

Notes de cours Python

2008–2009 - v 2.1

Bob CORDEAU
robert.cordeau@u-psud.fr

Mesures Physiques
IUT d'Orsay
www.iut-orsay.fr/dptmphy/Pedagogie/Welcome.html

26 août 2008



Ces notes reposent sur deux partis pris :

- le choix du langage Python ;
- le choix de logiciels libres :
 - Python (www.python.org)
 - Wing IDE (www.wingware.com)
 - gnuplot (sourceforge.net/projects/gnuplot)
 - ...

Ces notes reposent sur deux partis pris :

- le choix du langage Python ;
- le choix de logiciels libres :
 - Python (www.python.org)
 - Wing IDE (www.wingware.com)
 - gnuplot (sourceforge.net/projects/gnuplot)
 - ...

Remerciements :

Cette version a bénéficié des lectures attentives de Laurent POINTAL (LIMSI), Georges VINCENTS (IUT d'Orsay, M.P.) et Laurent GUITTET (LISI/ENSMA).

Sommaire du chapitre 1

- 1 Introduction à l'informatique
 - Environnement matériel
 - Environnement logiciel
 - Langages
 - Production des programmes
 - Méthodologie
 - Algorithme et programme
 - Importance des commentaires

Sommaire du chapitre 2

- 2 La « calculatrice » Python
 - Les types simples
 - Données, variables et affectation
 - Les entrées-sorties
 - Les séquences
 - Retour sur les références
 - Les dictionnaires
 - Les ensembles

Sommaire du chapitre 3

- 3 Le contrôle du flux d'instructions
 - Les instructions composées
 - Choisir
 - Opérateur ternaire
 - Boucler
 - Parcourir
 - Listes en compréhension
 - Ruptures de séquences

Sommaire du chapitre 4

- 4 Fonctions–Espaces de noms–Modules
 - Définition et exemple
 - Passage des arguments
 - Espaces de noms
 - Modules

Sommaire du chapitre 5

- 5 Les fichiers
 - Ouverture – Fermeture
 - Méthodes d'écriture
 - Méthodes de lecture
 - Le module « pickle »

Sommaire du chapitre 6

- 6 La programmation « OO »
 - Classes et instanciation d'objets
 - Méthodes
 - Méthodes spéciales
 - Héritage et polymorphisme
 - Notion de Conception Orientée Objet (COO)

Sommaire du chapitre 7

- 7 La programmation « OO » graphique
 - Programmes pilotés par des événements
 - Les bibliothèques graphiques
 - Les classes de widgets de Tkinter
 - Le positionnement des widgets
 - Un exemple

L'ordinateur

Définition

Automate déterministe à composants électroniques.

L'ordinateur

Définition

Automate déterministe à composants électroniques.

L'ordinateur comprend entre autres :

- un microprocesseur avec une UC (Unité de Contrôle), une UAL (Unité Arithmétique et Logique), une horloge, une mémoire cache rapide ;

L'ordinateur

Définition

Automate déterministe à composants électroniques.

L'ordinateur comprend entre autres :

- un microprocesseur avec une UC (Unité de Contrôle), une UAL (Unité Arithmétique et Logique), une horloge, une mémoire cache rapide ;
- de la mémoire vive (RAM), contenant les instructions et les données. La RAM est formée de cellules binaires (*bits*) organisées en mots de 8 bits (*octets*) ;

L'ordinateur

Définition

Automate déterministe à composants électroniques.

L'ordinateur comprend entre autres :

- un microprocesseur avec une UC (Unité de Contrôle), une UAL (Unité Arithmétique et Logique), une horloge, une mémoire cache rapide ;
- de la mémoire vive (RAM), contenant les instructions et les données. La RAM est formée de cellules binaires (*bits*) organisées en mots de 8 bits (*octets*) ;
- des périphériques : entrées/sorties, mémoires mortes (disque dur, CD-ROM...), réseau...

Deux sortes de programmes

- Le **système d'exploitation** : ensemble des programmes qui gèrent les ressources matérielles et logicielles ; il propose une aide au dialogue entre l'utilisateur et l'ordinateur : l'interface textuelle (interprète de commande) ou graphique (gestionnaire de fenêtres). Il est souvent multitâche et parfois multiutilisateur ;

Deux sortes de programmes

- Le **système d'exploitation** : ensemble des programmes qui gèrent les ressources matérielles et logicielles ; il propose une aide au dialogue entre l'utilisateur et l'ordinateur : l'interface textuelle (interprète de commande) ou graphique (gestionnaire de fenêtres). Il est souvent multitâche et parfois multiutilisateur ;
- les **programmes applicatifs** dédiés à des tâches particulières. Ils sont formés d'une série de commandes contenues dans un programme *source* écrit dans un langage « compris » par l'ordinateur.

Des langages de différents niveaux

- Chaque processeur possède un langage propre, directement exécutable : le **langage machine**. Il est formé de 0 et de 1 et n'est pas portable, mais c'est le seul que l'ordinateur « comprend » ;

Des langages de différents niveaux

- Chaque processeur possède un langage propre, directement exécutable : le **langage machine**. Il est formé de 0 et de 1 et n'est pas portable, mais c'est le seul que l'ordinateur « comprend » ;
- le **langage d'assemblage** est un codage alphanumérique du langage machine. Il est plus lisible que le langage machine, mais n'est toujours pas portable. On le traduit en langage machine par un *assembleur* ;

Des langages de différents niveaux

- Chaque processeur possède un langage propre, directement exécutable : le **langage machine**. Il est formé de 0 et de 1 et n'est pas portable, mais c'est le seul que l'ordinateur « comprend » ;
- le **langage d'assemblage** est un codage alphanumérique du langage machine. Il est plus lisible que le langage machine, mais n'est toujours pas portable. On le traduit en langage machine par un *assembleur* ;
- les **langages de haut niveau**. Souvent normalisés, ils permettent le portage d'une machine à l'autre. Ils sont traduits en langage machine par un *compilateur* ou un *interpréteur*.

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL...

Des centaines de langages ont été créés, mais l'industrie n'en utilise qu'une minorité.

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL...
- Années 60 (langages universels) : ALGOL, PL/1, PASCAL...

Des centaines de langages ont été créés, mais l'industrie n'en utilise qu'une minorité.

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL...
- Années 60 (langages universels) : ALGOL, PL/1, PASCAL...
- Années 70 (génie logiciel) : C, MODULA-2, ADA...

Des centaines de langages ont été créés, mais l'industrie n'en utilise qu'une minorité.

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL...
- Années 60 (langages universels) : ALGOL, PL/1, PASCAL...
- Années 70 (génie logiciel) : C, MODULA-2, ADA...
- Années 80–90 (programmation objet) : C++, LabView, Eiffel, Matlab...

Des centaines de langages ont été créés, mais l'industrie n'en utilise qu'une minorité.

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL...
- Années 60 (langages universels) : ALGOL, PL/1, PASCAL...
- Années 70 (génie logiciel) : C, MODULA-2, ADA...
- Années 80–90 (programmation objet) : C++, LabView, Eiffel, Matlab...
- Années 90–2000 (langages interprétés objet) : Java, Perl, tcl/Tk, Ruby, Python...

Des centaines de langages ont été créés, mais l'industrie n'en utilise qu'une minorité.

Deux techniques de production des programmes

- La **compilation** est la traduction du source en langage objet. Elle comprend au moins quatre phases (trois phases d'analyse — lexicale, syntaxique et sémantique — et une phase de production de code objet). Pour générer le langage machine il faut encore une phase particulière : l'**édition de liens**. La compilation est contraignante mais offre une grande vitesse d'exécution ;

Deux techniques de production des programmes

- La **compilation** est la traduction du source en langage objet. Elle comprend au moins quatre phases (trois phases d'analyse — lexicale, syntaxique et sémantique — et une phase de production de code objet). Pour générer le langage machine il faut encore une phase particulière : l'**édition de liens**. La compilation est contraignante mais offre une grande vitesse d'exécution ;
- dans la technique de l'**interprétation** chaque ligne du source analysé est traduite au fur et à mesure en instructions directement exécutées. Aucun programme objet n'est généré. Cette technique est très souple mais les codes générés sont peu performants : l'interpréteur doit être utilisé à chaque exécution. . .

Techniques de production de Python

- Technique mixte : l'**interprétation du bytecode compilé**. Bon compromis entre la facilité de développement et la rapidité d'exécution ;



Techniques de production de Python

- Technique mixte : l'**interprétation du bytecode compilé**. Bon compromis entre la facilité de développement et la rapidité d'exécution ;
- le *bytecode* (forme intermédiaire) est portable sur tout ordinateur muni de la **machine virtuelle Python**.



La construction des programmes : le génie logiciel

Plusieurs modèles sont envisageables, par exemple :

- la méthodologie **procédurale**. On emploie l'analyse descendante et remontante qui procède par raffinements successifs : on s'efforce de décomposer un problème complexe en sous-programmes plus simples. Ce modèle structure d'abord les actions ;

La construction des programmes : le génie logiciel

Plusieurs modèles sont envisageables, par exemple :

- la méthodologie **procédurale**. On emploie l'analyse descendante et remontante qui procède par raffinements successifs : on s'efforce de décomposer un problème complexe en sous-programmes plus simples. Ce modèle structure d'abord les actions ;
- la méthodologie **objet**. On conçoit des fabriques (*classes*) qui servent à produire des composants (*objets*) qui contiennent des données (*attributs*) et des actions (*méthodes*). Les classes dérivent (*héritage* et *polymorphisme*) de classes de base dans une construction hiérarchique.

La construction des programmes : le génie logiciel

Plusieurs modèles sont envisageables, par exemple :

- la méthodologie **procédurale**. On emploie l'analyse descendante et remontante qui procède par raffinements successifs : on s'efforce de décomposer un problème complexe en sous-programmes plus simples. Ce modèle structure d'abord les actions ;
- la méthodologie **objet**. On conçoit des fabriques (*classes*) qui servent à produire des composants (*objets*) qui contiennent des données (*attributs*) et des actions (*méthodes*). Les classes dérivent (*héritage* et *polymorphisme*) de classes de base dans une construction hiérarchique.

Python offre les *deux* techniques.

Programme et algorithme

Algorithme

Moyen d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions.

Programme et algorithme

Algorithme

Moyen d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions.

Donc, un algorithme se termine en un **temps fini**.

Programme et algorithme

Algorithme

Moyen d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions.

Donc, un algorithme se termine en un **temps fini**.

Programme

Un programme est la **traduction d'un algorithme** en un langage compilable ou interprétable par un ordinateur.

Programme et algorithme

Algorithme

Moyen d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions.

Donc, un algorithme se termine en un **temps fini**.

Programme

Un programme est la **traduction d'un algorithme** en un langage compilable ou interprétable par un ordinateur.

Il est souvent écrit en plusieurs parties, dont une qui *pilote* les autres, le **programme principal**.

La présentation des programmes

Un programme *source* est destiné à l'être humain. Pour en faciliter la lecture, il doit être judicieusement commenté.

La présentation des programmes

Un programme *source* est destiné à l'être humain. Pour en faciliter la lecture, il doit être judicieusement commenté.

La signification de parties non triviales doit être expliquée par un **commentaire**.

Un commentaire commence par le caractère `#` et s'étend jusqu'à la fin de la ligne :

```
#-----  
# Voici un commentaire  
#-----  
  
9 + 2 # En voici un autre
```

Les expressions booléennes

- Deux valeurs possibles : `False`, `True`.

