Module 0-M02

INTRODUCTION À LA PROGRAMMATION

Introduction à l'algorithmique
Éléments de Python

Camille Coti
camille.coti@iutv.univ-paris13.fr
IUT de Villetaneuse - Département R&T - 2013-2014

Table des matières

Préambule

Ce polycopié est une version rassemblée de deux polycopiés utilisés précédemment dans le module M02 : le polycopié d'introduction à l'algorithmique, et celui d'introduction au langage de programmaton Python. Alors que l'introduction à la programmation est souvent enseignée de façon linéaire, en présentant tout d'abord la notion d'algorithmes, puis leur mise en place dans un langage de programmation, le choix a été fait ici de présenter directement et en parallèle l'implémentation des structures algorithmiques dans un langage de programmation. Le but est d'aborder plus rapidement la programmation de façon concrète, et d'associer les structures algorithmiques avec leur expression dans un langage de programmation clair et facilement lisible comme Python.

Les structures algorithmiques seront présentées de manière générale, en langage algorithmique. Leur expression en Python sera ensuite présentée : gardez bien à l'esprit que ce n'est qu'une implémentation de l'algorithme dans un langage particulier. L'algorithme lui-même peut être exprimé dans d'autres langages de programmation. Il est fondamental que vous compreniez la structure algorithmique elle-même avant de penser à son expression dans un langage de programmation, quel qu'il soit.

Sont présentés ici quelques éléments de Python, reprenant le contenu des slides qui sont affichés en cours et disponibles sur Internet. Ce n'est pas une présentation extensive du langage Python : vous trouverez pour cela de très bons livres et documents, dont beaucoup sont librement disponibles sur Internet. C'est simplement une (légère) introduction au langage, vous permettant de mettre le pied à l'étrier et, si besoin ultérieur, de vous documenter sur des sujets plus approfondis ensuite.

Beaucoup d'aspects du langages sont volontairement occultés, notamment l'approche orientée objet. Le module dans lequel nous nous inscrivons ici est une *introduction* à la programmation, vous présentant des concepts et vous donnant les compétences pour être en mesure de suivre la suite de la formation.

1 Introduction

1.1 Définitions

Un algorithme est une *série d'instructions* qui doit être exécutée par un programme. À partir des données d'entrée, un algorithme doit donner un résultat. Une métaphore fréquemment utilisée consiste à définir un algorithme comme la recette qui, à partir des ingrédients, permet d'obtenir un gâteau.

Un programme informatique est la traduction d'un algorithme dans un langage de programmation. Les algorithmes constituent un formalisme permettant de décrire cette série d'instructions *indépendamment d'un langage de programmation en particulier*. On décrit généralement les algorithmes dans un langage compréhensible par un humain ou *pseudocode*.

Un algorithme est donc constitué des éléments suivants :
- Un début et une fin
- Un nom
- Des données d'entrée
- Des données de sortie, qui sont le résultat du calcul effectué par l'algorithme
- Un ensemble d'instructions exécutées par l'algorithme

La conception correcte d'un programme informatique commence par la *bonne compréhension du problème* et la *conception d'un algorithme qui fait bien ce que l'on attend*. Un excellent programmeur n'obtiendra pas un programme correct si il utilise un algorithme faux. L'étude du problème et sa transcription algorithmique sont donc fondamentales dans la pratique de la programmation. De même, le coût d'un calcul est défini principalement par son algorithme. La résolution efficace d'un problème dépend donc de l'*efficacité de l'algorithme* utilisé par le programme.

1.2 Enseignement de Python dans votre formation

Le choix a été fait par l'équipe pédagogique d'enseigner Python dans votre filière pour les nombreux avantages qu'il présente : simplicité, haut niveau d'abstraction (le programmeur n'a pas à gérer la mémoire lui-même, souplesse des structures de données disponibles...). Il offre de plus un spectre d'utilisations très large et nous semble bien correspondre aux débouchés de votre formation.

Il présente les avantages d'un puissant *langage de script*
- Rapidité de développement
- Utilisation pour des scripts d'administration système, analyse de fichiers textuels (logs...)
- Langage pour le web : développement d'applications web, scripts CGI, serveurs...
- Accès aux bases de données relationnelles

... mais également ceux d'un tout aussi puissant *langage de programmation*
- Programmes complets en Python
- Interfaçage facile avec des bibliothèques dans d'autres langages (C, C++, Fortran...)
- Accès aux interfaces graphiques facilité
- Permet de se concentrer sur l'algorithme plutôt que l'implémentation : calcul scientifique pour les non-informaticiens...

1.3 Un langage interprété

Python est un langage interprété, par opposition aux langages compilés.

On dispose de deux moyens d'exécuter des scripts Python :

En *ligne de commande*: dans l'interpréteur interactif
- On lance l'interpréteur, dans lequel on tape des instructions

```
1 coti@maximum:~$ python
2 Python 2.7.3rc2 (default, Apr 22 2012, 22:30:17)
3 [GCC 4.6.3] on linux2
4 Type "help", "copyright", "credits" or "license" for
     more information.
5 >>> print 3
6 3
```

– Rapidité de mise en place
– Permet de tester des choses

On peut également *exécuter un script*. Le script doit être un fichier contenant le programme. Il existe deux possibilités d'exécuter un script.

On peut l'exécuter directement : on le lance, et le script appelle l'interpréteur. Il faut alors être attentif aux droits associés au fichier : celui-ci doit être exécutable (`+x`).

```
1 coti@maximum:~$ ./monscript.py
```

On peut également le lancer dans l'interpréteur : on appelle l'interpréteur en lui passant le chemin vers le script en paramètre.

```
1 coti@maximum:~$ python ./monscript.py
```

Si il est lancé seul, un script Python doit remplir deux conditions :
– Être *exécutable* (`+x`)
– Spécifier le chemin vers l'interpréteur : c'est le *shebang*
On place le shebang au début du fichier :

```
1 #!/usr/bin/python
```

Attention : si le shebang ne pointe pas vers le bon interpréteur, on a une erreur.

```
1 coti@thorim:/tmp$ cat demo.py
2 #!/usr/python
3 coti@thorim:/tmp$ ./demo.py
4 -bash: ./demo.py: /usr/python: bad
     interpreter: No such file or directory
```

Cette approche permet de faire coexister plusieurs versions de Python sur le système, et de contrôler quelle version de l'interpréteur est utilisée pour exécuter un script donné.

1.4 Commentaires en Python

Pour commenter une ligne de code on utilise `#`
– Une ligne commentée n'est pas exécutée
– Commente tout ce qui suit la ligne
– Commente une et une seule ligne : le commentaire s'arrête à la fin de la ligne

Pour commenter plusieurs lignes, on encadre la section à commenter par `'''` (triple quote) ou `"""` (triple double quote)
– Le commentaire commence au triple quote
– Il se termine au triple quote suivant
– Impossible d'imbriquer des commentaires sur plusieurs lignes

```
1 #!/usr/bin/python
2
3 '''
4 un commentaire
5 sur plusieurs lignes
6 '''
7
8 # un commentaire sur une ligne
```

1.5 Affichage d'une chaîne de caractères

On affiche une chaîne de caractères avec l'instruction `print`. On peut préciser les éléments à afficher de deux façons :

– En donnant une chaîne de caractères suivant directement l'instruction `print`

```
1 >>> print "toto"
2 toto
3 >>> a = 2
4 >>> print a
5 2
```

– Ou en passant n'importe quoi entre parenthèses

```
1 >>> print( a )
2 2
```

Les chaînes de caractères sont données entre guillemets, sinon elles sont interpretées comme des noms de variables

```
1 >>> print "toto"
2 toto
```

L'opérateur de concaténation des chaînes de caractères est `+`. Attention, le `+` est d'abord interprété comme un opérateur mathématique si il est utilisé sur autre chose que des chaînes de caractère.

```
1 >>> print a + 3
2 5
3 >>> print a, 3
4 2 3
```

On peut aussi afficher deux éléments de types différents à la suite avec `,`. Ils sont affichés séparés par un espace.

Attention : on ne peut concaténer que des variables de même type :

```
1 >>> print "toto" + 3
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError: cannot concatenate 'str' and 'int
    ' objects
```

La solution est alors de convertir la variable en chaîne de caractères.

```
1 >>> print "toto" + str( 3 )
2 toto3
3 >>> print "toto", 3
4 toto 3
```

2 Les variables

Une *variable* correspond à l'emplacement mémoire d'une donnée. Elle porte un nom, ce qui permet de l'identifier facilement.

Une variable peut être *d'entrée* (une entrée de l'algorithme), *de sortie* (le résultat du calcul effectué par l'algorithme) ou *interne à l'algorithme* (ni d'entrée ni de sortie mais utilisée à l'intérieur de l'algorithme).

2.1 Types de variables

Les variables sont souvent *typées*, c'est-à-dire qu'on définit quel type de donnée pourra être contenu dans cette variable. Par exemple :

- Entier : tout nombre entier
- Booléen : vrai ou faux
- Caractère
- Chaîne de caractères
- Tableau...

Ainsi, on ne peut pas affecter un nombre réel à virgule (par exemple, 3,14159) dans une variable entière. Il est alors nécessaire d'effectuer une conversion : la variable entière contiendra la partie entière du nombre réel à virgule. Attention, les conversions ne sont pas toujours possibles.

2.2 Affectation d'une valeur à une variable

Pour écrire une valeur dans une variable, on dit que l'on *affecte* cette valeur à la variable. On le note de la façon suivante :

$variable \leftarrow valeur$

Par exemple, si on définit une variable maVariable de type entier et que l'on lui affecte la valeur 42 dans l'algorithme suivant :

```
1 début
2 |  maVariable : Entier
3 |  maVariable ⟵ 42
4 fin
```

On commence par déclarer le type de la variable avant de lui affecter une valeur. En général, par soucis de lisibilité, les déclarations de variables sont intégralement faites en début d'algorithme quel que soit l'endroit où elles sont utilisées par la suite dans cet algorithme.

