()
```

Python - Modules - Multiprocessing

- Remarque : Les différents processus sont en concurrence et l'affichage n'est pas tout le temps correct. Il faut donc pour éviter ce problème empêcher les processus d'afficher en même temps.
- La solution : Créer un verrou.

```
1  def mon_processus( name, verrou ):
2    verrou.acquire()
3    info( "processus["+name+"]" )
4    print "Hello", name
5    verrou.release()
6  if __name__ == '__main__':
7    info( "main" )
8    verrou = Lock()
9    names = ["gwenael", "nicolas", "francois", "paul"]
10   for name in names:
11     p = Process(target=mon_processus, args=(name,verrou))
12     p.start()
```

Python - Modules - Multiprocessing

- Nous pouvons avoir de l'échange de données entre les différents processus.
  - Avec l'usage des Queues (FIFO d'objet)
  - Avec l'usage des Pipes communication par socket
  - Avec l'usage d'objet partagé, deux types possible Value et Array.

Python - Modules - Expressions Régulières (RE)

- **Exemple 1** :

```
>>> s = "Bonjour à tous"
>>> "tous" in s
True
```

- **Exemple 2** :

```
>>> import re
>>> if re.search("Python", "On est en formation Python"):
        print " le motif est trouvé :)"
    else:
        print "pas de motif :s"
```

Éléments de syntaxe des méta-caractères :

Opérateur	Description
^	début de chaîne de caractères ou de ligne
$	fin de chaîne de caractères ou de ligne
.	n'importe quel caractère (mais un caractère quand même)
[ABC]	le caractère A ou B ou C (un seul caractère)
[A-Z]	n'importe quelle lettre majuscule
[a-z]	n'importe quelle lettre minuscule
[0-9]	n'importe quel chiffre
[A-Za-z0-9]	n'importe quel caractère alphanumérique
[^AB]	n'importe quel caractère sauf A et B
\	caractère d'échappement (pour protéger certains caractères)
*	0 à n fois le caractère précédent ou l'expression entre parenthèses précédente $\Rightarrow \divideontimes$
+	1 à n fois le caractère précédent ($\divideontimes$)
?	0 à 1 fois le caractère précédent ($\divideontimes$)
{n}	n fois le caractère précédent ($\divideontimes$)
{n,m}	n à m fois le caractère précédent ($\divideontimes$)
{n,}	au moins n fois le caractère précédent ($\divideontimes$)
{,m}	au plus m fois le caractère précédent ($\divideontimes$)
(CG|TT)	chaînes de caractères CG ou TT

Python - Modules - RE

re.search() et re.match()

- search() : permet de rechercher un motif (pattern) au sein d'une chaîne de caractères avec une syntaxe de la forme search(motif, chaine).
  - Si le motif est trouvé Python renvoie une instance MatchObject.
- match() : ressemble à re.search sauf quelle permet de rechercher l'expression régulière qui correspond (match) au début de la chaîne (à partir du premier caractère).

- **Exemple** :

```
1 >>> maPhrase = "Je suis en formation"
2 >>> re.search('en', maPhrase)
3 <_sre.SRE_Match object at 0x1660440>
4 >>> re.match('en', maPhrase)
```

Python - Modules - RE

re.compile()

- compile() : permet de compiler l'expression régulière et renvoie un objet de type expression régulière.

- **Exemple 1** :

```
>>> regex = re.compile("^Je")
>>> regex
<_sre.SRE_Pattern object at 0x16aff80>
```

- **Exemple 2** :

```
>>> maPhrase = "Salut. Je suis en formation"
>>> regex.search(maPhrase)
>>> maPhrase = "Je suis en formation"
>>> regex.search(maPhrase)
<_sre.SRE_Match object at 0x1660440>
```

Python - Modules - RE

Les groupes

- groupe() : permet de récupérer des parties du motif trouvé grâce à l'utilisation des parenthèses.
- groupe(i) : permet de récupérer la i^{eme} partie du motif trouvé.

- **Exemple** :

```
>>> regex = re.compile('([0-9]+)\.([0-9]+)')
>>> resultat = regex.search("pi vaut 3.14")
>>> resultat.group(0)
'3.14'
>>> resultat.group(1)
'3'
>>> resultat.group(2)
'14'
```

Python - Modules - RE

start() et end()

Il est possible de retrouver le positionnement du motif trouvé grâce au méthodes start() et end()

- **Exemple** :

```
>>> regex = re.compile('([0-9]+)\.([0-9]+)')
>>> resultat = regex.search("pi vaut 3.14")
>>> resultat.group(0)
'3.14'
>>> resultat.group(1)
'3'
>>> resultat.group(2)
'14'
>>> resultat.start()
8
>>> resultat.end()
12
```

Remarque

La méthode search() ne renvoie que la première zone qui correspond à l'expression régulière, même s'il en existe plusieurs.

- **Exemple** :

```
1 >>> resultat = regex.search("pi vaut 3.14 et e vaut 2.72")
2 >>> resultat.group(0)
3 '3.14'
```

Python - Modules - RE

findall()

Afin de récupérer les différentes occurrences d'un mofit on utilise la méthode findall()

- **Exemple** :

