Apprendre le langage PYTHON

I. Introduction au langage PYTHON
II. Premier pas en Python
II.1. Les variables
II.2. Les opérateurs
II.3. Les types simples
II.4. Type complexe
II.5. Importation des packages
II.6. Instruction d'Affectation
II.7. Les fonctions d'Entrées / sorties
III. Contrôles de flux d'instructions
III.1. Instructions conditionnels
III.2. Les instructions répétitives
IV. Structures des données
IV.1. Séquences
IV.2. Ensembles
IV.3. Dictionnaire
V. Fonctions
VI. Fichiers
Documentation en ligne

I. Introduction au langage PYTHON

1. Représentation de python

Python est un langage portable, dynamique, extensible, gratuit, qui permet une approche modulaire et orientée objet de la programmation. Python est développé depuis 1989 par « Guido van Rossum » et de nombreux contributeurs bénévoles. En 1995, sortie de Python 1.6.1, compatible GPL, Actuellement, en version 3.3.

2. Caractéristiques de langage PYTHON

Les principales caractéristiques de Python sont:

- Python est portable sur de nombreux OS (Unix, Mac, Windows, ...)
- Python convient aussi bien à des **scripts** d'une dizaine de lignes qu'à des **projets complexes** de plusieurs dizaines de milliers de lignes.
- La **syntaxe** de Python est **très simple** et, combinée à des **types de données évolués** (listes, dictionnaires,...), conduit à des programmes à la fois très compacts et très lisibles. A fonctionnalités égales, un programme Python est souvent de 3 à 5 fois plus court qu'un programme C ou C++ (ou même Java) équivalent, ce qui représente en général un temps de développement de 5 à 10 fois plus court et une facilité de maintenance largement accrue.
- Python est gratuit et utilisable sans restriction dans des projets commerciaux.
- Python détruit lui-même les objets créés lorsque plus aucune référence ne pointe sur eux.
- Il n'y a pas de pointeurs explicites en Python.
- Python est orienté-objet. Il supporte l'héritage multiple et la surcharge des opérateurs.
- Python est extensible: On peut l'interfacer avec d'autres langages (C, C++, Delphi, Java, ...)
- La bibliothèque standard de Python est très riche et de nombreuses bibliothèques gratuites peuvent être facilement ajoutées.
- Langage interprété et édition de lien dynamique.
- Lorsque Python a besoin d'une fonction définie autre part, il la charge .
- Langage fortement typé, mais typé dynamiquement (Rien besoin de déclarer et si vraiment on veut déclarer un objet on fait a=None (non typé)),
- Fichier source « *.py », fichier objet « *.pyc » et « *.pyo » mais on s'en fiche.
- Dans tes fichiers, tu mets ce que tu veux : Une classe, deux classes, pas de classe du tout.
- Langage impératif, objet et fonctionnel.
- Introspection et modification dynamique du code.

3. Installation du langage PYTHON

Python peut fonctionner sur de multiples systèmes d'exploitation (Windows, Linux, Mac, ...) mais aussi sur différentes plate-forme (Java, .Net, ...). La plus utilisée est l'implémentation ***CPython*** écrite en C et plus fréquemment appelé ***Python***. C'est celle qu'il est impératif d'utiliser pour débuter avec Python, les autres n'étant utiles que si on souhaite travailler également avec Java (***Jython***) ou .net (***IronPython***).

- téléchargement : Le site officiel de Python (www.python.org)
- La version courante (Python 3.3) pour Windows, Linux, Max OS .
- Il n'y a rien de plus simple que d'installer Python à partir de la version ***Windows installer*** (.msi). Vous avez le choix entre plusieurs fichiers d'installation à télécharger dépendant de votre version de windows (32 bits ou 64 bits) et de la version de Python sélectionnée.
- Après avoir téléchargé le fichier, "**python-X.X.msi**" (où X.X correspond au numéro de la version récupérée), un double-clic sur celui-ci permet de lancer l'installation de l'environnement Python sous Windows.
- Il suffit d'exécuter le fichier et de suivre les instructions. Il est conseillé de conserver le répertoire par défaut de l'installation, en général ***c:\python33***. En gros, cela consiste donc à toujours cliquer sur le bouton ***Next***