2.3 Les variables en Python

En Python, le typage des variables est *implicite* : on ne déclare pas les variables, elles sont créées au moment de leur initialisation et leur type est "deviné" par l'interpréteur.

On affecte une valeur à une variable avec l'opérateur = :

```
1  i = 3
2  j = 0.5
```

Les noms de variables doivent commencer par une lettre (majuscule ou minuscule) comprise entre A et Z ou un signe _. Ils ne peuvent pas commencer par un chiffre ni contenir de caractère accentué. Les majuscule et les minuscules ont leur importante : `Var` désignera une autre variable que `var`. Bien sûr, les mots réservés du langage ne peuvent pas être utilisés comme noms de variables.

2.4 Types de variables

2.4.1 Booléens

Python fournit un type booléen. Les variables de ce type peuvent prendre deux valeurs : `True` ou `False`.

On peut alors effectuer des opérations booléennes sur ces variables : ET, OU, et NON, respectivement `and`, `or` et `not`. Attention à la priorité entre les opérations : l'opération `not` a la priorité la plus basse devant `and` et `or`. Pour éviter les erreurs et lever les ambiguités (pensez à la personne qui relira votre code dans quelques mois, cette personne pouvait être vous-même), n'hésitez pas à utiliser des parenthèses.

```
1  >>> a = True
2  >>> b = False
3  >>> a and b
4  False
5  >>> a or b
6  True
7  >>> not a
8  False
9  >>> not ( a and b )
10 True
```

On peut également utiliser `&` pour `and`, `|` pour `or` et `^` pour le *ou exclusif* (xor).

```
1  >>> a = True
2  >>> b = False
3  >>> a & b
4  False
5  >>> a | b
6  True
7  >>> True ^ True
8  False
```

2.4.2 Entiers

Les variables de types entiers servent à stocker des nombres entiers. En interne, il en existe deux types : le type `integer`, codé sur au moins 32 bits, et le type `long integer` ou `long`. Les nombres trop grands pour être codés sur un `integer` sont codés sur un `long`.

On peut également forcer le type `long` en suffixant la valeur assignée à une variable par un `L` :

```
1  >>> b = 1L
```

Le nombre le plus grand pouvant être encodé par un `integer` est donné par la constante `sys.maxint`. Attention, cette valeur dépend du système sur lequel le programme s'exécute : la seule certitude est que l'on dispose d'au moins 32 bits, cette valeur est donc supérieure ou égale à 4 294 967 296. Le minimum est égal à `- sys.maxint -1`.

```
1  >>> print sys.maxint
2  9223372036854775807
```

2.4.3 Réels

Les nombres réels décimaux sont représentés sous forme de nombres à virgule flottante. La manière de représenter les données dans un ordinateur ne permet que de coder des nombres entiers : on a donc défini un système de codage des nombres réels, définie par la norme IEEE 754. On les appelle nombres *à virgule flottante*, car la virgule n'est pas toujours située au même endroit.

Un nombre à virgule flottante est constitué selon la norme IEEE 754 de trois éléments :
- Si le nombre est signé, on utilise un *bit de signe*
- Une *mantisse*, qui est un facteur multiplicateur retranché de 1 et compris entre 0 et 1
- Un *exposant*

En notant s le signe (-1 ou 1), m la mantisse et e l'exposant, on retrouve un nombre réel avec la formule :

$$nombre = s \times (1 + m) \times 2^e$$

On voit que l'on effectue une double approximation lorsqu'un nombre réel est converti en nombre réel IEEE 754 pour être utilisé sur une machine : la première approximation a lieu sur l'exposant, la seconde sur la mantisse.

Les trois éléments (signe, exposant et mantisse) sont représentés en mémoire de la façon suivante :

signe	exposant	mantisse

En Python, on dispose du type `float` pour encoder les nombres à virgule flottante. Ici encore, leur implémentation et donc les valeurs pouvant être encodées dépendent de la machine utilisée. La seule certitude est que l'exposant est encodé sur au moins 11 bits, et la mantisse sur 52 bits. En ajoutant le bit de signe, on obtient un total de 64 bits.

On peut lire les informations sur le type `float` disponible avec la constante `sys.float_info`.

```
>>> print sys.float_info
sys.floatinfo(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.2204460492503131e-16, radix=2, rounds=1)
```

2.4.4 Complexes

Les nombres complexes sont encodés par deux nombres de type `float`, représentant la partie réelle et la partie imaginaire. On indique la partie imaginaire avec le caractère `j` ou `J`. On peut accéder à la partie réelle et à la partie imaginaire d'un complexe `z` avec, respectivement, `z.real` et `z.imag`.

```
>>> z = 9+5J
>>> print z.imag
5.0
>>> print z.real
9.0
>>> print z
(9+5j)
```

On note que les parties réelle et imaginaire sont affichés comme des nombres réels et non pas comme des entiers.

2.4.5 Chaînes de caractères

Les variables peuvent aussi contenir des chaînes de caractères. Celles-ci doivent être déclarées entre ' ou ". Si la chaîne de caractères contient ce caractère, il doit être échappé en le précédant d'un \. Ainsi, le caractère suivant le \ne sera pas interprété. De même, l'intérieur d'une chaine déclarée entre " n'est pas interprété : par conséquent, un ' placé entre deux " ne sera pas interprété.

```
1    str = "Vive la prog"
2    str2 = 'Python c\'est bon'
3    str3 = "Pruch'ella duri"
```

On peut concaténer deux chaînes de caractères grâce à l'opérateur `+`.

```
1 >>> str = "Vive la prog"
2 >>> str2 = " et vive Python"
3 >>> str + str2
4 'Vive la prog et vive Python'
```

2.5 Introspection et conversions

Les types sont explicites en Python. Cependant, il est possible d'effectuer certaines opérations sur les types des variables.

2.5.1 Obtenir le type d'une variable

L'*introspection* est, par définition, la connaissance qu'une entité a d'elle-même. En programmation, cela correspond à la connaissance qu'a une entité (ici, une variable) de sa structure. On utilise la fonction `type()` :

```
1    >>> i = 5
2    >>> type( i )
3    <type 'int'>
4    >>> j = 6L
5    >>> type( j )
6    <type 'long'>
7    >>> z = 9+5J
8    >>> type( z )
9    <type 'complex'>
10   >>> type( z.real )
11   <type 'float'>
```

2.5.2 Conversion de type

On peut convertir une variable d'un type vers un autre, par exemple pour effectuer une opération sur deux variables de types différents. Pour cela, on utilise une fonction particulière qui *construit* une variable du type donné et qui porte le nom de ce type. Par exemple, on obtient un entier avec la fonction `int`.

```
>>> a = 7.0
>>> type( a )
<type 'float'>
>>> b = int( a )
>>> print b
7
>>> type( b )
<type 'int'>
```

Dans l'exemple ci-dessus, la variable `a` est de type `float`. On crée une variable `b` qui contient la même valeur que `a` convertie en `int`.

Les conversions sont également utiles pour former des chaines de caractères à partir de différents types, par exemple pour les afficher. Par exemple, si on essaye d'afficher la concaténation d'une chaîne de caractères et d'un entier, on reçoit une erreur :

```
>>> i = 5
>>> type( i )
<type 'int'>
>>> print 'la valeur de i est ' + i
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

Il est donc nécessaire de convertir l'entier en une chaîne de caractères avec la fonction `str()` :

```
>>> print 'la valeur de i est ' + str( i )
la valeur de i est 5
```

Attention aux pertes de précision et aux arrondis effectués lors des conversions. Par exemple, lorsqu'un nombre réel est converti en entier, la partie décimale du nombre (située après la virgule) est perdue.

```
>>> r = 4.8
>>> int( r )
4
```

On peut tester si un nombre réel (de type `float`) peut être convertible en entier sans arrondi, avec la fonction `is_integer()`

```
>>> r = 4.2
>>> r.is_integer()
False
>>> d = 4.0
>>> d.is_integer()
True
```

Lorsque l'on effectue une opération sur deux variables de types différents dont l'un est convertible en l'autre sans perte de précision, Python effectue la conversion implicitement. Par exemple, si on veut additionner une variable de type `int` avec une variable de type `long`, l'opération sera effectuée sur des `long` :

```
>>> m = 5
>>> n = 7L
>>> o = m + n
>>> type( o )
<type 'long'>
```

Pour convertir un nombre réel décimal en entier de façon sure, c'est-à-dire en sachant quel arrondi va être effectué, on utilisera une fonction mathématique effectuant explicitement cette opération. On peut arrondir, obtenir la partie entière ou le plafond d'un nombre avec respectivement `round()`, `math.floor()` et `math.ceil()`.

```
>>> import math
>>> i = 2.6
>>> j = math.ceil( i )
>>> k = math.floor( i )
>>> k = round( i )
>>> print i, j, k
2.5 3.0 3.0
```

2.6 Collections

2.6.1 Les tableaux

Un tableau est un type particulier de variable. Il contient un *ensemble de variables* de même type, stockées de façon contiguë en mémoire. Par exemple, voici un tableau d'entiers :

3	5	2	1	12	5	2	9

La taille d'un tableau est fixe. Ils peuvent être de la dimension que l'on souhaite : unidimensionnels comme dans l'exemple ci-dessus, bidimensionnels (dans le cas d'une matrice, par exemple), etc. Dans ce cas, les données sont identifiées par leur indice dans chacune des dimensions.

On déclare un tableau en donnant sa taille et le type de données qu'il contiendra :
$monTab[10] : Tableau\ d'entiers$ déclare le tableau $monTab$ de taille 10 contenant des entiers.

Les données contenues dans un tableau sont numérotées, à partir de 0 ou de 1 selon la convention. La plupart des langages modernes commencent la numérotation à partir de 0 : c'est pourquoi dans ce cours nous suivrons cette convention. L'indice utilisé pour désigner les données d'un tableau est généralement de type entier. Ainsi, on désigne la huitième case du tableau $monTab$ en utilisant $monTab[7]$.

Dans le cas d'un tableau multidimensionnel, on le déclarera en donnant sa taille dans toutes les dimensions. Par exemple, pour un tableau bidimensionnel de taille 10×10 :
$maMatrice[10][10] : Tableau\ d'entiers$. On accède aux données en donnant leur rang dans chacune des dimensions, par exemple : $maMatrice[4][3]$ donnera l'élément situé en quatrième position sur la cinquième ligne [1].

Python fournit des structures de données permettant de stocker des ensembles, appelés *collections*. Il en existe trois types : les listes, les tuples et les dictionnaires. Il est possible d'effectuer un certain nombre d'opérations spécifiques sur chacune de ces collections.

Une particularité est que les éléments contenus dans les collections ne sont pas forcément tous du même type.

1. L'ordre ligne / colonne dépend du langage d'implémentation : par exemple le C donne d'abord la ligne puis la colonne, tandis que le Fortran donne d'abord la colonne puis la ligne. Dans ce cours nous utiliserons la convention ligne-colonne (comme en C), plus répandue actuellement.

2.6.2 Les listes

Une liste est un ensemble d'éléments. Ces éléments ne sont pas triés. La liste est modifiable.

On définit une liste en donnant l'ensemble initial de ses éléments entre deux crochets et en les séparant par des virgules. On peut par la suite ajouter ou retirer des éléments de la liste. On définit une liste vide simplement avec une paire de crochets.