Les expressions booléennes

- Deux valeurs possibles : `False`, `True`.
- Opérateurs de comparaison : `==`, `!=`, `>`, `>=`, `<`, `<=`

```
2 > 8           # False  
2 <= 8 < 15    # True
```

Les expressions booléennes

- Deux valeurs possibles : `False`, `True`.
- Opérateurs de comparaison : `==`, `!=`, `>`, `>=`, `<`, `<=`

```
2 > 8           # False
2 <= 8 < 15    # True
```

- Opérateurs logiques (concept de *shortcut*) : `not`, `or`, `and`

```
(3 == 3) or (9 > 24)    # True (dès le premier membre)
(9 > 24) and (3 == 3)   # False (dès le premier membre)
```

Les expressions booléennes

- Deux valeurs possibles : `False`, `True`.
- Opérateurs de comparaison : `==`, `!=`, `>`, `>=`, `<`, `<=`

```
2 > 8           # False
2 <= 8 < 15    # True
```

- Opérateurs logiques (concept de *shortcut*) : `not`, `or`, `and`

```
(3 == 3) or (9 > 24)    # True (dès le premier membre)
(9 > 24) and (3 == 3)  # False (dès le premier membre)
```

- Les opérateurs logiques et de comparaisons sont à valeurs dans `False`, `True`

Le type entier (1/2)

- Opérations arithmétiques

```
20 + 3      # 23
20 - 3      # 17
20 * 3      # 60
20 ** 3     # 8000
20 / 3      # 6 (division entière)
20 % 3      # 2 (modulo)
```

Le type entier (2/2)

- Les entiers longs (seulement limités par la RAM)

```
2 ** 40      # 1099511627776L
3 * 72L      # 216L
```

Le type entier (2/2)

- Les entiers longs (seulement limités par la RAM)

```
2 ** 40      # 1099511627776L
3 * 72L      # 216L
```

- Opérations sur les bases

```
07 + 01      # 8
oct(7+1)     # '010' (octal)
0x9 + 0x2    # 11
hex(9+2)     # '0xb' (hexadécimal)
int('10110', 2) # 22
```

Le type flottant

- Les flottants sont notés avec un « point décimal » ou en notation exponentielle :

2.718

3e8

6.023e23

Le type flottant

- Les flottants sont notés avec un « point décimal » ou en notation exponentielle :

```
2.718
```

```
3e8
```

```
6.023e23
```

- Ils supportent les mêmes opérations que les entiers, sauf :

```
20.0 / 3 # 6.666666666666667
```

```
20.0 // 3 # 6 (division entière forcée)
```

Le type flottant

- Les flottants sont notés avec un « point décimal » ou en notation exponentielle :

```
2.718  
3e8  
6.023e23
```

- Ils supportent les mêmes opérations que les entiers, sauf :

```
20.0 / 3 # 6.666666666666667  
20.0 // 3 # 6 (division entière forcée)
```

- L'import du module « math » autorise toutes les opérations mathématiques usuelles :

```
from math import sin, pi  
print sin(pi/4) # 0.70710678118654746
```

Le type complexe

- Les complexes sont écrits en notation cartésienne formée de deux flottants.

Le type complexe

- Les complexes sont écrits en notation cartésienne formée de deux flottants.
- La partie imaginaire est suffixée par **j** :

```
1j                # 1j
(2+3j) + (4-7j)   # (6-4j)
(9+5j).real       # 9.0
(9+5j).imag       # 5.0
(abs(3+4j))       # 5.0 : module
```

Les données et les variables

Variable

C'est un **nom** donné à une valeur.

Les données et les variables

Variable

C'est un **nom** donné à une valeur.

Informatiquement, c'est une **référence** à une adresse mémoire.

Les données et les variables

Variable

C'est un **nom** donné à une valeur.

Informatiquement, c'est une **référence** à une adresse mémoire.

- Les noms des variables sont conventionnellement écrits en minuscule. Ils commencent par une lettre ou le caractère souligné (_), puis éventuellement, des lettres, des chiffres ou le caractère souligné.

Les données et les variables

Variable

C'est un **nom** donné à une valeur.

Informatiquement, c'est une **référence** à une adresse mémoire.

- Les noms des variables sont conventionnellement écrits en minuscule. Ils commencent par une lettre ou le caractère souligné (_), puis éventuellement, des lettres, des chiffres ou le caractère souligné.
- Ils doivent être différents des mots réservés de Python.

Les mots réservés de Python 2.5

and	def	finally	in	print	yield
as	del	for	is	raise	
assert	elif	from	lambda	return	
break	else	global	not	try	
class	except	if	or	while	
continue	exec	import	pass	with	

Les mots réservés de Python 2.5

and	def	finally	in	print	yield
as	del	for	is	raise	
assert	elif	from	lambda	return	
break	else	global	not	try	
class	except	if	or	while	
continue	exec	import	pass	with	

Remarques :

Ne pas redéfinir les constantes None, True et False.

L'affectation

Affectation

On **affecte** une variable par une valeur en utilisant le signe `=` (qui *n'a rien à voir* avec l'égalité en math!).

L'affectation

Affectation

On **affecte** une variable par une valeur en utilisant le signe **=** (qui *n'a rien à voir* avec l'égalité en math!).

Dans une affectation, le membre de gauche **reçoit** le membre de droite (ce qui nécessite deux temps d'horloge différents).

L'affectation

Affectation

On **affecte** une variable par une valeur en utilisant le signe **=** (qui *n'a rien à voir* avec l'égalité en math!).

Dans une affectation, le membre de gauche **reçoit** le membre de droite (ce qui nécessite deux temps d'horloge différents).

```
a = 2 # prononcez : a "reçoit" 2
```

L'affectation

Affectation

On **affecte** une variable par une valeur en utilisant le signe `=` (qui *n'a rien à voir* avec l'égalité en math!).

Dans une affectation, le membre de gauche **reçoit** le membre de droite (ce qui nécessite deux temps d'horloge différents).

```
a = 2 # prononcez : a "reçoit" 2
```

La valeur d'une variable peut évoluer au cours du temps (la valeur antérieure est perdue) :

```
a = a + 1 # 3 (incréméntation)
```

```
a = a - 1 # 2 (décréméntation)
```

Affecter n'est pas comparer !

l'affectation a un effet mais n'a pas de valeur :

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

Affecter n'est pas comparer !

l'affectation a un effet mais n'a pas de valeur :

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

la comparaison a une valeur mais n'a pas d'effet :

```
>>> a == b  
False  
>>>
```

Détails de l'affectation en Python (1/3)

Dans l'affectation : `a = 3`

- `a` est une **référence**
- `3` est un **objet** qui comporte :
 - une **adresse**, donnée par la fonction `id()` ;
 - un **type**, donné par la fonction `type()` ;
 - une **valeur**, modifiable ou non suivant son type.

Détails de l'affectation en Python (1/3)

Dans l'affectation : `a = 3`

- `a` est une **référence**
- `3` est un **objet** qui comporte :
 - une **adresse**, donnée par la fonction `id()` ;
 - un **type**, donné par la fonction `type()` ;
 - une **valeur**, modifiable ou non suivant son type.

Par conséquent, dans l'affectation `b = a` :

- 1 `b` est un alias de `a`, qui incrémente un compteur de références ;
- 2 `a` et `b` sont deux références qui pointent sur le même objet.

Détails de l'affectation en Python (2/3)

Le compteur de référence d'un objet est **incrémenté** lorsque :

- l'objet est créé :

```
x = 3.14
```

Détails de l'affectation en Python (2/3)

Le compteur de référence d'un objet est **incrémenté** lorsque :

- l'objet est créé :

```
x = 3.14
```

- des alias de celui-ci sont créés :

```
y = x
```

Détails de l'affectation en Python (2/3)

Le compteur de référence d'un objet est **incrémenté** lorsque :

- l'objet est créé :

```
x = 3.14
```

- des alias de celui-ci sont créés :

```
y = x
```

- il est transmis à une fonction (nouvelle référence locale) :

```
maFonction(x)
```

Détails de l'affectation en Python (2/3)

Le compteur de référence d'un objet est **incrémenté** lorsque :

- l'objet est créé :

```
x = 3.14
```

- des alias de celui-ci sont créés :

```
y = x
```

- il est transmis à une fonction (nouvelle référence locale) :

```
maFonction(x)
```

- il fait partie d'un objet conteneur :

```
maListe = [123, x, 'abc']
```

Détails de l'affectation en Python (3/3)

Le compteur de référence d'un objet est **décrémenté** lorsque :

- on sort de la portée d'une référence locale (par exemple lorsque `maFonction()` se termine);

Détails de l'affectation en Python (3/3)

Le compteur de référence d'un objet est **décrémenté** lorsque :

- on sort de la portée d'une référence locale (par exemple lorsque `maFonction()` se termine);
- des alias de l'objet sont explicitement détruits :

```
del y # ou del x
```

Détails de l'affectation en Python (3/3)

Le compteur de référence d'un objet est **décrémenté** lorsque :

- on sort de la portée d'une référence locale (par exemple lorsque `maFonction()` se termine);
- des alias de l'objet sont explicitement détruits :

```
del y # ou del x
```

- un alias est réassocié à un autre objet (nouvelle référence) :

```
x = 123
```

Détails de l'affectation en Python (3/3)

Le compteur de référence d'un objet est **décrémenté** lorsque :

- on sort de la portée d'une référence locale (par exemple lorsque `maFonction()` se termine);
- des alias de l'objet sont explicitement détruits :

```
del y # ou del x
```

- un alias est réassocié à un autre objet (nouvelle référence) :

```
x = 123
```

- l'objet est explicitement enlevé d'un objet conteneur :

```
maListe.remove(x)
```

Détails de l'affectation en Python (3/3)

Le compteur de référence d'un objet est **décrémenté** lorsque :

- on sort de la portée d'une référence locale (par exemple lorsque `maFonction()` se termine);
- des alias de l'objet sont explicitement détruits :

```
del y # ou del x
```

- un alias est réassocié à un autre objet (nouvelle référence) :

```
x = 123
```

- l'objet est explicitement enlevé d'un objet conteneur :

```
maListe.remove(x)
```

- le conteneur lui-même est désalloué ou lorsqu'on sort de sa portée :

```
del maListe
```

Instructions d'affectation : différentes formes

Outre l'affectation simple, on peut aussi utiliser les formes suivantes :

```
a = 4    # forme de base
a += 2   # idem à : a = a + 2 si a est déjà référencé
c = d = 8 # cibles multiples (affectation de droite à gauche)
e, f = 2.7, 5.1 # affectation de tuple (par position)
e, f = f, e     # échange les valeurs de e et f
g, h = ['G', 'H'] # affectation de liste (par position)
```

Les entrées

- L'instruction `raw_input()` effectue toujours une saisie en mode texte que l'on peut ensuite transtyper :

```
f1 = raw_input("Entrez un flottant : ")  
f1 = float(f1) # transtypage en flottant  
# ou plus brièvement :  
f2 = float(raw_input("Entrez un autre flottant : "))
```

Les entrées

- L'instruction `raw_input()` effectue toujours une saisie en mode texte que l'on peut ensuite transtyper :

```
f1 = raw_input("Entrez un flottant : ")  
f1 = float(f1) # transtypage en flottant  
# ou plus brièvement :  
f2 = float(raw_input("Entrez un autre flottant : "))
```

- L'instruction `input()` permet se saisir une entrée au clavier. Comme c'est une évaluation, elle effectue un *typage dynamique*. Elle permet également d'afficher une invite :

```
n = input("Entrez un entier : ")
```

Les sorties

- L'instruction `print` permet d'afficher des sorties à l'écran :

```
a = 2
b = 5
print a # 2
print "Somme :", a + b # 7
print "Différence :", a - b, # -3
print "produit :", a * b # 10
```

Les sorties

- L'instruction `print` permet d'afficher des sorties à l'écran :

```
a = 2
b = 5
print a                # 2
print "Somme :", a + b # 7
print "Différence :", a - b, # -3
print "produit :", a * b    # 10
```

- Le séparateur virgule (,) permet d'empêcher le retour à la ligne.

Qu'est-ce qu'une séquence ?

Définition

Une séquence est un conteneur ordonné d'éléments indicés par des entiers.

Qu'est-ce qu'une séquence ?

Définition

Une séquence est un conteneur ordonné d'éléments indicés par des entiers.

Python dispose de trois types prédéfinis de séquences :

- les chaînes (normales, brutes et en Unicode) ;
- les listes ;
- les tuples.

Fonctions et méthodes

On peut agir sur les séquence en utilisant des fonctions (notation procédurale) et des méthodes (notation objet).

Fonctions et méthodes

On peut agir sur les séquence en utilisant des fonctions (notation procédurale) et des méthodes (notation objet).

Pour appliquer une fonction on utilise directement l'opérateur `()` :

```
seq = "abc"  
lo = len(seq) # 3
```

Fonctions et méthodes

On peut agir sur les séquence en utilisant des fonctions (notation procédurale) et des méthodes (notation objet).