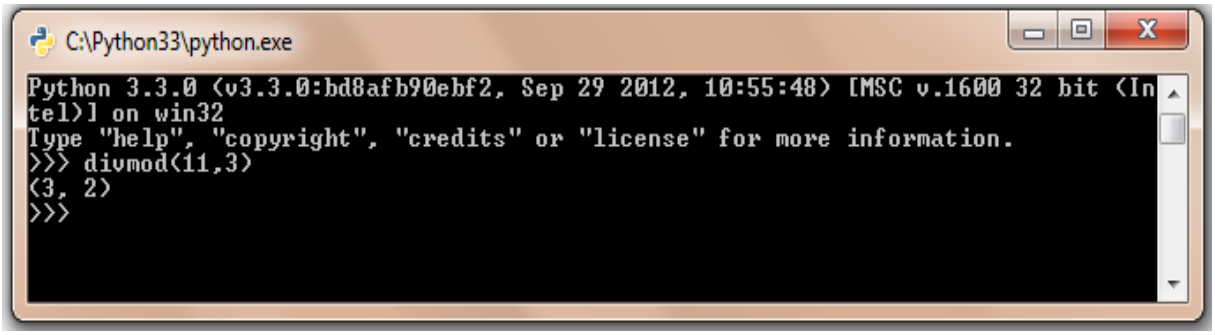
4. *Les deux modes d'exécution d'un code Python*

• Soit on enregistre un ensemble de commandes Python dans un fichier grâce à un éditeur (on parle alors d'un *script Python*) que l'on exécute par une commande ou par une touche du menu de l'éditeur ;

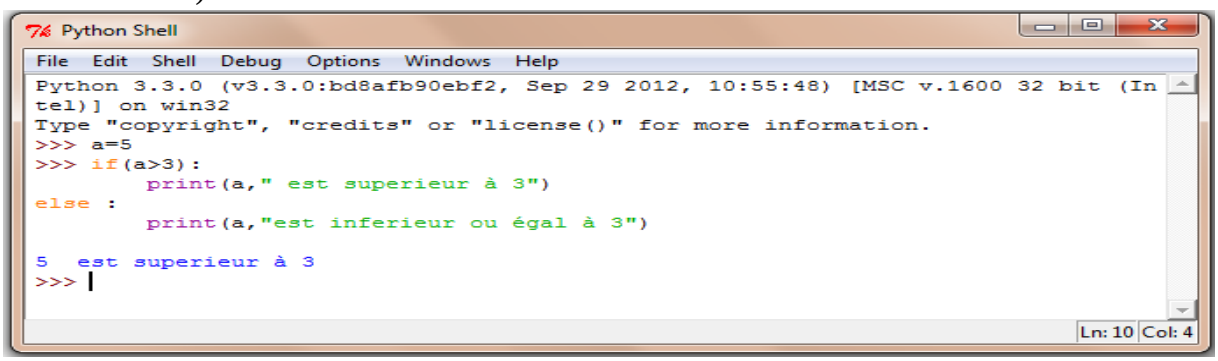
• Soit on utilise un interpréteur (par exemple IDLE) pour obtenir un résultat immédiat grâce à l'interpréteur Python embarqué dans IDLE qui exécute la *boucle d'évaluation*.

5. *Lancement de PYTHON*

Pour lancer le Shell interactif (ou console Python), il suffit d'exécuter le fichier ***python.exe*** se trouvant dans le répertoire ***c:/python33/.*** Vous pouvez aussi l'exécuter par le menu ***Démarrer -> Tous les programmes -> Python 3.3 -> Python (command line)*** (voir la fenêtre suivante)



Vous pouvez aussi l'exécuter par le menu en bas à gauche de votre écran ***Démarrer -> Tous les programmes ->Python 3.3 -> IDLE (Python GUI)*** (termes dépendant de la version de votre Windows). La fenêtre suivante s'affiche :



Vous pouvez écrire et exécuter de la même façon que précédemment des instructions Python. L'avantage est que l'interface est un peu plus élaboré avec quelques fonctionnalités intéressantes comme la coloration syntaxique, les call-tips ou encore l'autocomplétion (avec la touche Tabulation). Le signe >>> est le prompt de l'interpréteur Python. Vous pouvez alors taper du code que Python interprétera lorsque vous le validerez par la touche ***Entrée***. Si votre instruction n'est pas complète, Python n'exécutera pas le code immédiatement et passera à la ligne suivante. Une fois l'instruction terminée et validée, vous retrouverez alors le prompt >>> qui vous permettra d'écrire une nouvelle instruction.

