Langage Python

Table des matières

Python est un langage de programmation objet, multi-paradigme et multiplateformes. Il est placé sous une licence libre et fonctionne sur la plupart des plates-formes informatiques, des supercalculateurs aux ordinateurs centraux, de Windows à Unix en passant par GNU/Linux, Mac OS, ou encore Android.

1.Introduction

1.1. L'interpréteur Python

Il existe deux manières de coder en Python. La première, est d'écrire des fichiers de code source (dont l'extension est .py), puis d'exécuter ces fichiers. La seconde, est d'utiliser l'interpréteur de commandes.

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600 32 bit (In
tel)] on win32 Type "help", "copyright", "credits" or "license" for more information.
>>>
```

1.2. Mon premier programme

Nous allons écrire un programme en mode console qui affichera « Hello world! ».

```
>>> print("Hello World !")
```

On peut écrire une chaîne de caractères de différentes façons :

- entre guillemets ("ceci est une chaîne de caractères")
- entre triples guillemets ("""ceci est une chaîne de caractères""")
- entre apostrophes ('ceci est une chaîne de caractères')

Si cette dernière chaîne contient un apostrophe, il faut utiliser le caractère d'échappement « \ ».

```
chaine = 'J\'aime le Python!'
```

Les caractères spéciaux commencent par un anti-slash « \ » suivis d'une lettre. Exemple :

- \n : retour à la ligne
- \t : tabulation

La fonction print est dédiée à l'affichage uniquement. Le nombre de ses paramètres est variable, c'est-à-dire qu'on peut lui demander d'afficher une ou plusieurs variables.

```
>>> a = 3
>>> print(a)
>>> a = a + 3
>>> b = a - 2
>>> print("a =", a, "et b =", b)
```

1.3. Les commentaires

Un commentaire permet d'ajouter des annotations dans le codes source et de mieux vous expliquer à quoi peut servir telle ou telle ligne de code.

Il y a plusieurs manières d'insérer un commentaire selon la longueur du commentaire.

Ceci est un commentaire sur une ligne

ou

'' Ceci est un commentaire sur une ligne ''

ou

""" Ceci est
un commentaire
sur plusieurs lignes """

2. Les variables

2.1. Déclarer une variable

Une variable est constituée :

- d'une valeur : par exemple le nombre qu'elle stocke.
- d'un nom : c'est ce qui permet de la manipuler.

```
>>> a = 5
>>> b = 32
>>> a,b = b,a # permutation
>>> a
32
>>> b
5
>>>
```

Un nom de variable ne peut contenir que des lettres et des chiffres et doit commencer par une lettre ; les espaces et accents sont interdits et le langage est sensible à « la casse ».

Python utilise différents types de variables : les nombres entiers (int), les nombres flottants (float), booléen (bool), complexe (complex) et les chaînes de caractères (str).

Remarque : dans l'exemple ci-dessus, on constate que l'affectation en Python peut concerner plusieurs variables à la fois. Cette technique de sequence unpacking a pour seules contraintes que :

- le terme à droite du signe = est un iterable (tuple, liste, string, ...),
- le terme à gauche soit écrit comme un tuple ou une liste,
- les deux termes ont la même longueur.

2.2. Afficher le type d'une variable

La fonction type renvoie le type de la variable passée en paramètre.

```
>>> type(3.4)
<class 'float'>
>>> type("un essai")
<class 'str'>
>>>
```

2.3. Récupérer une saisie

La fonction input() interagit avec l'utilisateur : cette instruction interrompt le programme et attend la saisie de l'utilisateur. input() accepte un paramètre facultatif : le message à afficher à l'utilisateur.

```
>>> # Test de la fonction input
>>> annee = input("Saisissez une année : ")
Saisissez une année : 2009
>>> print(annee)
'2009'
>>>
```

Le type de la variable annee après l'appel à input() est une chaîne de caractères (apostrophes qui encadrent la valeur de la variable). Pour convertir une variable vers un autre type, il faut utiliser le nom du type comme une fonction.

```
>>> type(annee)
<type 'str'>
>>> # On veut convertir la variable en un entier, on utilise
>>> # donc la fonction int qui prend en paramètre la variable
>>> # d'origine
>>> annee = int(annee)
>>> type(annee)
<type 'int'>
>>> print(annee)
2009
>>>
```

Exemple :

```
# Programme testant si une année, saisie par l'utilisateur, est bissextile ou non

annee = input("Saisissez une année : ")        # saisie de l'année à tester
annee = int(annee)          # Risque d'erreur si l'utilisateur n'a pas saisi un nombre

if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
    print "L'année saisie est bissextile."
else:
    print "L'année saisie n'est pas bissextile."
```

Remarques :

- La fonction input() renvoie une valeur dont le type correspond à ce que l'utilisateur a entré. Si l'utilisateur souhaite entrer une chaîne de caractères, il doit l'entrer comme telle, c'est-à-dire incluse entre des apostrophes ou des guillemets.
- Pour cette raison, il sera souvent préférable d'utiliser dans vos scripts la fonction similaire raw_input(), laquelle renvoie toujours une chaîne de caractères. Vous pouvez ensuite convertir cette chaîne en nombre à l'aide de int() ou de float().

3. Les opérations de base

Opération	**Signe**	**Remarque**
Addition	+	
Soustraction	-	
Multiplication	*	
Division	/	Division entière
Modulo	%	Reste de la division entière

Décalage gauche[1]	<<	Multiplication par 2
Décalage droite	>>	Division par 2
and	&	Opération bit à bit
or	|	
xor	^	
Partie réelle	real	Méthode appliquée à un nombre complexe
Partie imaginaire	imag	Méthode appliquée à un nombre complexe

Attention à respecter les types lors des opérations : diviser deux entiers donnera un entier même si la variable qui reçoit le résultat est un flottant.