```
1  >>> l = []
2  >>> type( l )
3  <type 'list'>
4  >>> m = [ 5, 7.6, "bonjour"]
```

Une liste est *ordonnée*, c'est-à-dire que l'ordre dans lequel ses éléments sont disposés a de l'importance. Ainsi, on peut accéder à un élément perticulier par son indice dans la liste en utilisant l'opérateur crochets. Attention, la numérotation des indices commence à 0.

```
1  >>> print m[0]
2  5
```

On obient le nombre d'éléments dans une liste grâce à la fonction `len()` :

```
1  >>> len( m )
2  3
```

On peut ajouter un élément à la fin d'une liste grâce à la fonction `append()`, ou à un endroit précis de la liste avec la fonction `insert()` en lui passant en premier argument l'indice où l'élément doit être inséré.

```
1 >>> jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
2 >>> jour.append( 'dimanche' )
3 >>> print jour
4 ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'dimanche']
5 >>> jour.insert( 5, 'samedi')
6 >>> print jour
7 ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi',
8 'dimanche']
```

On peut retirer un élément d'une liste avec la fonction `remove()` en lui passant l'élément à retirer. Si l'élément se trouve plusieurs fois dans la liste, seul le premier est retiré.

```
1 >>> jour.remove( 'samedi' )
2 >>> print jour
3 ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'dimanche']
```

La fonction `pop()` permet de retirer un élément d'une liste en le retournant. L'élément retiré est celui situé à l'indice passé en argument ou, à défaut, le dernier élément de la liste.

```
1 >>> jour.pop()
2 'dimanche'
3 >>> print jour
4 ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
5 >>> jour.pop(2)
6 'mercredi'
7 >>> print jour
8 ['lundi', 'mardi', 'jeudi', 'vendredi']
```

2.6.3 Les tuples

Un tuple est une structure de données proche des listes, mais non modifiable.

On définit un tuple entre parenthèses. Ses éléments sont séparés par des virgules. Hormis les opérations de modification des éléments du tuple, on dispose des mêmes fonctions sur les tuples que sur les listes.

```
>>> jour = ('lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi')
>>> jour.append( 'dimanche' )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Les tuples présentent deux avantages sur les listes. Le premier est qu'ils sont implémentés de façon plus simple et donc utilisent moins de ressources que des listes (en particulier en mémoire). Ils peuvent également être utilisé pour s'assurer qu'un ensemble ne sera pas modifié par une autre partie du programme. Les restrictions de ce type servent souvent de garde-fou pour prévenir certains bugs.

Cependant, comme ils ne peuvent pas être modifiés, ils sont plus contraignants que les listes.

2.6.4 Les dictionnaires

Un dictionnaire est une structure de donnée qui effectue une association entre une *clé* et une *valeur*. On accède aux valeurs par leur clé.

On définit un dictionnaire en utilisant des accolades. On peut l'initialiser en donnant des couples clé-valeur séparés par des virgules, en séparant une clé de sa valeur par des signes ':'.

```
>>> dic = { 'pomme' : 'fruit' , 'poire' : 'fruit', 'poireau' :
'legume' }
```

On accède aux valeurs en donnant leur clé entre crochets. De même, on peut insérer une nouvelle paire clé-valeur en affectant la valeur à sa clé passée entre crochets.

```
>>> dic['tomate'] = 'fruit'
>>> print dic
{'poire': 'fruit', 'tomate': 'fruit', 'poireau': 'legume',
 'pomme': 'fruit'}
>>> dic['poireau']
'legume'
```

On peut obtenir la liste des clés et des valeurs avec les fonctions `keys()` et `values()`. On peut tester si une clé est présente dans le dictionnaire avec la fonction `has_key()`.

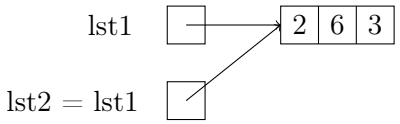
```
>>> dic.keys()
['poire', 'tomate', 'poireau', 'pomme']
>>> dic.values()
['fruit', 'fruit', 'legume', 'fruit']
>>> dic.has_key( 'tomate' )
True
```

2.7 Attention aux références

Attention : Python fonctionne en désignant les variables par leur *référence*. Une référence désigne l'adresse de début de l'espace mémoire occupé par la variable.

Ainsi, si on déclare par exemple une liste `lst1`, `lst1` contient en réalité la référence de cette liste. Si on copie `lst1` dans une autre variable, c'est en réalité la référence de la liste qui sera copiée :

```
1 >>> lst1 = [ 2, 6, 3 ]
2 >>> lst2 = lst1
3 >>> print lst2
4 [2, 6, 3]
```



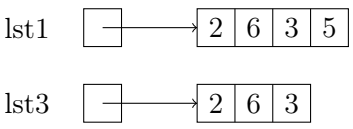
Si on effectue une modification sur la liste pointée par `lst2`, cette modification sera effective sur la liste pointée par `lst1`, étant donné qu'il s'agit en réalité de la même entité en mémoire.

```
1 >>> lst2.append( 5 )
2 >>> print lst1
3 [2, 6, 3, 5]
```

Pour effectuer une vraie copie d'un élément contenu en mémoire, on doit donc effectuer une copie explicite en créant un nouvel élément qui contiendra l'élément à copier.

Dans le cas d'une liste, on crée une nouvelle liste avec la fonction `list()`, et on lui passe en argument la liste à copier. On constate ensuite que si la nouvelle liste est modifiée, la liste d'origine n'est pas affectée.

```
1 >>> lst3 = list( lst1 )
2 >>> print lst3
3 [2, 6, 3, 5]
4 >>> lst3.append( 1 )
5 >>> print lst3, lst1
6 [2, 6, 3, 5, 1] [2, 6, 3, 5]
```



3 Les structures de contrôle

3.1 Les tests

Un *test* évalue la valeur d'une expression booléenne et effectue une action si cette expression est vraie, une autre action si cette expression n'est pas vraie. On l'appelle également *structure conditionnelle*, car l'action réalisée est soumise à une condition.

L'algorithme suivant présente un exemple de test. Si la condition *condition* est validée, alors on effectue l'action *action1*. Sinon, on effectue l'action *action2*.

```
1 début
2 |  si condition alors
3 |  |  action1
4 |  sinon
5 |  |  action2
6 |  fin si
7 fin
```

La deuxième partie est optionnelle : il est possible de ne pas en définir. Dans ce cas, l'algorithme ne fait rien dans le cas où la condition n'est pas réalisée.

L'algorithme suivant présente un deuxième exemple de test. Dans le cas où la variable $maVariable$ est inférieure à 5, on lui ajoute 1. Dans les autres cas, on ne fait rien.

```
1 début
2 |  si maVariable < 5 alors
3 |  |  maVariable ⟵ maVariable + 1
4 |  fin si
5 fin
```

Dans la condition, on peut tester :
- l'égalité entre deux variables : $var1 == var2$
- la non-égalité entre deux variables : $var1 != var2$
- une relation d'ordre entre deux variables : $var1 > 0$
- ou toute expression renvoyant $Vrai$ ou $Faux$

Ces conditions peuvent être combinées en suivant les règles de l'algèbre de Boole. Attention à mettre des parenthèses autour des expressions pour éviter les ambiguïtés, par exemple : $(var1 == var2) et (var1 > 0)$.

Les règles de syntaxe sont les suivantes :
- La *condition* est donnée entre les mot-clés **si** et **alors**
- L'action réalisée si la condition est vérifiée est donnée après le mot-clé **alors**
- Si on donne une action à réaliser si la condition n'est pas vérifiée, elle est introduite par le mot-clé **sinon**
- Le test est terminé par le mot-clé **finsi**
- Pour plus de lisibilité, le contenu de chaque bloc est décalé d'un cran vers la droite (on parle d'*indentation*) et indiqué par une ligne verticale à gauche

3.2 Expressions booléennes

Le contrôle du flux d'exécution d'un programme se base sur l'évaluation d'expressions booléennes. Les deux valeurs possibles pour un booléen sont `True` et `False` (respectivement, vrai et faux). On dispose d'*opérateurs booléens* permettant d'effectuer des opérations logiques sur des valeurs booléennes.

Opérateur	Opération effectuée	Exemple
&	ET	a & b
|	OU	a | b
∧	XOR	a ∧ b

On dispose d'*opérateurs de comparaison* permettant de comparer deux variables et retournant un booléen.

Opérateur	Opération effectuée	Exemple
==	est égal à	a == b
!=	est différent de	a != b
>	supérieur à	a > b
>=	supérieur ou égal à	a >= b
<	inférieur à	a < b
<=	inférieur ou égal à	a <= b

En outre, on dispose de la négation `!` qui permet d'obtenir la négation d'une expression booléenne. Par exemple, `!True` vaut `False`.