IDLE en plus d'être un Shell interactif est aussi un éditeur spécialement conçu pour des programmes Python. Pour ouvrir IDLE en mode éditeur, vous pouvez, à partir du mode interactif, aller dans le menu **File - New Window ou bien raccourci clavier : Ctrl+N. S'ouvrira alors l'éditeur d'IDLE.*

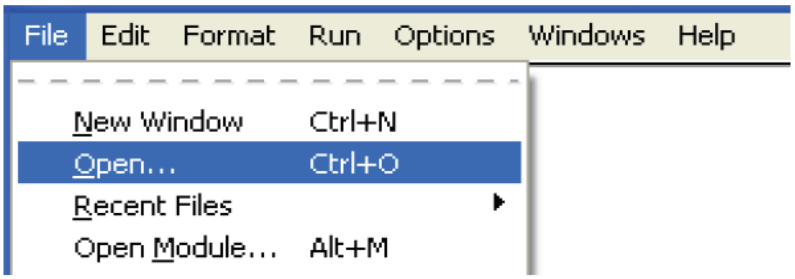
6. *Exécution d'un programme écrit en PYTHON*

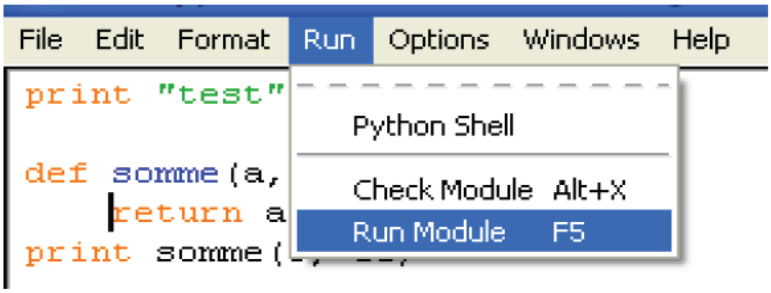
Une autre possibilité pour accéder à l'éditeur de ***IDLE*** est tout d'abord de créer à la main un fichier texte (par exemple avec le ***bouton droit de la souris -> Nouveau -> Document texte*** que vous renommez en un fichier d'extension ***.py***.

Vous pouvez ensuite ouvrir ce fichier avec ***IDLE*** en cliquant dessus avec le ***bouton droit de la souris -> Edit With IDLE***. En fonction de la configuration de ***IDLE***, il se peut que cela ouvre en même temps le shell interactif de ***IDLE***.

Sous ***IDLE***, l'exécution d'un script Python est très simple. Il vous faudra tout d'abord charger dans l'éditeur votre script s'il n'y est pas déjà présent par le menu ***File -> Open***.

Ensuite pour l'exécuter, il suffit d'appuyer sur la touche ***F5*** (ou bien aller dans le menu ***Run -> Run Module***). L'exécution se fera dans la fenêtre interactive de ***IDLE***.





II. *Premiers pas en Python*

→ ***Algorithme :*** *est une suite finie de règles à appliquer dans un ordre déterminé à un nombre fini de données pour arriver avec certitude, en un nombre fini d'étapes, à un certain résultat et cela, indépendamment des données. Leur écriture est indépendante du langage choisi.*

→ ***Programme:*** *est la traduction d'un algorithme en un langage compilable ou interprétable par un ordinateur qui peut ensuite être exécuté pour effectuer le traitement souhaité. Il est souvent écrit en plusieurs parties, dont une qui pilote les autres, le programme principal.*
Qu'il soit écrit en Basic, en Pascal, en C, en Python, en français, un algorithme reste le même, ce qui favorise l'apprentissage d'un nouveau langage informatique.

→ ***Les commentaires :*** *Pour en faciliter la lecture d'un programme source, il doit être judicieusement commenté. La signification de parties non triviales doit être expliquée par un commentaire.*

En Python, un commentaire commence par le caractère # et s'étend jusqu'à la fin de la ligne.
*Autrement dit, toute information aidant à la compréhension du programme mais n'en faisant pas partie comme dans l'exemple qui suit :

```
>>> x = 5    # affectation de la valeur 5 à la variable x
```

→ ***Mot clés :*** *Comme tout langage, Python permet de manipuler des données grâce à un vocabulaire de mots réservés et grâce à des types de données. En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme noms de variables les « mots réservés »*

1. *Les variables dans Python*

Définition

Une variable est un identifiant associé à une valeur. Informatiquement, c'est une référence d'objet située à une adresse mémoire. Les variables permettent de manipuler des informations en les nommant. Elles jouent un rôle semblable aux inconnues dans une équation mathématique. Les identifiants sont sensibles à la casse et ne doivent pas être un mot clé. Une variable est caractérisée par :

- **Un identificateur :** il peut contenir des lettres, des chiffres, des blancs soulignés mais il ne peut commencer par un chiffre. Minuscules et majuscules sont différenciées. Ils doivent être différents des mots réservés de Python.