```
a, b = 5, 2 + 3j
c = float(a) / float(b.real)  # au moins un des deux nombres doit être convertit
print c
```

Comme pour les entiers, les calculs sur les flottants sont, naturellement, réalisés par le processeur. Cependant contrairement au cas des entiers où les calculs sont toujours exacts, les flottants posent un problème de précision. Cela n'est pas propre au langage python, mais est dû à la technique de codage des nombres flottants sous forme binaire.

Si le problème se pose bien en termes de nombres rationnels, il est alors tout à fait possible de le résoudre avec exactitude avec le module decimal qui permet de résoudre le problème.

from decimal import Decimal

Decimal('0.3') - Decimal('0.1') == Decimal('0.2')

3.1. Raccourcis d'écriture

Il existe des techniques permettant de raccourcir l'écriture des opérations.

Au lieu d'écrire : nombre = nombre + 1

on écrira : nombre += 1

De même pour : nombre = nombre - 1

on pourra écrire : nombre -= 1

Il existe d'autres raccourcis qui fonctionnent sur le même principe qui fonctionnent pour toutes les opérations de base :

```
nombre = 9
nombre *= 5;          // ... nombre vaut 45
nombre /= 3;          // ... nombre vaut 15
nombre %= 2;          // ... nombre vaut 1 (car 15 = 2 x 7 + 1)
```

3.2. La bibliothèque mathématique

En langage Python, il existe ce qu'on appelle des bibliothèques « standard », c'est-à-dire des bibliothèques toujours utilisables. Ce sont en quelque sorte des bibliothèques « de base » qu'on

1 Voir cours 'Systèmes numériques'

utilise très souvent.

Les fonctions intégrées au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des fichiers séparés que l'on appelle des modules.

Il existe un grand nombre de modules pré-programmés qui sont fournis d'office avec Python. Vous pouvez en trouver d'autres chez divers fournisseurs. Souvent on essaie de regrouper dans un même module des ensembles de fonctions apparentées que l'on appelle des bibliothèques.

Le module math, par exemple, contient les définitions de nombreuses fonctions mathématiques telles que sinus, cosinus, tangente, racine carrée, etc. Pour pouvoir utiliser ces fonctions, il vous suffit d'incorporer la ligne suivante au début de votre script :

from math import *

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant toutes les fonctions (c'est là la signification du symbole *) du module math, lequel contient une bibliothèque de fonctions mathématiques pré-programmées.

Voici quelques fonctions principales :

fonction	**description**
fabs[2]	Cette fonction retourne la valeur absolue d'un nombre flottant, c'est-à-dire |x|.
ceil	Cette fonction renvoie le premier nombre entier après le nombre décimal qu'on lui donne.
floor	C'est l'inverse de la fonction précédente : elle renvoie le nombre directement en dessous.
pow	Cette fonction permet de calculer la puissance d'un nombre.
sqrt	Cette fonction calcule la racine carrée d'un nombre.
sin, cos, tan	Ce sont les trois fonctions sinus, cosinus et tangeante utilisées en trigonométrie. Ces fonctions attendent une valeur en radians.
asin, acos, atan	Ce sont les fonctions arc sinus, arc cosinus et arc tangente utilisées en trigonométrie.
exp	Cette fonction calcule l'exponentielle d'un nombre.
log	Cette fonction calcule le logarithme népérien d'un nombre.
log10	Cette fonction calcule le logarithme base 10 d'un nombre.

4. Les conditions

Une condition peut être écrite en Python sous différentes formes. On parle de structures conditionnelles.

4.1. if… else

Les symboles suivant permettent de faire des comparaisons simples ou multiples pour les conditions.

2 Équivalent à la fonction abs de la librairie standard stdlib.h (ne foctionne que sur des entiers)

Symbole	Signification
==	Est égal à
>	Est supérieur à
<	Est inférieur à
>=	Est supérieur ou égal à
<=	Est inférieur ou égal à
!=	Est différent de

Symbole	Signification
and	ET logique
or	OU logique

Pour introduire une condition :

1. on utilise le mot if, qui en anglais signifie « si »
2. on ajoute à la suite la condition en elle-même, suive de :
3. on indente les instructions à exécuter si la condition est **vraie**
4. **Facultativement** : on utilise le mot else, qui en anglais signifie « sinon », suivi de :
5. puis on indente les instructions à exécuter si la condition est **fausse**.

```
Age = 8             # initialisation de la variable

if age < 12:        # test
    print "Salut gamin !"     # action si condition vraie
else:               # SINON
    """ action si condition fausse """
```

Dans le cas des conditions multiples, on utilise les opérateurs logiques.

```
Age = 8             # initialisation des variables
garcon = False      # attenstion à la Majuscule !

if age < 12 and garcon:         # test
    print "Salut Jeune homme"       # action si condition vraie
else:
    if age < 12 and garcon == False:        # SINON
       print "Bonjour Mademoiselle"         # action si condition vraie
```

Le mot clé elif est une contraction de « else if », que l'on peut traduire très littéralement par « sinon si ». Dans l'exemple que nous venons juste de voir, l'idéal serait d'écrire :

```
if age < 12 and garcon:         # test
    print "Salut Jeune homme"       # action si condition vraie
elif age < 12 and garcon == False:          # SINON SI
       print "Bonjour Mademoiselle"         # action si condition vraie
```

4.2. Conditions imbriquées

Les structures à base de if… else suffisent pour traiter n'importe quelle condition.

Pour une imbrication de if... else trop importante, on utilisera : elif... else.

