

Initiation à la programmation impérative avec Python

Avertissement au lecteur :

Ce polycopié n'est pas un document scolaire de référence sur le cours de programmation, c'est seulement l'ensemble de mes propres notes de cours mises en forme. Il ne contient donc pas les explications détaillées qui ont été données en cours. En particulier tous les développements systématiques des exemples, expliquant comment le langage Python effectuerait les traitements, sont absents de ces notes. On trouvera également parfois quelques simples allusions à des concepts élémentaires, largement développés en cours mais qui sont routiniers pour tout informaticien.

G. Bernot

COURS 1

1. Les objectifs du cours
2. Comment fonctionne un ordinateur, dans les grandes lignes
3. Quelques unités de capacité d'information
4. Les langages de programmation
5. Un exemple en Python
6. Les structures de données
7. Les booléens
8. Les entiers relatifs

COURS 2

1. Les nombres réels
2. Les chaînes de caractères
3. Opération de substitution de chaînes de caractères
4. Les expressions conditionnelles
5. Les définitions de fonctions

COURS 3

1. Les définitions de procédures
2. Variables locales et variables globales
3. Compléments sur les chaînes de caractères, et `while`

COURS 4

1. Les listes
2. Instructions de modification de liste

COURS 5

1. Instructions de modification de liste (suite)
2. Exemples de programmes sur les listes

COURS 6

1. Les dictionnaires
2. Exemples de programmes sur les dictionnaires

COURS 7

1. Compléments mineurs mais utiles sur le type `str`
2. Lecture de fichiers

COURS 8

1. Lecture de fichiers (suite)
2. Écriture de fichiers

COURS 9

1. Les modules en Python
2. Gestion du système de fichiers avec le module `os`

COURS 10

1. Gestion du système de fichiers avec le module `os` (suite)
2. Système de fichiers, compléments du sous-module `os.path`

1 Les objectifs du cours

L'objectif de ce cours est d'apprendre à programmer « proprement ». On verra qu'utiliser un langage de programmation pour faire enchaîner à l'ordinateur, rapidement, des opérations rébarbatives sur des données n'est pas difficile. De même modifier un programme écrit la veille pour l'adapter aux besoins du lendemain ou pour supprimer des erreurs oubliées la veille, c'est facile. C'est une toute autre paire de manches que d'écrire un programme *fiable* qu'on pourra modifier non pas le lendemain de sa création mais six mois plus tard, quand on a oublié comment il marche... Pire encore : reprendre un programme écrit par un autre et le modifier sans introduire d'erreurs. Programmer *proprement*, c'est adopter un *style de programmation* clair et *lisible* qui facilite les modifications et la maintenance du programme. C'est ce qu'on va apprendre dans ce cours.

Pour ce faire, on va utiliser le langage *Python* mais le choix du langage n'est pas d'une importance capitale car avec les connaissances de « programmation dans les règles de l'art » acquise dans ce cours, il sera facile de passer d'un langage à un autre. Les concepts de base d'une programmation bien abordée sont les mêmes dans tous les langages.

Sur le fond, (presque) tout est facile en programmation. Face à des questions d'apparence compliquées, on applique des méthodes de découpage de problèmes en sous-problèmes et on combine les solutions intermédiaires. On se ramène toujours à utiliser les structures classiques de la programmation que nous verrons dans ce cours.

2 Comment fonctionne un ordinateur, dans les grandes lignes

Le cœur d'un ordinateur est son *processeur* : c'est une sorte de « centrale de manipulations » qui effectue des transformations électriques rapides, correspondant à des manipulations des *symboles* codés par ces signaux électriques. Il marche de concert avec une mémoire qui alimente le processeur en « données » c'est-à-dire, encore, des suites de symboles (signaux électriques).

Le codage mentionné plus haut contient principalement trois niveaux : les valeurs logiques (vrai/faux), les valeurs numériques (i.e. les nombres), et les commandes c'est-à-dire les manipulations à effectuer sur les codages des données précédentes. L'élément de base de ces codages est l'information *binnaire*, « le courant peut passer » ou « ne peut pas passer », et est matériellement réalisé par un genre de transistor largement miniaturisé.

Les commandes successives à effectuer sont placées en des endroits déterminés de la mémoire et sont lues les unes après les autres, ce qui conduit à enchaîner des commandes et finalement à faire des calculs complexes. Le premier usage de la mémoire est donc de mémoriser le programme des opérations (commandes) à effectuer. Pour ce faire le processeur possède une zone dans laquelle il note la place de la mémoire où se trouve la prochaine opération à effectuer, une zone dans laquelle il a recopié l'opération en cours, et quelques zones qui servent à manipuler les données traitées. Ces zones sont appelées des registres.

Maintenant que l'on a vu comment transitent les commandes successives à effectuer dans le processeur, parlons des données : il ne servirait à rien de faire tant de manipulations si elles ne sortaient jamais du processeur. Pour cela il y a des commandes de stockage des registres du processeur vers la mémoire. De même, pour alimenter le processeur en données, il existe des commandes de chargement des données (copies de la mémoire vers les registres).

Ainsi, on voit que si l'on conçoit un programme de commandes judicieux, on peut modifier à volonté l'état de la mémoire pour lui faire stocker des résultats de calculs aussi complexes et sophistiqués que l'on veut. La question qui se pose maintenant est d'exploiter cette mémoire en la montrant à l'extérieur sous une forme compréhensible pour l'Homme, ou réexploitable ultérieurement. C'est le rôles des *périphériques*.

Avec ce que l'on a vu jusqu'à maintenant, s'il y a une coupure de courant alors tout est perdu car les « mini-transistors » ne conservent pas leur état sans courant. Donc certains périphériques sont des mémoires « de masse » comme disque dur, disquettes, CDrom, DVDrom, etc. ou certaines mémoires « flash » (=clés USB) ayant une durée raisonnable de conservation des données.

Il faut également pouvoir, naturellement, entrer les données et les ordres, d'où des périphériques comme clavier, instruments de mesure, etc. Enfin il faut montrer à l'utilisateur les résultats des manipulations symboliques par exemple via un écran, une imprimante... Il faut également partager et échanger des informations avec d'autres ordinateurs \implies les réseaux.

3 Quelques unités de capacité d'information

- Un *bit* est une valeur logique (vrai/faux)
- Un *octet* (en anglais : byte) est une suite de 8 bits ; il permet de coder soit des entiers de 0 à 255, soit des caractères
- Un *Ko* se prononce un *kilo-octet* (en anglais : Kb, kilo-byte) ; c'est une suite de 1024 octets (pas exactement 1000 parce que 1024 est une puissance de 2, ce qui facilite l'adressage sur ordinateur).
- Un *Mo* se prononce *Méga-octet* (Mb en anglais) ; c'est une suite de 1024 Ko.
- Un *Go* se prononce *Giga-octet* (Gb) ; suite de 1024 Mo.
- Enfin un *Tera-octet* est une suite de 1024 Go.

4 Les langages de programmation

Ce qu'on vient de voir sur la structure d'un ordinateur donne déjà des capacités de programmation importantes : il suffit de mettre en mémoire une suite d'opérations à effectuer, et le processeur les chargera et les effectuera les unes après les autres. C'est ce qu'on appelle le *langage machine*. Comme son nom l'indique, c'est un langage très efficace pour une machine. . . mais il est très indigeste à lire (et *a fortiori* à écrire) pour un être humain. Plutôt que d'écrire du langage machine, l'humain préfère écrire des ordres à l'ordinateur dans un langage plus évolué (par exemple Python).