Exemple : - {A1 ,A2, AN, F_x, B, s, Somme, ….} se sont des identificateurs des variables 1A , F&d : ne peuvent pas être utilisé comme des noms des variables

- **Un type :** c'est une information sur le contenu de la variable qui indique au compilateur Python la manière de manipuler cette information : Entier (integer -int) , réel (float), chaine de caractères (string – str)

Exemple : On désire calculer la somme de n nombres (u1, . . . ,un). Or l'ordinateur ne sait bien souvent manipuler que deux nombres à la fois. Par conséquent, il est nécessaire de créer une variable intermédiaire qu'on appellera par exemple 's' de manière à conserver le résultat des sommes intermédiaires :

```
s = 0  # pour tous les indices de 1 à n
for i in range(1,n) :
s = s + u [i]    # on ajoute le ième élément à 's'
```

→ ***Les constantes :*** *sont le contraire des variables, ce sont toutes les valeurs numériques, chaînes de caractères, ..., tout ce qui n'est pas désigné par un nom. Les constantes possède un type mais pas d'identificateur.*

2. *Les opérateurs*

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

Opérateur	Description	Exemple
Opérateurs arithmétiques	**+ :** addition - : soustraction * : multiplication ** : puissance / : division flottante ; // : division entière. % : reste de la division entière Abs() : valeur absolue	20 + 3 # 23 20 - 3 # 17 20 * 3 # 60 20 ** 3 # 8000 20 / 3 # 6.666666666666667 20 // 3 # 6 (division entière) 20 % 3 # 2 (modulo) abs(3 - 20) # valeur absolue
Operateurs de comparaison	==, !=, >, >=, <, <=	2 > 8 # False 2 <= 8 < 15 # True
Opérateurs logiques	not, or, and	Not 0 # true (3 == 3) or (9 > 24) # True (dès le 1er membre) (9 > 24) and (3 == 3) # False (dès le 1er membre)
Opérations sur les bases		07 + 01 # 8 oct(7+1) # '010' (octal) 0x9 + 0x2 # 11 hex(9+2) # '0xb' (hexad´ecimal) int('10110', 2) # 22

→ La fonction **divmod(a,b)** retourne un couple de valeurs (quotient, reste), résultat de la division entière de a par b. Il s'agit de l'unique couple (q,r) de NxR vérifiant a=bq+r avec $0 \leq r < b$.
Exemple : divmod(11,2) # donne (5,1)

→ En python 2, la division appliquée à des entiers retourne le résultat de la division entière ;

→ En Python 3, l'opérateur « / » retourne toujours le résultat de la division réelle. L'opérateur « // » retourne résultat de la division entière.

3. *Les types simples*

→ **Les nombres entiers : «** int »

Le type « **int** » n'est limité en taille que par la mémoire de la machine. Les entiers littéraux sont décimaux par défaut, mais on peut aussi utiliser d'autres bases.

Exemple : L'entier 2013 écrit en base 10 peut se représenter en binaire, octal et hexadécimal en utilisant les syntaxes suivantes :

```
>>> 2013 # décimal
2013
>>> 0b11111011101 # binaire
2013
>>> 0o3735 # octal
2013
>>> 0x7dd # hexadecimal
2013
```

→ **Les booléens : bool**

Une expression booléenne à deux valeurs possibles : False, True. Les opérations logiques et de comparaisons sont évaluées afin de donner des résultats booléens dans False, True.
Python attribue à une expression booléenne la valeur False si c'est :

- La constante False ;
- La constante None ;
- Une séquence ou une collection vide ;
- Une donnée numérique de valeur 0.

Tout le reste vaut True.

→ **Les nombres flottants**

Les flottants supportent les mêmes opérations que les entiers. Un float est noté avec un point décimal ou en notation exponentielle. **Exemple :** 2.718 , .02 , 3e8 , 6.023e23

4. *Le type complexe*

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

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

• Un module mathématique spécifique (**cmath**) leur est réservé:

```
import cmath
print(cmath.phase(-1 + 0j)) # 3.14159265359
print(cmath.polar(3 + 4j)) # (5.0, 0.9272952180016122)
print(cmath.rect(1., cmath.pi/4)) #
(0.707106781187+0.707106781187j)
```

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

%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 (format exponentiel)
%g : un flottant (format « optimal » suivant sa longueur)

```
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'
```

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

5. *Importation des fonctions*

L'import du module math par exemple autorise toutes les opérations mathématiques usuelles:

Importer les fonctions de la bibliothèque Package « math »	**Importer que sin & pi à partir du package « math »**

`import math`
`print(math.sin(math.pi/4)) # 0.7071067811865475`
`print(math.degrees(math.pi)) # 180.0`
`print(math.factorial(9)) # 362880`
`print(math.log(1024, 2)) # 10.0`
	`from math import sin, pi`
	`print( sin(pi/3))`
	`# 0.8660254037844386`

6. *L'instruction d'affectation*

On affecte une variable par une valeur en utilisant le signe = (qui n'a rien à voir avec l'égalité en mathématiques !). Dans une affectation, le membre de gauche reçoit le membre de droite.

```
>>>A = 5     # A reçoit 5
```

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

1. 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 ;
2. Crée et mémorise une valeur (membre de droite) ;
3. Établit un lien entre le nom de la variable et l'adresse de la valeur correspondante.