```
jour = "mercredi"

if  jour == "lundi" or jour == "mardi":          # début de la semaine
    print "courage !!!"
elif  jour == "mercredi":                        # milieu de la semaine
```

```
    print "c'est le jour des enfants"
elif  jour == "jeudi" or jour == "vendredi":    # fin de la semaine
    print "bientôt le we !"
else:    # il faut traiter les autres jours (cas par défaut)
    print "vive le week end !"
```

Le mot-clé else: est le traitement par défaut quelle que soit la valeur de la variable.

5. Les boucles

5.1. Tant que : while

Une boucle permet de répéter des instructions plusieurs fois.

- les instructions sont d'abord exécutées dans l'ordre, de haut en bas
- à la fin des instructions, on retourne à la première
- et ainsi de suite...

Tant que la condition est remplie, les instructions sont réexécutées. Dès que la condition n'est plus remplie, on sort de la boucle.

```
nombre_de_lignes = 1

# on boucle tant que on n'est pas arrivé à 100 lignes
while nombre_de_lignes <= 100:
    print("ligne n", nombre_de_lignes)    # affiche le n° de ligne
    nombre_de_lignes += 1                 #incrément du nombre de ligne
```

Il faut TOUJOURS s'assurer que la condition sera fausse au moins une fois. Si elle ne l'est jamais, alors la boucle s'exécutera à l'infini !

Nous avons vu qu'en python les blocs de code sont définis par leur indentation. Cette convention a une limitation lorsqu'on essaie de définir un bloc vide. Pour cela, il existe une instruction pass, qui ne fait rien et que l'on évitera d'utiliser !!

```
liste = range(10)

print 'avant', liste
while liste.pop() != 5:
    pass
print 'après', liste
```

L'assertion de ce test est difficilement évaluable quant à la condition de terminaison.

Une erreur surviendra si `liste = range(4)`

On utilisera plutôt :

```
liste = range(10)

print 'avant', liste
for i in range(5):
    if i > len(liste):
        liste.pop()
print 'après', liste
```

5.2. Faire Tantque : do while

Il n'existe pas de boucle Faire... Tanque en Python !

5.3. Boucle : for

Le but de la boucle for est de répéter certaines instructions pour chaque élément d'une liste. Contrairement à d'autres langages, l'usage d'une liste est donc nécessaire !

```
# on boucle tant que on n'est pas arrivé à 100 lignes
for ligne in range (1, 101):
```

```
    print("ligne n", ligne)    # affiche le n° de ligne
    ligne += 1                 # incrément du nombre de ligne
```

Cette boucle est utilisée lorsque l'on connait à l'avance le nombre d'instructions à répéter.

Remarque : les chaînes de caractères sont des séquences… de caractères que l'on peut parcourir.
Exemple :

```
# ce programme filtre les voyelles d'une chaîne de caractères
chaine = "Bonjour"

for lettre in chaine:
    if lettre in "AEIOUYaeiouy":    # lettre est une voyelle
        print lettre
    else:                           # la lettre n'est pas une voyelle
        print "*"
```

Remarque : dans une boucle for, sur un objet mutable, il ne faut pas modifier le sujet de la boucle. Lorsqu'on est dans ce cas de figure, il suffit de faire la boucle sur une shallow copy[3] de l'objet grâce à la fonction copy.

```
from copy import copy
# on veut enlever de l'ensemble toutes les chaines
# qui ne commencent pas par 'a'
ensemble = {'marc', 'albert'}

# si on fait d'abord une copie tout va bien
for valeur in copy(ensemble):
    if valeur[0] != 'a':
        ensemble.discard(valeur)

print ensemble
```

6. Les tableaux

En Python, les tableaux sont traités comme des listes qui peuvent contenir une mixité de types de variables. Ils contiennent les références aux objets mais pas les objets eux-mêmes.

6.1. Déclarer un tableau

Pour déclarer un tableau, il faut respecter la syntaxe suivante :

<nom du tableau> = [<contenu du tableau>]

Ex : a = [4,7,9,15] # tableau de 4 entiers

Il est possible d'initialiser un tableau d'entiers avec la fonction range.

Ex : a = range(10) # crée le tableau a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

6.2. Parcourir un tableau

Pour accéder à un élément du tableau, il suffit de donner son indice dans le tableau. Il faut se souvenir que :

- les indices commencent comme toujours à zéro
- le premier indice debut est inclus

3 copie superficielle

- le second indice fin est exclu
- on obtient en tout fin-debut items dans le résultat

```
a = [4, 'toto', 9.2, True]          # déclaration du tableau
for i in range(0,4):                # ou bien : for i in range(0,len(a)):
        print a[i]
```

Un slice permet de désigner toute une plage d'éléments d'une séquence :

chaine = "abcdefghijklmnopqrstuvwxyz"

```
c = chaine[2:6];        print c     # affiche : cdef
c = chaine[:3];         print c     # affiche : abc (l'indice de départ peut être omis)
c = chaine[3:];         print c     # affiche : defghijklmnopqrstuvwxyz
c = chaine[-3:];        print c     # affiche : xyz (si indice < 0, on commence par la fin)
c = chaine[3:-3];       print c     # affiche : defghijklmnopqrstuvw
c = chaine[3:-3:2];     print c     # affiche : dfhjlnprtv (le 3ème argument indique le pas)
```

La fonction enumerate permet d'itérer sur une liste avec l'indice dans la liste :

```
villes = ["Paris", "Nice", "Lyon"]
for i, ville in enumerate(villes):
        print i, ville
```

6.3. Tableaux multi dimensionnels

Un tableau multidimensionnel n'est rien d'autre qu'un tableau contenant au minimum deux tableaux.

Chaque dimension est désignée par des crochets [].