C'est là qu'intervient un raisonnement astucieux :

1. Un programme écrit par un humain en Python est finalement un texte, c'est-à-dire une suite de caractères qui peut donc être stockée dans la mémoire de l'ordinateur.
2. Les programmes en langage machine sont des suites de commandes qui doivent aussi être stockées en mémoire.
3. Or un programme en langage machine est finalement un processus qui transforme des données en mémoire en d'autres données en mémoire, quelles qu'elles soient. Un programme en langage machine peut donc considérer un programme écrit en Python comme des données, de même qu'il peut considérer un autre programme en langage machine comme des données.
4. *Donc*, si quelques spécialistes se chargent de la corvée d'écrire un programme en langage machine qui transforme :
 - un (texte de) programme en python
 - en une suite de commande en langage machine*alors* nous pouvons écrire un texte en Python, laisser le programme des spécialistes en faire un programme en langage machine, et faire tourner le résultat.

Cette technique peut s'appliquer de différentes façons que nous ne détaillerons pas ici. Elle est appelée selon les cas la *compilation* ou l'*interprétation* du langage (ici Python). Ceci permet d'avoir l'impression que c'est le programme écrit en Python qui « tourne » directement sur l'ordinateur.

D'une certaine façon, l'interprétation d'un langage de programmation joue pour l'informatique un rôle inverse de la transcription et la traduction pour la biologie moléculaire : la transcription et la traduction permettent de construire des structures de plus haut niveau à partir du « langage machine » qu'est le génome, alors que l'interprétation produit du langage machine à partir de textes bien structurés.

Grâce à cette technique classique, la notion de *langage de programmation* devient plus large qu'une simple suite de commandes à l'ordinateur.

Un langage de programmation est un « vocabulaire » restreint et des règles de formation de « phrases » très strictes pour donner des instructions à un ordinateur. Le *moins* par rapport au français ou aux mathématiques reste malgré tout sa pauvreté, mais le *plus* est qu'aucune « phrase » (= *expression*) n'est ambiguë : il n'existe pas 2 interprétations possibles. On peut alors :

- regrouper puis abstraire un grand nombre de données élémentaires (nombres, caractères. . .) pour caractériser une *structure de données* abstraite (**protéine** = suite de ses acides aminés *et* ses conformations possibles *et* ses sites actifs en fonction de conditions, etc.)
- regrouper des suites de commandes élémentaires (additions, multiplications, statistiques, classifications, décisions. . .) pour organiser le *contrôle du programme* et le rendre plus lisible et logique (déterminer si deux protéines ont de l'**affinité** = inventorer les conditions du compartiment qui les contient, calculer la conformation la plus probable, inventorer les sites actifs qui en résultent, comparer deux à deux si un site d'une protéine colle avec un site de l'autre protéine, etc.)

Donc : ce qui définit les langages de programmation, c'est

- la façon de représenter symboliquement les **structures de données** (e.g. protéine)
- et la façon de gérer le **contrôle** des programmes (que faire et dans quel ordre pour résoudre une question d'affinité de protéines).

5 Un exemple en Python

On reviendra plus précisément sur chacune des notions utilisées ici. Il s'agit simplement pour l'instant de *voir* à quoi ressemble un programme et son utilisation.

```
>>> borne = 100
>>> def evalue (n) :
...     if n < (borne / 2) :
...         print "petit"
...     elif n < borne :
...         print "moyen"
...     else :
...         print "grand"
...
>>> evalue (70)
moyen
>>> borne = 50
>>> evalue (70)
grand
```

Selon le système d'exploitation avec lequel on travaille, la présentation peut différer (les « >>> » ou les « ... ») mais en revanche, ce qu'on tape au clavier et les réponses de l'ordinateur sont *toujours* identiques.

... ICI EXPLICATION DÉTAILLÉE DE CHACUNE DES LIGNES ...

Puisqu'un langage de programmation est défini par ses *structures de données* et par son *contrôle*, il suffit de connaître les parties les plus utiles de ces deux aspects en Python pour savoir programmer en Python. Mieux : sur le fond, les principales structures de données et les principales primitives de contrôle *sont les mêmes* pour tous les langages dits impératifs (qui eux-mêmes représentent la quasi-totalité des langages industriels). Les seules variations d'un langage à l'autre sont les choix des mots clefs : par exemple certains langages utilisent « **function** », « **procedure** » ou encore « **let** » au lieu de « **def** ». C'est dire à quel point un cours de programmation peut être universel, sous réserve de ne pas se noyer dans les particularités de détail du langage choisi.

Nous allons commencer par les structures de données les plus simples.

6 Les structures de données

Une structure de données est définie par trois choses :

1. Un *type* qui est simplement un nom permettant de classifier les données relevant de cette structure de donnée. Par exemple, "grand", avec ses guillemets autour, est une donnée de type **string** (chaîne de caractères en français) et "petit" et "moyen" sont deux autres données de type **string** elles aussi. [Analogie avec les espèces]
2. Un ensemble qui définit avec précision quelles sont les *données pertinentes* de ce type. Par exemple l'ensemble de toutes les suites de caractères, de n'importe quelle longueur, sachant qu'un caractère peut aussi bien être une lettre qu'un chiffre, une ponctuation ou un caractère dit « de contrôle ». [idem]
3. L'ensemble des *opérations* que l'ordinateur peut utiliser pour effectuer des « calculs » sur les données précédentes. Cela comprend le nom de l'opération, par exemple **print**, et comment l'utiliser, c'est-à-dire quels *arguments* elle accepte et la nature du résultat. Dans l'exemple précédent, on a vu par exemple que **print** accepte un argument de type **string** (en réalité il en accepte aussi d'autres, on verra ça plus tard) et a pour résultat d'écrire à l'écran cette chaîne de caractère (sans les guillemets). [Analogie, mange(chat,souris) donne chat plus gros, mais ne mange pas lion et ne maigrira pas, à ceci près que l'ordinateur donne *toujours* le même résultat]

7 Les booléens

Il s'agit d'un ensemble de seulement 2 données pertinentes, dites *valeurs de vérité* : {True,False}. Le type est appelé **bool**, du nom de George Boole [1815-1864] : mathématicien anglais qui s'est intéressé aux propriétés algébriques des valeurs de vérité.

Opérations qui travaillent dessus :

Opération	Entrée	Sortie
not	bool	bool
and	bool×bool	bool
or	bool×bool	bool

De manière similaire aux tables de multiplications, on écrit facilement les tables de ces fonctions puisque le type est fini (et petit). On le fera en TD.

Exemples de calculs sur les booléens :

```
>>> print True
True
>>> print False
False
>>> print true
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
>>> print True and False
False
>>> type(True)
<type 'bool'>
>>> True and False
False
>>> True and
  File "<stdin>", line 1
    True and
    ^
SyntaxError: invalid syntax
```

Comme on le voit, Python offre aussi une opération `type` qui écrit quel est le type d'une donnée.

8 Les entiers relatifs

Théoriquement le type `int` correspond à l'ensemble \mathbb{Z} des mathématiques. En fait il est borné en Python, avec borne dépendante de la machine, mais assez grande pour ignorer ce fait ici. (`int` comme « integers »).