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

```
A = A + 1    # 6   (incrémentation)
A = A – 1    # 5   (décrémentation)
```

→ **Autres formes des instructions d'affectation**

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)
```

7. *Les fonctions d'entrées/sorties*

→ **Les entrées**

Il s'agit de réaliser une *saisie* à l'écran : la fonction standard **input()** interrompt le programme, afficher une éventuelle invite et attend que l'utilisateur entre une donnée et la valide par la touche ***Entrée*** .

La fonction standard **input()** effectue toujours une saisie en *mode texte* (la saisie est une chaîne) dont on peut ensuite changer le type (on dit aussi *transtyper*) :

Exemple 1

```
nb_etudiant = input("Entrez le nombre d'étudiants : ")
print(type(nb_etudiant)) # <class 'str'> (c'est une chaîne)
```

Exemple 2

```
f1 = input("\n Entrez un flottant : ")
f1 = float(f1) # transtypage en flottant
                    # ou plus brièvement :
f2 = float(input("Entrez un autre flottant : "))
print(type(f2)) # <class 'float'>
```

Exemple 3

```
Texte = input("Tapez une phrase : ")
                          # transtypage l'entrée en  entier
n =int( input("Entrez un entier : "))
                          # transtypage l'entrée en flottant
f1 = float(input("Entrez un flottant : "))
```

→ **Les sorties**

En mode « calculatrice », Python *lit-évalue-affiche*, mais la fonction **print()** reste indispensable aux affichages dans les scripts :

```
a, b = 2, 5
print(a, b) # 2 5
print(”Somme :”, a + b) # Somme : 7
print(a - b, ”est la différence”) # -3 est la différence
print(”Le produit de”, a, ”par”, b, ”vaut :”, a * b)
# Le produit de 2 par 5 vaut : 10
print() # affiche une nouvelle ligne
# pour afficher un espace à la place de la nouvelle ligne:
print(a, end=” ”) # 2 (et ne va pas à la ligne)
```

- *Remarque1 :* Une instruction sur plusieurs lignes

Il n'est pas non plus possible d'utiliser deux lignes pour écrire une affectation à moins de conclure chaque ligne qui n'est pas la dernière par le symbole **«** \ » .

L'exemple suivant est impossible.

```
x =
    5
```

Il devrait être rédigé comme suit:

```
>>> x = \
        5
```

Avec ce symbole **« \ »,** les longues lignes peuvent être écrites sur plusieurs de manière plus lisibles, de sorte qu'elles apparaissent en entier.

-*Remarque2 :* plusieurs instructions sur une même ligne

Il est possible de créer plus qu’une instruction sur la même ligne (ils sont séparé entre eux par un point virgule « ; »), de réaliser deux affectations sur une même ligne.

```
>>> x = 5 ; X=4.
```

III. *Contrôles de flux d'instructions*

1. *Les instructions conditionnelles if -- [elif] -- [else]*

→ **Syntaxe :**. Une instruction composée se compose :

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

• D'un bloc d'instructions indenté **par rapport à la ligne d'en-tête**

Toutes les instructions au même niveau d'indentation appartiennent au même bloc

1er format	2eme format	3éme format

if (condition):
  <instruction>
	if (condition):
	  <instruction>
	else:
	  <instruction>
		if (condition):
		  <instruction>
		elif (condition):
		  <instruction>
		else:
		  <instruction>

`x = int(input("enter un entier: "))`
if (x < 0):
 print ("x est Non positif")
	`x = int(input("enter un entier: "))`
	if (x < 0):
	 print ("x est négatif")
	else:
	 print ("x n'est pas négatif")
		`x = int(input("enter un entier: "))`
		if (x < 0):
		 print ("x est négatif")
		elif (x % 2):
		 print ("x est positif et impair")
		else:
		 print ("x est positif et est pair")

→ **Syntaxe compacte d'une alternative**

Pour trouver, par exemple, le minimum de deux nombres, on peut utiliser l'opérateur ternaire (repris du compilateur C) :

Ecriture classique	Utilisation de l'opérateur ternaire
x, y = 4, 3 if x < y: min = x else: min = y print("Plus petit : ", min) # 3	x, y = 4, 3 min = x if x < y else y print("Plus petit : ", min) # 3

→ Test d'une valeur booléenne :

if (x): # mieux que (if x is True:) ou que (if x == True:)

2. *Les instructions répétitives : les boucles*

L'une des tâches que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus fondamentales : la boucle de répétition construite autour de l'instruction while.

→ *La boucle :for -- [else]* : Répéter une portion de code

Sa forme est : **For** < nom_var> **in range** (<val_initial > , <val_final >) **:**

<Actions>

Cette action permet de répéter une action (ou un groupe d'actions représenté ici par <Actions>) un nombre de fois déterminé par les valeurs initiale et finale du paramètre < nom_var>.

Exemple pour i=1 a 4 faire lire (note) finfaire	**En Python** for i in range (1,5) : note=float(input('saisir une note'))

→ La fonction range

for i in range(10): print(i, end=" | ") # 0 |1|2|3|4|5|6|7|8|9|	for i in range(5,10): print(i, end=" , ") #5,6,7,8,9	for i in range(1,10,2): print(i, end=" * ") # 1*3*5*7*9

→ Parcourir une séquence