```
>>> regex = re.compile('[0-9]+\.[0-9]+')
>>> resultat = regex.findall("pi vaut 3.14
          et e vaut 2.72")
>>> resultat
['3.14', '2.72']
>>> regex = re.compile('([0-9]+)\.([0-9]+)')
>>> resultat = regex.findall("pi vaut 3.14
          et e vaut 2.72")
>>> resultat
[('3', '14'), ('2', '72')]
```

Python - Modules - RE

sub()

Il est possible de faire des substitutions (remplacements) de motifs grâce à la méthode sub().

- Pour délimiter le nombre de motifs a remplacer on utilise count(n).

- **Exemple** :

```
1 >>> regex.sub('quelque chose',"pi vaut 3.14 et e vaut 2.72"
      )
2 'pi vaut quelque chose et e vaut quelque chose'
3 >>> regex.sub('quelque chose',"pi vaut 3.14  et e vaut 2.72
      ", count=1)
4 'pi vaut quelque chose et e vaut 2.72'
```

Exercice - Modules - RE

- Écrire un le programme python avec l'expression régulière qui retrouve les tags d'un fichier HTML.
- Écrire un le programme python avec l'expression régulière qui récupéré le texte des tags HTML
- Écrire un le programme python avec l'expression régulière qui permute le 1er et le dernier caractère d'un mot
- Écrire un script python qui demande aux utilisateurs leur nom, adresse et numéro de téléphone. Testez chaque entrée pour la précision, par exemple, il ne faut pas qu'il y ai de lettres dans un numéro de téléphone. Un numéro de téléphone doit avoir une certaine longueur. Une adresse doit avoir un certain format, etc Demandez à l'utilisateur de répéter la saisie quand elle est incorrect.

Sommaire

Python - Qualité du code

- Python - débogueur
- Nous avons la possibilité d'utiliser le débogueur Eclipse
- Ou de lancer le débogueur python directement dans une console avec la commande suivante :

```
python -m pdb fichier.py
```

- Nous pouvons alors lister le fichier, ajouter des points d'arrêt, afficher une variable, exécuter le code pas-à-pas.

Python - Qualité du code - pdb (1/2)

- En ligne de commande :
  - python –m pdb monfichier.py
- Depuis une IDE
  - pdb.run('monModule.maFonc()')
- Directement dans le script
  - Inclure pdb.set_trace()
- Plus d'information sur
  `http://docs.python.org/library/pdb.html`

Python - Qualité du code - pdb (2/2)

- Quelques commande du pdb :

n	next, exécute une ligne
q	quit, quitte le programme
p	print, affiche la valeur de la variable souhaitée
c	continue, relance l'exécution du programme
l	list, affiche les 11 lignes du code qui sont sur le point d'être exécutées
s	step into, rentre dans les fonctions
r	return, exécute le code jusqu'à la sortie de la fonction courante
b	breakpoint, définit des points d'arrêt

Python - Qualité du code - Pylint

Pylint est un outils d'analyse de code pour améliorer la lisibilité du code en donnant une note sur 10 en fonction de critère.
Installation :

- 1er méthode :
  - Installer « logilab-astroid » et « logilab-common » packages
  - Installer pylint
  - Utiliser la commande : python setup.py install
  - `http://www.logilab.org/card/pylint_manual#installation`
- 2eme méthode :
  - Installer setuptools depuis : `http://pypi.python.org/pypi/setuptools`
  - utiliser easy_install : easy_install pylint (`http://packages.python.org/distribute/easy_install.html`)
- Utilisation :

pylint <option> fichier.py

Python - Qualité du code - Pylint

Pylint et le fichier de sortie (the output). Il y a 5 types de messages :

- (C) convention, violation des standards de programmation
- (R) refactor, mauvaise utilisation du code
- (W) warning, problems spécifique a python
- (E) error, dû à des bugs dans le code
- (F) fatal, une erreur qui a causé l'arret de pylint
- Tableau de rapport sur le code

Python - Qualité du code - Pylint

Pylint et les commandes optionnelles :

- Master : –rcfile=<file>
- Commands : –help-msg=<msg-id>
  Exemple : pylint –help-msg=C0111
- Message control : –disable=<msg-ids>
  Exemple : pylint –reports=n –include-ids=y –disable=W0402 fichier.py
- Reports : –files-output=<y_or_n>
  –reports=<y_or_n>
  Exemple : pylint –reports=n fichier.py
  –include-ids=<y_or_n>
  Exemple : pylint –reports=n –include-ids=y fichier.py
  –output-format=<format>
- Documentation :
  `http://www.logilab.org/card/pylint_tutorial`

Python - Qualité du code - pydoc

- Comment définir de la documentation dans mon code Python ?
  Utilisation des " " " Texte de documentation " " "
- Quel module utiliser
  Le module **pydoc**
- Comment générer la documentation dans un fichier HTML ?
  python -m pydoc -w nomRepertoireOuFichierSansExtention
- Il existe d'autres générateurs de documentation tel que : *Doxygen* ou *Sphinx*

Python - Qualité du code - Test Unitaire

Pourquoi écrire des test unitaires ?

- S'assurer que chaque fonctionnalité créée respecte un comportement défini
- Eviter de passer un temps important à chercher des bugs
- Les tests montrent un code exemple pour les autres développeurs (facilitent le passage de relais entre développeurs)

Python - Qualité du code - Test Unitaire

Il existe dans Python plusieurs modules qui permettent de faire de la vérification par tests unitaires.

- Doctest
- Unittest
- Nose

La vérification de code par tests unitaires se fait sur 3 niveaux.

Python - Qualité du code - Test Unitaire

- 1er niveau :
  Test de petites portions de code (méthodes / fonctions). Cela permet de s'assurer que chaque brique du programme fonctionne.

- 2ème niveau :
  Test d'interaction entre des parties (test lançant plusieurs tests de niveau 1)
- 3ème niveau :
  Test global qui répond à une problématique métier (authentification utilisateur)

Python - Qualité du code - Test Unitaire

Pour le premier niveau, nous pouvons utiliser "doctest".

doctest recherche dans le code source des commentaires ressemblant à une exécution de Python en mode interactif.

- Il exécute ces parties et teste le résultat
- Par défaut : doctest affiche la sortie du test que si ce dernier a échoué

Python - Qualité du code - Test Unitaire

- Utiliser le package doctest
- Utiliser la méthode de test :
  doctest.testmod()
- Plus d'information avec :
  help(doctest)
- Documentation :
  `http://docs.python.org/library/doctest.html`

Python - Qualité du code - Test Unitaire

- Exemple :