Pour appliquer une fonction on utilise directement l'opérateur `()` :

```
seq = "abc"  
lo = len(seq) # 3
```

On applique une méthode à un objet en utilisant la notation **pointée** :

```
seq = "abracadabra"  
seq2 = seq.upper() # 'ABRACADABRA'
```

Les chaînes de caractères : notations

Trois notations disponibles :

- Les guillemets permettent d'inclure des apostrophes :

```
c1 = "L'eau vive"
```

Les chaînes de caractères : notations

Trois notations disponibles :

- Les guillemets permettent d'inclure des apostrophes :

```
c1 = "L'eau vive"
```

- Les apostrophes permettent d'inclure des guillemets :

```
c2 = ' est "froide" !'
```

Les chaînes de caractères : notations

Trois notations disponibles :

- Les guillemets permettent d'inclure des apostrophes :

```
c1 = "L'eau vive"
```

- Les apostrophes permettent d'inclure des guillemets :

```
c2 = ' est "froide" !'
```

- Les triples guillemets ou triples apostrophes conservent la mise en page (lignes multiples) :

```
c3 = """Usage :  
-h : help  
-q : quit"""
```

Les chaînes de caractères : opérations

- Longueur :

```
s = "abcde"  
len(s) # 5
```

Les chaînes de caractères : opérations

- Longueur :

```
s = "abcde"  
len(s) # 5
```

- Concaténation :

```
s1 = "abc"  
s2 = "defg"  
s3 = s1 + s2 # 'abcdefg'
```

Les chaînes de caractères : opérations

- Longueur :

```
s = "abcde"  
len(s) # 5
```

- Concaténation :

```
s1 = "abc"  
s2 = "defg"  
s3 = s1 + s2 # 'abcdefg'
```

- Répétition :

```
s4 = "Fi! "  
s5 = s4 * 3 # 'Fi! Fi! Fi! '
```

Les chaînes de caractères : méthodes (1/3)

- `split()` : scinde une chaîne en une liste de mots :

```
'ab*cd'.split('*') # ['ab', 'cd'] (on indique le séparateur)
```

Les chaînes de caractères : méthodes (1/3)

- `split()` : scinde une chaîne en une liste de mots :

```
'ab*cd'.split('*') # ['ab', 'cd'] (on indique le séparateur)
```

- `join()` : concatène une liste de chaînes en une chaîne :

```
'-'.join(['ci', 'joint']) # 'ci-joint'
```

Les chaînes de caractères : méthodes (1/3)

- `split()` : scinde une chaîne en une liste de mots :

```
'ab*cd'.split('*') # ['ab', 'cd'] (on indique le séparateur)
```

- `join()` : concatène une liste de chaînes en une chaîne :

```
'-'.join(['ci', 'joint']) # 'ci-joint'
```

- `find()` : donne la position d'une sous-chaîne dans la chaîne :

```
'abracadabra'.find('bra') # 1 (le premier indice vaut 0)
```

Les chaînes de caractères : méthodes (1/3)

- `split()` : scinde une chaîne en une liste de mots :

```
'ab*cd'.split('*') # ['ab', 'cd'] (on indique le séparateur)
```

- `join()` : concatène une liste de chaînes en une chaîne :

```
'-'.join(['ci', 'joint']) # 'ci-joint'
```

- `find()` : donne la position d'une sous-chaîne dans la chaîne :

```
'abracadabra'.find('bra') # 1 (le premier indice vaut 0)
```

- `count()` : donne le nombre de sous-chaînes dans la chaîne :

```
'abracadabra'.count('bra') # 2
```

Les chaînes de caractères : méthodes (2/3)

- `lower()` : convertit en minuscules :

```
'PETIT'.lower() # 'petit'
```

Les chaînes de caractères : méthodes (2/3)

- `lower()` : convertit en minuscules :

```
'PETIT'.lower() # 'petit'
```

- `upper()` : convertit en majuscules :

```
'grand'.upper() # 'GRAND'
```

Les chaînes de caractères : méthodes (2/3)

- `lower()` : convertit en minuscules :

```
'PETIT'.lower() # 'petit'
```

- `upper()` : convertit en majuscules :

```
'grand'.upper() # 'GRAND'
```

- `capitalize()` : convertit la première lettre en majuscule :

```
'michelle'.capitalize() # 'Michelle'
```

Les chaînes de caractères : méthodes (2/3)

- `lower()` : convertit en minuscules :

```
'PETIT'.lower() # 'petit'
```

- `upper()` : convertit en majuscules :

```
'grand'.upper() # 'GRAND'
```

- `capitalize()` : convertit la première lettre en majuscule :

```
'michelle'.capitalize() # 'Michelle'
```

- `title()` : la première lettre de tous les mots en majuscule :

```
'un beau titre !'.title() # 'Un Beau Titre !'
```

Les chaînes de caractères : méthodes (3/3)

- `swapcase()` : intervertit les casses :

```
'bOB'.swapcase() # 'Bob'
```

Les chaînes de caractères : méthodes (3/3)

- `swapcase()` : intervertit les casses :

```
'bOB'.swapcase() # 'Bob'
```

- `strip()` : supprime les blancs en début et fin de chaîne :

```
' Trop de blancs '.strip() # 'Trop de blancs'
```

Les chaînes de caractères : méthodes (3/3)

- `swapcase()` : intervertit les casses :

```
'bOB'.swapcase() # 'Bob'
```

- `strip()` : supprime les blancs en début et fin de chaîne :

```
' Trop de blancs '.strip() # 'Trop de blancs'
```

- `replace()` : remplace une sous-chaîne par une autre :

```
'abracadabra'.replace('a', 'o') # 'obrocodobro'
```

Les chaînes de caractères : indiçage

- On indique, entre crochets, la position d'un caractère par un *indice* qui commence à 0 :

```
s = "abcdefg"  
s[0]    # a  
s[2]    # c  
s[-1]   # g (négatif : on commence par la fin)
```

Les chaînes de caractères : indiçage

- On indique, entre crochets, la position d'un caractère par un *indice* qui commence à 0 :

```
s = "abcdefg"
s[0]    # a
s[2]    # c
s[-1]   # g (négatif : on commence par la fin)
```

- On peut extraire une sous-chaîne par *découpage* :

```
s[1:3]  # 'bc'
s[3:]   # 'defg'
s[:3]   # 'abc'
```

Les chaînes de caractères : indiçage

- On indique, entre crochets, la position d'un caractère par un *indice* qui commence à 0 :

```
s = "abcdefg"  
s[0]    # a  
s[2]    # c  
s[-1]   # g (négatif : on commence par la fin)
```

- On peut extraire une sous-chaîne par *découpage* :

```
s[1:3]  # 'bc'  
s[3:]   # 'defg'  
s[:3]   # 'abc'
```

Les chaînes de caractères ne sont pas modifiables !

Les chaînes de caractères : formatage

Les **formats** permettent de contrôler finement l'affichage :

```
n = 100
"%d en base 10 : %d" % (n, n) # '100 en base 10 : 100'
"%d en base 8 : %o" % (n, n) # '100 en base 8 : 144'
"%d en base 16 : %x" % (n, n) # '100 en base 16 : 64'

pi = 3.1415926535897931
print "%4.2f" %(pi) # 3.14
print "%.4e" %(pi) # 3.1415e+000
print "%g" %(pi) # 3.14159

msg = "Résultat sur %d échantillons : %4.2f" % (n, pi)
print msg # 'Résultat sur 100 échantillons : 3.14'
```

Les principaux formats

- **%d** : entier signé
- **%u** : entier non-signé
- **%o** : un octal non-signé
- **%x** : un hexadécimal non-signé
- **%s** : une chaîne de caractères
- **%f** : un flottant
- **%e** : un flottant (notation exponentielle)
- **%g** : un flottant (le plus court de %e et %f)

Les principaux formats

- **%d** : entier signé
- **%u** : entier non-signé
- **%o** : un octal non-signé
- **%x** : un hexadécimal non-signé
- **%s** : une chaîne de caractères
- **%f** : un flottant
- **%e** : un flottant (notation exponentielle)
- **%g** : un flottant (le plus court de %e et %f)

On peut aussi contrôler le champ d'affichage : "%7.2f" permet d'écrire un flottant sur 7 caractères dont 2 après le point décimal.

Les chaînes brutes et en Unicode

Python dispose de variantes très utiles pour coder le type chaîne :

- Les chaînes brutes sont précédées d'un **r** ou d'un **R**.
Elles n'interprètent pas les *séquences d'échappement* comme `\` ou `\n`.
- Les chaînes en Unicode sont précédées d'un **u** ou d'un **U**.
- Les littéraux chaînes Unicode bruts commencent par **ur**, et non par **ru**.

Les listes : définition et exemples

Définition

Collection hétérogène, ordonnée et modifiable d'éléments séparés par des virgules, et entourée de crochets.

Les listes : définition et exemples

Définition

Collection hétérogène, ordonnée et modifiable d'éléments séparés par des virgules, et entourée de crochets.

```
couleurs = ['trèfle', 'carreau', 'coeur', 'pique']
print couleurs # ['trèfle', 'carreau', 'coeur', 'pique']
print couleurs[1] # carreau
couleurs[1] = 14
print couleurs # ['trèfle', 14, 'coeur', 'pique']

list1 = ['a', 'b']
list2 = [4, 2.718]
list3 = [list1, list2] # liste de listes
print list3 # [['a', 'b'], [4, 2.718]]
```

Les listes : méthodes et indiciage

Mêmes notations que pour les chaînes de caractères :

```
nombres = [17, 38, 10, 25, 72]

nombres.sort()      # [10, 17, 25, 38, 72]
nombres.append(12)  # [10, 17, 25, 38, 72, 12]
nombres.reverse()   # [12, 72, 38, 25, 17, 10]
nombres.index(17)   # 4
nombres.remove(38)  # [12, 72, 25, 17, 10]

nombres[1:3]        # [72, 25]
nombres[:2]         # [12, 72]
nombres[:]          # [12, 72, 25, 17, 10]
nombres[-1]         # 10
```

Les listes : initialisations

Utilisation de la répétition et de l'instruction `range()` :

```
truc = []           # liste vide
machin = [0.0] * 3 # [0.0, 0.0, 0.0]

range(4)           # créé la liste [0, 1, 2, 3]
range(4, 8)       # créé la liste [4, 5, 6, 7]
range(2, 9, 2)    # créé la liste [2, 4, 6, 8]

chose = range(6)  # [0, 1, 2, 3, 4, 5]

print 3 in chose # True (teste l'appartenance)
```

Les listes : technique de « *slicing* » avancée (1/2)

Insertion d'éléments

Dans le membre de gauche d'une affectation, il faut obligatoirement indiquer une « tranche » pour effectuer une insertion ou une suppression.

Les listes : technique de « *slicing* » avancée (1/2)

Insertion d'éléments

Dans le membre de gauche d'une affectation, il faut obligatoirement indiquer une « tranche » pour effectuer une insertion ou une suppression.

Le membre de droite doit lui-même être une liste.

Les listes : technique de « *slicing* » avancée (1/2)

Insertion d'éléments

Dans le membre de gauche d'une affectation, il faut obligatoirement indiquer une « tranche » pour effectuer une insertion ou une suppression.

Le membre de droite doit lui-même être une liste.

```
mots = ['jambon', 'sel', 'confiture']  
mots[2:2] = ['miel']           # insertion en 3e position  
mots[4:4] = ['beurre']        # insertion en 5e position  
print mots                    # ['jambon', 'sel', 'miel', 'confiture', 'beurre']
```

Les listes : technique de « *slicing* » avancée (2/2)

Suppression/remplacement d'éléments

Mêmes remarques : une « tranche » dans le membre de gauche, une liste dans le membre de droite.

Les listes : technique de « *slicing* » avancée (2/2)

Suppression/remplacement d'éléments

Mêmes remarques : une « tranche » dans le membre de gauche, une liste dans le membre de droite.

```
mots = ['jambon', 'sel', 'miel', 'confiture', 'beurre']  
  
mots[2:4] = []      # effacement par affectation d'une liste vide  
print mots        # ['jambon', 'sel', 'beurre']  
mots[1:3] = ['salade']  
print mots        # ['jambon', 'salade']  
mots[1:] = ['mayonnaise', 'poulet', 'tomate']  
print mots        # ['jambon', 'mayonnaise', 'poulet', 'tomate']
```

Les tuples

Définition

Collection hétérogène, ordonnée et non modifiable d'éléments séparés par des virgules, et entourée de parenthèses.