```
for lettre in "ciao": print (letter) # 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
```

→ *La boucle* ***while -- [else]***

Sa forme est: **tant que** < condition> **faire** < bloc instructions > **finfaire**
Cette action spécifie qu'une instruction (ou groupe d'instructions) doit être répétée tant que la condition reste vérifiée. La partie < bloc instructions > étant répétée jusqu'à ce que la condition devienne **fausse**; il faut donc que cette partie soit capable de modifier les paramètres intervenant dans les conditions afin de sortir du 'tant que'.

Syntaxe de base

Nous avons ainsi construit notre deuxième *boucle de programmation*, laquelle répète un certain nombre de fois le bloc d'instructions indentées.

Exemple

ecrire('Entrez un entier [1 .. 10] : ')
lire (n)
Tantque (n < 1) or (n > 10):
  ecrire('Entrez un entier [1 .. 10], S.V.P: ')
  lire (n)
finTq
	# 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: ')

Comme l'instruction **« if »** abordée précédemment, l'instruction **while** amorce une *instruction composée*. Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lequel doit obligatoirement se trouver en retrait. Comme vous l'avez appris chapitre précédent, toutes les instructions d'un même bloc doivent être indentées exactement au même niveau (c'est-à-dire décalées à droite d'un même nombre d'espaces). Voici comment cela fonctionne :

- Avec l'instruction **while**, Python commence par évaluer la validité de la *condition* fournie entre parenthèses (Celles-ci sont optionnelles. Nous ne les avons utilisées que pour clarifier notre explication).
- Si la condition se révèle fausse, alors tout le bloc qui suit est ignoré et l'exécution du programme se termine.
- Si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant *le corps de la boucle*

→ **Exceptions :** A fin de rendre les applications plus robustes, il est nécessaire de gérer les erreurs d'exécution des parties sensibles du code.
Le mécanisme des exceptions sépare d'un côté la séquence d'instructions à exécuter lorsque tout se passe bien et, d'un autre côté, une ou plusieurs séquences d'instructions à exécuter en

cas d'erreur. Lorsqu'une erreur survient, un *objet exception* est passé au mécanisme de propagation des exceptions, et l'exécution est transférée à la séquence de traitement *ad hoc*.

→ **Le mécanisme s'effectue en deux phases**

• la *levée* d'exception lors de la détection d'erreur ;
• le *traitement* approprié.

→ **Syntaxe :**. La séquence normale d'instructions est placée dans un bloc **try**. Si une erreur est détectée (levée d'exception), elle est traitée dans le bloc **except** approprié (le gestionnaire d'exception).

```
try:
... # séquence normale d'exécution
except <exception_1> as e1:
... # traitement de l'exception 1
except <exception_2> as e2:
... # traitement de l'exception 2
...
else:
... # clause exécutée en l'absence d'erreur
finally:
... # clause toujours exécutée
```

→ **Syntaxe complète d'une exception**

Toutes les exceptions levées par Python sont des instances de sous-classe de la classe Exception. La hiérarchie des sous-classes offre une vingtaine d'exceptions standard.

→ **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**

Passe immédiatement au début de la boucle « for » ou « while » en cours contenant l'instruction ; reprend `a 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 saut´e la valeur 5
```

→ **Exceptions : try -- except -- [else]**

L' 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 (Voir la liste des exceptions standards dans la documentation Python):

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

IV. *Structures des donnes*

*Dans le chapitre 2, on a présenté les types de données simples, mais Python offre beaucoup plus : « **les conteneurs** ». De façon générale, un conteneur est un objet composite destiné à contenir d'autres objets. Ce chapitre détaille les séquences, les ensembles et les dictionnaires.*

1. *Les séquences*

Une séquence est un conteneur ordonné d'éléments indexés par des entiers. Python dispose de trois types prédéfinis de séquences :

- Les chaînes;
- Les listes ;
- Les tuples.