```
1  # Nom du fichier : etape1.py
2  def addition(a, b):
3    """
4    C'est une fonction qui fait l'addition de deux entiers
5    >>> addition(4, 5)
6    9
7    """
8    return a + b
9
10 if __name__ == "__main__":
11   import doctest
12   doctest.testmod()
```

Python - Qualité du code - Test Unitaire

Nous pouvons aussi utiliser le module unittest.

Ce module supporte :

- Les tests automatiques
- Des fonctions d'initialisation et de finalisation de test (connexion à une base de données)
- L'agrégation de tests
- Indépendance des tests dans le rapport final

Il est utilisé le plus souvent pour les tests de niveau deux et trois.

Python - Qualité du code - Test Unitaire

Pour utiliser le module unittest :

- import unittest
- Les tests font partis d'une classe héritée de unittest.TestCase
- Le nom des méthodes doivent commencer par « test »
- Les tests sont exécutés dans l'ordre alphabétique
- setUp est une méthode appelée au début de chaque test
- tearDown à la fin

Python - Qualité du code - Test Unitaire

- Exemple :

```
1  # Nom du  fichier : etape2.py
2  import unittest
3  from etape1 import *
4
5  class TestAddition(unittest.TestCase):
6    def setUp(self):
7      self.a = 5.
8      self.b = 6.
9    def test001_Addition(self):
10     self.assertEqual( addition(self.a, self.b), self.a +
           self.b)
11
12 if __name__ == "__main__" :
13   unittest.main()
```

Python - Qualité du code - Test Unitaire

- Exemple :

```
1  # Nom du fichier etape3.py
2  import unittest
3  from etape1 import *
4
5  class TestDivision(unittest.TestCase):
6    def setUp(self):
7      self.a = 5.
8      self.b = 6.
9    def test001_Division(self):
10     self.assertEqual( division(self.a, self.b), self.a /
           self.b)
11   def test002_Division(self):
12     self.assertRaises( ZeroDivisionError, division, self.a,
           0 )
13   def test003_DivisionFull(self):
14     self.assertAlmostEqual(division(1., 3),0.3333, 4)
15
16 if __name__ == "__main__" :
17   unittest.main()
```

Python - Qualité du code - Test Unitaire

- Exemple d'agrégation de tests

```
1  import unittest
2
3  def TestProgram():
4    from etape2 import TestAddition
5    from etape3 import TestDivision
6
7    suite = unittest.TestSuite()
8    suite.addTest(unittest.makeSuite(TestAddition))
9    suite.addTest(unittest.makeSuite(TestDivision))
10
11   return suite
12
13 if __name__ == "__main__" :
14   unittest.TextTestRunner(verbosity=2).run(TestProgram())
```

Python - Qualité du code - Test Unitaire

- nosetests est un module qui étend unittest et doctest
- Il permet de lancer rapidement des tests en détectant automatiquement les doctests et unittests
- Il permet aussi de faire des tests de couverture de code, ou de profiling

Name	Stmts	Miss	Cover	Missing
etape1	7	2	71%	27-28
etape2	10	1	90%	15
etape3	14	1	93%	21
etape4	10	1	90%	16
TOTAL	41	5	88%	

Sommaire

Python - Interface graphique

- Python fourni en standard un module d'interfaces graphiques appelé TkInter.
- Il est possible d'utiliser des modules tiers pour l'utilisation de bibliothèque graphique différente, comme GTK, Qt ou wxPython.

Python - Interface graphique

Concepts universels et jargon

- **Widget** (window gadget), on dit aussi composant graphique
- **L'interface d'une application** : une fenêtre ou cadre d'application contenant d'autres widgets.
- **D'autres cadres** de niveau supérieur (i.e. non incluses dans le cadre d'application) apparaissent et disparaissent : les boîtes de dialogue
- **Les composants** sont cibles d'actions de l'utilisateur (souris, clavier) ces actions du monde extérieur sont représentées dans le programme par des événements :
  - Événements simples. Ex. : clic avec la souris
  - Événements élaborés, ou commandes. Ex. : choix d'un item dans un menu

Python - Interface graphique

- Première application graphique : l'affichage d'une fenêtre et d'un message.

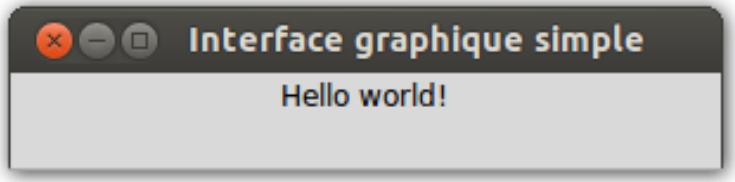


FIGURE : Mon hello world !

  - Fenêtre simple :