Les opérations qui travaillent dessus :

Opérations						<u>Entrée</u>	<u>Sortie</u>
+	-	*	/	%	**	<code>int×int</code>	<code>int</code>
<code>==</code>	<code><</code>	<code>></code>	<code><=</code>	<code>>=</code>	<code>!=</code>	<code>int×int</code>	<code>bool</code>

Exemples de calculs sur les entiers :

```
>>> type(46)
<type 'int'>
>>> 5 * 7
35
>>> print 5 * 7
35
>>> 5 / 2
2
>>> 5 % 2
1
>>> 5 ** 3
125
```

On remarque que la division `/` est *euclidienne* (pas de virgule dans le résultat) et que `%` donne le reste. Pour les comparaisons :

```
>>> 5 < 3
False
>>> 5 == 3 + 2
True
>>> 5 = 3 + 2
  File "<stdin>", line 1
```

```
SyntaxError: can't assign to literal
```

```
>>> 5 <= 5
```

```
True
```

On remarque que les opérations de calcul `+` `-` `*` `/` `%` `**` sont *prioritaires* sur les opérations de comparaison `==` `<` `>` `<=` `>=` `!=` car sinon « `5 == 3 + 2` » n'aurait pas eu plus de sens que « `False + 2` » (on n'additionne pas des choux et des carottes).

1 Les nombres réels

Historique : on dit aussi « les nombres flottants » par opposition aux 2 chiffres après la virgule genre caisse enregistreuse.

Le type `float` représente l'ensemble des nombres réels, avec cependant une précision limitée par la machine, mais les erreurs seront négligeables dans le cadre de ce cours. Par abus d'approximations on considère donc que `float` correspond à l'ensemble \mathbb{R} .

Opérations	Entrée	Sortie
+ - * / **	float×float	float
int	float	int
float	int	float
== < > <= >= !=	float×float	bool

On dispose des opérations suivantes :

Exemples d'utilisation :

```
>>> type(46.8)
<type 'float'>
>>> 5.0 / 2.0
2.5
>>> 5. / 2.
2.5
>>> 5.0 / 2
2.5
>>> 5 / 2
2
>>> 7.5 * 2
15.0
>>> 7.5 * 2
15.0
>>> int(7.5 * 2)
15
>>> int(7.75)
7
>>> float(3)
3.0
```

2 Les chaînes de caractères

Il existe 256 « caractères » qui couvrent à peu près tout ce que l'on peut taper au clavier en une fois, en utilisant les touches de nombres, lettres ou ponctuations, éventuellement en combinaison avec la touche *majuscule* (*shift* en anglais) ou bien la touche *contrôle*. Une *chaîne de caractères*, comme son nom l'indique, est une suite finie de caractères.

Le type des chaînes de caractères est généralement appelé `string` mais en Python il est appelé de manière abrégée `str`. Pour produire une chaîne de caractères, il suffit de l'écrire entre guillemets, simple ou doubles :

```
>>> "toto"
'toto'
>>> 'toto'
'toto'
>>> toto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'toto' is not defined
>>> toto = 56
>>> toto
56
```

Les guillemets sont nécessaires car sinon, Python interprète la chaîne de caractères comme le nom d'une variable, et la commande est alors interprétée par Python en « donner la valeur de `toto` ».

L'opération la plus courante sur les chaînes de caractères est la concaténation, notée avec un « + ». Les autres opérations fréquentes sont la longueur (nombre de caractères) notée `len` et les comparaisons (ordre du dictionnaire).

```
>>> "toto" + "tutu"
'tototutu'
>>> len("toto")
4
>>> "toto" < "tutu"
True
>>> "ab" < "aab"
False
>>> "a b" == "ab"
False
```

Noter l'absence d'espace entre "toto" et "tutu" après concaténation.

Opérations	Entrée	Sortie
+	<code>str</code> × <code>str</code>	<code>str</code>
<code>len</code>	<code>str</code>	<code>int</code>
<code>==</code> <code><</code> <code>></code> <code><=</code> <code>>=</code> <code>!=</code>	<code>str</code> × <code>str</code>	<code>bool</code>

3 Opération de substitution de chaînes de caractères

La structure de données des chaînes de caractères a encore une opération qui mérite une section d'explication à elle seule : la *substitution*, notée « % ».

Si la chaîne de caractères s_0 contient « %s » et si s_1 est une autre chaîne de caractères, alors s_0 % s_1 est la chaîne obtenue en remplaçant dans s_0 les deux caractères %s par la chaîne s_1 . Par exemple :

```
>>> "bonjour %s ; il faut beau aujourd'hui." % "Pierre Dupond"
"bonjour Pierre Dupond ; il faut beau aujourd'hui."
```

Plus généralement, s'il y a plusieurs « %s », il faut mettre entre parenthèses autant de chaînes (s_1, \dots, s_n) après l'opération %. Par exemple :

```
>>> nom = "Dupond"
>>> prenom="Pierre"
>>> genre = "homme"
>>> "Ce %s s'appelle %s et c'est un %s" % (prenom,nom,genre)
"Ce Pierre s'appelle Dupond et c'est un homme"
```

Si l'on veut mettre un entier relatif, on utilise « %d » au lieu de « %s » (d comme décimal et s comme string).

```
>>> age = 41
>>> "%s %s a %d ans." % (prenom,nom,age)
'Pierre Dupond a 41 ans.'
```

Pour un nombre réel, c'est « %f » (f comme float).

```
>>> "%s %s mesure %fm." % (prenom,nom,1.85)
'Pierre Dupond mesure 1.850000m.'
```

et si l'on veut imposer le nombre de chiffres après la virgule :

```
>>> "%s %s mesure %.2fm." % (prenom,nom,1.85)
'Pierre Dupond mesure 1.85m.'
```

Les différents encodages des substitutions présentés ici sont les plus utiles. Il y en a d'autres, plus rarement utilisés, qu'on trouve aisément dans tous les manuels.

4 Les expressions conditionnelles

Abandonnons quelques instants la description des structures de données classiques en programmation pour étudier deux éléments cruciaux du contrôle : les expressions conditionnelles et les définitions de fonctions ou de procédures.

Une expression conditionnelle « de base » est de la forme *SI condition ALORS action1 SINON action2*. En python cela donne par exemple, comme on l'a déjà vu :

```
>>> if "a b" == "ab" :
...     print "L'espace ne compte pas."
... else :
...     print "L'espace compte."
...
L'espace compte.
```

Dans le cas où l'on ne souhaite rien faire lorsque la condition est fausse, on peut omettre la partie `else`.

```
>>> if len("abc") != 3 :
...     print "YA UN PROBLEME !"
...
>>>
```

Enfin il arrive qu'on ait des « cascades » d'expressions conditionnelles et dans ce cas le mot-clef `elif` est une contraction de `else if` (exemple dans la section suivante).

5 Les définitions de fonctions

Lorsque certains calculs sont assez bien identifiés pour porter un nom, on a intérêt à leur donner un nom, ce qui simplifiera leur usage ultérieurement. Pour cela on définit des *fonctions*. Par exemple :

```
>>> def complementaire (n) :
...     if n == 'A' :
...         return 'T'
...     elif n == 'T' :
...         return 'A'
...     elif n == 'G' :
...         return 'C'
...     elif n == 'C' :
...         return 'G'
...     else :
...         return 'N'
...
>>> complementaire ('G')
'C'
>>> complementaire ('N')
'N'
>>> complementaire(4)
'N'
>>> complementaire(complementaire('A'))
'A'
```

... ICI ON COMMENTE CHAQUE MOT CLEF DU LANGAGE, la ligne vide de fin de def ...