→ *Les chaînes de caractères*

Le type de données non modifiable *str* représente une séquence de caractères.

1. Les guillemets permettent d'inclure des apostrophes : c1 = "L'eau vive"
2. Les apostrophes permettent d'inclure des guillemets : c2 = ' est "froide" !'

Les opérations sur les chaines de caractères : fonctions vs méthodes

On peut agir sur une chaîne en utilisant des fonctions (notion procédurale) ou des méthodes.

Opération	Description	Exemple
lens	Longueur	s = "abcde" len(s) # 5
+	Concaténation	s1 = "abc" s2 = "defg" s3 = s1 + s2 # 'abcdefg
*	Répétition	s4 = "Fi! " s5 = s4 * 3 # 'Fi! Fi! Fi! '
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-chaine dans la chaîne	'abracadabra'.find('bra') # 1 (le 1er indice vaut 0)
count()	donne le nombre de sous-chaînes dans la chaîne	'abracadabra'.count('bra') # 2
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!
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'

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 listes*

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

```
# Exemple
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]]
```

Initialisation : 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)
```

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
```

Parcours d'une liste en utilisant ***for - else***

```
une_sequence = [2, 5, 9, 7, 11]
cible = int(input("Entrez un entier : "))
for i in une_sequence:
if i == cible:
sauve = i
break # voici l'interruption !
else:
print(cible, "n'est pas dans",
une_sequence)
sauve = None
```

- ***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 3ème position
mots[4:4] = ['beurre']  # insertion en 5ème position
print (mots) # ['jambon', 'sel', 'miel', 'confiture', 'beurre']
```

- *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']
```

Si l'on désire réaliser une vraie copie d'un objet, on utilise 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 (a) # [1, 2, 3, 5]
```

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

Exercice

Écrire une fonction **conjuguer(verbe)** qui prend en paramètre un verbe du premier groupe à l'infinitif, et qui affiche la conjugaison de ce verbe au présent de l'indicatif. Attention: n'oubliez pas l'élision du "je" pour les verbes qui commencent par une voyelle (`J'aime`, `Je déteste`). Par exemple, **conjuguer('aimer')** donne:

```
J'aime
Tu aimes
Il aime
Nous aimons
Vous aimez
Ils aiment
```

→ *Liste en compréhension*

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	Deuxième forme	Troisième forme
result1 = [x+1 for x in une_seq]	result3 = [x+1 for x in une_seq if x > 23]	result5 = [x+y for x in une_seq for y in une_autre]

# a le même effet que : 	
result2 = []
for x in une_seq:
result2.append(x+1)
	# a le même effet que :
	result4 = []
	for x in une_seq:
	if x > 23:
	result4.append(x+1)
		# a le même effet que :
		result6 = []
		for x in une_seq:
		for y in une_autre:
		result6.append(x+y)

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

```
s = sum([i for i in range(10)])    # à le même effet que :
```

```
s = 0
for i in range(10):
    s = s + i
```

2. *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']) : diff´erence
x | y # set(['a', 'c', 'b', 'd', 'x']) : union
x & y # set(['b', 'd']) : intersection
```

3. *Les dictionnaires*

→ ***Dictionnaires : Définition :*** Collection de couples (clé : valeur) entourée d'accolades. Les dictionnaires ne sont pas ordonnés, on ne peut donc pas les indicer.

```
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 : itérateurs***

Les itérateurs sont des objets spécifiques permettant de parcourir un dictionnaire. On dispose d'itérateurs sur les cl´es, les valeurs et les items :

```
d = {'a':1, 'b':2, 'c':3, 'd':4} # un dictionnaire
for it1 in d.keys(): print (it1) # a c b d
for it2 in d.values(): print (it2 )# 1 3 2 4
for it3 in d.items(): print (it3) # ('a', 1) ('c', 3) ('b', 2) ('d', 4)
```

V. *Les fonctions*

Les fonctions sont les éléments structurants de base de tout langage procédural. Elles offrent différents avantages :

- ✓ ***Évite la répétition*** *: on peut « factoriser » une portion de code qui se répète lors de l'exécution en séquence d'un script ;*
- ✓ ***Met en relief les données et les résultats*** *: entrées et sorties de la fonction ;*
- ✓ ***Permet la réutilisation*** *: mécanisme de l'import ;*
- ✓ ***Décompose une tâche complexe en tâches plus simples*** *: conception de l'application.*

→ **Définition :** Ensemble d'instructions regroupées sous un *nom* et s'exécutant à la demande. On doit définir une fonction à chaque fois qu'un bloc d'instructions se trouve à plusieurs reprises dans le code ; il s'agit d'une « mise en facteur commun ».

→ Syntaxe

Le bloc d'instructions est obligatoire. S'il est vide, on emploie l'instruction pass. La documentation (facultative) est fortement conseillée.