```
a = [
    [4,'toto',9],
    ['a',4,3]
    ]

for i in range(len(a)): # dimension 2, NB : length renvoie la taille du tableau
    for j in range(len(a[i])):      # dimension 3
        print a[i][j]               # affichage ligne, colonne
```

6.4. Opérations sur les listes

On dispose de plusieurs méthodes pour ajouter des éléments dans une liste :

- append(<élément>) : ajouter un élément à la fin de la liste
- insert(<indice>, <élément>) : insérer un élément à l'indice <indice> dans la liste

Exemple :

```
a = [10,20,30]      # déclaration du tableau
a.append(40)        # ajoute 40 en fin de liste
a.insert(1,15)      # ajoute 15 en 2ème position

for i in range(0,len(a)):
    print a[i]
```

On dispose de plusieurs méthodes pour supprimer des éléments dans une liste :

- remove(<élément>) : supprime l'élément <élément> de la liste
- pop(<élément>) : extrait et supprime l'élément <élément> de la liste
- del tableau[<indice>] : supprime un élément à l'indice <indice> dans la liste tableau

```
a = [10,15,20,25,30,40] # déclaration du tableau
a.remove(40)       # supprime l'élément 40
a.pop(3)           # supprime l'élément 40
del a[1]           # supprime l'élément en 2ème position
```

La méthode reverse renverse la liste, le premier élément devient le dernier

La méthoe sort tri une liste par ordre croissant ou décroissant pour sort(reverse = True).

```
a = [20,10,30]           # déclaration du tableau
a.sort()           # tri la liste
a.reverse()        # renverse la liste

for i in range(0,len(a)):
    print a[i]
```

Une liste étant maillable, on peut ajouter ou supprimer une série d'élements avec un slice :

```
a = [4,'toto',9.2,True]        # déclaration du tableau
# remplace élément 'toto' par 3 éléments 'pim','pam','poum'
a[1:2] = ['pim','pam','poum']
print a
```

7. Les tuple

Les tuple sont des listes immuables. On les déclare de la même façon qu'un tableau mais il faut mettre des paranthèses à la place des crochets.

Ex : t = (4,'toto',9.2,True) # déclaration du tuple

En réalité, la parenthèse est superflue, il se trouve toutefois qu'elle est largement utilisée pour améliorer la lisibilité des programmes.

Remarque : pour un tuple avec un seul élément, il est nécessaire de rajouter une virgule.

Les tuple étant immuable, il faut les convertir en liste pour effectuer dessus des opérations.

Ex : ma_liste = list(t)

8. Les fonctions

Une fonction est une série d'instructions qui effectue des actions qui sont répétées plusieurs fois dans le code sans avoir à les réécrire à chaque fois.

Lorsqu'on appelle une fonction, il y a trois étapes.

1. L'entrée: on donne des informations à la fonction en lui passant des paramètres avec lesquelles travailler).
2. Le traitement : grâce aux informations qu'elle a reçues en entrée.
3. La sortie : une fois qu'elle a fini son traitement, la fonction renvoie un résultat.

La syntaxe pour coder un fonction est donc :

```
def nomFonction(parametres):
        // Insérez vos instructions ici
```

- def, mot-clé qui est l'abréviation de « define » (définir, en anglais) et qui constitue le prélude à toute construction de fonction.
- Le nom de la fonction, qui se nomme exactement comme une variable. N'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.
- La liste des paramètres qui seront fournis lors d'un appel à la fonction. Les paramètres sont séparés par des virgules et la liste est encadrée par des parenthèses ouvrante et fermante.
- Les deux points qui clôturent la ligne.

Remarque : Les parenthèses sont obligatoires, quand bien même votre fonction n'attendrait aucun paramètre.

Exemple : on veut afficher le cube des nombres de 1 à 9.

```
def cube(valeur):
    """
    function:     calcule le cube d'une valeur passée en paramètre
    in:           valeur, valeur à calculer
    out:          résultat du paramètre élevé au cube
    """
    return valeur * valeur * valeur    # il faut utiliser le mot-clef return

# programme principal
for i in range (1, 10):
    print("la valeur de ", i, "^3 est ", cube(i))    # appel de la fonction
```

Remarque : nous avons placé une chaîne de caractères, sans la capturer dans une variable, juste en-dessous de la définition de la fonction. Cette chaîne est ce qu'on appelle une docstring que l'on pourrait traduire par une chaîne d'aide. Si vous tapez help(cube), c'est ce message que vous verrez apparaître. Documenter vos fonctions est également une bonne habitude à prendre. Comme vous le voyez, on indente cette chaîne et on la met entre triple guillemets. Si la chaîne figure sur une seule ligne, on pourra mettre les trois guillemets fermants sur la même ligne ; sinon, on préférera sauter une ligne avant de fermer cette chaîne, pour des raisons de lisibilité. Tout le texte d'aide est indenté au même niveau que le code de la fonction.

On peut appeler des paramètres par leur nom. Cela est utile pour une fonction comptant un certain nombre de paramètres qui ont tous une valeur par défaut.

Soit la fonction :