Le mot-clef `return` indique ce que la fonction fournit comme résultat. Dans chaque cas, il faut donc n'avoir qu'un seul `return` possible. On peut alors réutiliser la fonction à toutes les sauces dans d'autres fonctions, exactement comme si le langage Python la fournissait parmi ses opérations :

```
>>> def nucleotide(n) :
...     return complementaire(n) != 'N'
...
>>> nucleotide ('A')
True
>>> nucleotide ('B')
False
>>> def transcription (adn) :
...     if adn == 'A' :
...         return 'U'
...     else :
...         return complementaire(adn)
...
>>> transcription ('A')
'U'
>>> transcription ('B')
'N'
>>> transcription ('G')
'C'
```

Noter qu'*une variable indique une place* : `adn`, aurait pu être remplacé par `n` ou `glop`...

1 Les définitions de procédures

Il est possible de définir des « fonctions » qui ne fournissent aucun résultat ! On appelle cela des *procédures*. Naturellement cela est d'un intérêt moindre et on préférera utiliser des fonctions chaque fois que possible.

Pour écrire une procédure, il suffit de ne jamais utiliser `return` dans la programmation. Par exemple, on peut se contenter d'écrire à l'écran le résultat du calcul, sans le fournir pour autant comme résultat « déclaré ».

```
>>> def trans (n) :
...     if n == 'A' :
...         print 'U'
...     else :
...         print complémentaire(n)
...
>>> trans ('A')
U
>>> trans ('G')
C
```

La différence ne saute pas aux yeux, si ce n'est que les guillemets n'apparaissent pas dans le « résultat » lors des appels de `trans`. En fait, `trans` imprime lui même U ou C à l'écran (commande `print`), alors que pour `transcription`, c'est le langage Python qui se chargeait d'écrire le résultat. La différence majeure, c'est que `trans` ne retourne aucun résultat. En voici la preuve :

```
>>> len(transcription('A'))
1
>>> type(transcription('A'))
<type 'str'>
>>> len(trans('A'))
U
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'NoneType' has no len()
>>> type(trans('A'))
U
<type 'NoneType'>
```

On remarque que `trans` est appelé dans chacune des deux commandes précédentes parce qu'il imprime « bêtement » à l'écran son résultat. Cependant il ne le fournit pas à la fonction qui l'appelle, donc `len` qui attend une chaîne de caractères, produit une erreur, et le « résultat » de `trans` est sans type.

Ainsi donc, on n'utilisera les procédures que pour imprimer divers résultats finaux à l'écran (IHM = Interface Homme-Machine).

2 Variables locales et variables globales

Lorsqu'une fonction ou une procédure utilise une variable, elle est toujours « locale » à cette fonction ou procédure. Cela signifie que sitôt que l'on est sorti de la fonction, on a « oublié » la variable.

```
>>> def triple (n) :
...     resultat = n * 3
...     return resultat
...
>>> triple (4)
12
>>> resultat
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
NameError: name 'resultat' is not defined
```

Mieux : si la variable était définie préalablement, elle n'est pas modifiée par la fonction :

```
>>> resultat = 128
>>> triple(2)
6
>>> resultat
128
```

En revanche, une variable *globale* qui n'est pas modifiée par la fonction est visible dans la fonction :

```
>>> def teste (n) :
...     if n > resultat :
...         print "plus grand"
...     else :
...         print "plus petit"
...
>>> teste (4)
plus petit
>>> teste (200)
plus grand
```

MAIS une variable ne peut pas être *à la fois* globale et locale :

```
>>> def ajuste (n) :
...     if n > resultat :
...         resultat = n
...
>>> ajuste (200)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in ajuste
UnboundLocalError: local variable 'resultat' referenced before assignment
```

Moralité : en Python, par défaut, une fonction ou une procédure ne peut pas modifier une variable globale. Ce serait de toute façon un style de programmation dangereux, puisque l'état des variables serait modifié implicitement. Mieux vaut écrire :

```
>>> def ajuste (n) :
...     if n > resultat :
...         return n
...     else :
...         return resultat
...
>>> resultat = ajuste(200)
>>> resultat
200
```

3 Compléments sur les chaînes de caractères, et while

Revenons un peu aux structures de données et commençons par des moyens complémentaires d'accéder aux caractères dans une liste.

Il est souvent pratique de pouvoir extraire le n -ième caractère d'une liste. Si s est une chaîne de caractères, alors l'opération d'*accès direct* $s[i]$ fournit le $(i+1)$ -ième caractère de la liste (c'est à dire que le premier caractère est en fait numéroté 0, le deuxième est numéroté 1, etc).

```

>>> "abcdef"[0]
'a'
>>> "abcdef"[1]
'b'
>>> "abcdef"[2]
'c'
>>> "abcdef"[5]
'f'
>>> "abcdef"[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range

```

On peut alors par exemple calculer le brin complémentaire d'une portion de génome (et plus seulement d'un seul caractère à la fois comme le fait la fonction `complementaire` programmée au cours précédent). Il faut alors parcourir le brin d'origine « à l'envers » :

```

>>> def brinCompl (c) :
...     resultat = ""
...     i = len(c)
...     while i > 0 :
...         i = i - 1
...         resultat = resultat + complementaire(c[i])
...     return resultat
...
>>> brinCompl ("AAATCCGT")
'ACGGATTT'

```

À cette occasion, on rencontre une nouvelle commande de contrôle : `while`, qui s'utilise syntaxiquement comme un `if` sans `else`, mais dont le bloc de programme est exécuté et ré-exécuté tant que la condition reste vraie. ATTENTION : si l'on gère mal le programme, une instruction `while` peut boucler indéfiniment (par exemple si l'on oublie `i = i - 1`). La boucle ci-dessus met donc deux variables locales à jour : l'indice `i` des caractères que l'on traite les uns après les autres et la chaîne de caractères, construite pas à pas, qui fournira le résultat final.

Encore une opération qui est parfois utile sur les chaînes de caractères : `s0 in s` prend en entrée deux chaînes de caractères et retourne un booléen qui indique si `s0` est une sous-chaîne (consécutive) de `s` (test d'appartenance).

```

>>> "to" in "tatetitotu"
True
>>> "io" in "tatetitotu"
False

```

Pour calculer le nombre de voyelles d'une chaîne de caractères, on peut alors programmer :

```

>>> def nbvoyelles (s) :
...     n = 0
...     i = 0
...     while i < len(s) :
...         if s[i] in "aeiouyAEIOUY" :
...             n = n + 1
...             i = i + 1
...     return n
...
>>> nbvoyelles("toto")
2

```

(cette fois on a parcouru la chaîne dans l'ordre).

Terminons sur les chaînes de caractères en signalant qu'il est possible d'extraire plus de un caractère à la fois avec la forme suivante : `s[i : j]`. Cette forme fournit la chaîne de caractères commençant à l'indice `i` et se terminant à `j-1` (et non pas `j`, ce serait trop simple...).

```
>>> "abcdef"[2:5]
'cde'
>>> "abcdef"[2:2]
''
```

Cela permet d'extraire n'importe quelle sous-chaîne d'une chaîne de caractères.

1 Les listes

Une *liste* en Python est une suite finie d'éléments quelconques. Le type des listes est appelé `list`.

L'ensemble des listes est constitué des expressions de la forme

$$[e_1, \dots, e_n]$$

où les e_i sont des données absolument quelconques (elles peuvent même être aussi elles-mêmes des listes).

```
>>> [2+3,"toto",2.5]
[5, 'toto', 2.5]
>>> type([5,"toto",2.5])
<type 'list'>
>>> [5,[9,5.5],"toto"]
[5, [9, 5.5], 'toto']
>>> type([5,[9,5.5],"toto"])
<type 'list'>
```

Il se peut qu'il n'y ait aucun élément dans la liste ; il s'agit alors de la *liste vide*, notée `[]`.