Ces opérations peuvent être combinées : attention aux parenthèses pour éviter les ambiguïtés, sources d'erreurs !

3.3 Tests et alternatives en Python

Comme vu en algorithmique, un *test* évalue la valeur d'une expression booléenne et effectue une action si cette expression est vraie, une autre action si cette expression n'est pas vraie. On l'appelle également *structure conditionnelle*, car l'action réalisée est soumise à une condition.

La syntaxe des tests en Python utilise les mot-clés `if`, `else` et `elif`.

Le mot-clé `if` sert à introduire la condition à tester. Il est suivi de la condition, éventuellement entre parenthèses, et la ligne doit se terminer par le signe `:`. Le bloc contenant l'action à effectuer si la condition à réaliser est indenté d'un cran supplémentaire.

```
1 a = 5
2 if a > 0:
3     print "a est positif"
```

Lorsque la condition n'est pas réalisée, on introduit l'alternative avec le mot-clé `else`. De même, le bloc contenant le code de l'action à effectuer si l'action n'est pas réalisée (et donc si l'on passe dans l'alternative) est indenté d'un cran supplémentaire.

```
1 a = 5
2 if a >= 0:
3     print "a est positif ou nul"
4 else:
5     print "a est negatif"
```

Lorsque la condition n'est pas réalisée, on peut introduire une alternative conditionnelle. Cela permet d'introduire un nouveau test : le programme peut tester plusieurs possibilités, sans être limité à deux cas seulement. On l'introduit grâce au mot-clé `elif`

```
1 a = 5
2 if a > 0:
3     print "a est positif"
4 elif a == 0:
5     print "a est nul"
6 else:
7     print "a est negatif"
```

3.4 Les boucles

Une boucle sert à *répéter une action*. On distingue deux sortes de boucles :
- La boucle **Pour**
- La boucle **Tant que**

La boucle *Pour* nécessite de connaître le nombre de répétitions à effectuer avant le début de la boucle. L'arrêt de la boucle *Tant que* est déterminé par une condition qui n'est pas intrinsèque à la boucle, mais souvent à son contenu : par exemple, une condition sur une variable modifiée à l'intérieur de la boucle.

3.4.1 Boucle *Pour*

La boucle *Pour* effectue une action en commençant par l'initialisation d'un compteur. À la fin de chaque répétition, on effectue une action sur ce compteur (par exemple, on l'augmente d'une certaine valeur). La boucle se termine lorsque la condition de bouclage n'est plus vraie.

Par exemple, examinons l'algorithme suivant. On a un compteur i initialisé à la valeur 0. L'action est réalisée pour i allant de 0 à 9 par un pas de 1, c'est-à-dire qu'à chaque répétition, la valeur du compteur i est augmentée de 1.

```
1 début
2 |  tab[10] : Tableau d'entiers
3 |  pour i ⟵ 0 a 9 pas 1 faire
4 |  |  tab[i] = 2 * i
5 |  fin pour
6 fin
```

Dans le cas de l'algorithme ci-dessus, le compteur i est initialisé à 0 et la condition de la boucle est que i doit être inférieur ou égal à 9. Lorsque la valeur de i dépasse 9, la boucle se termine. Le pas vaut ici 1, mais il peut être différent de 1 (par exemple, pour aller de 2 en 2...), voire être négatif (pour aller à reculons).

Les règles de syntaxe sont les suivantes :
- Les conditions sur le compteur sont définies entre les mot-clés **pour** et **faire**
- L'initialisation du compteur est donnée comme une affectation ($i \longleftarrow 0$), sa valeur finale est donnée après le mot-clé **à**, le pas est donné après le mot-clé **pas**
- L'action à effectuer est donnée entre les mot-clés **faire** et **finpour**
- Pour plus de lisibilité, le contenu du bloc définissant l'action à effectuer est décalé d'un cran vers la droite (on parle d'*indentation*) et indiqué par une ligne verticale à gauche

3.4.2 Boucle *Tant que ... Faire*

La boucle *Tant que* effectue une action tant qu'une condition donnée est vraie. Avant chaque répétition, on évalue la valeur de la condition. Si elle est vraie, alors on effectue l'action définie dans la boucle. Sinon, on sort de la boucle.

L'algorithme suivant est un exemple de boucle *tant que ... faire*. Nous définissons une variable entière *puissance* initialisée à la valeur 1. À chaque répétition, on multiplie par 2 la valeur de *puissance*. La boucle s'arrête lorsque la valeur de *puissance* atteint ou dépasse 100.

```
1 début
2 |  puissance : Entier
3 |  puissance ⟵ 1
4 |  tant que puissance < 100 faire
5 |  |  puissance ⟵ puissance ∗ 2
6 |  fin tq
7 fin
```

Les règles de syntaxe sont les suivantes :
- La condition de boucle est définie après le mot-clé **Tant que**
- L'action à effectuer est donnée entre les mot-clés **faire** et **fintq**
- Pour plus de lisibilité, le contenu du bloc définissant l'action à effectuer est décalé d'un cran vers la droite (on parle d'*indentation*) et indiqué par une ligne verticale à gauche

3.4.3 Boucle *Faire ... Tant que*

À la différence de la boucle *tant que ... faire*, la boucle *faire ... tant que* évalue la valeur de la condition *après* avoir effectué l'action définie à l'intérieur de la boucle. On effectue donc l'action au moins une fois, et on décide ensuite si on la répète.

Par exemple, l'action à effectuer devant un panneau de stop correspond à une boucle *faire ... tant que*. L'algorithme ci-dessus décrit le comportement que doit suivre une voiture lorsqu'elle se trouve face à un panneau de stop. Il faut d'abord s'arrêter (ligne 4), puis regarder s'il y a des voitures. Tant que des voitures arrivent (ligne 5), on répète l'action dans la boucle (ligne 4). Lorsqu'il n'y a plus de voitures, le test situé ligne 5 est faux : on sort de la boucle (ligne 6).

```
1 début
2 |  vitesse : Entier
3 |  faire
4 |  |  vitesse ⟵ 0
5 |  tant que voitures_arrivent == Vrai;
6 |  vitesse ⟵ 50
7 fin
```

3.4.4 Équivalence entre les boucles *Pour* et *Tant que*

On remarque qu'il est facile d'écrire une boucle *Pour* au moyen d'une boucle *Tant que*. On initialise le compteur avant d'entrer dans la boucle, la condition de la boucle *Tant que* est l'inverse de la condition d'arrêt de la boucle *Pour* (dans la boucle *Pour* il s'agit d'une condition d'arrêt, tandis que dans la boucle *Tant que* il s'agit d'une condition de continuation) et le compteur est modifié en ajoutant le pas à la fin de l'action à effectuer dans la boucle :

```
1 début
2 |  compteur : Entier
3 |  compteur ⟵ 0
4 |  tant que compteur < 10 faire
5 |  |  action
6 |  |  compteur ⟵ compteur + 1
7 |  fin tq
8 fin
```

Cependant, une boucle *Pour* est souvent plus lisible et moins source d'erreurs qu'une boucle *Tant que*. Lorsque le nombre de répétitions est connu avant l'entrée dans la boucle et ne dépend pas du résultat de l'action à effectuer, on privilégie donc plutôt une boucle *Pour*. À l'inverse, la boucle *Tant que* est plutôt utilisée lorsque la condition d'arrêt de la boucle dépend du calcul effectué.

Cette équivalence illustre par ailleurs le fait qu'en algorithmique et en programmation en général, il y a souvent plusieurs façons de résoudre un problème. Les critères de choix entre les différentes solutions peuvent être :

- L'efficacité : par exemple, minimiser le nombre d'opérations à effectuer, ou l'espace mémoire utilisé ;
- La simplicité et la lisibilité : un algorithme simple présentera moins de risques d'erreur, s'il est lisible il sera plus facile de s'y replonger plus tard ou, pour une autre personne, de se familiariser avec le travail de quelqu'un d'autre.

3.4.5 Itération

La boucle `for` permet d'itérer parmi des éléments. On lui passe un ensemble : pour chaque élément de cet ensemble, on effectue un passage dans la boucle en récupérant la valeur de cet élément affectée à un *itérateur*.

En langage algorithmique, cela correspondrait à une boucle `pour` :

```
1 début
2 |  tab[10] : Tableau d'entiers
3 |  pour i dans tab faire
4 |  |  afficher(i)
5 |  fin pour
6 fin
```

On utilise le mot-clé `for`, suivi de la variable qui servira d'itérateur. On introduit l'ensemble considéré avec le mot-clé `in` suivi de cet ensemble (ou la variable le contenant). Enfin, la ligne est terminée par `:`. Le bloc d'actions à effectuer est indenté d'un cran.

```
1 lst = [1, 4, 6]
2 for i in lst:
3     print i
```

Dans l'exemple ci-dessus, on itère sur les éléments contenus dans la liste `lst`. Pour chaque élément de cette liste, on exécute la ligne 3, en disposant de cet élément dans la variable `i`.

On peut obtenir une liste d'entiers avec la fonction `range()`. Cette fonction peut prendre 1, 2 ou 3 arguments entiers :

- Avec un seul argument, elle retourne la liste d'entiers consécutifs entre 0 et l'entier passé en argument exclu

```
1 >>> print range( 5 )
2 [0, 1, 2, 3, 4]
```

  L'entier passé en argument est bien exclu : on a la liste d'entiers positifs strictement inférieurs à 5.
- Avec deux arguments, elle retourne la liste d'entiers consécutifs entre le premier et le deuxième argument exclu

```
1 >>> print range( 2, 5 )
2 [2, 3, 4]
```

- Avec trois arguments, le troisième entier correspond au pas entre deux entiers de la liste

```
1 >>> print range( 1, 10, 3 )
2 [1, 4, 7]
```

3.4.6 Boucle conditionnelle

Une boucle conditionnelle définit un bloc d'actions à effectuée tant qu'une conditions est réalisée. La condition est évaluée *avant* le passage dans le corps de la boucle. Si la condition est réalisée, on exécute le corps de la boucle et on revient à l'évaluation de la condition. Sinon, on sort de la boucle.