```
def ma_fonction(a=1, b=2, c=3, d=4, e=5):
    print("a =", a, "b =", b, "c =", c, "d =", d, "e =", e)
```

Il existe de nombreuses façons d'appeler cette fonction :

Instruction	**Résultat**
ma_fonction()	a = 1 b = 2 c = 3 d = 4 e = 5
ma_fonction(4)	a = 4 b = 2 c = 3 d = 4 e = 5
ma_fonction(b=8, d=5)	a = 1 b = 8 c = 3 d = 5 e = 5
ma_fonction(b=35, c=48, a=4, e=9)	a = 4 b = 35 c = 48 d = 4 e = 9

6.1. Fonctions de manipulation des chaînes

Les chaînes de caractères sont des objets fréquemment utilisés. Le langage Python fournit une pléthore de méthodes dédiées aux opérations sur des chaînes. Voici les principales :

méthode	**description**
split	découpe une chaine selon un séparateur pour obtenir une liste
join	reconstruit une chaîne à partir d'une liste
replace	remplace une sous-chaîne par une autre, avec une limite éventuelle sur le nombre de remplacements
strip	enlève les espaces dans une chaîne
find	cherche une sous-chaîneet renvoie le plus petit index où on trouve la sous-chaîne
rfind	identique à find mais en partant de la fin de la chaîne
count	compte le nombre d'occurrences d'une sous-chaîne
upper	Transforme une chaîne de caractères en majuscules
lower	Transforme une chaîne de caractères en minuscules
swapcase	Échange les majuscules en minuscules et inversement

8.2. Les modules

Un module contient des fonctions et des variables ayant toutes un rapport entre elles. Il n'y a qu'à importer le module et utiliser ensuite toutes les fonctions et variables prévues. La syntaxe est facile à retenir : le mot-clé import, qui signifie « importer » en anglais, suivi du nom du module.

Il existe un grand nombre de modules disponibles avec Python sans qu'il soit nécessaire d'installer des bibliothèques supplémentaires. La fonction help permet de connaître les fonctions contenues dans un module.

Exemple :

```
>>> help("math")
Help on built-in module math:

NAME
    math

FILE
    (built-in)

DESCRIPTION
    This module is always available. It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the hyperbolic arc cosine (measured in radians) of x.

    asin(...)
-- Suite  --
```

Pour connaître la description d'une fonction contenue dans un module, il suffit de passer son nom en paramètre de la fonction help.

```
>>> help("math.sqrt")
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.

>>>
```

import math crée un espace de noms dénommé « math », contenant les variables et fonctions du module math. Quand on lance math.sqrt(25), Python exécute la fonction sqrt contenue dans l'espace de noms math. Ainsi, les variables et fonctions sont stockées à part, à l'abri dans un espace de noms, sans risque de conflit avec vos propres variables et fonctions. Mais dans certains cas, il peut être nécessaire de changer le nom de l'espace de noms dans lequel sera stocké le module importé.

```
import math as mathematiques
mathematiques.sqrt(25)
```

Le module math est importé dans l'espace de noms dénommé « mathematiques » au lieu de math. Cela permet de mieux contrôler les espaces de noms des modules.

Il existe une autre méthode d'importation qui permet de ne plus préfixer le nom des fonctions. Celle-ci charge la fonction depuis le module indiqué et la place dans l'interpréteur au même plan que les fonctions existantes, comme print par exemple.

```
>>> from math import *
>>> sqrt(4)
2
>>> fabs(5)
5
```

On peut appeler toutes les variables et fonctions d'un module en tapant « * » à la place du nom de la fonction à importer.

8.3. Écriture de modules

La création de module consiste à enregistrer les scripts dans un fichier d'extension .py. Ce fichier devra être ensuite importé dans le corps du programme principal. Si le programme contient des accents, il est nécessaire de préciser à Python l'encodage de ces accents (utf-8 par exemple).

Exemple : fichier bissextile.py

```
def bissextile():
    """ Programme testant si une année, saisie par l'utilisateur, est bissextile
ou non """

    annee = input("Saisissez une année : ")     # année à tester
        annee = int(annee)                      # on attend un nombre

    if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
        print "L'année saisie est bissextile."
    else:
        print "L'année saisie n'est pas bissextile."
```

Exemple : fichier main.py

```
# -*-coding:utf_8 -*
import os    # On importe le module os qui dispose de variables
        # et de fonctions utiles pour dialoguer avec votre
        # système d'exploitation
from bissextile import *

bissextile()

# On met le programme en pause pour éviter qu'il ne se referme (Windows)
os.system("pause")
```

8.4. Les packages

Les packages son des répertoires dans lesquels peuvent se trouver d'autres répertoires (d'autres packages) ou des fichiers (des modules).

Exemple de hiérarchie de répertoires et de fichiers :

- un répertoire du nom de la bibliothèque contenant :
  - un répertoire evenements contenant :
    - un module clavier ;
    - un module souris ;
    - …
  - un répertoire effets contenant différents effets graphiques ;

- un répertoire objets contenant les différents objets graphiques de notre fenêtre (boutons, zones de texte, barres de menus…).

Pour utiliser la bibliothèque il existe plusieurs moyens :

1. import nom_bibliotheque

   Cette ligne importe le package contenant la bibliothèque. Pour accéder aux sous-packages, il faut utiliser un point « . » afin de modéliser le chemin menant au module ou à la fonction.

   nom_bibliotheque.evenements # Pointe vers le sous-package evenements

   nom_bibliotheque.evenements.clavier # Pointe vers le module clavier

2. from nom_bibliotheque.objets import bouton

   Pour importer qu'un seul module (ou qu'une seule fonction) d'un package :

Pour créer un package, il suffit donc de créer un répertoire portant le nom du package puis y déposer soit des modules ou d'autres dossirs (packages).

Pour définir un package, il faut obligatoirement créer dans le répertoire (celui, donc, que l'on veut exposer à python), un fichier nommé __init__.py. Le code placé dans __init__.py est chargé d'initialiser la librairie. Le fichier peut être vide mais doit absolument exister !

9. Les exceptions

9.1. Le bloc try

Python lève des exceptions quand il trouve une erreur dans le code (erreur de syntaxe ou exécution d'une opération) et s'interrompt. Pour éviter de faire planter le programme on utilise alors les mécanismes d'exceptions.

Nous allons mettre les instructions à tester dans un premier bloc et les instructions à exécuter en cas d'erreur dans un autre bloc.