Beaucoup d'opérations travaillent sur les listes, à commencer par les opérations d'accès direct que nous venons de voir sur les chaînes de caractères. Ces dernières se transcrivent sur les listes de manière naturelle :

```
>>> ["toto",5,"tutu",5.5][1]
5
>>> ["toto",5,"tutu",5.5][1:3]
[5, 'tutu']
```

Le test d'appartenance ne fonctionne que pour un unique élément dans une liste, et non pas pour une sous-liste contrairement aux chaînes de caractères.

```
>>> "glop" in ["toto",5,"tutu",5.5]
False
>>> "tutu" in ["toto",5,"tutu",5.5]
True
>>> "tu" in ["toto",5,"tutu",5.5]
False
>>> [5,"tutu"] in ["toto",5,"tutu",5.5]
False
>>> [5,"tutu"] in ["toto",[5,"tutu"],5.5]
True
```

Comme pour les chaînes de caractères, on trouve également les opérations : `len` pour la longueur (nombre d'éléments dans la liste) et `+` pour la concaténation de deux listes.

```
>>> def homogene (l) :
...     if len(l) == 0 :
...         return True
...     else :
...         t = type(l[0])
...         i = 1
...         vu = True
...         while vu and i < len(l) :
...             vu = vu and t == type(l[i])
...             i = i + 1
```

```

...         return vu
...
>>> homogene (["toto",5,"tutu",5.5])
False
>>> homogene (["toto","glop","titi"])
True
>>> homogene ([])
True

```

Lorsqu'une liste `l` est homogène, les opérations `min(l)` et `max(l)` fournissent respectivement le plus petit et le plus grand élément de la liste. On peut également énumérer les éléments d'une liste homogène avec `for` :

```

>>> def somme (l) :
...     s = 0
...     for n in l :
...         s = s + n
...     return s
...
>>> somme ([])
0
>>> somme ([2,2,5,3])
12
>>> min ([2,2,5,3])
2
>>> max ([2,2,5,3])
5

```

2 Instructions de modification de liste

Contrairement aux chaînes de caractères, une variable de type liste peut être partiellement modifiée sans réaffecter toute la variable. Par exemple :

```

>>> coursesAfaire=["beurre","pain","artichaut","vin rouge"]
>>> coursesAfaire
['beurre', 'pain', 'artichaut', 'vin rouge']
>>> coursesAfaire[2]="avocat"
>>> coursesAfaire
['beurre', 'pain', 'avocat', 'vin rouge']

```

alors que :

```

>>> nom="Samon"
>>> nom[1]
'a'
>>> nom[1]='i'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

```

Ainsi, si `v` est une variable de type liste, alors l'instruction « `v[i]=expr` » remplace l'élément d'indice `i` dans la liste par la valeur du calcul de l'expression `expr`.

On peut aussi remplacer une portion entière de liste par une autre liste :

```

>>> coursesAfaire[1:3]=["baguette","chou","vinaigre"]
>>> coursesAfaire
['beurre', 'baguette', 'chou', 'vinaigre', 'vin rouge']

```

Remarquez que dans l'instruction « `v[i :j]=expr` » on remplace les indices de `i` à `(j - 1)`...

1 Instructions de modification de liste (suite)

Rappel : dans l'instruction « $v[i:j]=expr$ » on remplace les indices de i à $(j-1)$...

```
>>> coursesAfaire = ['beurre', 'baguette', 'chou', 'vinaigre', 'vin rouge']
>>> coursesAfaire[1:3]="toto"
>>> coursesAfaire
['beurre', 't', 'o', 't', 'o', 'vinaigre', 'vin rouge']
>>> coursesAfaire[1:3]=8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
```

Par défaut, l'expression doit être de type liste mais Python « fait ce qu'il peut » pour transformer l'expression en liste (ici, une chaîne en liste de caractères). Plus généralement, une structure de données qui peut être transformée en liste de manière naturelle est dite « itérable » en Python.

Il est également possible de supprimer un élément ou une tranche d'éléments d'une variable de type liste :

```
>>> coursesAfaire
['beurre', 't', 'o', 't', 'o', 'vinaigre', 'vin rouge']
>>> del coursesAfaire[2]
>>> coursesAfaire
['beurre', 't', 't', 'o', 'vinaigre', 'vin rouge']
>>> del coursesAfaire[1:4]
>>> coursesAfaire
['beurre', 'vinaigre', 'vin rouge']
```

Évitez à tout prix de prendre des indices hors des bornes, le comportement n'est pas garanti; cela ne produit pas forcément une erreur!

```
>>> del coursesAfaire[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> del coursesAfaire[1:6]
>>> coursesAfaire
['beurre']
```

En revanche, il est possible de donner des indices négatifs : cela signifie « en partant de la fin ». Ainsi « $v[-i]=expr$ » est équivalent à « $v[\text{len}(v)-i]=expr$ » :

```
>>> coursesAfaire=["beurre","pain","artichaut","vin rouge"]
>>> coursesAfaire[-1]
'vin rouge'
>>> coursesAfaire[1:-1]
['pain', 'artichaut']
>>> del coursesAfaire[1:-1]
>>> coursesAfaire
['beurre', 'vin rouge']
```

Enfin pour les tranches d'éléments (avec le « : »), un indice manquant va jusqu'au bout de la liste :

```
>>> coursesAfaire=["beurre","pain","artichaut","vin rouge"]
>>> coursesAfaire[1:]
['pain', 'artichaut', 'vin rouge']
```

```

>>> coursesAfaire[2:]
['artichaut', 'vin rouge']
>>> coursesAfaire[:2]
['beurre', 'pain']
>>> coursesAfaire[:]
['beurre', 'pain', 'artichaut', 'vin rouge']

```

2 Exemples de programmes sur les listes

Compresser une liste de nucléotides où l'on suppose qu'il y a beaucoup de répétitions successives de nucléotides. L'idée est de ne pas recopier plusieurs fois un nucléotide répété mais de mémoriser dans la liste son nombre d'occurrences. Par exemple la fonction `comprime` appliquée à `['A','A','A','G','C','T','T','C','C','C','G']` retourne comme résultat `['A',3,'G','C','T',2,'C',3,'G']`.

```

>>> def comprime (l) :
...     r = []
...     compteur = 0
...     precedent = ""
...     for nucl in l :
...         if nucl == precedent :
...             compteur = compteur + 1
...         else :
...             # mémoriser le nucleotide precedent
...             if compteur > 1 :
...                 r = r + [ precedent , compteur ]
...             elif compteur == 1 :
...                 r = r + [ precedent ]
...             # préparer la suite
...             precedent = nucl
...             compteur = 1
...     #traiter le dernier nucleotide en sortant du for
...     if compteur >= 2 :
...         return r + [ precedent , compteur ]
...     else :
...         return r + [ precedent ]

```

et pour décompresser :

```

>>> def deploie (l) :
...     r = []
...     precedent = ""
...     for elem in l :
...         if type(elem) != type(0) :
...             r = r + [elem]
...             precedent = elem
...         else :
...             while elem > 1 :
...                 r = r + [precedent]
...                 elem = elem - 1
...     return r

```

1 Les dictionnaires

Et si je dois acheter 5 pains, 2 artichauts, 2 plaquettes de beurre et 1 bouteille de vin ?

Il faut cette fois *associer*, d'une part, des entités (`beurre`, `pain`, etc), et d'autre part des informations sur chaque entité (ici une quantité entière mais ce pourrait être une information aussi complexe que l'on veut). La structure de données qui mémorise cela est classiquement appelée une « liste d'association » mais en raison de la ressemblance avec un dictionnaire qui associe à chaque mot une définition, Python appelle ce type `dict` (comme dictionnaire).