```
1 from Tkinter import *
2 window = Tk()
3 window.title("Une interface graphique simple")
4 window.minsize(300, 40)
5 widget = Label(window, text="Hello world!")
6 widget.pack()
7 window.mainloop()
```

Python - Interface graphique

- Il existe une panoplie de Widgets :

Button	Un bouton classique, à utiliser pour provoquer l'exécution d'une commande.
Canvas	Un espace pour disposer divers éléments graphiques
Checkbutton	Une case à cocher qui peut prendre deux états distincts.
Entry	Un champ d'entrée, dans lequel l'utilisateur du programme pourra insérer un texte quelconque.
Frame	Une surface rectangulaire dans la fenêtre, où l'on peut disposer d'autres widgets..
Label	Un texte.
Listbox	Une liste de choix proposés à l'utilisateur, généralement présentés dans une sorte de boîte..
Menu	Un menu(déroulant ou pop up).
Radiobutton	Représente (par un point noir dans un petit cercle) une des valeurs d'une variable qui peut en posséder plusieurs.

Python - Interface graphique

- Il existe une panoplie de Widgets (suite) :

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.
...	...

Python - Interface graphique

- On va Ajouter un bouton "Ok" :

```
1 from Tkinter import *
2
3 window = Tk()
4 window.title("Interface graphique simple")
5 window.minsize(300, 40)
6 widget = Label(window, text="Hello world!")
7 widget.pack()
8 Button(window, text="Ok", bg='gray65').pack(side=BOTTOM)
9 window.mainloop()
```

Python - Interface graphique

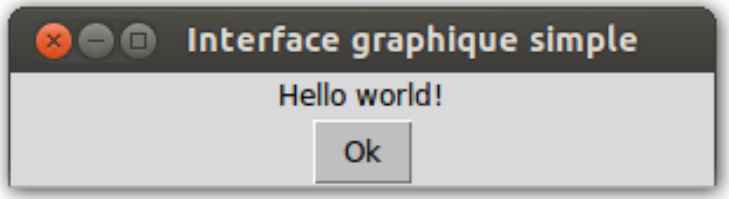


FIGURE : Mon hello world avec un bouton.

- Quand on veut écrire des programmes conséquents, la bonne idée est d'utiliser une ou plusieurs classes.

Python - Interface graphique

- Adaptons notre 'Hello world' et ajoutons deux boutons :

FIGURE : Mon hello world avec deux boutons.

Python - Interface graphique

```
1  class MyApp:
2      def __init__(self, window):
3          frame = Frame(window)
4          frame.pack()
5          self.button_quit = Button(frame, text="Quitter", fg="
               red", command=frame.quit )
6          self.button_quit.pack(side=LEFT)
7          self.button_hello = Button(frame, text="Hello", fg="
               blue", command=self.say_hello )
8          self.button_hello.pack()
9      def say_hello(self):
10         print "Hello world !"
11 if __name__ == "__main__":
12     window = Tk()
13     window.title("Interface graphique simple")
14     window.minsize(300, 40)
15     app = MyApp(window)
16     window.mainloop()
```

Python - Interface graphique

- toto
- Utiliser un champ de saisie de texte :