Les tuples

Définition

Collection hétérogène, ordonnée et non modifiable d'éléments séparés par des virgules, et entourée de parenthèses.

```
monTuple = ('a', 2, [1, 3])
```

Les tuples

Définition

Collection hétérogène, ordonnée et non modifiable d'éléments séparés par des virgules, et entourée de parenthèses.

```
monTuple = ('a', 2, [1, 3])
```

- Les tuples s'utilisent comme les listes mais leur parcours est plus rapide ;

Comme les chaînes de caractères, les tuples ne sont pas modifiables !

Les tuples

Définition

Collection hétérogène, ordonnée et non modifiable d'éléments séparés par des virgules, et entourée de parenthèses.

```
monTuple = ('a', 2, [1, 3])
```

- Les tuples s'utilisent comme les listes mais leur parcours est plus rapide ;
- Ils sont utiles pour définir des constantes.

Comme les chaînes de caractères, les tuples ne sont pas modifiables !

Le problème des références (1/3)

L'opération d'affectation, apparemment innocente, est une réelle difficulté de Python.

```
i = 1  
msg = "Quoi de neuf ?"  
e = 2.718
```

Le problème des références (1/3)

L'opération d'affectation, apparemment innocente, est une réelle difficulté de Python.

```
i = 1  
msg = "Quoi de neuf ?"  
e = 2.718
```

Dans l'exemple ci-dessus, l'affectation réalise plusieurs opérations :

- crée et mémorise un *nom* de variable dans l'espace de noms courant (c'est en fait une *adresse*) ;

Le problème des références (1/3)

L'opération d'affectation, apparemment innocente, est une réelle difficulté de Python.

```
i = 1  
msg = "Quoi de neuf ?"  
e = 2.718
```

Dans l'exemple ci-dessus, l'affectation réalise plusieurs opérations :

- crée et mémorise un *nom* de variable dans l'espace de noms courant (c'est en fait une *adresse*) ;
- lui attribue *dynamiquement* un type bien déterminé ;

Le problème des références (1/3)

L'opération d'affectation, apparemment innocente, est une réelle difficulté de Python.

```
i = 1  
msg = "Quoi de neuf ?"  
e = 2.718
```

Dans l'exemple ci-dessus, l'affectation réalise plusieurs opérations :

- crée et mémorise un *nom* de variable dans l'espace de noms courant (c'est en fait une *adresse*) ;
- lui attribue *dynamiquement* un type bien déterminé ;
- crée et mémorise une *valeur* (membre de droite) ;

Le problème des références (1/3)

L'opération d'affectation, apparemment innocente, est une réelle difficulté de Python.

```
i = 1  
msg = "Quoi de neuf ?"  
e = 2.718
```

Dans l'exemple ci-dessus, l'affectation réalise plusieurs opérations :

- crée et mémorise un *nom* de variable dans l'espace de noms courant (c'est en fait une *adresse*) ;
- lui attribue *dynamiquement* un type bien déterminé ;
- crée et mémorise une *valeur* (membre de droite) ;
- établit un *lien* entre le nom de la variable et l'adresse de la valeur correspondante.

Le problème des références (2/3)

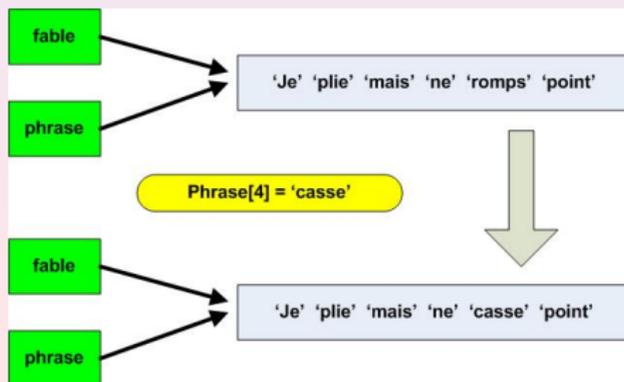
Une conséquence de ce mécanisme est que, si un objet modifiable est affecté, tout changement sur un objet modifiera l'autre :

```
fable = ['Je', 'plie', 'mais', 'ne', 'romps', 'point']  
phrase = fable  
fable[4] = 'casse'
```

Le problème des références (2/3)

Une conséquence de ce mécanisme est que, si un objet modifiable est affecté, tout changement sur un objet modifiera l'autre :

```
fable = ['Je', 'plie', 'mais', 'ne', 'romps', 'point']  
phrase = fable  
fable[4] = 'casse'
```



Le problème des références (3/3)

Si l'on désire réaliser une *vraie* copie d'un objet, on doit utiliser le module `copy` :

Le problème des références (3/3)

Si l'on désire réaliser une *vraie* copie d'un objet, on doit utiliser le module `copy` :

```
import copy
a = [1, 2, 3]
b = a          # une référence
b.append(4)
print a       # [1, 2, 3, 4]
c = copy.copy(a) # une copie de "surface"
c.append(5)
print c       # [1, 2, 3, 4, 5]
print a       # [1, 2, 3, 4]
```

Le problème des références (3/3)

Si l'on désire réaliser une *vraie* copie d'un objet, on doit utiliser le module `copy` :

```
import copy
a = [1, 2, 3]
b = a          # une référence
b.append(4)
print a       # [1, 2, 3, 4]
c = copy.copy(a) # une copie de "surface"
c.append(5)
print c       # [1, 2, 3, 4, 5]
print a       # [1, 2, 3, 4]
```

Dans les rares occasions où l'on veut aussi que chaque élément et attribut de l'objet soit copié séparément et de façon récursive, on emploie la fonction `copy.deepcopy()`.

Les dictionnaires : définition et exemples

Définition

Collection de couples *clé* : *valeur* entourée d'accolades.

Les dictionnaires : définition et exemples

Définition

Collection de couples *clé* : *valeur* entourée d'accolades.

```
dico = {}          # dictionnaire vide

dico['I'] = 'je'
dico['she'] = 'elle'
dico['you'] = 'vous'
print dico        # {'I':'je', 'she':'vous', 'you':'vous'}
print dico['I']   # 'je'
del dico['I']
print dico        # {'she':'vous', 'you':'vous'}
dico['we'] = 'nous'
```

Les dictionnaires : méthodes

Les méthodes suivantes sont spécifiques aux dictionnaires :

```
dico.keys()           # ['we', 'she', 'you']
dico.values()         # ['nous', 'elle', 'vous']
dico.items()          # [('we', 'nous'), ('she', 'elle'), ('you', 'vous')]
dico.has_key('I')     # False
dico.has_key('we')    # True
```

Les dictionnaires : méthodes

Les méthodes suivantes sont spécifiques aux dictionnaires :

```
dico.keys()          # ['we', 'she', 'you']
dico.values()        # ['nous', 'elle', 'vous']
dico.items()         # [('we', 'nous'), ('she', 'elle'), ('you', 'vous')]
dico.has_key('I')    # False
dico.has_key('we')   # True
```

Les dictionnaires ne sont pas ordonnés.
On ne peut donc pas les indicer.

Les dictionnaires : itérateurs

Définition

Les *itérateurs* sont des objets spécifiques permettant de parcourir un dictionnaire.

Les dictionnaires : itérateurs

Définition

Les *itérateurs* sont des objets spécifiques permettant de parcourir un dictionnaire.

On dispose d'itérateurs sur les clés, les valeurs et les items :

```
d = {'a':1, 'b':2, 'c':3, 'd':4} # un dictionnaire

for it1 in d.iterkeys():
    print it1,      # a c b d

for it2 in d.itervalues():
    print it2,     # 1 3 2 4

for it3 in d.iteritems():
    print it3,     # ('a', 1) ('c', 3) ('b', 2) ('d', 4)
```

Les ensembles

- Ils sont générés par la fonction `set()` :

```
x = set('abcd')  
y = set('bdx')
```

Les ensembles

- Ils sont générés par la fonction `set()` :

```
x = set('abcd')  
y = set('bdx')
```

- Opérations sur les ensembles :

```
'c' in x # True : appartenance  
x - y   # set(['a', 'c']) : différence  
x | y   # set(['a', 'c', 'b', 'd', 'x']) : union  
x & y   # set(['b', 'd']) : intersection
```

Les instructions composées

Syntaxe

Elles se composent :

- d'une ligne d'en-tête terminée par deux-points ;

Les instructions composées

Syntaxe

Elles se composent :

- d'une ligne d'en-tête terminée par deux-points ;
- d'un bloc d'instructions indenté au même niveau.

Les instructions composées

Syntaxe

Elles se composent :

- d'une ligne d'en-tête terminée par deux-points ;
- d'un bloc d'instructions indenté au même niveau.

On peut imbriquer des instructions composées pour réaliser des structures de décision complexes.

```
if -- [elif] -- [else]
```

Contrôler une alternative :

if -- [elif] -- [else]

Contrôler une alternative :

```
if x < 0:
    print "x est négatif"
elif x % 2:
    print "x est positif et impair"
else:
    print "x n'est pas négatif et est pair"

# Test d'une valeur booléenne :
if x: # mieux que (if x is True:) ou que (if x == True:)
```

Syntaxe compacte de choix d'une alternative

Trouver, par exemple, le minimum de deux nombres :

Syntaxe compacte de choix d'une alternative

Trouver, par exemple, le minimum de deux nombres :

```
x, y = 4, 3
```

```
# Ecriture classique :
```

```
if x < y:  
    plusPetit = x  
else:  
    plusPetit = y
```

```
# Utilisation de l'opérateur ternaire :
```

```
plusPetit = x if x < y else y
```

```
while -- [else]
```

Répéter une portion de code :

while -- [else]

Répéter une portion de code :

```
cpt = 0
while x > 0:
    x = x // 2 # division avec troncature
    cpt += 1  # incrémentation
print "L'approximation de log2 de x est", cpt

# Exemple de saisie filtrée
n = input('Entrez un entier [1 .. 10] : ')
while (n < 1) or (n > 10):
    n = input('Entrez un entier [1 .. 10], S.V.P. : ')
```

for -- [else]

Parcourir une séquence :

for -- [else]

Parcourir une séquence :

```
for lettre in "ciao":  
    print lettre, # c i a o  
  
for x in [2, 'a', 3.14]:  
    print x,      # 2 a 3.14  
  
for i in range(5):  
    print i,      # 0 1 2 3 4
```

Les listes en compréhension (1/3)

Définition

Une liste en compréhension est une expression qui permet de générer une liste de manière très compacte.

Les listes en compréhension (1/3)

Définition

Une liste en compréhension est une expression qui permet de générer une liste de manière très compacte.

Elle est équivalente à une boucle `for` qui construirait la même liste en utilisant la méthode `append()`.

Les listes en compréhension (1/3)

Définition

Une liste en compréhension est une expression qui permet de générer une liste de manière très compacte.

Elle est équivalente à une boucle `for` qui construirait la même liste en utilisant la méthode `append()`.

Les listes en compréhension sont utilisables sous trois formes.

Première forme :

```
result1 = [x+1 for x in une_seq]
# a le même effet que :
result2 = []
for x in une_seq:
    result2.append(x+1)
```

Les listes en compréhension (2/3)

Deuxième forme :

```
result3 = [x+1 for x in une_seq if x > 23]
# a le même effet que :
result4 = []
for x in une_seq:
    if x > 23:
        result4.append(x+1)
```

Les listes en compréhension (2/3)

Deuxième forme :

```
result3 = [x+1 for x in une_seq if x > 23]
# a le même effet que :
result4 = []
for x in une_seq:
    if x > 23:
        result4.append(x+1)
```

Troisième forme :

```
result5 = [x+y for x in une_seq for y in une_autre]
# a le même effet que :
result6 = []
for x in une_seq:
    for y in une_autre:
        result6.append(x+y)
```

Les listes en compréhension (3/3)

Une utilisation fréquente très « pythonique », le calcul d'une somme :

```
s = sum([i for i in xrange(10)])  
# a le même effet que :  
s = 0  
for i in xrange(10):  
    s = s + i
```

Interrompre : `break`

Sort immédiatement de la boucle `for` ou `while` en cours contenant l'instruction et passe outre le `else` éventuel :

Interrompre : break

Sort immédiatement de la boucle `for` ou `while` en cours contenant l'instruction et passe outre le `else` éventuel :

```
for x in range(1, 11):
    if x == 5:
        break
    print x,

print "\nBoucle interrompue pour x =", x
# affiche :
# 1 2 3 4
# Boucle interrompue pour x = 5
```

Interrompre : continue

Passer immédiatement au début de la boucle `for` ou `while` en cours contenant l'instruction ; reprend à la ligne de l'en-tête de la boucle :

Interrompre : continue

Passer immédiatement au début de la boucle `for` ou `while` en cours contenant l'instruction ; reprend à la ligne de l'en-tête de la boucle :

```
for x in range(1, 11):
    if x == 5:
        continue
    print x,

print "\nLa boucle a sauté la valeur 5"
# affiche :
# 1 2 3 4 6 7 8 9 10
# La boucle a sauté la valeur 5
```

Exceptions : `try -- except -- [else]`

Définition

Opération effectuée à l'exécution par Python lorsqu'une erreur est détectée.