L'ensemble des dictionnaires est constitué des expressions de la forme

$$\{ e_1 : d_1 , \dots , e_n : d_n \}$$

où les e_i sont des données *élémentaires* quelconques et les d_i sont des données quelconques. Par donnée élémentaire, on entend ici une donnée qui n'est pas elle-même une liste ou un dictionnaire.

```
>>> coursesPrecises = { "baguette":2 , "vin":1 , "saumon":2 }
>>> coursesPrecises[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
>>> coursesPrecises["vin"]
1
```

On constate que l'on n'accède pas au contenu d'un dictionnaire par un indice entier donnant la position dans le dictionnaire mais, et c'est plus simple, par le nom des entités appartenant au dictionnaire.

Un dictionnaire peut être hétérogène :

```
>>> chelou = { "vin":2 , (4,5,6):"c'est un triplet" , 3.5:78 }
>>> type(chelou)
<type 'dict'>
>>> chelou[(4,5,6)]
"c'est un triplet"
>>> chelou[3.5]
78
>>> chelou[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
```

L'accès à un dictionnaire se fait par les entités, pas par les informations qui lui sont associées.

On peut compléter ou modifier un dictionnaire avec une instruction d'affectation :

```
>>> coursesPrecises
{'baguette': 2, 'saumon': 2, 'vin': 1}
>>> coursesPrecises["saumon"]=1
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'vin': 1}
>>> coursesPrecises["gateau"]=8
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'vin': 1, 'gateau': 8}
```

On peut aussi supprimer une entité (et son information associée) avec `del`, obtenir le nombre d'entités avec `len` et tester si une entité est dans le dictionnaire avec `in`.

```

>>> del coursesPrecises["vin"]
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'gateau': 8}
>>> len(coursesPrecises)
3
>>> "gateau" in coursesPrecises
True
>>> "chou" in coursesPrecises
False

```

Enfin, on peut *itérer* un bloc d'instructions sur toutes les entités d'un dictionnaire, un peu comme avec un `while` sur une chaîne de caractères ou sur une liste. Pour cela on utilise `for...in...`

```

>>> def total (d) :
...     s=0
...     for e in d :
...         s = s + d[e]
...     return s
...
>>> total(coursesPrecises)
11
>>> total(chelou)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in total
TypeError: unsupported operand type(s) for +: 'int' and 'str'

```

Moralité : un dictionnaire avec des entités ayant toutes le même type et des informations ayant toutes le même type, c'est souvent mieux !

2 Exemples de programmes sur les dictionnaires

Si les clefs doivent être des données élémentaires, les informations qui leur sont associées peuvent en revanche être aussi complexes que l'on veut. Entre autres, ce peuvent être elles-mêmes des dictionnaires.

```

>>> anniv = { "Pierre" : {"jour":3,"mois":"jan","an":1965} ,
...          "Paul" : {"jour":18,"mois":"nov","an":1998} ,
...          "Irène" : {"jour":25,"mois":"mar","an":1982} }
>>> anniv["Irène"]
{'mois': 'mar', 'jour': 25, 'an': 1982}
>>> anniv["Paul"]["an"]
1998

```

Supposons que l'on veuille écrire une fonction `lesAges` qui prend en entrée un dictionnaire `d` d'anniversaires similaire à `anniv` et une date courante `t`, et retourne en sortie la liste des âges des personnes du dictionnaire `d`.

```

def lesAges (d,t) :
    r = []
    for p in d :
        r = r + [...???....]
    return r

```

calculer l'âge d'une personne en fonction de son « dictionnaire anniversaire » et du dictionnaire représentant la date courante est de prime abord compliqué. Dans de tels cas, il faut toujours procéder de la manière suivante :

- on fait l'hypothèse qu'il existe des fonctions qui résolvent les problèmes compliqués (ici faire la différence de deux dates en nombre d'années entières : `diffDates(t1,t2)`);
- on programme comme si ces fonctions existaient (ici, `r = r + [diffDates(d[p],t)]`); naturellement Python n'acceptera cette programmation qu'après avoir réellement programmé ces fonctions;

- on s'attaque ensuite, une par une, aux fonctions identifiées par le processus précédent, et on leur applique exactement la même approche;
- la programmation est terminée lorsque les fonctions deviennent suffisamment simples pour ne plus avoir besoin de faire appel à des fonctions hypothétiques.

Ici, pour programmer `diffDates`, si la date de `t1` vient après celle de `t2` dans l'année, il suffit de faire la soustraction des années; sinon il faut soustraire 1 au résultat :

```
def diffDates (t1,t2) :
  if apresAnnee(t1,t2) :
    return t1["an"] - t2["an"]
  else :
    return t1["an"] - t2["an"] - 1
```

La fonction hypothétique qu'il faut maintenant programmer est `apresAnnee`. Ce qui est difficile est alors de comparer des mois qui ne sont pas des entiers mais des chaînes de caractères. Qu'à cela ne tienne, on fait l'hypothèse que la fonction `numero` fournit le numéro du mois.

```
def apresAnnee (u,v) :
  if numero(u["mois"]) > numero(v["mois"]) :
    return True
  elif numero(u["mois"]) == numero(v["mois"]) :
    return u["jour"] > v["jour"]
  else :
    return False
```

ou mieux :

```
def apresAnnee (u,v) :
  if numero(u["mois"]) == numero(v["mois"]) :
    return u["jour"] >= v["jour"]
  else :
    return numero(u["mois"]) > numero(v["mois"])
```

et enfin :

```
def numero (m) :
  if m in ["jan", "Jan", "janv", "Janv", "janvier", "janvier"] :
    return 1
  elif m in ["fev", "Fev", "fevrier", "Fevrier", "fév", "Fév", "février", "Février"] :
    return 2
  elif ...
    ...
  else :
    print "numero: mois mal formé !"
    return 0
```

1 Compléments mineurs mais utiles sur le type str

Rappel : la fonction `int` convertit en entier (si possible) l'argument qu'on lui donne. Pour les `float`, ça tronque, pour les chaînes de caractères *ne contenant que des chiffres*, ça marche aussi. Au passage, `str` fait le contraire.

```
>>> int(12.7)
12
>>> int("12")
12
>>> int("12.7")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.7'
>>> str(126)
'126'
```

Si l'on veut ignorer les caractères qui ne sont pas des chiffres, il faut le programmer soi-même :

```
>>> def intsouple (s) :
...     r = 0
...     for c in s :
...         if c >= '0' and c <= '9' :
...             r = 10 * r + int(c)
...         #sinon on ignore le caractère
...     return r
...
>>> intsouple ("to3to4glop0")
340
```

La « fonction » `raw_input` prend en entrée une chaîne de caractères et fournit en sortie une chaîne de caractère. Elle écrit à l'écran la chaîne qu'on lui donne en argument et attend que l'utilisateur tape une ligne (terminée par un retour chariot); le résultat de la « fonction » est alors la chaîne de caractère qu'a tapée l'utilisateur.

```
>>> def askint () :
...     s = raw_input("entrez un entier positif : ")
...     n = intsouple(s)
...     if str(n) != s :
...         print "Vous tapez à coté des touches !"
...     return n
...
>>> 2 * askint()
entrez un entier positif : 45
90
>>> 2 * askint()
entrez un entier positif : glop9u0
Vous tapez à coté des touches !
180
```

2 Lecture de fichiers

Un fichier est une suite de caractères mémorisés sur le disque dur de la machine dans un endroit caractérisé par un « nom de fichier ». Les contraintes techniques font qu'on doit charger du disque dur vers la mémoire ces caractères les uns après les autres pour pouvoir travailler dessus. Ainsi, en programmation, on gère un fichier comme un « flux de caractères » sur lequel on peut « ouvrir un robinet » pour charger un certain nombre de caractères, en partant du début du fichier. Après usage, il faut « fermer le robinet ».