En langage algorithmique, cela correspondrait à une boucle `tant que` :

```
1 début
2 |  i : Entier
3 |  i ⟵ 0
4 |  tant que i < 10 faire
5 |  |  affiche(i)
6 |  |  i ⟵ i + 1
7 |  fin tq
8 fin
```

En Python, on utilise le mot-clé `while`, suivi de la condition à évaluer. La ligne est terminée par `:`. Le bloc d'actions à effectuer est indenté d'un cran.

```
1 i = 0
2 while i < 10 :
3         print i
4         i = i + 1
```

3.5 Structuration par blocs et imbrication

Comme vu dans les règles de syntaxe des tests et des boucles, les structures de contrôle obéissent à certaines règles de présentation telles que l'indentation vers la droite et l'utilisation d'une ligne verticale pour bien marquer les actions à effectuer dans les différentes parties de ces structures. Les parties de ces structures sont appelées des *blocs*.

L'algorithme suivant présente deux blocs successifs.

```
1  début
2  |  tableau[10] : Tableau d'entiers
3  |  i : Entier
4  |  pour i ⟵ 0 a 9 pas 1 faire
5  |  |  tableau[i] ⟵ i × i
6  |  fin pour
7  |  i ⟵ 0
8  |  tant que tableau[i] < 8 faire
9  |  |  i ⟵ i + 1
10 |  fin tq
11 fin
```

La boucle *Pour* est un premier bloc. Ce bloc commence à la ligne 4 et termine à la ligne 6. Il commence avec le mot-clé **pour**, qui marque le début de la boucle, et termine avec le mot-clé **finpour**, qui marque la fin de la boucle. La boucle *Tant que* est un deuxième bloc. Ce bloc commence à la ligne 8 et termine à la ligne 10. Il commence avec le mot-clé **tant que**, qui marque le début de la boucle, et

termine avec le mot-clé **fintq**, qui marque la fin de la boucle. La ligne 5 est un bloc (constitué d'une seule ligne) : il s'agit du bloc d'instructions de la boucle *Pour*. De même, la ligne 9 contient le bloc d'instructions de la boucle *Tant que*.

Des blocs peuvent également être *imbriqués*. Un exemple de blocs imbriqués est illustré par l'algorithme ci-dessous.

```
1  début
2  |  matrice[10][10] : Tableau d'entiers
3  |  x : Entier
4  |  y : Entier
5  |  pour x ⟵ 0 a 9 pas 1 faire
6  |  |  pour y ⟵ 0 a 9 pas 1 faire
7  |  |  |  si x! = y alors
8  |  |  |  |  matrice[x][y] ⟵ x × y
9  |  |  |  sinon
10 |  |  |  |  matrice[x][y] ⟵ 0
11 |  |  |  fin si
12 |  |  fin pour
13 |  fin pour
14 fin
```

Cet algorithme présente deux boucles *Pour* imbriquées. La première boucle répète la deuxième boucle en faisant varier son indice x. La deuxième boucle remplit la matrice en faisant varier son indice y. La première boucle parcourt les lignes de la matrice, la deuxième boucle parcourt les colonnes de chaque ligne. Enfin, un test est imbriqué dans la deuxième boucle et affecte une valeur à la case du tableau correspondant aux indices x et y en fonction de la valeur de x et y.

Le bloc de la boucle principale commence à la ligne 5 et termine à la ligne 13. Le bloc imbriqué dans la boucle principale commence à la ligne 6 et termine à la ligne 12. Le bloc imbriqué dans le bloc imbriqué commence à la ligne 7 et termine à la ligne 11. Enfin, ce bloc contient des blocs d'une ligne à la ligne 8 et à la ligne 10.

3.5.1 Indentation des blocs en Python

La syntaxe du langage Python repose énormément sur la notion de *bloc*. Les blocs sont délimités par les niveaux d'indentation du code vers la droite : tout le code qui se trouve indenté au moins au même niveau se trouve dans le même bloc. On peut *imbriquer* un bloc dans un autre en indentant le bloc imbriqué d'un niveau supplémentaire. Chaque niveau d'indentation est représenté par une *tabulation*.

Examinons l'exemple suivant. Il ne contient pas de "vrai" code, mais des lignes indentées de façon à former des blocs.

```
1 bloc principal
2       début d'un sous-bloc
3       suite du sous-bloc
4             bloc imbriqué dans le sous-bloc
5       on revient dans le sous-bloc
6 retour au bloc principal
```

Le bloc principal s'étend de la première à la sixième ligne. Toutes les lignes de ce bloc commencent à gauche, ou plus à droite. La ligne 2 est décalée d'une tabulation vers la droite : il s'agit d'un sous-bloc. Toutes les lignes situées à *au moins* une tabulation du bord gauche du code font partie de ce sous-bloc : ainsi, les lignes 2 à 5 font partie du sous-bloc. Un autre sous-bloc est imbriqué dedans : il s'agit de la ligne 4. On peut revenir dans un bloc à la fin d'un sous-bloc en revenant au niveau d'indentation correspondant : la ligne 6 est située dans le bloc principal.

4 Programmation structurée

> "... diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour mieux les résoudre."
>
> *Discours de la méthode (1637)*
> René Descartes

Il est possible de découper un problème en sous-problèmes indépendants les uns des autres. On suppose que chacun des sous-problèmes est résolu : on dispose alors d'une abstraction permettant de les combiner et de repousser le plus possible la résolution elle-même.

Un autre avantage de la programmation structurée est qu'elle permet de factoriser et de réutiliser un algorithme plusieurs fois à différents endroits du programme.

Pour cela, on découpe souvent un algorithme en *fonctions* et/ou en *procédures*. Le corps de l'algorithme appelle ces fonctions et procédures pour effectuer les actions et les calculs qui y sont définis. On leur passe souvent des *arguments* ou *paramètres*, qui sont les variables d'entrée de la fonction ou de la procédure.

4.1 Arguments

On distingue trois types d'arguments :
- Les arguments d'entrée : ils sont utilisés par la fonction ou la procédure mais leur valeur n'est pas modifiée par les actions effectuées à l'intérieur de la fonction ou de la procédure ;
- Les arguments de sortie : leur valeur n'est pas utilisée mais on leur affecte une valeur à l'intérieur de la fonction ou la procédure, et la variable utilisée comme paramètre par l'appelant est utilisée ensuite ;
- les arguments d'entrée/sortie : ils sont utilisés par la fonction ou la procédure et leur valeur est modifiée à l'intérieur de la fonction ou de la procédure.

4.2 Fonctions

À l'image d'une fonction mathématique $f : x \mapsto f(x)$, une fonction effectue un calcul et produit un résultat. On dit que ce résultat est *retourné* par la fonction. Il peut être utilisé dans un calcul ou affecté à une variable.

Par exemple, l'algorithme suivant appelle une fonction qui calcule le carré d'une variable passée en argument et retourne la valeur du carré. La fonction `carre` est définie en-dessous : elle prend comme argument un entier (qui est appelé *nombre*) et retourne un entier. Le calcul est effectué ligne 3.

```
1 début programme
2 |   nombreDepart : Entier
3 |   calcul : Entier
4 |   calcul ⟵ carre ( nombreDepart )
5 fin programme
```

```
1 début fonction carre( nombre: Entier ): Entier
2 |   resultat : Entier
3 |   resultat ⟵ nombre × nombre
4 |   retourner resultat
5 fin fonction
```

Le mot-clé *retourner* signifie que l'on sort de la fonction en renvoyant la valeur indiquée. Par exemple, la ligne 4 de la fonction `carre` contient l'instruction **retourner**. On sort donc de la fonction `carre` et elle retourne la valeur de la variable *resultat*, qui a été calculée dans la fonction.

La ligne 3 de l'algorithme principal appelle la fonction `carre`. On passe alors dans la fonction. La valeur retournée par la fonction est affectée à la variable *calcul* à la ligne 3 de l'algorithme principal.

La fonction ci-dessous retourne la plus petite puissance de 2 supérieure à 200. La condition de la boucle **tant que** est toujours vérifiée : la boucle est donc *infinie*. Cependant, il y a un test à l'intérieur de la boucle: si la condition est réalisée (si le nombre dépasse 200), la ligne 6 est exécutée. On appelle alors l'instruction **retourner** : la fonction retourne la valeur de la variable *petitepuissance*. On sort alors de la fonction.

```
1 début fonction puissance( ): Entier
2 |  petitepuissance : Entier
3 |  tant que Vrai faire
4 |  |  petitepuissance ⟵ petitepuissance × 2
5 |  |  if petitepuissance > 200 then
6 |  |  |  retourner petitepuissance
7 |  |  end if
8 |  fin tq
9 fin fonction
```

Les règles de syntaxe sont les suivantes :

- Une fonction commence par les mot-clés **début fonction** et se termine par le mot-clé **fin fonction**
- On donne ensuite le *nom* de la fonction et les *paramètres* qui doivent lui être passés, ainsi que leur type
- Pour les paramètres de sortie ou d'entrée/sortie, on précise **sortie** ou **entrée/sortie**. Pour les paramètres d'entrée, on n'a pas à le préciser
- Le *type retourné* par la fonction est précisé à la fin de la ligne de définition de la fonction, après deux points (**:**)
- Une fonction *retourne obligatoirement quelque chose*. On ne peut sortir d'une fonction que par le mot-clé **retourner** suivi du nom de la variable retournée, ou d'une valeur

4.3 Procédures

À l'inverse d'une fonction, une procédure ne retourne rien. Elle est utilisée pour effectuer une action qui ne renvoie pas de résultat (par exemple, rafraîchir un affichage) ou pour modifier la valeur de variables qui lui sont passées en paramètres.