Exceptions : try -- except -- [else]

Définition

Opération effectuée à l'exécution par Python lorsqu'une erreur est détectée.

```
from math import sin

for x in range(-3, 4):
    try:
        print '%.3f' %(sin(x)/x), # cas normal
    except:
        print 1.0,                # gère l'exception en 0
```

Gérer ses propres exceptions : `raise`

L'instruction `raise` permet de lever *volontairement* une exception :

Gérer ses propres exceptions : raise

L'instruction `raise` permet de lever *volontairement* une exception :

```
if not 0<=x<=1:  
    raise ValueError, "x n'est pas dans [0 .. 1]"
```

Gérer ses propres exceptions : raise

L'instruction `raise` permet de lever *volontairement* une exception :

```
if not 0<=x<=1:  
    raise ValueError, "x n'est pas dans [0 .. 1]"
```

(Voir la liste des exceptions standards dans la documentation Python)

Définition et syntaxe

Fonction

Groupe d'instructions regroupé sous un *nom* et s'exécutant à la demande (*appel*). Les fonctions sont les éléments structurants de base de tout langage procédural.

Définition et syntaxe

Fonction

Groupe d'instructions regroupé sous un *nom* et s'exécutant à la demande (*appel*). Les fonctions sont les éléments structurants de base de tout langage procédural.

Syntaxe

C'est une instruction composée :

```
def nom_fonction(paramètres):  
    """Documentation de la fonction."""  
    <bloc_instructions>
```

Définition et syntaxe

Fonction

Groupe d'instructions regroupé sous un *nom* et s'exécutant à la demande (*appel*). Les fonctions sont les éléments structurants de base de tout langage procédural.

Syntaxe

C'est une instruction composée :

```
def nom_fonction(paramètres):  
    """Documentation de la fonction."""  
    <bloc_instructions>
```

Le bloc d'instructions est **obligatoire**. S'il ne fait rien, on emploie l'instruction `pass`. La documentation est *fortement* conseillée.

Exemples de fonction et d'appel

```
def double(x):  
    return 2*x  
  
def table(base):  
    n = 1  
    while n < 11:  
        print n * base,  
        n += 1  
  
print double(8) # 16 : "appel" de la fonction double  
# Affiche la table de multiplication par 8 :  
table(8)      # "appel" de la fonction table
```

Mécanisme général

Passage par affectation

Chaque argument de la définition de la fonction correspond, dans l'ordre, à un paramètre de l'appel.

Mécanisme général

Passage par affectation

Chaque argument de la définition de la fonction correspond, dans l'ordre, à un paramètre de l'appel.

La correspondance se fait par *affectation*.

Mécanisme général

Passage par affectation

Chaque argument de la définition de la fonction correspond, dans l'ordre, à un paramètre de l'appel.

La correspondance se fait par *affectation*.

```
def ma_fonction(x, y, z):  
    pass  
  
# Appel de la fonction :  
## le passage d'arguments se fait dans l'ordre, par affectation :  
ma_fonction(7, 'k', 2.718) # x = 7, y = 'k', z = 2.718
```

Un ou plusieurs paramètres, pas de retour

```
def table(base, debut, fin):  
    """Affiche la table des <base> de <debut> à <fin>."""  
    n = debut  
    while n <= fin:  
        print n, 'x', base, '=', n * base,  
        n += 1  
  
# exemple d'appel :  
table(7, 2, 11)
```

Un ou plusieurs paramètres, utilisation du retour

```
from math import pi

def cube(x):
    return x**3

def volumeSphere(r):
    return 4.0 * pi * cube(r) / 3.0

# Saisie du rayon et affichage du volume
rayon = float(raw_input('Rayon : '))
print 'Volume de la sphère =', volumeSphere(rayon)
```

La valeur d'un paramètre peut être une fonction

```
def tabuler(fonction, borneInf, borneSup, nbPas):  
    """Affichage des valeurs de <fonction>.  
    On doit avoir (borneInf < borneSup) et (nbPas > 0)  
    """  
    h = (borneSup - borneInf) / float(nbPas)  
    x = borneInf  
    while x <= borneSup:  
        y = fonction(x)  
        print "f(%.2f) = %.2f" % (x, y)  
        x += h  
  
def maFonction(x):  
    return 2*x**3 + x - 5  
  
tabuler(maFonction, -5, 5, 10)
```

Paramètres avec valeur par défaut

```
def initPort(speed=9600, parity="paire", data=8, stops=1):  
    print "Initialisation à", speed, "bits/s"  
    print "parité :", parity  
    print data, "bits de données"  
    print stops, "bits d'arrêt"  
  
# Appels possibles :  
initPortort()  
initPort(parity="nulle")  
initPort(2400, "paire", 7, 2)
```

Note : on utilise de préférence des valeurs par défaut **non-modifiables** (entiers, booléens, flottants, complexes, chaînes, tuples) car la modification d'un paramètre par un premier appel est visible les fois suivantes.

Nombre d'arguments arbitraire : passage d'un tuple (1/2)

```
def somme(*args):  
    """Renvoie la somme de <tuple>."""  
    resultat = 0  
    for nombre in args:  
        resultat += nombre  
    return resultat  
  
# Exemples d'appel :  
print somme(23)           # 23  
print somme(23, 42, 13)  # 78
```

Nombre d'arguments arbitraire : passage d'un tuple (2/2)

Il est aussi possible de passer un tuple (en fait une séquence) à l'appel qui sera *décompressé* en une liste de paramètres d'une fonction « classique » :

```
def somme(a, b, c):  
    return a+b+c  
  
# Exemple d'appel :  
elements = (2, 4, 6)  
print somme(*elements) # 12
```

Nombre d'arguments arbitraire : passage d'un dictionnaire

```
def un_dict(**kargs):  
    return kargs  
  
# Exemples d'appels  
## par des paramètres nommés :  
print un_dict(a=23, b=42) # affiche : {'a': 23, 'b': 42}  
  
## en fournissant un dictionnaire :  
mots = {'d': 85, 'e': 14, 'f': 9}  
print un_dict(**mots) # affiche : {'e': 14, 'd': 85, 'f': 9}
```

Portée des objets

Portée

Les noms des objets sont créés lors de leur *première affectation*, mais ne sont visibles que dans certaines régions de la mémoire.

Portée des objets

Portée

Les noms des objets sont créés lors de leur *première affectation*, mais ne sont visibles que dans certaines régions de la mémoire.

- **Portée globale** : celle du module `__main__`. Un dictionnaire gère les objets globaux : l'instruction `globals()` fournit les couples `variable : valeur` ;

Portée des objets

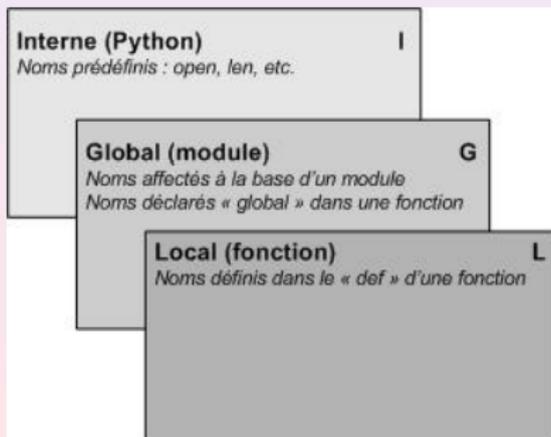
Portée

Les noms des objets sont créés lors de leur *première affectation*, mais ne sont visibles que dans certaines régions de la mémoire.

- **Portée globale** : celle du module `__main__`. Un dictionnaire gère les objets globaux : l'instruction `globals()` fournit les couples `variable : valeur` ;
- **Portée locale** : les objets internes aux fonctions (et aux classes) sont locaux. Les objets globaux ne sont *pas modifiables* dans les portées locales. L'instruction `locals()` fournit les couples `variable : valeur`.

Résolution des noms : règle *LGI*

La recherche des noms est d'abord locale (**L**), puis globale (**G**), enfin interne (**I**) :



Exemple de portée (1/3)

```
# x et fonc sont affectés dans le module : globaux

def fonc(y): # y et z sont affectés dans fonc : locaux
    # dans fonc : portée locale
    z = x + y
    return z

x = 99
print fonc(1) # affiche : 100
```

Exemple de portée (2/3)

```
# x et fonc sont affectés dans le module : globaux

def fonc(y): # y, x et z sont affectés dans fonc : locaux
    x = 3 # ce nouvel x est local et masque le x global
    z = x + y
    return z

x = 99
print fonc(1) # affiche : 4
```

Exemple de portée (3/3)

```
# x et fonc sont affectés dans le module : globaux

def fonc(y): # y et z sont affectés dans fonc : locaux
    global x # permet de modifier x ligne suivante
    x += 2
    z = x + y
    return z

x = 99
print fonc(1) # affiche : 102
```

Définition

Module

Fichier *indépendant* permettant d'élaborer des bibliothèques de fonctions ou de classes.

Définition

Module

Fichier *indépendant* permettant d'élaborer des bibliothèques de fonctions ou de classes.

Avantages des modules :

- réutilisation du code ;

Définition

Module

Fichier *indépendant* permettant d'élaborer des bibliothèques de fonctions ou de classes.

Avantages des modules :

- réutilisation du code ;
- la documentation et les tests peuvent être intégrés au module ;

Définition

Module

Fichier *indépendant* permettant d'élaborer des bibliothèques de fonctions ou de classes.

Avantages des modules :

- réutilisation du code ;
- la documentation et les tests peuvent être intégrés au module ;
- réalisation de services ou de données partagés ;

Définition

Module

Fichier *indépendant* permettant d'élaborer des bibliothèques de fonctions ou de classes.

Avantages des modules :

- réutilisation du code ;
- la documentation et les tests peuvent être intégrés au module ;
- réalisation de services ou de données partagés ;
- partition de l'espace de noms du système.

Import d'un modules

Deux syntaxes possibles :

- la commande `import <nom_module>` importe la totalité des objets du module :

```
import Tkinter
```

Import d'un modules

Deux syntaxes possibles :

- la commande `import <nom_module>` importe la totalité des objets du module :

```
import Tkinter
```

- la commande `from <nom_module> import obj1, obj2...` n'importe que les objets `obj1, obj2...` du module :

```
from math import pi, sin, log
```

Exemple

Un module `cube_m.py` :

```
def cube(y):  
    u"""Calcule le cube du paramètre.""" # docstring  
    return y**3  
  
# Auto-test -----  
if __name__ == "__main__": # ignoré lors d'un import  
    help(cube)             # affiche le docstring de la fonction  
    print "cube de 9 :", cube(9)
```

Utilisation de ce module :

```
from cube_m import cube  
  
for i in xrange(10):  
    print "cube de", i, "=", cube(i)
```

Bibliothèques tierces

Outre les modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines :

- scientifique ;

Le site pypi.python.org/pypi recense plus de 4500 packages !

Bibliothèques tierces

Outre les modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines :

- scientifique ;
- bases de données ;

Le site pypi.python.org/pypi recense plus de 4500 packages !

Bibliothèques tierces

Outre les modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines :

- scientifique ;
- bases de données ;
- tests fonctionnels et contrôle de qualité ;

Le site pypi.python.org/pypi recense plus de 4500 packages !

Bibliothèques tierces

Outre les modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines :

- scientifique ;
- bases de données ;
- tests fonctionnels et contrôle de qualité ;
- 3D ;

Le site pypi.python.org/pypi recense plus de 4500 packages !