Il y a beaucoup de fichiers sur un disque dur, donc beaucoup de robinets potentiels. L'ensemble de tous ces robinets potentiels constitue le type de données appelé `file`.

Les opérations qui travaillent sur le type `file` sont nombreuses. La première de toutes consiste à « ouvrir un fichier », ce qui met le robinet correspondant à disposition du programme qui a ouvert le fichier. La fonction `open` prend en argument un nom de fichier (qui est une chaîne de caractères) et retourne un objet de type `file` (le robinet).

Par la suite, le programme disposera du « robinet », qu'il pourra ouvrir à chaque instant pour récupérer des quantités déterminées de caractères, ceci avec la « fonction » `read` qui prend en argument le nombre de caractères qu'on veut lire. La première ouverture de robinet « lit » les premiers caractères du fichiers; l'ouverture suivante, toujours avec `read` reprendra à partir du premier caractère pas encore lu du fichier, et ainsi de suite.

Après utilisation du fichier, il est *très important* de « rendre le robinet au système » pour que d'autres programmes puissent l'utiliser ou le modifier à leur tour. La fonction `close` assure cette tâche en rendant le robinet de nouveau inaccessible au programme. Imaginons un fichier sur le disque, appelé « toto.txt », qui contiendrait la suite de caractères « TitiTrucMachinChouette ». On peut par exemple écrire :

```
>>> rob = open("toto.txt")
>>> w = rob.read(4)
>>> x = rob.read(4)
>>> y = rob.read(6)
>>> z = rob.read(8)
>>> rob.close()
>>> w
'Titi'
>>> x
'Truc'
>>> y
'Machin'
>>> z
'Chouette'
```

On observe une syntaxe nouvelle : au lieu d'écrire `read(rob,4)` ou `close(rob)`, on écrit `rob.read(4)` et `rob.close()`. Ceci est dû au fait que le robinet `rob` n'est pas une donnée comme celles qu'on a vues jusqu'à maintenant : c'est un *objet*.

Un objet en informatique est une entité qui peut « réagir » à un programme de plusieurs manières différentes en fonction de l'état dans lequel il est. Par exemple, après `open`, l'état de l'objet `rob` est d'être en début de fichier. Ainsi la première lecture `rob.read(4)` vaut "Titi" et change de manière implicite l'état de `rob` : maintenant le robinet en est au cinquième caractère, et on le voit parce que le prochain appel de `rob.read(4)` ne fournit plus Titi mais Truc.

Les opérations qui travaillent spécifiquement sur des *objets* en informatique sont appelées des *méthodes* et s'utilisent avec cette notation « pointée » qui se veut rappeler qu'une méthode ne prend pas seulement son objet en argument mais peut aussi changer son état.

`close` est finalement la méthode « d'apoptose du robinet »... et change radicalement l'état de son objet en le faisant disparaître (le robinet disparaît du programme mais le fichier reste parfaitement visible dans le disque dur).

D'autres méthodes utiles sont les suivantes :

`readline()` : cette méthode lit autant de caractères que nécessaire pour aller jusqu'en fin de ligne (le retour-chariot inclus). Par exemple si `gamin.txt` contient les 4 lignes

```
premier doigt
deuxième doigt
second doigt
troisième doigt
```

alors on peut programmer :

```
>>> def affiche (f) :
...     fich = open(f)
```

```

...     i = 1
...     ligne = fich.readline()
...     while ligne != "":
...         print "%i : %s" % (i,ligne)
...         i = i + 1
...         ligne = fich.readline()
...     fich.close()
...
>>> affiche("gamin.txt")
1 : premier doigt

2 : deuxième doigt

3 : second doigt

4 : troisième doigt

```

Lorsque l'objet fichier arrive en fin de fichier (plus rien à lire), la méthode `readline` retourne une chaîne vide. La procédure précédente s'arrête donc à la fin du fichier. Remarquons aussi que les trois premières lignes sont suivies d'une ligne vide : c'est dû au fait que `readline` inclut le retour-chariot *et* que `print` en écrit également un (au passage, une ligne vide en milieu de fichier n'arrête pas le `while` car `s` contient alors le retour-chariot). Enfin, si le fichier ne se termine pas par un retour-chariot, alors le dernier `readline` fournit les caractères qui restent jusqu'à la fin de fichier, sans retour-chariot final.

`tell` : cette méthode retourne la position du robinet, c'est-à-dire le nombre de caractères déjà lus.

```

>>> r = open("toto.txt")
>>> r.read(4)
'Titi'
>>> r.read(4)
'Truc'
>>> r.tell()
8L
>>> r.close()

```

Le « L » signifie « entier long » car la taille d'un fichier peut être vraiment très grande et python encode la taille dans ce type, qui se manipule exactement comme les entiers de type `int`.

Ajoutons qu'on peut faire un `for` sur un fichier : cela énumère les lignes du fichier :

```

>>> f = open("gamin.txt")
>>> for ligne in f :
...     print len(ligne)
...
14
15
13
15
>>> f.close()

```

1 Lecture de fichiers (suite)

Ajoutons qu'on peut faire un `for` sur un fichier : cela énumère les lignes du fichier :

```
>>> f = open("gamin.txt")
>>> for ligne in f :
...     print len(ligne)
...
14
15
13
15
>>> f.close()
```

ICI LE COURS DE JEAN-PAUL COMET RESTE À RÉDIGER. ...

Écriture d'une procédure `more(fichier)` qui affiche à l'écran le contenu du fichier. Attention, la difficulté réside dans la dernière ligne, il faut tester si la dernière ligne contient un `"\n"` ou non.

2 Ecriture de fichiers

Il est possible « d'entrer un flux contraire dans un robinet » c'est-à-dire d'écrire dans un fichier. Il faut alors ouvrir le fichier non plus en lecture mais en écriture :

`open(nom, 'w')` est une fonction qui crée un fichier de ce `nom`. Si un fichier du même nom existait, il est remis à zéro (vidé) par `open`.

Ensuite, il suffit d'utiliser la méthode `write` qui prend en argument une chaîne de caractères et a pour effet de l'écrire dans le fichier.

```
>>> f = open("nouveau.txt", "w")
>>> f.write("Toto")
>>> f.write("Tutu")
>>> f.write("Glop\n")
>>> f.write("ligne numero 2 seulement\n")
>>> f.close()
```

Le fichier contient alors :

```
TotoTutuGlop
ligne numero 2 seulement
```

et l'on remarque qu'il faut explicitement écrire, avec `write`, les retour-chariots sous la forme « `\n` ».

Écriture dans un fichier, en revenant sur la fonction `open` qui peut prendre 2 arguments. Le deuxième pouvant être `"r"`, `"w"`, `"a"` ou `"U"`.

1 Les modules en Python

En ce point du cours, nous avons étudié la majorité des structures de données classiques (manquent les *arbres* et les *graphes* qui relèvent de cours plus avancés) et la quasi-totalité des structures de contrôle (le *traitement d'exceptions* et la *récurtivité* relèvent de cours plus avancés). On peut même affirmer que quel que soit le problème posé, s'il existe un programme qui le résoud alors nous avons vu tous les éléments pour le faire.