```
1  from Tkinter import *
2
3  def fetch():
4      print "Texte: < %s > " %ent.get()
5
6  root = Tk()
7  root.title("Gestion des évènements")
8  root.minsize(300, 40)
9  ent = Entry(root)
10 ent.pack(side=TOP, fill =X)
11 ent.bind("<Return>", (lambda event : fetch()))
12 b1 = Button(root, text= "Fetch", command=fetch)
13 b1.pack(side=LEFT)
14 b2 = Button(root, text="Quit", command= root.destroy)
15 b2.pack(side=RIGHT)
16 root.mainloop()
```

Python - Interface graphique

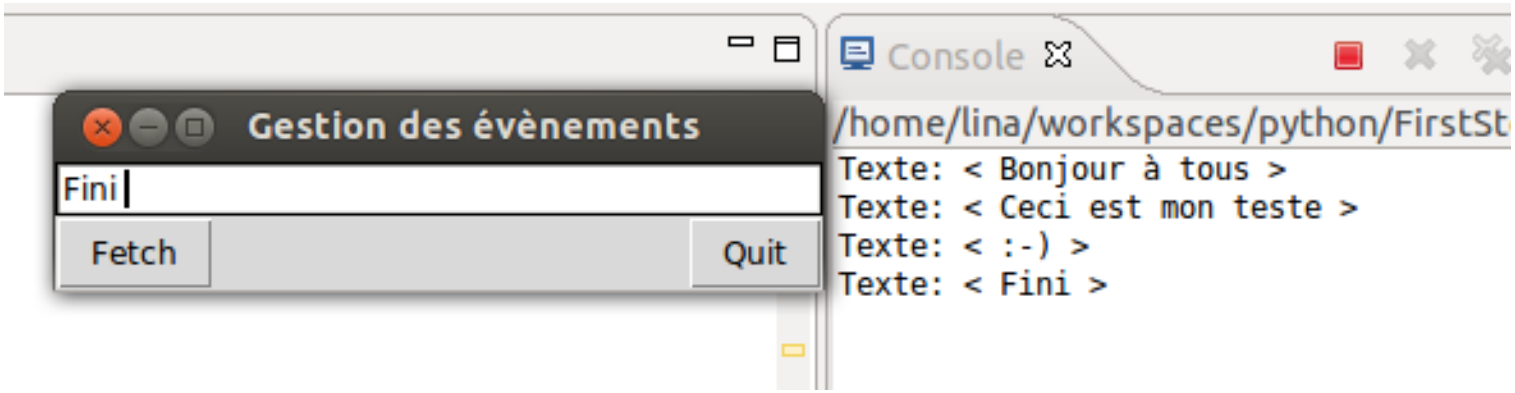


FIGURE : Ma saisie de texte.

Exercice - Interface graphique

Application qui récupère l'identifiant et le mot de passe d'une personne

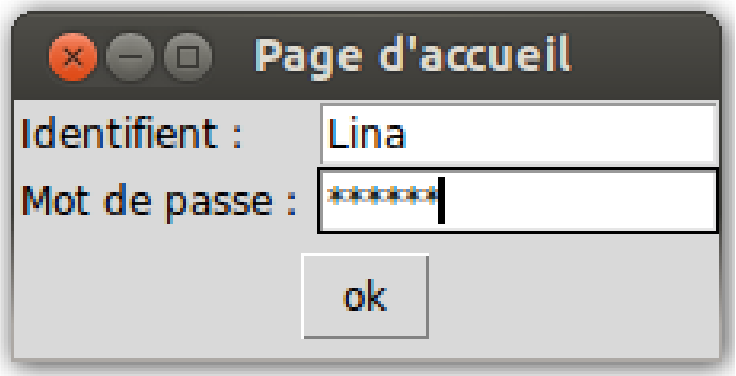


FIGURE : Interface Login.

Python - Interface graphique

- Source pour l'application Login :

```
1  def affiche():
2      print "affichage : "
3      print "id : ", e1.get()
4      print "mp : ",e2.get()
5      e1.delete(0, END)
6      e2.delete(0, END)
7      e1.focus()
8
9  root = Tk()
10 root.title("Page d'accueil ")
11 Label(root, text="Identifient : ").grid(row=0, sticky=W)
12 Label(root, text="Mot de passe : ").grid(row=1, sticky=W)
```

Python - Interface graphique

- Source pour l'application Login (suite) :

```
1
2 e1 = Entry(root, width=15)
3 e2 = Entry(root, show="*", width=15)
4 e1.grid(row=0, column=1)
5 e2.grid(row=1, column=1)
6
7 b = Button(root, text = "ok", command=affiche)
8 b.grid(row=2,columnspan=2, pady=5 )
9 root.mainloop()
```

Exercice - Interface graphique

On modifie notre application Login pour lui ajouter une liste pour les identifiants.

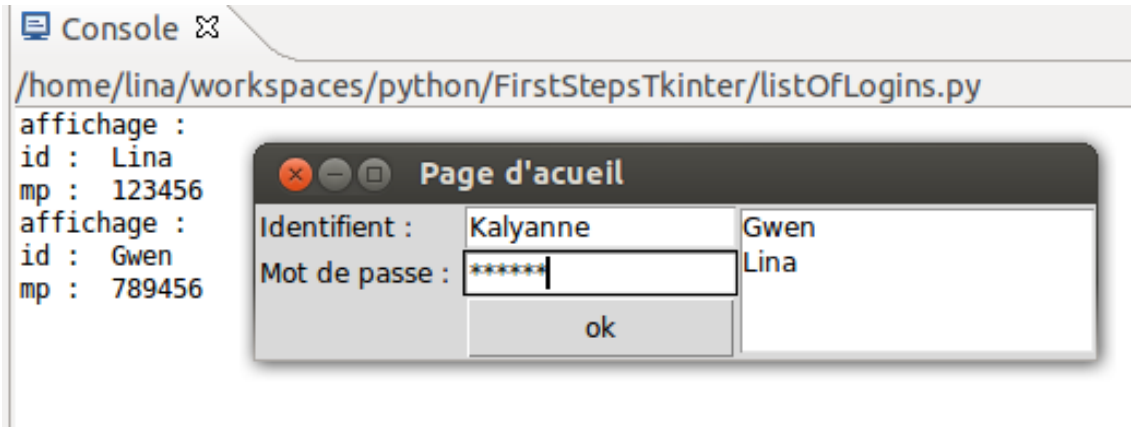


FIGURE : Interface Login avec une liste des identifiants.

Python - Interface graphique

- Source pour l'application Login avec la liste des identifiants :