Bibliothèques tierces

Outre les modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines :

- scientifique ;
- bases de données ;
- tests fonctionnels et contrôle de qualité ;
- 3D ;
- ...

Le site pypi.python.org/pypi recense plus de 4500 packages !

Exemple : la bibliothèque Unum

Elle permet de calculer en tenant compte des unités du système S.I.

Voici un exemple de session interactive :

Exemple : la bibliothèque Unum

Elle permet de calculer en tenant compte des unités du système S.I.

Voici un exemple de session interactive :

```
-- Welcome in Unum Calculator (ver 04.00) --
>>> d = 1609 * M
>>> t = 11.7 * S
>>> v = d/t
>>> v
137.521367521 [m/s]
>>> a = v/t
>>> a
11.753963036 [m/s2]
>>>
```

Ouverture et fermeture des fichiers

Le type fichier est un type pré-défini de Python.

Ouverture et fermeture des fichiers

Le type fichier est un type pré-défini de Python.

Principaux *modes* d'ouverture :

```
f1 = open("monFichier_1", "r") # en lecture
f2 = open("monFichier_2", "w") # en écriture
f3 = open("monFichier_3", "a") # en ajout
```

Python utilise les fichiers en mode *texte* par défaut (noté t). Pour les fichiers *binaires* on précise le mode b. Tant que le fichier n'est pas fermé, son contenu n'est pas garanti sur le disque.

Ouverture et fermeture des fichiers

Le type fichier est un type pré-défini de Python.

Principaux *modes* d'ouverture :

```
f1 = open("monFichier_1", "r") # en lecture  
f2 = open("monFichier_2", "w") # en écriture  
f3 = open("monFichier_3", "a") # en ajout
```

Python utilise les fichiers en mode *texte* par défaut (noté t). Pour les fichiers *binaires* on précise le mode b. Tant que le fichier n'est pas fermé, son contenu n'est pas garanti sur le disque.

Une seule méthode de fermeture :

```
f1.close()
```

Écriture séquentielle

Méthodes d'écriture :

```
f = open("truc.txt", "w")
s = 'toto'
f.write(s)          # écrit la chaîne s dans f
l = ['a', 'b', 'c']
f.writelines(l)    # écrit les chaînes de la liste l dans f
f.close()

# on peut aussi utiliser le "print étendu" :
f = open("truc.txt", "w")
print >> f, "abcd" # écrit dans f en mode ajout
f.close()
```

Lecture séquentielle

Méthodes de lecture :

```
f = open("truc.txt", "r")
s = f.read()      # lit tout le fichier --> string
s = f.read(n)    # lit au plus n octets --> string
s = f.readline() # lit la ligne suivante --> string
s = f.readlines() # lit tout le fichier --> liste de strings

for ligne in f:
    print ligne  # bon procédé de parcours d'un fichier

f.close()
```

Le module « pickle »

Plus que du stockage, il permet la *conservation* des types :

Le module « pickle »

Plus que du stockage, il permet la *conservation* des types :

```
import pickle

a, b = 5, 2.83
f = open("monFichier.txt", "w") # en écriture
pickle.dump(a, f)
pickle.dump(b, f)
f.close()

f = open("monFichier.txt", "r") # en lecture
t = pickle.load(f)
print t, type(t) # 5 <type 'int'>
t = pickle.load(f)
print t, type(t) # 2.83 <type 'float'>
f.close()
```

La programmation *Orientée Objet*

Les *classes* sont les éléments structurants de base de tout langage orienté objet.

Une **classe** Python possède plusieurs caractéristiques :

- on appelle un objet classe comme s'il s'agissait d'une fonction.
L'objet créé (*instance*) sait à quelle classe il appartient ;

La programmation *Orientée Objet*

Les *classes* sont les éléments structurants de base de tout langage orienté objet.

Une **classe** Python possède plusieurs caractéristiques :

- on appelle un objet classe comme s'il s'agissait d'une fonction. L'objet créé (*instance*) sait à quelle classe il appartient ;
- Une classe possède deux sortes d'attributs : des *données* et des fonctions (appelées *méthodes*) ;

La programmation *Orientée Objet*

Les *classes* sont les éléments structurants de base de tout langage orienté objet.

Une **classe** Python possède plusieurs caractéristiques :

- on appelle un objet classe comme s'il s'agissait d'une fonction. L'objet créé (*instance*) sait à quelle classe il appartient ;
- Une classe possède deux sortes d'attributs : des *données* et des fonctions (appelées *méthodes*) ;
- Python définit des *méthodes spéciales* ;

La programmation *Orientée Objet*

Les *classes* sont les éléments structurants de base de tout langage orienté objet.

Une **classe** Python possède plusieurs caractéristiques :

- on appelle un objet classe comme s'il s'agissait d'une fonction. L'objet créé (*instance*) sait à quelle classe il appartient ;
- Une classe possède deux sortes d'attributs : des *données* et des fonctions (appelées *méthodes*) ;
- Python définit des *méthodes spéciales* ;
- Une classe peut *hériter* d'autres classes.

L'instruction class

C'est une instruction composée : en-tête (avec *docstring*) + corps indenté.

```
class C(object):  
    """Documentation de la classe."""  
    x = 23
```

L'instruction class

C'est une instruction composée : en-tête (avec *docstring*) + corps indenté.

```
class C(object):  
    """Documentation de la classe."""  
    x = 23
```

Dans cet exemple, C est le nom de la *classe* (qui commence conventionnellement par une majuscule), object est un type prédéfini ancêtre de tous les autres types, et x est un *attribut de classe*, local à C.

L'instanciation et ses attributs

On crée un objet en appelant le nom de sa classe :

```
a = C()      # a est un objet de la classe C
print dir(a) # affiche les attributs de l'objet a
print a.x    # affiche 23. x est un attribut de classe
a.x = 12     # modifie l'attribut d'instance (attention...)
print C.x    # 23, l'attribut de classe est inchangé
a.y = 44     # nouvel attribut d'instance

b = C()      # b est un autre objet de la classe C
print b.x    # 23. b connaît son attribut de classe, mais...
print b.y    # AttributeError: C instance has no attribute 'y'
```

Retour sur les espaces de noms (1/2)

Les espaces de noms sont implémentés par des *dictionnaires* pour les modules, les classes et les instances.

Retour sur les espaces de noms (1/2)

Les espaces de noms sont implémentés par des *dictionnaires* pour les modules, les classes et les instances.

- **Noms non qualifiés** (exemple `dimension`) l'affectation crée ou change le nom dans la *portée* locale courante. Ils sont cherchés suivant la règle LGI.

Retour sur les espaces de noms (1/2)

Les espaces de noms sont implémentés par des *dictionnaires* pour les modules, les classes et les instances.

- **Noms non qualifiés** (exemple `dimension`) l'affectation crée ou change le nom dans la *portée* locale courante. Ils sont cherchés suivant la règle LGI.
- **Noms qualifiés** (exemple `dimension.hauteur`) l'affectation crée ou modifie l'attribut dans l'espace de noms de l'objet. Un attribut est cherché dans l'objet, puis dans toutes les classes dont l'objet dépend (mais pas dans les modules).

Retour sur les espaces de noms (2/2)

L'exemple suivant affiche le dictionnaire lié à la classe C puis la liste des attributs liés à une instance de C.

Retour sur les espaces de noms (2/2)

L'exemple suivant affiche le dictionnaire lié à la classe C puis la liste des attributs liés à une instance de C.

```
class C(object):
    x = 20

print C.__dict__ # {'__dict__': <attribute '__dict__' of 'C'
                  objects>, 'x': 20, '__module__': '__main__', '__weakref__':
                  <attribute '__weakref__' of 'C' objects>, '__doc__': None}
a = C()
print dir(a)     # ['__class__', '__delattr__', '__dict__', '__
                  __doc__', '__getattr__', '__hash__', '__init__', '__
                  __module__', '__new__', '__reduce__', '__reduce_ex__', '__
                  __repr__', '__setattr__', '__str__', '__weakref__', 'x']
```

Syntaxe

Méthode

Une méthode s'écrit comme une fonction *du corps de la classe* avec un premier paramètre **self** obligatoire. **self** représente l'objet sur lequel la méthode sera appliquée.

Syntaxe

Méthode

Une méthode s'écrit comme une fonction *du corps de la classe* avec un premier paramètre **self** obligatoire. **self** représente l'objet sur lequel la méthode sera appliquée.

```
class C(object):
    x = 23          # x et y : attributs de classe
    y = x + 5
    def affiche(self): # méthode affiche()
        self.z = 42  # attribut d'instance
        print C.y,  # dans une méthode, on qualifie un attribut
                    # de classe,
        print self.z # mais pas un attribut d'instance
ob = C()          # instanciation de l'objet ob
ob.affiche()     # 28 42 (à l'appel, ob affecte self)
```

Les méthodes spéciales

Ces méthodes portent des noms pré-définis, précédés et suivis de deux caractères de soulignement.

Les méthodes spéciales

Ces méthodes portent des noms pré-définis, précédés et suivis de deux caractères de soulignement.

Elles servent :

- à initialiser l'objet instancié ;

Les méthodes spéciales

Ces méthodes portent des noms pré-définis, précédés et suivis de deux caractères de soulignement.

Elles servent :

- à initialiser l'objet instancié ;
- à modifier son affichage ;

Les méthodes spéciales

Ces méthodes portent des noms pré-définis, précédés et suivis de deux caractères de soulignement.

Elles servent :

- à initialiser l'objet instancié ;
- à modifier son affichage ;
- à surcharger ses opérateurs ;

Les méthodes spéciales

Ces méthodes portent des noms pré-définis, précédés et suivis de deux caractères de soulignement.

Elles servent :

- à initialiser l'objet instancié ;
- à modifier son affichage ;
- à surcharger ses opérateurs ;
- ...

Le constructeur

Lors de l'instanciation d'un objet, la méthode `__init__` est automatiquement invoquée. Elle permet d'effectuer toutes les initialisations nécessaires :

Le constructeur

Lors de l'instanciation d'un objet, la méthode `__init__` est automatiquement invoquée. Elle permet d'effectuer toutes les initialisations nécessaires :

```
class C(object):  
    def __init__(self, n):  
        self.x = n      # initialisation de l'attribut d'instance x  
  
uneInstance = C(42)   # paramètre obligatoire, affecté à n  
print uneInstance.x  # 42
```

Surcharge des opérateurs (1/2)

La *surcharge* permet à un opérateur de posséder un sens différent suivant le type de leurs opérandes. Par exemple, l'opérateur **+** permet :

Surcharge des opérateurs (1/2)

La *surcharge* permet à un opérateur de posséder un sens différent suivant le type de leurs opérandes. Par exemple, l'opérateur `+` permet :

```
x = 7 + 9           # addition entière  
s = 'ab' + 'cd'    # concaténation
```

Surcharge des opérateurs (1/2)

La *surcharge* permet à un opérateur de posséder un sens différent suivant le type de leurs opérandes. Par exemple, l'opérateur `+` permet :

```
x = 7 + 9           # addition entière  
s = 'ab' + 'cd'   # concaténation
```

Python possède des méthodes de surcharge pour :

- tous les types (`__call__`, `__str__`, ...);

Surcharge des opérateurs (1/2)

La *surcharge* permet à un opérateur de posséder un sens différent suivant le type de leurs opérandes. Par exemple, l'opérateur `+` permet :

```
x = 7 + 9           # addition entière  
s = 'ab' + 'cd'    # concaténation
```

Python possède des méthodes de surcharge pour :

- tous les types (`__call__`, `__str__`, ...);
- les nombres (`__add__`, `__div__`, ...);

Surcharge des opérateurs (1/2)

La *surcharge* permet à un opérateur de posséder un sens différent suivant le type de leurs opérandes. Par exemple, l'opérateur `+` permet :

```
x = 7 + 9           # addition entière  
s = 'ab' + 'cd'    # concaténation
```

Python possède des méthodes de surcharge pour :

- tous les types (`__call__`, `__str__`, ...);
- les nombres (`__add__`, `__div__`, ...);
- les séquences (`__len__`, `__iter__`, ...).