On peut réécrire la fonction *puissance*() définie ci-dessous au moyen d'une procédure en mettant le résultat du calcul dans une variable de sortie passée en paramètre de la procédure. On précise alors dans la déclaration de la procédure que l'argument passé **petitepuissance** est de type **Entier** et qu'il s'agit d'un argument de **sortie** :

```
1 début procédure puissance( petitepuissance: Entier Sortie )
2 |  petitepuissance ⟵ 1
3 |  tant que Vrai faire
4 |  |  petitepuissance ⟵ petitepuissance × 2
5 |  |  if petitepuissance > 200 then
6 |  |  |  retourner
7 |  |  end if
8 |  fin tq
9 fin procédure
```

De même que pour une fonction, on peut sortir d'une procédure à tout moment en appelant le mot-clé **retourner**. Cependant, on l'appelle seul : on ne précise pas de valeur à retourner, car une procédure ne retourne *rien*.

Les règles de syntaxe sont les suivantes :
- Une procédure commence par les mot-clés **début procédure** et se termine par le mot-clé **fin procédure**
- On donne ensuite le *nom* de la procédure et les *paramètres* qui doivent lui être passés, ainsi que leur type
- Pour les paramètres de sortie ou d'entrée/sortie, on précise **sortie** ou **entrée/sortie**. Pour les paramètres d'entrée, on n'a pas à le préciser
- On sort de la procédure soit en arrivant à la fin, soit par le mot-clé **retourner**

4.4 Définition en Python d'une fonction ou d'une procédure

En Python, la définition d'une fonction et d'une procédure sont identiques. La seule différence est dans ce qui est retourné (rappel : une fonction retourne une valeur, une procédure ne retourne rien).

La définition d'une fonction est introduite par le mot-clé `def`
- Suivi du nom de la fonction
- Entre parenthèses, ses arguments (parenthèses vides si aucun)
- Enfin, la ligne est terminée par deux points `:`

Le corps de la fonction est un bloc : on l'*indente* donc d'un cran vers la droite.

On peut sortir d'une fonction de deux façons :
- En arrivant à la fin de la fonction : retour à l'indentation de niveau maximal (complètement à gauche), dans le cas d'une procédure
- En exécutant le mot-clé `return`
  - Soit seul : fin d'une procédure (ne retournant rien)
  - Soit suivi d'une variable qui est retournée par la fonction

Par exemple :

```
1 def carre( a ):
2     c = a * a
3     return c
```

La fonction ci-dessus calcule le carré d'un nombre passé en paramètre, et retourne la valeur calculée. Il s'agit bien d'une fonction : elle retourne une valeur.

À l'inverse, une procédure ne retourne pas de valeur. Par exemple, la procédure suivante affiche le carré d'un nombre passé en paramètre :

```
1 def afficheCarre( a ):
2     c = a * a
3     print c
```

4.5 Appel d'une fonction

On appelle une fonction *par son nom*, en lui passant ses *paramètres entre parenthèses.*

```
1 def carre( a ):
2     c = a * a
3     return c
4
5 n = carre( 5 )
6 print n
7 b = 2
8 m = carre( b )
9 print m
```

Attention : il n'y a pas de vérification du type des paramètres passés lors de l'appel d'une fonction. On peut passer n'importe quel type de variable : il n'y a pas de vérification immédiate. Cela a des avantages mais aussi des inconvénients.

C'est une source d'erreurs qui ne sont pas directement détectées. Si une fonction ne peut être appelée qu'avec certains types d'arguments, l'erreur ne sera pas relevée lors de l'appel de la fonction mais plus tard, par exemple lors de l'utilisation de la variable de type incorrect dans la fonction (au mieux...).

Cela apporte cependant une certaine flexibilité en rendant possible l'appel de la fonction avec des arguments de différents types. Par exemple, la fonction suivante peut être appelée avec n'importe quel type d'argument :

```
1 def maFonction( a ):
2     print "parametre : " + str( a ) + " de type " + str( type( a ) )
3     return type( a )
4
5 # ailleurs dans le programme
6 maFonction( 5 )
7 maFonction( "toto" )
```

Attention, si les paramètres sont modifiés dans le corps de la fonction ou de la procédure, en sortie de cette fonction ou de cette procédure l'appelant a toujours la valeur initiale et non pas la valeur modifiée.

Par exemple :

```
1 def decremente( n ):
2     n = n - 1
3
4 a = 3
5 decremente( a )
6 print a
```

affichera 3 : le paramètre a été décrémenté dans le corps de la procédure, mais c'est toujours sa version non modifiée dont on dispose dans le bloc appelant en sortie de la procédure. La raison pour cela est que la variable a été *copiée* dans l'espace mémoire de la procédure lors de l'appel : on n'a modifié qu'une copie de la variable, et non pas la variable elle-même.

4.6 Paramètres par défaut

Python permet d'utiliser des paramètres de fonctions *optionnels* et des paramètres *nommés*. Les paramètres optionnels doivent avoir une valeur par défaut : c'est cette valeur qui sera utilisée si le paramètre n'est pas fourni lors de l'appel de la fonction.

Lorsque tous les paramètres ne sont pas fournis lors de l'appel de la fonction, il faut préciser à quoi correspondent ceux qui sont passés en utilisant leur nom.

Par exemple, la fonction suivante prend trois paramètres dont deux sont optionnels :

```
1 def maFonction( a, b = 1, c = 0 ):
2     return a + b + c
```

Les paramètres `b` et `c` sont optionnels : ils ont respectivement la valeur 1 et 0 par défaut. C'est cette valeur qui sera prise si on ne les passe pas lors de l'appel de la fonction.

On peut alors appeler cette fonction de différentes façons. On peut passer tous les paramètres optionnels, ou aucun. Dans le premier cas, tous les paramètres passés seront pris en compte. Dans le deuxième cas, le ou les paramètre(s) passé(s) seront considérés comme correspondant aux premiers paramètres pris par la fonction, et pour les autres les valeurs par défaut seront utilisées.

```
1 maFonction( 2, 4, 6 )
2 maFonction( 2 )
```

Si on passe une partie des paramètres optionnels, deux cas se présentent. On peut passer les premiers dans l'ordre des paramètres acceptés par la fonction, auquel cas ceux-ci seront considérés dans l'ordre fourni et les paramètres suivant auront leur valeur par défaut :

```
1 maFonction( 1, 2 )          # ici b vaut 2
```

On peut aussi ne pas passer le premier paramètre optionnel mais en passer d'autres situés plus loin dans la liste des paramètres optionnels. Dans ce cas, il faut utiliser le nom du paramètre pour préciser duquel il s'agit :

```
1 maFonction( 1, c = 2 )      # ici c vaut 2
```

Attention : les arguments obligatoires doivent *toujours* être situés avant les arguments optionnels.

4.7 Point d'entrée dans le programme

L'exécution d'un script Python commence par la *première ligne en-dehors de toute fonction*. L'interpréteur lit séquentiellement les lignes de script et exécute la première ligne du bloc de plus haut niveau qui ne soit pas une définition de fonction.

```
1 def maFonction( a ):
2         print "parametre : " + str( a ) + " de type " + str( type( a ) )
3         return type( a )
4
5 maFonction( 5 ) # premiere ligne executee
```

Ci-dessus, la définition de fonction est chargée, mais la première instruction exécutée est la ligne 5 où figure l'appel de la fonction.

Les programmes Python complexes sont souvent composés de plusieurs *modules* (plus de détails section 6). On dispose d'une variable `__name__` contenant le nom du module dans lequel on se trouve, ou `__main__` dans le cas où on se trouve dans le script qui est en train d'être exécuté.

Généralement, on commence par tester si on se trouve dans le module principal, puis on effectue les appels de fonctions.

```
1 if __name__ == "__main__":
2     appel_fonction()
```

L'avantage est que le code ainsi écrit est plus facile à lire : on sait où commence le programme. Par analogie avec d'autres langages, on définit souvent une fonction `main()` qui est la seule instruction exécutée à l'entrée dans le programme. Les instructions du script et les appels aux autres fonctions sont placées dans cette fonction `main()` :

```
1 def main():
2     res = appel_fonction()
3     print res
4
5 if __name__ == "__main__":
6     main()
```

4.8 Visibilité des variables

4.8.1 Variables locales

Une variable déclarée dans une fonction ou une procédure n'est visible, c'est-à-dire accessible, que *dans le corps* de cette fonction ou de cette procédure.

4.8.2 Variables globales

Une *variable globale* est une variable faisant partie d'un script ou d'un module et ayant portée dans tout ce script ou ce module. En particulier, elle peut être utilisée (en lecture ou en écriture) dans toutes les fonctions et procédures définies dans ce script ou ce module.

Les constantes fournies par les modules Python sont donc des variables globales.

Attention : bien que séduisantes de prime abord, les variables globales sont souvent sources d'erreurs. Évitez autant que possible d'en utiliser. Généralement, moins un programme utilise de variables globales, mieux c'est.

En Python, une variable globale doit :
- Être utilisée au moins une fois au plus haut niveau d'exécution dans le script
  - Pour être vue de toutes les fonctions appelées dans ce script
- Être déclarée comme globale, avec le mot-clé `global`, quand elle est utilisée dans une fonction

4.8.3 Déclaration d'une variable globale

L'idée est qu'une variable globale est vue dans toute l'arborescence d'appels de fonction qui descend de l'endroit où elle est déclarée (c'est-à-dire, utilisée pour la première fois). C'est-à-dire que si elle est déclarée dans un bloc d'instructions, toutes les fonctions appelées dans ce bloc pourront voir cette variable.

Pour déclarer une variable globale, on la met généralement
- Au plus haut niveau d'indentation
- En haut du script, avant les déclarations de fonctions

L'intérêt de les mettre en haut du script ou du module est de regrouper les variables globales : ainsi, on améliore la lisibilité du code (on sait où trouver les variables globales, il n'y en a pas d'autres cachées dans les méandres du code), et on diminue les risques d'erreurs.

4.8.4 Utilisation d'une variable globale

L'utilisation d'une variable globale dans une fonction nécessite de déclarer le fait qu'on veut utiliser une variable globale. Pour cela, on utilise le mot-clé `global`.

```
1 def maFonction():
2     global varGlob
3     print varGlob
```

Sans cette précaution, l'interpéteur considérera qu'il s'agit d'une variable locale.