```
try:
        # Bloc à essayer
except:
        # Bloc qui sera exécuté en cas d'erreur
```

Dans l'ordre, nous trouvons :

- Le mot-clé try suivi des deux points « : » : bloc d'instructions à essayer.
- Le mot-clé except suivi des deux points « : » : bloc d'instructions qui sera exécuté si une erreur est trouvée dans le premier bloc. Il se trouve au même niveau d'indentation que try.

Cette méthode est assez grossière car elle intercepte n'importe quelle exception liée à cette instruction et Python peut lever des exceptions qui ne signifient pas nécessairement qu'il y a eu une erreur.

Exemple :

```
try:
    resultat = numerateur / denominateur
except:
    print "Une erreur est survenue... laquelle ?"
```

Ici, plusieurs erreurs sont susceptibles d'intervenir, chacune levant une exception différente :

- NameError : l'une des variables numerateur ou denominateur n'a pas été définie (elle n'existe pas).
- TypeError : l'une des variables numerateur ou denominateur ne peut diviser ou être divisée (les chaînes de caractères ne peuvent être divisées, ni diviser d'autres types, par exemple).
- ZeroDivisionError : si denominateur vaut 0, cette exception sera levée.
- ...

On peut préciser le type de l'exception à traiter au niveau du bloc except.

```
try:
    resultat = numerateur / denominateur
except NameError:
    print "La variable numerateur ou denominateur n'a pas été définie."
except TypeError:
    print "La variable numerateur ou denominateur possède un type incompatible."
except ZeroDivisionError:
    print "La variable denominateur est égale à 0."
```

Dans un bloc try, le mot -clé else va permettre d'exécuter une action si aucune erreur ne survient dans le bloc. Voici un exemple :

```
try:
    resultat = numerateur / denominateur
except NameError:
    print "La variable numerateur ou denominateur n'a pas été définie."
except TypeError:
    print "La variable numerateur ou denominateur possède un type incompatible."
except ZeroDivisionError:
    print "La variable denominateur est égale à 0."
else:
    print("Le résultat obtenu est", resultat)
```

Enfin, le mot-clé finally permet d'exécuter du code après un bloc try, quelle que soit le résultat de l'exécution dudit bloc.

```
try:
    # Test d'instruction(s)
except TypeDeLException:
    # Traitement en cas d'erreur
finally:
    # Instruction(s) exécutée(s) qu'il y ait eu des erreurs ou non
```

9.2. Les assertions

Les assertions sont un moyen simple de s'assurer, avant de continuer, qu'une condition est respectée. En général, on les utilise dans des blocs try … except. Sa syntaxe est la suivante :

assert test

Si le test renvoie True, l'exécution se poursuit normalement. Sinon, une exception AssertionError est levée.

Exemple : dans le programme testant si une année est bissextile, on pourrait vouloir s'assurer que l'utilisateur ne saisit pas une année inférieure ou égale à 0.

```
annee = input("Saisissez une année supérieure à 0 :")
```

```
try:
    annee = int(annee)     # Conversion de l'année
    assert annee > 0
except ValueError:
    print "Vous n'avez pas saisi un nombre."
except AssertionError:
    print "L'année saisie est inférieure ou égale à 0."
```

9.3. Lever une exception

Pour lever une exception, il faut utiliser le mot-clé raise. Sa syntaxe est la suivante :

raise TypeDeLException("message à afficher")

Exemple :

```
annee = input()     # L'utilisateur saisit l'année
try:
    annee = int(annee) # On tente de convertir l'année
    if annee <= 0:
        raise ValueError("l'année saisie est négative ou nulle")
except ValueError:
    print "La valeur saisie est invalide (l'année est peut-être négative)."
```

10. Les objets

Les objets sont des variables plus complexes qui peuvent être constitués d'une ou plusieurs autres variables (attributs) et d'une ou de plusieurs fonctions (méthodes). Ainsi, un objet peut lui-même contenir un objet ! Un objet peut représenter absolument ce que l'on veut : une chaise, une voiture, un concept philosophique, une formule mathématique, etc.

Pour cela, il faut déclarer ce qu'on appelle une classe et lui donner un nom. On construit ensuite un objet en faisant référence à cette classe. Construire un objet s'appelle l'instanciation.

Exemple :

```
class Personne:
        """déclaration de la classe"""
        nom = ""
        prenom = ""
        age = 33

qui = Personne()                # instanciation de la classe Personne
qui.nom         = "Dupont"      # initialisation des attributs
qui.prenom      = "Jean"
print "Je m'appelle {0} {1} et j'ai {2} ans." . format(qui.prenom, qui.nom, qui.age)
```

10.1. Les constructeurs

Parmi les différents types de méthode, il existe un type particulier : les constructeurs. Ces constructeurs sont des méthodes qui construisent l'objet désigné par la classe. Un constructeur porte le nom __init__.

Exemple :

class Personne:

```
        """Classe définissant une personne caractérisée par :
        - son nom
        - son prénom
        - son âge
        - son lieu de résidence"""

        def __init__(self, nom, prenom):       # le constructeur
                """Chaque attribut va être instancié avec une valeur par défaut"""
                self.nom        = nom
                self.prenom     = prenom
                self.age        = 33
                self.residence  = "Paris"

        def ma_residence(self):
                """Cette méthode affiche le lieu de résidence"""
                print "J'habite {0}." . format(self.residence)

qui = Personne('Dupont', 'Jean')
qui.ma_residence()
```

10.2. L'héritage

L'héritage est une fonctionnalité objet qui permet de déclarer que telle classe sera elle-même modelée sur une autre classe, qu'on appelle la classe parente, ou la classe mère. Concrètement, si une classe B hérite de la classe A, les objets créés sur le modèle de la classe B auront accès aux méthodes et attributs de la classe A, on dit que la classe A est la fille de la classe B et que la classe B est le parent (ou la superclasse) de la classe A.

Exemple :

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom, prenom):
        self.nom    = nom
        self.prenom = prenom

class AgentSpecial(Personne):
    """Classe définissant un agent spécial.
    Elle hérite de la classe Personne"""
    def __init__(self, nom, prenom, matricule):
        """Un agent se définit par son nom et son matricule"""
        # On appelle explicitement le constructeur de Personne :
        Personne.__init__(self, nom, prenom)
        self.matricule = matricule

    def mon_matricule(self):
        """Cette méthode affiche le matricule"""
        print "Mon matricule est {0}." . format(self.matricule)

### Programme principal : ###
qui = AgentSpecial('Dupont', 'Jean', '007')
print "Je m'appelle {0} {1}." . format(qui.prenom, qui.nom)
qui.mon_matricule()
```

Pour contrôler les capacités des classes à utiliser les attributs et méthodes les unes des autres, il est possible de les déclarer privés grâce au double souligné __ pour que les éléments ne soient accessibles qu'à la classe elle-même.

Exemple :

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom, prenom):
        self.nom    = nom
        self.prenom = prenom

class AgentSpecial(Personne):
    """Classe définissant un agent spécial qui hérite de la classe Personne"""
    def __init__(self, nom, prenom, matricule):
        Personne.__init__(self, nom, prenom)
        # l'attribut __matricule ne peut être initialisé qu'à travers ce
constructeur
        self.__matricule = matricule

    def __private(self):
        """Cette méthode affiche le matricule"""
        # elle ne peut être appelée qu'à l'intérieur de la classe
        print "Mon matricule est {0}." . format(self.__matricule)

    def mon_matricule(self):
        """Cette méthode affiche le matricule"""
        self.__private()

### Programme principal : ###
qui = AgentSpecial('Dupont', 'Jean', '007')
print "Je m'appelle {0} {1}." . format(qui.prenom, qui.nom)
qui.mon_matricule()
qui.__matricule = '001'       # provoque une erreur
qui.__private()               # provoque une erreur
```

D'autres méthodes prédéfinies sont couramment utilisées :

- __repr__ : affichage de l'objet sur une commande print par exemple
- __add__ : redéfinition de l'opérateur + (addition)
- __div__ : redéfinition de l'opérateur / (division)
- __gt__ : redéfinition de l'opérateur >
- ...

Exemple :

```
class Vector2(object):
    """Classe de base : les vecteurs"""
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    # Definir la methode __add__() permet d'ecrire v1 + v2 au lieu de v1.add(v2)
    def __add__(self, v):
        return self.__class__(self.x + v.x, self.y + v.y)
```

```
class Complex(Vector2):
    """Classe des nombres complexes"""
    def __init__(self, x, y):
        # Appel au constructeur de la classe parente
        super(Complex, self).__init__(x, y)
        #Vector2.__init__(self, x, y)

    # Definition de la representation : x + y*i
    def __repr__(self):
        y = self.im()
        if y < 0:
            sign = "-"
        else:
            sign = "+"

        return "%.2f %s %.2f*i" % (self.real(), sign, abs(y))

    # accesseurs des attributs parties reelles et imaginaires
    def real(self):
        return self.x

    def im(self):
        return self.y
```

11. Les fichiers

11.1. Ouverture de fichier

Sous Python, l'accès aux fichiers est assuré par l'intermédiaire d'un « objet-fichier » que l'on crée à l'aide de la fonction interne open(). Après avoir appelé cette fonction, vous pouvez lire et écrire dans le fichier en utilisant les méthodes spécifiques de cet objet-fichier.

La fonction open est disponible sans avoir besoin de rien importer. Elle prend en paramètre :

- le chemin (absolu ou relatif) menant au fichier à ouvrir ;
- le mode d'ouverture.

Le mode est donné sous la forme d'une chaîne de caractères. Voici les principaux modes :

- 'r' : ouverture en lecture (Read).
- 'w' : ouverture en écriture (Write). Le contenu du fichier est écrasé. Si le fichier n'existe pas, il est créé.
- 'a' : ouverture en écriture en mode ajout (Append). On écrit à la fin du fichier sans écraser l'ancien contenu du fichier. Si le fichier n'existe pas, il est créé.

Remarque : on peut ajouter à tous ces modes le signe b pour ouvrir le fichier en mode binaire.

```
>>> mon_fichier = open("fichier.txt", "r")
```

Remarque : si d'autres applications, ou d'autres morceaux du code, souhaitent accéder à ce fichier, ils ne pourront pas car le fichier sera déjà ouvert. La méthode à utiliser est close :

```
>>> mon_fichier.close()
```

11.2. Lecture d'un fichier

La méthode read renvoie tout ou une partie du contenu du fichier, que l'on capture dans une chaîne de caractères. Le paramètre de la méthode indique le nombre de caractères à lire. S'il est omis, tout le fichier est lu.