```
1  def affiche():
2      print "affichage : "
3      print "id : ", e1.get()
4      maliste.insert(0,e1.get())
5      print "mp : ",e2.get()
6      e1.delete(0, END)
7      e2.delete(0, END)
8      e1.focus()
9
10 root = Tk()
11 root.title("Page d'acueil ")
12 #root.minsize(300, 40)
13 Label(root, text="Identifient : ").grid(row=0, sticky=W)
14 Label(root, text="Mot de passe : ").grid(row=1, sticky=W)
```

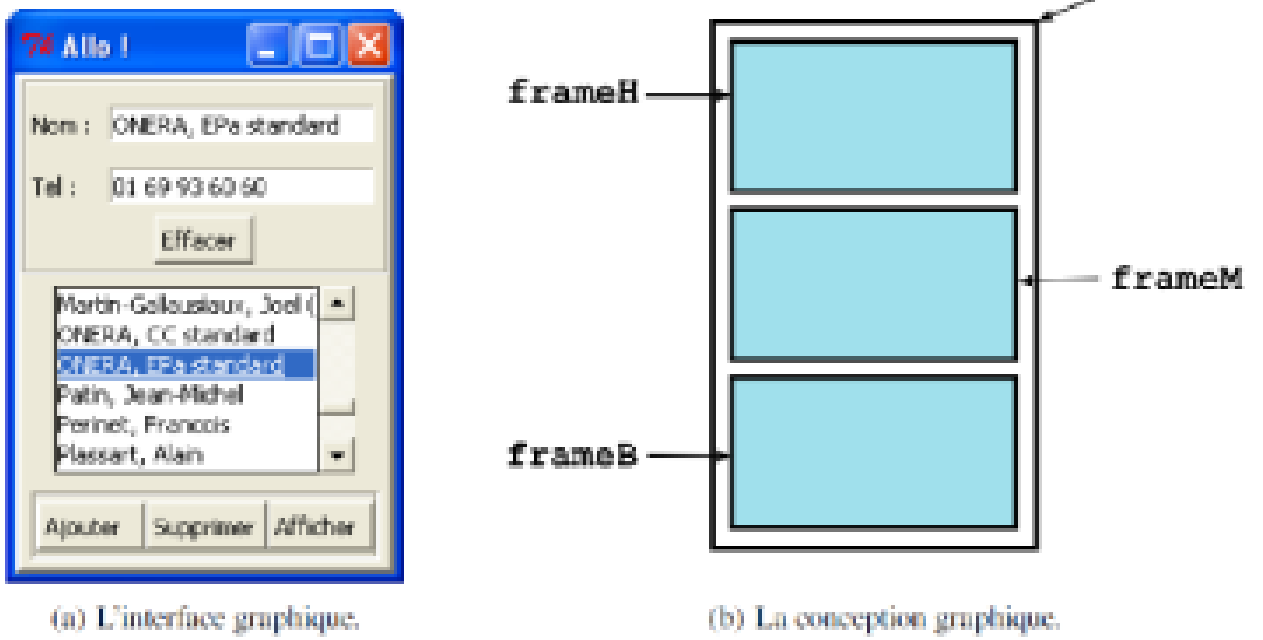
Python - Interface graphique

- Source pour l'application Login avec la liste des identifiants (suite) :

```
1  e1 = Entry(root, width=15)
2  e2 = Entry(root, show="*", width=15)
3
4  e1.grid(row=0, column=1)
5  e2.grid(row=1, column=1)
6
7  maliste= Listbox(root, height=4)
8  maliste.grid(row =0,rowspan=3, column=3)
9  b = Button(root, text = "ok", command=affiche, width=12)
10 b.grid(row=2,column=1 )
11
12 root.mainloop()
```

Exercice - Interface graphique

Il s'agit de créer un script de gestion d'un carnet téléphonique.
L'aspect de l'application est illustré dans la figure suivante :



(a) L'interface graphique. (b) La conception graphique.

FIGURE : Interface voulue.

Sommaire

Python - Extension avec C

Objectif : Appeler du code C depuis Python.

Il existe trois possibilités pour créer un module C pour Python.

1. Chargement de bibliothèques dynamiques,
2. Ecrire un module avec l'API Python/C,
3. Utiliser SWIG (Simplified Wrapper and Interface Generator).

Python - Extension avec C

1. **Chargement de bibliothèques dynamiques** :

   ctypes est une bibliothèque de fonctions étrangère pour Python.

   Elle fournit des types de données compatibles C, et permet l'appel de fonctions dans les DLL ou bibliothèques partagées.

   - On charge des des bibliothèques en y accédant comme des attributs de ces objets.
   - cdll, pydll : avec la convention 'cdecl' sur toutes les plates-formes
   - windll, oledll : avec la convention 'stdcall' sur Windows

Python - Extension avec C

- Exemple d'utilisation de bibliothèque :

```
1 from ctypes import *
2 # exemple de chargement sous Unix
3 libc = cdll.LoadLibrary("libc.so.6")
4 # ou bien
5 libc = CDLL("lib.so.6")
6 # exemple de chargement sous Windows
7 libc = windll.LoadLibrary("msvcrt")
8 # ou bien
9 libc = WinDLL("msvcrt")
```

- Appel de fonction :
  - On appelle les fonctions comme des fonctions Python
  - Exemple :