Considérons par exemple le programme suivant :

```
1 #!/usr/bin/python
2
3 globVar = 0
4
5 def fonction():
6     global globVar
7     globVar = 5
8
9 def fonc2():
```

```
10      globVar = 2
11
12  def main():
13      global globVar
14      print globVar
15      globVar = 1
16      print globVar
17      fonction()
18      print globVar
19      fonc2()
20      print globVar
21
22  if __name__ == "__main__":
23      main()
```

Nous avons déclaré une variable globale `globVar` initialisée à 0. La fonction `main()` déclare l'utiliser avec l'instruction `global globVar`. Elle affiche sa valeur et la modifie, puis l'affiche à nouveau : nous sommes en train de travailler sur la variable globale. Puis la fonction `fonction()` est appelée. Elle travaille également sur la variable globale `globVar` et modifie sa valeur pour la mettre à 5. L'affichage effectué par la fonction `main()` en sortie de `fonction()` donne bien la valeur modifiée. Puis la fonction `fonc2()` est appelée. Elle ne fonctionne pas sur la variable globale `globVar` mais sur une variable locale portant ce nom. De ce fait, l'affichage effectué par la fonction `main()` en sortie de `fonc2()` donne la valeur de la variable globale `globVar`, qui est toujours à 5.

Il est donc nécessaire de redoubler d'attention et de précautions dans l'utilisation de variables globales.

4.9 Approche descendante

En découpant un problème en sous-problèmes, qui sont eux-mêmes découpés en sous-problèmes, on en vient à concevoir un algorithme qui fait appel à des solutions à ces sous-problèmes. On définit alors ces solutions dans des fonctions ou dans des procédures, qui font elles-mêmes appel à d'autres fonctions et procédures.

Par exemple, considérons un problème de géométrie. On veut calculer la longueur de l'hypoténuse d'un triangle rectangle, en connaissant la longueur de ses deux autres côtés.

L'approche naïve et inefficace conduit à l'algorithme suivant :

```
1  début fonction hypo( cote1: Entier, cote2: Entier ): Entier
2      /* variables internes */
3      carre1 : Entier
4      carre2 : Entier
5      hypotenusecarre : Entier
6      hypotenuse : Entier
7      /* on calcule la somme des carrés des côtés */
8      carre1 ⟵ cote1 × cote1
9      carre2 ⟵ cote2 × cote2
10     hypotenusecarre ⟵ carre1 + carre2
11     /* on prend la racine carrée de la somme et on retourne le résultat */
12     hypotenuse ⟵ √hypotenusecarre
13     retourner hypotenuse
14 fin fonction
```

Tous les calculs sont définis à la suite les uns des autres. Cependant, le théorème de Pythagore qui calcule de la longueur de l'hypoténuse d'un triangle rectangle se définit comme suit : *"le carré*

de l'hypoténuse d'un triangle rectangle est égal à la somme des carrés des longueurs des deux autres côtés".

On découpe alors notre algorithme comme suit :

```
1  début fonction hypo( cote1: Entier, cote2: Entier ): Entier
2  |  /* variables internes */
3  |  carre1 : Entier
4  |  carre2 : Entier
5  |  hypotenusecarre : Entier
6  |  hypotenuse : Entier
7  |  /* on calcule la somme des carrés des côtés */
8  |  carre1 ⟵ carré( cote1 )
9  |  carre2 ⟵ carré( cote2 )
10 |  hypotenusecarre ⟵ carre1 + carre2
11 |  /* on prend la racine carrée de la somme et on retourne le résultat */
12 |  hypotenuse ⟵ racine( hypotenusecarre )
13 |  retourner hypotenuse
14 fin fonction
```

Cet algorithme présente l'avantage de ne pas demander de savoir faire les calculs eux-mêmes : *on retarde le plus possible le moment d'effectuer les calculs*. Par exemple, un calcul de racine carrée est compliqué à mettre en place sur une machine. On préfère appeler une fonction qui l'effectue une fois pour toutes, plutôt que de devoir penser tout de suite à la façon dont on va faire ce calcul. À ce moment de la conception de l'algorithme, on n'a pas à savoir comment on va la calculer. On se donne un *niveau d'abstraction* qui permet de penser à la structure générale de l'algorithme, puis de *descendre dans les détails* de la résolution des sous-problèmes.

L'algorithme ci-dessus fait appel à deux fonctions : **carré()** et **racine()**. On donne leur algorithme séparément :

```
1 début fonction carré( nombre: Entier ): Entier
2 |  moncarre : Entier
3 |  moncarre ⟵ nombre × nombre
4 |  retourner moncarre
5 fin fonction
```

```
1 début fonction racine( nombre: Entier ): Entier
2 |  maracine : reel
3 |  maracine ⟵ √nombre
4 |  retourner maracine
5 fin fonction
```

De cette façon, on se concentre spécifiquement et de manière isolée sur les calculs eux-mêmes. Lorsqu'on conçoit un algorithme, il vaut mieux écrire de nombreuses fonctions et procédures courtes, plutôt qu'une seule et longue fonction.

5 Exceptions

Il existe en Python [2] un mécanisme permettant de signaler un évènement survenu dans une fonction à l'appelant de cette fonction.

2. Comme dans d'autres langages tels que Java par exemple

On appelle ce mécanisme le fait de *soulever une exception*. L'exécution d'un bloc d'instructions s'arrête à un endroit arbitraire et remonte dans le code appelant. L'appelant peut (doit) *attraper l'exception* pour éventuellement agir en conséquence.

L'idée est de signaler un évènement et d'arrêter l'exécution de la séquence d'instructions. Il peut s'agir d'une erreur (par exemple, un dépassement d'indice dans un tableau) mais aussi d'une condition validée (par exemple, une valeur trouvée par une routine de recherche).

On peut soulever *différents types d'exceptions* : cela permet à l'appelant de savoir quel évènement est survenu, et ainsi de traiter en conséquence.

5.1 Lever une exception

On lève une exception en utilisant le mot-clé `raise`. On peut lever une exception définie par le système.

```
1 if type( idx ) != IntType:
2     raise TypeError
```

Dans le code qui précède, on teste le type d'une variable `idx`. Si il ne s'agit pas d'un entier, on lève l'exception `TypeError`.

On peut passer un paramètre à certaines exceptions lorsqu'elles sont levées. Par exemple, dans le cas de `TypeError` on peut passer la variable dont le type est erroné.

```
1 if type( idx ) != IntType:
2     raise TypeError( idx )
```

5.2 Attraper une exception

Les instructions susceptibles de lever une exception doivent être exécutées dans un bloc `try`. Directement à la suite du bloc `try`, on met les instructions de gestion des exceptions, c'est-à-dire la conduite à tenir en cas d'exception soulevée dans une fonction appelée dans le bloc `try`.

Les exceptions soulevées sont traitées dans des clauses `except`, qui sont interprétées les unes après les autres. La structure est donc la suivante :

```
1 try:
2     bloc d'instructions
3 except:
4     traitement des exceptions
```

On peut avoir plusieurs clauses de gestion des exceptions, afin de différencier les actions à effectuer en fonction du type d'exception soulevée. Dans ce cas, on écrit les clauses `except` les unes après les autres en les distinguant par le type d'exception dont il est question. Si l'on utilise la clause `except` seule sans type d'exception, toutes les exceptions seront attrapées dans cette clause.

```
1 try:
2     bloc d'instructions
3 except Exception1:
4     traitement des exceptions de type Exception1
5 except Exception2:
6     traitement des exceptions de type Exception2
7 except:
8     traitement de tous les autres types d'exceptions
```

On peut aussi avoir un même traitement pour plusieurs types d'exceptions. Dans ce cas, on les précise à la suite les unes des autres, dans une clause `except` :

```
1  try:
2      bloc d'instructions
3  except Exception1:
4      traitement des exceptions de type Exception1
5  except Exception2, Exception3:
6      traitement des exceptions de type Exception2 ou
           Exception3
7  except:
8      traitement de tous les autres types d'exceptions
```

Une exception, lorsqu'elle est soulevée, est un objet disponible en mémoire. On peut accéder à l'exception elle-même au moment où elle est attrapée. C'est une partie de la puissance du système d'exceptions : des données sont remontées à l'appelant. Pour accéder à une exception dans une clause `except`, on nomme la variable correspondante :

```
1  try:
2      bloc d'instructions
3  except Exception e:
4      traitement des exceptions
5      on accède à la variable e
```

Par exemple, on peut afficher son type. La plupart des exceptions peuvent être affichées, c'est-à-dire qu'une fonction particulière aura été définie pour retourner une chaîne de caractères donnant des informations sur l'exception :

```
1  try:
2      bloc d'instructions
3  except Exception e:
4      print type( e )
5      print e
```

Lorsqu'aucune exception n'a été soulevée, on peut exécuter une séquence d'instructions définie grâce au bloc `else` :

```
1  try:
2      bloc d'instructions
3  except Exception1:
4      traitement des exceptions de type Exception1
5  except Exception2:
6      traitement des exceptions de type Exception2
7  except:
8      traitement de tous les autres types d'exceptions
9  else:
10     traitement si aucune exception n'a été soulevée
```

On peut aussi définir un bloc `finally`, qui sera toujours exécuté, quelle que soit l'exception soulevée ou même si aucune exception n'est soulevée :

```
try:
    bloc d'instructions
except Exception1:
    traitement des exceptions de type Exception1
except Exception2:
    traitement des exceptions de type Exception2
except:
    traitement de tous les autres types d'exceptions
else:
    traitement si aucune exception n'a été soulevée
finally:
    bloc toujours exécuté
```

5.3 Informations remontées par les exceptions

L'intérêt des exceptions est savoir ce qui s'est mal passé dans la fonction qui a levé l'exception. On peut notamment s'intéresser aux valeurs remontées dans l'exception. Suivant la définition de l'exception, on peut lui passer une ou plusieurs valeurs en l'appelant.