Surcharge des opérateurs (2/2)

Soient deux instances, *obj1* et *obj2*, les méthodes spéciales suivantes permettent d'effectuer les opérations arithmétiques courantes :

Surcharge des opérateurs (2/2)

Soient deux instances, *obj1* et *obj2*, les méthodes spéciales suivantes permettent d'effectuer les opérations arithmétiques courantes :

Nom	Méthode spéciale	Utilisation
opposé	<code>--neg--</code>	<code>-obj1</code>
addition	<code>--add--</code>	<code>obj1 + obj2</code>
soustraction	<code>--sub--</code>	<code>obj1 - obj2</code>
multiplication	<code>--mul--</code>	<code>obj1 * obj2</code>
division	<code>--div--</code>	<code>obj1 / obj2</code>

Exemple de surcharge

```
class Vecteur2D(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, autre): # addition vectorielle
        return Vecteur2D(self.x + autre.x, self.y + autre.y)
    def __str__(self):       # affichage d'un Vecteur2D
        return "Vecteur(%g, %g)" % (self.x, self.y)

v1 = Vecteur2D(1.2, 2.3)
v2 = Vecteur2D(3.4, 4.5)

print v1 + v2 # affiche : Vecteur(4.6, 6.8)
```

Héritage et polymorphisme

Définition

L'*héritage* est le mécanisme qui permet de se servir d'une classe préexistante pour en créer une nouvelle qui possédera des fonctionnalités différentes ou supplémentaires.

Héritage et polymorphisme

Définition

L'*héritage* est le mécanisme qui permet de se servir d'une classe préexistante pour en créer une nouvelle qui possédera des fonctionnalités différentes ou supplémentaires.

Définition

Le *polymorphisme* est la faculté pour une méthode portant le même nom mais appartenant à des classes distinctes héritées d'effectuer un travail différent. Cette propriété est acquise par la technique de la surcharge.

Exemple d'héritage et de polymorphisme

Dans l'exemple suivant, la classe `Carre` hérite de la classe `Rectangle`, et la méthode `__init__` est polymorphe :

Exemple d'héritage et de polymorphisme

Dans l'exemple suivant, la classe Carre hérite de la classe Rectangle, et la méthode `__init__` est polymorphe :

```
class Rectangle(object):
    def __init__(self, longueur=30, largeur=15):
        self.L, self.l, self.nom = longueur, largeur, "rectangle"

class Carre(Rectangle):
    def __init__(self, cote=10):
        Rectangle.__init__(self, cote, cote)
        self.nom = "carré"

r = Rectangle()
print r.nom # 'rectangle'
c = Carre()
print c.nom # 'carré'
```

Conception des classes

Suivant les relations que l'on va établir entre les objets de notre application, on peut concevoir nos classes de deux façons possibles :

- la **composition** qui repose sur la relation **a-un** ou sur la relation **utilise-un** ;

Conception des classes

Suivant les relations que l'on va établir entre les objets de notre application, on peut concevoir nos classes de deux façons possibles :

- la **composition** qui repose sur la relation **a-un** ou sur la relation **utilise-un** ;
- la **dérivation** qui repose sur la relation **est-un**.

Conception des classes

Suivant les relations que l'on va établir entre les objets de notre application, on peut concevoir nos classes de deux façons possibles :

- la **composition** qui repose sur la relation **a-un** ou sur la relation **utilise-un** ;
- la **dérivation** qui repose sur la relation **est-un**.

Bien sûr, ces deux conceptions peuvent cohabiter, et c'est souvent le cas !

Composition

Composition

Collaboration de plusieurs classes distinctes via une *association* (utilise-un) ou une *aggrégation* (a-un).

Composition

Composition

Collaboration de plusieurs classes distinctes via une *association* (utilise-un) ou une *aggrégation* (a-un).

La classe composite bénéficie de l'ajout de fonctionnalités d'autres classes qui n'ont rien en commun.

Composition

Composition

Collaboration de plusieurs classes distinctes via une *association* (utilise-un) ou une *aggrégation* (a-un).

La classe composite bénéficie de l'ajout de fonctionnalités d'autres classes qui n'ont rien en commun.

L'implémentation Python utilisée est généralement l'instanciation de classes dans le constructeur de la classe composite.

Composition : exemple

```
class Point(object):
    def __init__(self, x, y):
        self.px, self.py = x, y

class Segment(object):
    u"""Classe composite utilisant la classe distincte Point."""
    def __init__(self, x1, y1, x2, y2):
        self.orig = Point(x1, y1) # Segment "a-un" Point origine,
        self.extrem = Point(x2, y2) # et "a-un" Point extrémité

    def __str__(self):
        return "Segment : [(%g, %g), (%g, %g)]" % (self.orig.px,
            self.orig.py, self.extrem.px, self.extrem.py)

s = Segment(1.0, 2.0, 3.0, 4.0)
print s # Segment : [(1, 2), (3, 4)]
```

Dérivation

Dérivation

Décrit la création de sous-classes par spécialisation.

Dérivation

Dérivation

Décrit la création de sous-classes par spécialisation.

On utilise le mécanisme de l'*héritage*.

Dérivation

Dérivation

Décrit la création de sous-classes par spécialisation.

On utilise le mécanisme de *l'héritage*.

L'implémentation Python utilisée est généralement l'appel dans le constructeur de la classe dérivée du constructeur de la classe parente, soit nommément, soit grâce à l'instruction `super`.

Dérivation : exemple

```
class Rectangle(object):
    def __init__(self, longueur=30, largeur=15):
        self.L, self.l, self.nom = longueur, largeur, "rectangle"

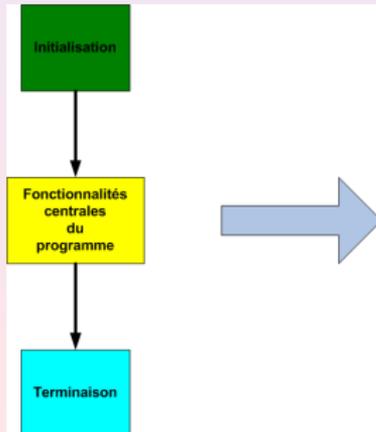
class Carre(Rectangle): # héritage simple
    u"""Sous-classe spécialisée de la super-classe Rectangle."""
    def __init__(self, cote=20):
        # appel au constructeur de la super-classe de Carre :
        super(Carre, self).__init__(cote, cote)
        self.nom = "carré" # surcharge d'attribut
```

Deux styles de programmation

En programmation graphique objet, on remplace le déroulement *séquentiel* du script par une **boucle d'événements** :

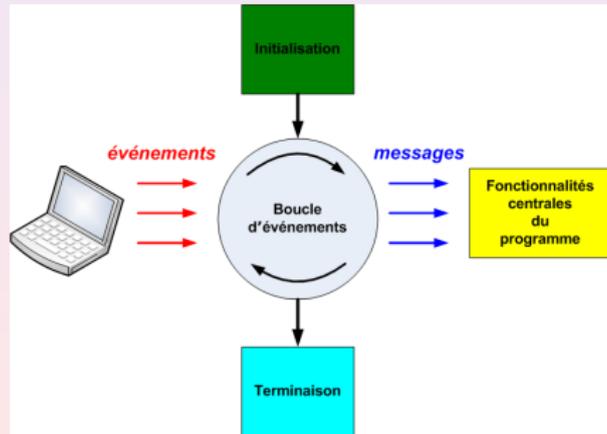
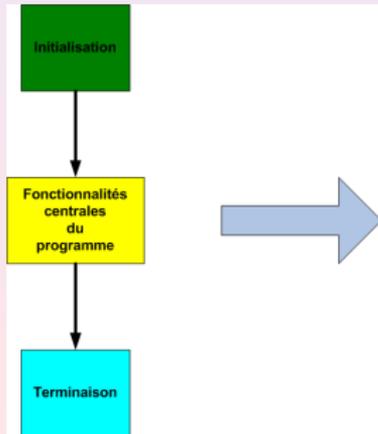
Deux styles de programmation

En programmation graphique objet, on remplace le déroulement *séquentiel* du script par une **boucle d'événements** :



Deux styles de programmation

En programmation graphique objet, on remplace le déroulement *séquentiel* du script par une **boucle d'événements** :



Les bibliothèques graphiques

- Parmi les différentes bibliothèques graphiques utilisables dans Python (GTK+, wxPython, Qt...), la bibliothèque Tkinter, issue du langage tcl/Tk est très employée, car elle est installée de base dans toute les distributions Python.

Les bibliothèques graphiques

- Parmi les différentes bibliothèques graphiques utilisables dans Python (GTK+, wxPython, Qt...), la bibliothèque Tkinter, issue du langage tcl/Tk est très employée, car elle est installée de base dans toute les distributions Python.
- Tkinter facilite la construction de *GUI* (*Graphic User Interface*) simples. Après l'avoir importé, il suffit de créer, configurer et positionner les widgets utilisés, puis d'entrer dans la boucle principale de gestion des événements.

Tkinter : exemple simple

```
from Tkinter import *

base = Tk()
texte = Label(base, text="Bienvenue !", fg="red")
texte.pack()
bouton = Button(base, text="Quit", command=base.destroy)
bouton.pack()

base.mainloop()
```

Tkinter : exemple simple

```
from Tkinter import *  
  
base = Tk()  
texte = Label(base, text="Bienvenue !", fg="red")  
texte.pack()  
bouton = Button(base, text="Quit", command=base.destroy)  
bouton.pack()  
  
base.mainloop()
```



Les widgets de Tkinter (1/3)

Tk fenêtre de plus haut niveau

Frame contenant pour organiser d'autres widgets

Label zone de message

Button bouton avec texte ou image

Message zone d'affichage multi-lignes

Entry zone de saisie

Checkbutton bouton à deux états

Radiobutton bouton à deux états exclusifs

Les widgets de Tkinter (2/3)

Scale glissière à plusieurs positions

PhotoImage sert à placer des images sur des widgets

BitmapImage sert à placer des *bitmaps* sur des widgets

Menu associé à un **Menubutton**

Menubutton bouton ouvrant un menu d'options

Scrollbar ascenseur

Listbox liste de noms à sélectionner

Text édition de texte multi-lignes

Les widgets de Tkinter (3/3)

Canvas zone de dessins graphiques ou de photos

OptionMenu liste déroulante

ScrolledText widget **Text** avec ascenseur

PanedWindow interface à onglets

LabelFrame contenant avec un cadre et un titre

Spinbox un widget de sélection multiple

Gestion de la géométrie

Tkinter possède trois gestionnaires de positionnement :

Gestion de la géométrie

Tkinter possède trois gestionnaires de positionnement :

- Le **packer** : dimensionne et place chaque widget dans un widget conteneur selon l'espace requis par chacun d'eux.

Gestion de la géométrie

Tkinter possède trois gestionnaires de positionnement :

- Le **packer** : dimensionne et place chaque widget dans un widget conteneur selon l'espace requis par chacun d'eux.
- Le **gridder** : dimensionne et positionne chaque widget dans les cellules d'un tableau d'un widget conteneur.

Gestion de la géométrie

Tkinter possède trois gestionnaires de positionnement :

- Le **packer** : dimensionne et place chaque widget dans un widget conteneur selon l'espace requis par chacun d'eux.
- Le **gridder** : dimensionne et positionne chaque widget dans les cellules d'un tableau d'un widget conteneur.
- Le **placer** : dimensionne et place chaque widget w dans un widget conteneur exactement selon ce que demande w . C'est un placement absolu (usage peu fréquent).

« Saisie » : programmation OO (1/4)

L'en-tête et les imports (on utilise le module Pmw, *Python Mega Widgets*) :

```
#!/bin/env python
# -*- coding: UTF-8 -*-
"""Programmation OO graphique."""
# fichier : 7_005.py
# auteur : Bob Cordeau

# imports
from Tkinter import *
from tkSimpleDialog import *
import Pmw
```

« Saisie » : programmation OO (2/4)

La classe App :

```
# classes
class App:
    def __init__(self, master):
        self.result = Pmw.EntryField(master, entry_width=8,
                                     value='10',
                                     label_text='Affichage retourné : ',
                                     labelpos=W, labelmargin=1)
        self.result.pack(padx=15, pady=15)
```

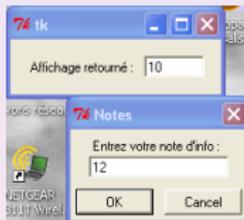
« Saisie » : programmation OO (3/4)

Le programme principal :

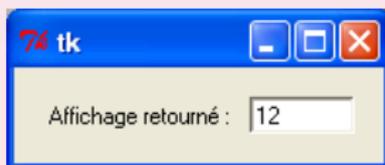
```
# programme principal -----  
root = Tk()  
display = App(root)  
retVal = askinteger("Notes",  
                    "Entrez votre note d'info :",  
                    minvalue=0, maxvalue=50)  
display.result.setentry(retVal)  
root.mainloop()
```

« Saisie » : programmation OO (4/4)

Le programme ouvre deux fenêtres. On saisit une note...