```
1 >>> libc = cdll.LoadLibrary("libc.so.6")
2 >>> libc.time()
```

Python - Extension avec C

Types de données ctype (1/2) :

ctypes	c	ctypes	c
c_char	char	c_ulonglong	unsigned long long
c_wchar	wchar_t	c_float	float
c_byte	char	c_double	double
long	unsigned char	c_longdouble	long double
c_short	short	c_char_p	char *
c_ushort	unsigned short	c_wchar_p	wchar_t *
c_int	int	c_void_p	void *
c_uint	unsigned int	c_long	long
c_ulong	unsigned long	c_longlong	long long (64bits)

Python - Extension avec C

Types de données ctype (2/2) :

- Exemple d'affectation de variables ctype et changement de sa valeur :

```
1  i = c_int(15)
2  print(i)
3  # c_int(15)
4  print(i.value)  # 15
5  i.value = 32
6  print(i.value)  # 32
7  c_s = c_char_p("toto")
8  print c_s.value # "toto"
9
10 # utilisation de la fonction pointer
11 i = c_int(15)
12 pi = pointer(i)
13 print(pi.contents)  # c_int(15)
14 print(pi.contents.value)  # 15
```

2. **Module avec l'API Python/C** :

- A l'installation de Python, est fournie une librairie "Python.h" qui sera appelée à la compilation de notre code C automatiquement.
- Pour créer notre module en C : le fichier demo.c
- On aura besoin de 4 parties :
  1. On inclut la librairie "Python.h"
  2. On créé le module "demo", qui sera ajouté au namespace Python pour la session en cours
  3. On définit un tableau de méthodes permettant de lister les signatures de méthodes (noms, entrées, sorties)
  4. On écrit la méthode demo_foo qui pourra être appelée par notre code Python via la fonction "foo()"

Python - Extension avec C

- Exemple :

```
1  #include "Python.h"
2  void initdemo(void){
3  PyImport_AddModule("demo");
4  // ajoute le module demo  dans le namespace Python
5  Py_InitModule("demo", demo_methods); }
6  // notre fonction (42 puissance 42)
7  static PyObject* demo_foo(PyObject *self, PyObject* args){
8  PyObject* a = PyInt_FromLong(42L);
9  PyObject* b = PyInt_FromLong(42L);
10 return PyNumber_Power(a, b, Py_None);}
11 // 4 arguments à définir pour chaque fonction
12 static PyMethodDef demo_methods[] = {
13 { "foo", // nom de la fonction appelable
14 demo_foo, // fonction c associée
15 METH_NOARGS, // arguments
16 "Return the meaning of everything." // commentaire pour
       help
17 },
18 { NULL, NULL} };
```

Python - Extension avec C

- Nous devons utiliser le système d'installeur Python qui se chargera de compiler le module.

```
$ python setup.py build
```

- où le setup.py est définie comme suit :

```
from distutils.core import setup, Extension
DemoModule = Extension('demo', sources = ['demo.c'])
setup( name = "Demonstration d'extension C",
version = '1.0',
description = 'Demonstration d\'extension C',
ext_modules = [DemoModule]
)
```

Python - Extension avec C

Utilisation :

- Aller dans le répertoire build/lib.linux - .../

```
>>> import demo
>>> demo.foo()
# 150130937545296572356771972164254457814047970
    568738777235893533016064L
>>> 42L ** 42L
# 150130937545296572356771972164254457814047970
    568738777235 893533016064L
>>> help(demo)
```

Python - Extension avec C

3. Simplified Wrapper and Interface Generator (SWIG) :

- Créer un fichier example.h :

```
1 int fact(int n);
```

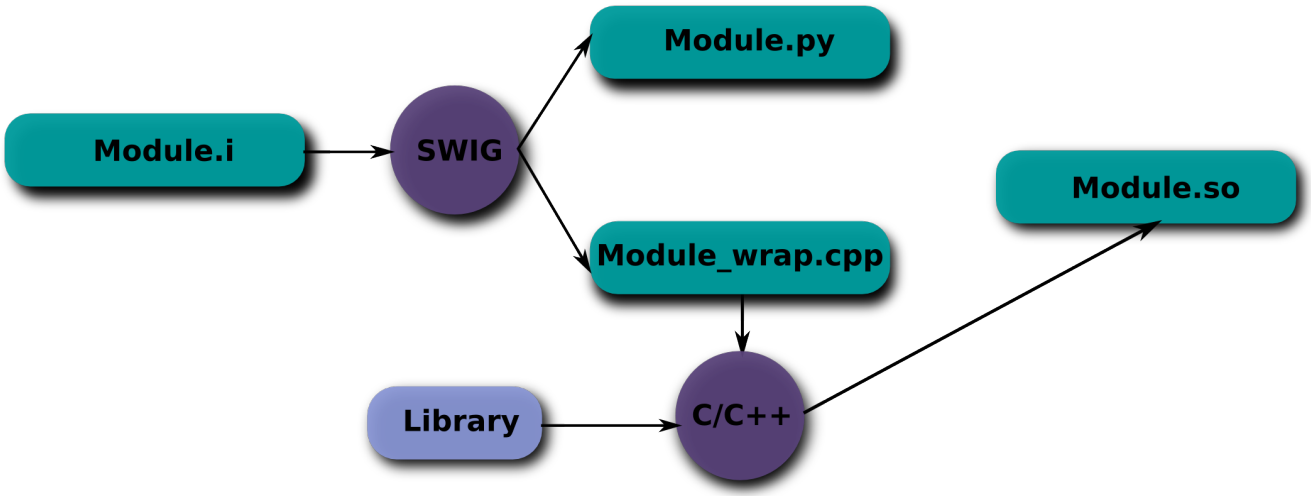
- Créer un fichier example.c