Il existe cependant des questions « standard » qu'il serait peu productif de résoudre chacun pour soi : ces questions sont tellement courantes que les programmes qui les résolvent sont stockés dans des *bibliothèques* (*libraries* en anglais). En Python, une telle bibliothèque est constituée de nombreux *modules* et un module est constitué de plusieurs fonctions (ou procédures) déjà programmées. Dans un module, on prend soin de regrouper les fonctions qui permettent de gérer un type de problème donné, bien souvent ce sont simplement les opérations associées à une structure de données adaptée au problème.

On peut même écrire soi-même des modules. Lorsque l'on veut écrire un gros logiciel, c'est généralement une bonne idée de le découper en modules (qui seront du même coup réutilisables par ailleurs), en suivant généralement le principe « au moins un module par nouvelle structure de donnée ». Il suffit alors d'importer les modules pour utiliser les fonctions qui y ont été programmées.

Pour créer un module, il suffit de programmer les fonctions qui le constituent dans un fichier portant le nom du module, suivi du suffixe « .py ». Depuis un (autre) programme en Python, il suffit alors d'utiliser la primitive `import` pour pouvoir utiliser ces fonctions. Par exemple, si l'on crée un fichier appelé `test.py` et contenant :

```
def voir (s) :  
    print "la chaine est : %s" % s  
    print "et voila!"
```

```
def double (n) :  
    return 2 * n
```

alors sous Python on peut écrire :

```
>>> import test  
>>> test.double(3)  
6  
>>> double (3)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'double' is not defined  
>>> test.voir ("toto")  
la chaine est : toto  
et voila!  
>>> test.toto (3)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'module' object has no attribute 'toto'
```

On remarque que :

- `import` est une instruction de contrôle de Python très particulière puisqu'il n'est pas nécessaire d'écrire `test` entre guillemets pour en faire une chaîne de caractères, pourtant, `import` ne considère pas `test` comme un nom de variable!
- On n'écrit pas le suffixe du fichier mais seulement le nom du module.
- Pour utiliser les fonctions et procédures du module, il faut les préfixer du nom du module avec un point au milieu.
- Un fichier `test.pyc` est créé automatiquement par Python. Il contient la version compilée des fonctions et procédures du module.

Ajoutons que l'on peut inclure une variable globale dans un module. L'usage est que son nom commence généralement par le caractère souligné « `_` » (par exemple `_test_MaVariableGlobale`).

Python trouve ses modules dans le répertoire courant, comme on vient de le voir, mais aussi dans des répertoires prédéfinis où se trouvent des modules utiles écrits par d'autres programmeurs.

Un module peut faire appel à un autre module : il suffit qu'il contienne lui-même une instruction `import`.

Les modules dits « standard » sont documentés dans de nombreux livres sur Python et, avec un peu d'habitude, l'appel à des modules renforce considérablement la rapidité de programmation. Libre à vous d'explorer la liste presque sans fin des modules existants (et qui s'enrichit tous les jours) : vous avez maintenant toutes les bases de programmation nécessaires pour comprendre leur usage à partir de leur documentation.

2 Gestion du système de fichiers avec le module `os`

Un module tout à fait central dans tous les langages de programmation est celui qui permet de gérer le système de fichiers présent sur le disque dur de la machine. Son nom est `os` (en minuscules) comme *operating system* (*système d'exploitation* en français).

Avant d'utiliser `os` pour gérer le système de fichiers, il faut savoir (À RÉDIGER ULTÉRIEUREMENT AVEC FIGURES) :

- qu'il peut y avoir plusieurs disques dur sur un ordinateur, ou qu'un disque dur peut être partagé en plusieurs partitions. Sous Mac ou un système Unix comme Linux cette séparation des partitions est gérée pour qu'on ait l'impression de n'avoir qu'un seul disque. Sous Windows, ces partitions sont vues comme plusieurs « lecteurs » différents numérotés A, B, C, D, etc. En général A et B sont présents pour des raisons historiques et sont réservés à deux lecteurs de disquettes. Le disque C est généralement celui sur lequel le système Windows est installé et les suivants sont des disques de données ou des lecteurs/graveurs de CD, DVD, etc.
- Un disque est organisé en arbre avec des répertoires (directories) qui peuvent contenir des sous-arbres et des fichiers (files) qui contiennent les « vraies » données et sont nécessairement des feuilles de l'arbre. . .
- Un nœud de l'arbre est repéré par son chemin depuis la racine, qui est souvent appelé son adresse.
- On note généralement « .. » le répertoire père d'un répertoire.
- Compte tenu de la taille de cet arbre et de la longueur des chemins, et du fait qu'on travaille longtemps sous un même répertoire, il est pratique d'avoir un répertoire dit « courant ». Cela permet de ne donner que les adresses à partir du répertoire courant.
- chemins relatif/absolu, mode d'un fichier, etc.

```
>>> import os
>>> os.getcwd()
'/home/bernot'
>>> os.chdir("foutoir")
>>> os.getcwd()
'/home/bernot/foutoir'
>>> os.chdir("dirTest")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory: 'dirTest'
>>> os.mkdir("dirTest")
>>> os.chdir("dirTest")
>>> os.chdir("..")
>>> os.getcwd()
'/home/bernot/foutoir'
>>> os.listdir('.')
['plan.jpg', 'oraux_membres_Jury.xls', 'dirTest']
```

Ce premier usage du module `os` met en évidence 4 premières fonctions ou procédures de ce module :

- `getcwd` retourne la chaîne de caractères du chemin de la racine au répertoire courant.
- `chdir` change le répertoire courant.
- `mkdir` fabrique un nouveau répertoire.
- `listdir` prend en argument un chemin et retourne la liste des noms de fichiers ou répertoires fils

Au passage, la variable `os.linesep` fournit la chaîne de caractère `"\n"` utilisée par le système d'exploitation sur lequel on est :

```
>>> os.linesep
'\n'
```

```
>>> print "toto%stutu" % os.linesep
toto
tutu
```

Il s'agit d'une variable, non d'une fonction, d'où l'absence de parenthèses. On peut entre autres tester `len(os.linesep)`, ce qui facilite certaines gestions de fichier vues précédemment.

1 Gestion du système de fichiers avec le module `os` (suite)

D'autres fonctions ou procédures utiles pour la gestion du système de fichiers sont :

- `listdir` prend en argument un chemin et retourne la liste des noms de fichiers ou répertoires fils

```
>>> os.listdir('.')  
['plan.jpg', 'oraux_membres_Jury.xls', 'dirTest']
```
- `rename` prend en argument les chemins source et cible : peut déplacer dans l'arbre, aussi bien un fichier qu'un sous-arbre complet
- `remove` prend en argument le chemin d'un fichier et le supprime
- `rmdir` comme `remove`, mais pour un répertoire vide
- `chmod` prend 2 arguments : le chemin et le mode (toujours à la manière Unix)

2 Système de fichiers, compléments du sous-module `os.path`

Le module `os` contient un « sous-module » `os.path` qui est automatiquement importé avec `os` et fournit d'autres fonctions bien utiles :

- `isfile` dit si le chemin pointe sur un fichier standard
- `isdir` ... si c'est un répertoire
- `islink` ... si c'est un lien symbolique (« raccourci » sous windows)
- `exists` dit si le chemin donné en argument existe (fonction booléenne)
- `getsize` taille en octets du fichier désigné par le chemin donné en argument
- `abspath` prend un chemin en entrée et retourne sa version en adresse absolue.
- `basename` fournit le dernier nom du chemin
- `dirname` le contraire