...qui est reportée dans la première fenêtre.



Annexes et Bibliographie

- La table ASCII
- Un exemple complet de script
- Un exemple de module
- Les expressions booléennes
- Les changements de base
- Bibliographie

n°	char	n°	char	n°	char	n°	char
32		56	B	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	\$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	,	63	?	87	W	111	o
40	(64	@	88	X	112	p
41)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[115	s
44	,	68	D	92	\	116	t
45	-	69	E	93]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	Δ

Un jeu de carte

L'exemple proposé programme un jeu de carte dont la règle est la suivante :

Un jeu de carte

L'exemple proposé programme un jeu de carte dont la règle est la suivante :

- On utilise un jeu de n cartes particulier : chaque carte porte un numéro, de 1 à n .

Un jeu de carte

L'exemple proposé programme un jeu de carte dont la règle est la suivante :

- On utilise un jeu de n cartes particulier : chaque carte porte un numéro, de 1 à n .
- Les cartes sont préalablement triées dans l'ordre croissant.

Un jeu de carte

L'exemple proposé programme un jeu de carte dont la règle est la suivante :

- On utilise un jeu de n cartes particulier : chaque carte porte un numéro, de 1 à n .
- Les cartes sont préalablement triées dans l'ordre croissant.
- Le jeu proprement dit consiste à :

Un jeu de carte

L'exemple proposé programme un jeu de carte dont la règle est la suivante :

- On utilise un jeu de n cartes particulier : chaque carte porte un numéro, de 1 à n .
- Les cartes sont préalablement triées dans l'ordre croissant.
- Le jeu proprement dit consiste à :
 - jeter la première carte ;

Un jeu de carte

L'exemple proposé programme un jeu de carte dont la règle est la suivante :

- On utilise un jeu de n cartes particulier : chaque carte porte un numéro, de 1 à n .
- Les cartes sont préalablement triées dans l'ordre croissant.
- Le jeu proprement dit consiste à :
 - jeter la première carte ;
 - mettre la seconde sous le paquet ;

Un jeu de carte

L'exemple proposé programme un jeu de carte dont la règle est la suivante :

- On utilise un jeu de n cartes particulier : chaque carte porte un numéro, de 1 à n .
- Les cartes sont préalablement triées dans l'ordre croissant.
- Le jeu proprement dit consiste à :
 - jeter la première carte ;
 - mettre la seconde sous le paquet ;
 - recommencer jusqu'à ce qu'il n'en reste qu'une.

Un jeu de carte

L'exemple proposé programme un jeu de carte dont la règle est la suivante :

- On utilise un jeu de n cartes particulier : chaque carte porte un numéro, de 1 à n .
- Les cartes sont préalablement triées dans l'ordre croissant.
- Le jeu proprement dit consiste à :
 - jeter la première carte ;
 - mettre la seconde sous le paquet ;
 - recommencer jusqu'à ce qu'il n'en reste qu'une.
- **Quel est le numéro de la carte restante ?**

Un en-tête général

```
#!/bin/env python          #1
# -*- coding: Latin-1 -*-  #2
"""Jeu de carte."""      #3
# fichier : a_000.py       #4
# auteur(s) : Adam et Eve #5
```

Un en-tête général

```
#!/bin/env python          #1
# -*- coding: Latin-1 -*-  #2
"""Jeu de carte."""      #3
# fichier : a_000.py       #4
# auteur(s) : Adam et Eve #5
```

- Ligne 1 : Assure le fonctionnement du script sur tous les systèmes.

Un en-tête général

```
#!/bin/env python          #1
# -*- coding: Latin-1 -*-  #2
"""Jeu de carte."""       #3
# fichier : a_000.py        #4
# auteur(s) : Adam et Eve  #5
```

- Ligne 1 : Assure le fonctionnement du script sur tous les systèmes.
- Ligne 2 : définit l'*encodage* (permet l'utilisation des accents usuels en français).

Un en-tête général

```
#!/bin/env python          #1
# -*- coding: Latin-1 -*-  #2
"""Jeu de carte."""       #3
# fichier : a_000.py       #4
# auteur(s) : Adam et Eve  #5
```

- Ligne 1 : Assure le fonctionnement du script sur tous les systèmes.
- Ligne 2 : définit l'*encodage* (permet l'utilisation des accents usuels en français).
- Ligne 3 : le *docstring* du script.

Un en-tête général

```
#!/bin/env python          #1
# -*- coding: Latin-1 -*-  #2
"""Jeu de carte."""       #3
# fichier : a_000.py       #4
# auteur(s) : Adam et Eve  #5
```

- Ligne 1 : Assure le fonctionnement du script sur tous les systèmes.
- Ligne 2 : définit l'*encodage* (permet l'utilisation des accents usuels en français).
- Ligne 3 : le *docstring* du script.
- Ligne 4 : nom du script.

Un en-tête général

```
#!/bin/env python #1
# -*- coding: Latin-1 -*- #2
"""Jeu de carte.""" #3
# fichier : a_000.py #4
# auteur(s) : Adam et Eve #5
```

- Ligne 1 : Assure le fonctionnement du script sur tous les systèmes.
- Ligne 2 : définit l'*encodage* (permet l'utilisation des accents usuels en français).
- Ligne 3 : le *docstring* du script.
- Ligne 4 : nom du script.
- Ligne 5 : noms des auteurs.

```
# programme principal -----
taille = int(raw_input("Taille du jeu de cartes : "))
while taille < 2:
    taille = int(raw_input("Taille > 1, S.V.P. : "))

# on constitue le jeu (1, 2, 3...)
t = [i+1 for i in range(taille)]

# on joue
while taille > 1:
    sauve = t[1]      # la deuxième carte
    t = t[2:]        # décalage du jeu de 2 crans à gauche
    t.append(sauve)  # sauve est mise sous le paquet
    taille -= 1      # on réduit la taille du jeu

print "\nCarte restante :", sauve
```

L'en-tête et les imports :

```
#!/bin/env python
# -*- coding: Latin-1 -*-
"""Module de vérification."""
# fichier : verif_m.py
# auteur : Bob Cordeau

# imports
from sys import exit
```

La fonction et son auto-test :

```
# fonctions
def verif(entree, reference, precision=0, comment=""):
    """Vérifie que l'entrée est égale à la référence à la
    précision près."""
    print u"%s[%s, %s]" % (comment, entree, reference),
    if abs(entree - reference) <= precision:
        print ": ok"
    else:
        print "### ERREUR ! ###"
        exit(1)

# auto-test -----
if __name__=='__main__':
    def f(x):
        return 7*x-5
    verif(f(2), 9, comment="Teste la fonction f : ")
```

Les variables logiques

Au 19^e siècle, le logicien et mathématicien George BOOLE restructura complètement la logique en un système formel. Aujourd'hui, l'algèbre de BOOLE trouve de nombreuses applications en informatique et dans la conception des circuits électroniques.

Les variables logiques

Au 19^e siècle, le logicien et mathématicien George BOOLE restructura complètement la logique en un système formel. Aujourd'hui, l'algèbre de BOOLE trouve de nombreuses applications en informatique et dans la conception des circuits électroniques.

C'est une logique à deux valeurs. Soit $\mathcal{B} = \{0, 1\}$, l'ensemble de définition, sur lequel on définit les opérateurs NON, ET et OU.

Les variables logiques

Au 19^e siècle, le logicien et mathématicien George BOOLE restructura complètement la logique en un système formel. Aujourd'hui, l'algèbre de BOOLE trouve de nombreuses applications en informatique et dans la conception des circuits électroniques.

C'est une logique à deux valeurs. Soit $\mathcal{B} = \{0, 1\}$, l'ensemble de définition, sur lequel on définit les opérateurs NON, ET et OU.

Les valeurs des variables sont parfois notées False et True.
Les opérateurs sont parfois notés respectivement \bar{a} , $a.b$ et $a + b$.

Les *tables de vérité* des opérateurs booléens

Table de vérité des opérateurs booléens de base :

Les tables de vérité des opérateurs booléens

Table de vérité des opérateurs booléens de base :

Opérateur unaire NON

a	$NON(a)$
0	1
1	0

Opérateurs binaires OU et ET

a	b	a OU b	a ET b
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Les *tables de vérité* des opérateurs composés

Table de vérité des opérateurs ou exclusif, équivalence et implication :

Les tables de vérité des opérateurs composés

Table de vérité des opérateurs ou exclusif, équivalence et implication :

Opérateurs composés

a	b	$a \text{ XOR } b$	$a \iff b$	$a \implies b$
0	0	0	1	1
0	1	1	0	1
1	0	1	0	0
1	1	0	1	1

Les bases arithmétiques

Définition

En arithmétique, une **base** désigne la valeur dont les puissances successives interviennent dans l'écriture des nombres, ces puissances définissant l'ordre de grandeur de chacune des positions occupées par les chiffres composant tout nombre

Les bases arithmétiques

Définition

En arithmétique, une **base** désigne la valeur dont les puissances successives interviennent dans l'écriture des nombres, ces puissances définissant l'ordre de grandeur de chacune des positions occupées par les chiffres composant tout nombre

Certaines bases sont couramment employées :

- la base 2 (système binaire), en électronique numérique et informatique ;

Les bases arithmétiques

Définition

En arithmétique, une **base** désigne la valeur dont les puissances successives interviennent dans l'écriture des nombres, ces puissances définissant l'ordre de grandeur de chacune des positions occupées par les chiffres composant tout nombre

Certaines bases sont couramment employées :

- la base 2 (système binaire), en électronique numérique et informatique ;
- la base 8 (système octal), en informatique ;

Les bases arithmétiques

Définition

En arithmétique, une **base** désigne la valeur dont les puissances successives interviennent dans l'écriture des nombres, ces puissances définissant l'ordre de grandeur de chacune des positions occupées par les chiffres composant tout nombre

Certaines bases sont couramment employées :

- la base 2 (système binaire), en électronique numérique et informatique ;
- la base 8 (système octal), en informatique ;
- la base 16 (système hexadécimal), fréquente en informatique ;

Les bases arithmétiques

Définition

En arithmétique, une **base** désigne la valeur dont les puissances successives interviennent dans l'écriture des nombres, ces puissances définissant l'ordre de grandeur de chacune des positions occupées par les chiffres composant tout nombre

Certaines bases sont couramment employées :

- la base 2 (système binaire), en électronique numérique et informatique ;
- la base 8 (système octal), en informatique ;
- la base 16 (système hexadécimal), fréquente en informatique ;
- la base 60 (système sexagésimal), dans la mesure du temps et des angles.

Les changements de base

Méthode

Un nombre dans une base n donnée s'écrit sous la forme d'addition des puissances successives de cette base.

Les changements de base

Méthode

Un nombre dans une base n donnée s'écrit sous la forme d'addition des puissances successives de cette base.

Par exemple :

$$57_{16} = (5 \times 16^1) + (7 \times 16^0) = 87_{10}$$

$$57_8 = (5 \times 8^1) + (7 \times 8^0) = 47_{10}$$



G rard SWINNEN

Apprendre   programmer avec Python

O'Reilly, 2005.



Gérard SWINNEN

Apprendre à programmer avec Python

O'Reilly, 2005.



Mark LUTZ

Python précis et concis

O'Reilly, 2005.



Gérard SWINNEN

Apprendre à programmer avec Python

O'Reilly, 2005.



Mark LUTZ

Python précis et concis

O'Reilly, 2005.



Alex MARTELLI

Python en concentré

O'Reilly, 2004.



Gérard SWINNEN

Apprendre à programmer avec Python

O'Reilly, 2005.



Mark LUTZ

Python précis et concis

O'Reilly, 2005.



Alex MARTELLI

Python en concentré

O'Reilly, 2004.



Wesley J. CHUN

Au cœur de Python. Version 2.5

CampusPress, 2007.