```
1 #include "example.h"
2 int fact(int n) {
3 if (n < 0){ return 0; }
4 if (n == 0) { return 1; }
5 else { return n * fact(n-1); }
6 }
```

Python - Extension avec C

- Créer un fichier example.i (extension de swig)

```
1 %module example
2 %{
3 #define SWIG_FILE_WITH_INIT
4 #include "example.h"
5 %}
6 #define A 6.56456
7 int fact(int n);
```



Python - Extension avec C

- Nous devons utiliser 'swig' pour créer le code C du wrapper

```
swig -python example.i
```

- Création du setup.py :

```
from distutils.core import setup, Extension
example_module = Extension('_example', sources=['
    example_wrap.c', 'example.c'], )
setup (name = 'example',
version = '0.1',
author = "Matelli",
description = """Exemple creation d'un module""",
ext_modules = [example_module],
py_modules = ["example"],
)
```

Python - Extension avec C

- Nous lançons l'installeur qui va compiler notre module (utilisation de GCC implicite)

```
python setup.py build
```

- Utilisation de la fonction : Aller dans le repertoire build/... (y ouvrir un interpréteur python)

```
>>> import _example
>>> _example.fact(5)
# 120
```

Python - Interpréteur Python dans C

- La première étape est de créer un programme en C qui appelle une fonction d'un module python. Le programme en C utilise l'API Python/C.

```c
/* Fichier call_function.c -
Exemple d'appel d'une fonction python depuis du C
*/
#include <Python.h>
int main(int argc, char *argv[])
{
/* Déclaration de tous les objets Python utilisés */
PyObject *pName, *pModule, *pDict, *pFunc, *pValue;
if (argc < 3)
{
printf("Usage: exe_name python_source function_name\n");
return 1;
}
```

Python - Interpréteur Python dans C

- (suite)

```
1  /* Initialisation de l'interpréteur python */
2  Py_Initialize();
3  /* Passage des arguments du C vers l'interpréteur */
4  PySys_SetArgv(argc, argv);
5  /* Création d'une chaine de caractères
6  du nom de fichier que l'on va importer */
7  pName = PyString_FromString(argv[1]);
8  /* Importation du module */
9  pModule = PyImport_Import(pName);
10 /* Récupération du dictionnaire regroupant les informations
11 du module (variable, fonction, classe, ...) */
12 pDict = PyModule_GetDict(pModule);
```

Python - Interpréteur Python dans C

- (suite)

```
1  /* Récupération d'une référence, en temps normal c'est une
2  fonction que l'on souhaite appeler */
3  pFunc = PyDict_GetItemString(pDict, argv[2]);
4  /* Nous testons si la référence est appelable */
5  if (PyCallable_Check(pFunc))
6  {
7  /* Nous appelons la fonction et récupérons sont retour */
8  pValue = PyObject_CallObject(pFunc, NULL);
9  } else {
10 PyErr_Print();
11 }
12 /* affichage en C du retour de la fonction */
13 printf("retour : %ld\n", PyInt_AsLong(pValue));
```

Python - Interpréteur Python dans C

- (suite)

```
1 /* Dé-référencement des objets pModule et pName */
2 Py_DECREF(pModule);
3 Py_DECREF(pName);
4 /* Fermeture de l'interpréteur Python */
5 Py_Finalize();
6 return 0;
7 }
```

Python - Interpréteur Python dans C

- Voici la phase de compilation du code C (version 2.7) :

  gcc -I/usr/include/python2.7 -lpython2.7 -lpthread -lm -g -finline-functions call_function.c -o call_function

  - -I/usr/include/python2.7 : pour avoir Python.h
  - -lpython2.7 : librairie python que sera liée
  - -lpthread : librairie standard POSIX pour les processus léger
  - -lm : librairie mathématique
  - -g : Information pour Debug
  - -finline-functions : intégration des fonctions simple lors de leur appel.

Python - Interpréteur Python dans C

- Il faut créer le module Python (fichier.py) qui sera appelé par notre programme en C .

```
1 def calc():
2   a = 1234*3
3   print "calcul 1234*3 fait", a
4   return a
```

- Nous pouvons maintenant tester notre appel en C.

```
1 $ call_function fichier calc
2 calcul 1234*3 fait 3702
3 retour : 3702
```

Sommaire

Conclusion

- Il est possible de transformer le python 2 en python 3. Pour en savoir plus :
  `http://docs.python.org/2/library/2to3.html#to3-reference`

Conclusion

Merci pour votre attention.