```
fd = open("fichier.txt", 'r')

end = False
while not end:
     text = fs.read(50)
     if text == "":
             end = True
     else:
             print text

fd.close()
```

11.3. Écriture dans un fichier

Vous pouvez utiliser le mode w ou le mode a. Le premier écrase le contenu éventuel du fichier, alors que le second ajoute ce que l'on écrit à la fin du fichier. Dans tous les cas, ces deux modes créent le fichier s'il n'existe pas.

Pour écrire dans un fichier, on utilise la méthode write en lui passant en paramètre la chaîne à écrire dans le fichier. Elle renvoie le nombre de caractères qui ont été écrits. Chaque nouvel appel de **write()** continue l'écriture à la suite de ce qui est déjà enregistré.

```
fd = open("fichier.txt", "w")

nbcar = fd.write("Premier test d'écriture dans un fichier via Python")
if nbcar == 0:
      print "erreur d'écriture"

fd.close()
```

Remarque : la méthode write n'accepte en paramètre que des chaînes de caractères. Si vous voulez écrire dans votre fichier des nombres, des scores par exemple, il vous faudra les convertir en chaîne avant de les écrire et les convertir en entier après les avoir lus.

11.4. Le mot-clé with

Il peut se produire des erreurs quand on lit, quand on écrit… et si l'on n'y prend garde, le fichier restera ouvert.

Il existe un mot-clé qui permet d'éviter cette situation :

```
with open(fichier, mode_ouverture) as variable:
      # Opérations sur le fichier
```

Si une exception se produit, le fichier sera tout de même fermé à la fin du bloc.

Le mot-clé with permet de créer un "context manager" (gestionnaire de contexte) qui vérifie que le fichier est ouvert et fermé, même si des erreurs se produisent pendant le bloc.

Il est inutile, par conséquent, de fermer le fichier à la fin du bloc with. Python va le faire tout seul, qu'une exception soit levée ou non.

TEST

1 - Après ces instructions, de quel type est la variable c ?

```
a = 8
b = 3
c = a / b
```

- int (entier)
- float (flottant)
- str (chaîne de caractères)

2 - Quelle est la variable de type str (chaîne de caractères) parmi les choix suivants ?

- 3
- 3.1
- "3"

3 - Quelle est la différence entre entrer une variable dans l'interpréteur interactif et utiliser la fonction print ?

- Aucune
- Dans l'interpréteur interactif, toutes les variables apparaissent entourées de guillemets.
- La fonction print est dédiée à l'affichage, l'interpréteur au débuggage.

4 - De quoi doit être composée une condition au minimum ?

- D'un bloc if
- D'un bloc if et elif
- D'un bloc if et else

5 - Considérant les instructions ci-dessous, si variable vaut 2.8, quel va être le résultat obtenu ?

```
if variable >= 3:
        print("1")
elif variable < -1:
        print("2")
else:
        print("3")
```

- Afficher 1.
- Afficher 2.
- Afficher 3.
- N'afficher rien.

6 - Considérant le prédicat combiné ci-dessous, dans quel cas sera-t-il True (vrai) ?

- predicat_a and predicat_b
- predicat_a est vrai, peu importe predicat_b.
- predicat_b est vrai, peu importe predicat_a.

- L'un d'eux est vrai, peu importe l'autre.
- predicat_a et predicat_b sont tous deux vrais.

7 - Comment Python identifie-t-il les instructions formant un bloc (par exemple à l'intérieur d'une condition) ?

- Grâce à l'indentation
- Grâce aux accolades ({}) entourant le bloc
- Grâce aux deux points en début de bloc

8 - Quel est l'avantage de la boucle while sur la boucle for ?

- Aucun.
- Elle est préférable pour parcourir des séquences.
- Elle fait la même chose en moins de lignes de code.
- Elle crée rarement de boucle infinie.
- Elle est plus utile pour vérifier une condition.

9 - Quel est l'avantage de la boucle for sur la boucle while ?

- Elle est préférable pour parcourir des séquences.
- Elle est plus utile pour vérifier une condition.
- C'est l'unique façon de parcourir des séquences.

10 - Sachant la définition de fonction ci-dessous, quel est l'appel INVALIDE parmi les choix suivants ?

def table(nombre, maximum=10):

- table(5, 20)
- table(12, maximum=5)
- table(8)
- table(maximum=15, nombre=4)
- table(7, entier=30)

11 - Qu'est-ce qu'un module en Python ?

- Un fichier contenant du code Python sans extension particulière.
- Un fichier contenant du code Python avec l'extension .py
- Un répertoire contenant des fichiers Python

12 - En utilisant l'instruction from ... import *, que peut-on importer ?

- Seulement des fonctions d'un module
- Seulement des modules d'un package
- Tout ce que contient un module ou package

13 - Qu'est-ce qui caractérise, au minimum, un package Python ?

- Un répertoire simple
- Un répertoire avec, au minimum, un fichier __init__.py dedans
- Un répertoire avec un fichier __init__.py et au moins un autre module ou package

14 - Quels sont les mot-clés minimums pour capturer une exception ?

- try et catch
- try et except
- try et else

15 - Quel mot-clé est utilisé pour lever une exception ?

- throw
- raise
- try

16 - Dans quel cas le bloc finally est-il exécuté ?

- Quand aucune exception ne se produit dans le bloc try.
- Quand une exception se produit dans le bloc try.
- Dans tous les cas.

17 - Dans quel cas l'instruction ci-dessous lèvera une exception ?

annee = int(entree)

- La variable annee n'existe pas.
- La variable entree est une chaîne de caractères.
- La variable entree ne peut être convertie en nombre.