```
def nomFonction(paramètres):
#Documentation de la fonction.
<bloc_instructions>
```

→ *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

```
def table(base, debut, fin):
n = debut
while n <= fin:
    print (n, 'x', base, '=', n * base,)
    n += 1
table(7, 2, 11)  # exemple d'appel :
```

Un ou plusieurs paramètres, utilisation

```
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))
```

- **Paramètres avec valeur par défaut**

```
def initport(speed=9600, parity="paire", data=8, stops=1):
    print( "Initialisation `a", speed, "bits/s")
    print ("parit´e :", parity)
    print (dat, "bits de données")
    print (stops, "bits d'arret")
# Appels possibles :
initport()
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**

```
def somme(*args):
resultat = 0
for nombre in args: resultat += nombre
return resultat
# Exemples d'appel :
print somme(23) # 23
print somme(23, 42, 13) # 78
```

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

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

- **Nombre d'arguments arbitraire : passage d'un dictionnaire**

```
def un_dict(**kargs):
return kargs
# Exemples d'appels
## par des param`eres nomm´es :
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}
# 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
```

VI. Les fichiers

Il faut utiliser une mémoire permanente comme par exemple le dique dur, la clé USB, le DVD,. . . pour conserver durablement les informations. Comme la plupart des langages, Python utilise classiquement la notion de fichier.

C'est un type pré-défini en Python, qui ne nécessite donc pas d'importer de module externe. Nous nous limiterons aux fichiers textuels (portables, lisible par un éditeur), mais signalons que les fichiers stockés en codage binaire sont plus compacts et plus rapides à gérer.

Gestion des fichiers : Python utilise les fichiers en mode *texte* par défaut (noté t) (pour les fichiers *binaires*, il faut préciser le mode b).

→ ***Ouverture des fichiers :*** Principaux *modes* d'ouverture des fichiers textuels :

```
f1 = open("monFichier_1", "r", encoding='utf-8') # en lecture
f2 = open("monFichier_2", "w", encoding='utf-8') # en écriture
f3 = open("monFichier_3", "a", encoding='utf-8') # en ajout
```

Le paramètre optionnel encoding assure les conversions entre les types byte et str. Les encodages les plus fréquents sont 'utf-8' (c'est l'encodage à privilégier en Python 3), 'latin1', 'ascii'...

→ ***Fermeture des fichiers***

Une seule méthode de fermeture : ***f1.close()***

Tant que le fichier n'est pas fermé, son contenu n'est pas garanti sur le disque.

→ ***Écriture séquentielle : Exemple***	→ ***Lecture séquentielle : Exemple***

f = open("fich1.txt", "w")
s = 'python 3.3\n'
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()
  # utilisation de l'option file de print
f2 = open("chaine.txt", "w")
print("abcd", file=f2)
f2.close()
	f = open("fich1.txt", "r")
	s = f.read() # lit tout le fichier --> string
	s = f.read(3) # lit au plus n octets --> string
	s = f.readline() # lit la ligne suivante --> string
	s = f.readlines() # lit tout le fichier --> liste de strings
	f.close()
	  # Affichage des lignes d'un fichier une à une
	f = open("fich1.txt") # mode "r" par défaut
	for ligne in f:
	  print(ligne[:-1]) # pour sauter le retour à la ligne
	f.close()

Documentation en ligne

1. Tutoriel de la documentation officielle en ligne : http://docs.python.org/tutorial/index.html

2. Livre et exercices corrigés en ligne sur Python 2 et 3 : Apprendre à programmer avec Python de Gérard Swinnen, 2009 - http://inforef.be/swi/python.htm

3. Livre (format pdf) en ligne sur Python 3 : Introduction à Python 3 de Robert CORDEAU, 2010 http://www.afpy.org/Members/bcordeau/Python3v1-1.pdf/download

4. Tutoriel en ligne : http://www.tutorialspoint.com/python/index.htm

5. Documentation en ligne : http://infohost.nmt.edu/tcc/help/lang/python/docs.html

6. Site officiel python : http://www.python.org/

7. Association francophone Python : http://www.afpy.org/python/tutoriels

8. Documentation en ligne de l'intégration C/C++/Python : http://docs.python.org/extending/extending.html