Par exemple, l'exception `TypeError` peut prendre en paramètre la variable concernée par l'erreur. On soulève l'exception en passant la variable incriminée en paramètre :

```
raise TypeError( var )
```

La valeur peut alors être récupérée par l'appelant qui attrape l'exception. Ces informations sont fournies dans des variables du module `sys`, notamment :

- `exc_info()` renvoie un tuple donnant des informations : valeur remontée, pile d'appels
- `exc_value` contient la valeur remontée par l'exception
- `exc_type` contient le type d'exception soulevée
- `exc_traceback` contient la pile d'appels au moment de lever l'exception

NB : le tuple renvoyé par `exc_info()` contient `exc_value`, `exc_type` et `exc_traceback`.

Lorsqu'une exception est soulevée, on peut donc récupérer des informations la concernant comme suit :

```
try:
    maFonction()
except:
    import sys
    print "Exception soulevee ", sys.exc_type, sys.
        exc_value
```

5.4 Créer un type d'exception

Le système propose un certain nombre de types d'exceptions prédéfinies. Cependant, il est souvent utile de définir son propre type d'exceptions. On défini dans ce cas une nouvelle *classe d'exception*.

Une classe d'exception doit définir deux fonctions :

- `__init__()` : fonction d'initialisation de l'entité exception lorsqu'elle est créée. Cette fonction ne retourne rien.
- `__str__()` : fonction qui permet à `print` d'afficher l'état de l'exception sous forme d'une chaîne de caractères. Cette fonction doit retourner une chaîne de caractères.

Par exemple, le code suivant défini une classe d'exception `ErreurNegIndice` qui est appelée lorsqu'un indice est négatif (ce qui est impossible dans un tableau par exemple).

```
1  class ErreurNegIndice( Exception ):
2
3      def __init__( self ):
4          self.str = "Indice negatif"
5
6      def __init__( self, idx ):
7          self.str = "Indice negatif: " + str( idx )
8
9      def __str__( self ):
10         return self.str
```

L'exception soulevée contient une variable de type chaîne de caractères : la variable `str`. Cette variable est initialisée à la création de la classe dans la fonction `__init__()`. On constate que cette fonction est ici *surchargée* : on peut l'appeler en ne lui passant aucun paramètre (première version) ou en lui passant l'indice fautif (deuxième version).

On lève cette exception avec l'instruction `raise` avec ou sans paramètre :

```
1  raise ErreurNegIndice( idx )
2  raise ErreurNegIndice
```

6 Modules

La programmation modulaire présente de nombreux avantages, tant du point de vue de la méthode de programmation qu'elle implique que de celui de la réutilisation du code écrit.

6.1 Définition

Un *module* est un *regroupement de procédures et de fonctions*, dont le but est de les mettre à disposition de plusieurs programmes appelant ces procédures et fonctions.

Ainsi, un module peut être utilisé par autant de programmes que souhaité : le code écrit une fois est réutilisé autant de fois que nécessaire. Cela permet de factoriser l'effort de développement, d'éviter les risques d'erreurs en recopiant du code, et facilite la maintenance de ce code : une erreur corrigée dans un module sera bénéfique pour tous les programmes utilisant ce module.

La programmation modulaire permet également de découper l'architecture d'un programme en autant de modules que de tâches indépendantes. Cette méthode induit une grande souplesse de programmation et une méthode efficace pour penser le programme à écrire. Par voie de conséquence, elle est également très utile dans un contexte de développement collaboratif : le programme est découpé en modules, chaque développeur écrit un module, et chacun appelle les fonctions du module écrit par les autres.

On peut citer quelques exemples de modules :

- Fonctions mathématiques : `math` : fournit des fonctions mathématiques telles que `ceil()` et `floor()`, `exp()`, `pow()` et `log()`, `sqrt()`, les fonctions de trigonométrie...
- La tortue utilisée en TP est disponible sous forme d'un module
- Modules de compression : `zlib`, `gzip`, `bz2`...
- Interfaces avec le système d'exploitation : `os`
- Génération de nombres aléatoires : `random`

6.2 Contenu d'un module

Comme présenté dans la définition introductive, un module contient :

- Des procédures
- Des fonctions

– Des constantes

Par exemple, dans le module `math` on trouve :

– Des fonctions mathématiques : `pow(x, y)` retourne x^y (`x` élevé à la puissance `y`)

– Des constantes mathématiques : π et e

Un module est donc une suite de définitions de constantes, de procédures et de fonctions.

Considérons par exemple un module qui tracerait des formes géométriques en utilisant la tortue vue en TP. Nous allons définir dans ce module deux procédures :

– `carre`, qui va tracer un carré

– `polygone`, qui va tracer un polygone régulier

ainsi qu'une constante :

– `VERSION`

Le nom de cette constante est écrit en majuscules, convention souvent suivie pour les constantes.

Attention : les constantes sont à manipuler avec extrême précaution ! Par définition, une constante ne doit pas être modifiée. Cependant, Python ne nous fournit pas de mécanisme permettant de protéger les constantes. Une règle de programmation à respecter est donc de ne *jamais modifier une constante.*

Le code de notre module est alors :

```
1  #!/usr/bin/python
2
3  from turtle import *
4
5  VERSION = 0.9
6
7  '''
8  Trace un carre
9  Arguments :
10  x : longueur des cotes
11 Retourne :
12  rien
13 '''
14
15 def carre( x ):
16     for i in range( 4 ):
17         forward( x )
18         left( 90 )
19
20 '''
21 Trace un polygone regulier
22 Arguments :
23  x : longueur des cotes
24  nb : nombre de cotes
25 Retourne :
26  rien
27 '''
28
29 def polygone( x, nb ):
30     for i in range( nb ):
31         forward( x )
32         left( 360/nb )
```

On peut prévoir d'exécuter le module comme un script. Cela permet notamment de le tester, afin de vérifier qu'il fonctionne correctement (utile notamment lors du développement du module). Pour cela, on utilise une fonction qui sera exécutée uniquement lorsque le module est lancé comme un script.

Nous avons vu dans la section 4 que la variable `__name__` contient le nom du module courant. Elle est toujours à la valeur `__main__` quand le script est éxécuté, sinon elle contient le nom du module. Nous pouvons alors écrire un bout de code qui ne sera exécuté que si le module est exécuté comme un script :

```
1 if __name__ == "__main__":
2     main()
```

Dans ce qui précède, la condition ligne 1 ne sera validée que si le module est exécuté en tant que script. La ligne 2 sera donc exécutée uniquement dans ce cas. Si le module est chargé par un script, le condition ne sera pas validée et la ligne 2 ne sera pas exécutée.

On peut donc écrire une fonction `main()` qui sera appelée dans ce cas et qui appellera les fonctions et procédures définies dans notre module.

```
1 def main():
2     print "Test du module mod_tortue"
3     carre( 30 )
4     polygone( 30, 6 )
5
6 if __name__ == "__main__":
7     main()
```

6.3 Utilisation d'un module

L'utilisation d'un module se fait en deux temps :
- Chargement du module : mot-clé `import`
- Utilisation du module en appelant les fonctions, procédures et constantes qui en font partie.

Il existe deux façons différentes d'importer un module. L'utilisation de son contenu dépend de la façon dont il est importé.

La première méthode consiste à importer *tout le module*. Par exemple, avec le module `math`, on utilisera :

```
1 import math
```

L'utilisation du contenu du module ainsi importé se fait en utilisant la notation *pointée* : on utilise le nom du module, un point, et le nom de ce que l'on veut utiliser. Par exemple, avec le module `math` importé comme ci-avant :

```
1 print math.pi
2 s = math.sin( 35 )
```

La deuxième façon de faire permet d'importer des parties spécifiques d'un module. On les précise de la façon suivante :

```
1 from math import ceil, floor, pi
```

On peut demander d'importer l'intégralité du contenu du module avec l'étoile :

```
1 from math import *
```

On peut appeler le contenu ainsi importé directement, en utilisant son nom seul :

```
1 n = ceil( pi )
2 m = floor( pi )
```

La deuxième méthode permet de mieux maîtriser ce qui est chargé en n'important que les parties nécessaires d'un module. C'est particulièrement intéressant pour limiter l'espace mémoire nécessaire dans le cas de l'utilisation d'une sous-partie réduite d'un module particulièrement volumineux. De plus, l'utilisation directe du nom des fonctions, des procédures et des constantes du module permet d'écrire en apparence plus rapidement du code. Cependant, cela introduit un risque de conflit dans les cas où plusieurs éléments provenant de différents modules portent le même nom.

6.4 Lister le contenu d'un module

On peut obtenir les variables et les fonctions fournies par un module en utilisant la fonction `dir()` sur un module que l'on a préalablement chargé. Par exemple, dans l'interpréteur de commandes interactif, on peut demander à examiner le module `time` :

```
1 >>> import time
2 >>> dir( time )
3 ['__doc__', '__file__', '__name__', '__package__', 'accept2dyear',
      'altzone', 'asctime', 'clock', 'ctime', 'daylight', 'gmtime',
     'localtime', 'mktime', 'sleep', 'strftime', 'strptime', '
     struct_time', 'time', 'timezone', 'tzname', 'tzset']
```

On voit notamment que ce module fournit une constante `__doc__`, qui documente ce module. On peut donc y accéder :

```
1 >>> print time.__doc__
2 This module provides various functions to manipulate time values.
3
4 There are two standard representations of time.  One is the number
5 of seconds since the Epoch, in UTC (a.k.a. GMT).  It may be an
     integer
6 [...]
7 strftime() -- convert time tuple to string according to format
     specification
8 strptime() -- parse string to time tuple according to format
     specification
9 tzset() -- change the local timezone
```

Le module qui réunit les fonctions et les constantes fournies par le système par défaut : le module `__builtin__`. On peut donc notamment obtenir leur liste avec l'instruction `dir( __builtin__ )`. On constate notamment la constante `__doc__` :

```
1 >>> print __builtin__.__doc__
2 Built-in functions, exceptions, and other objects.
3
4 Noteworthy: None is the 'nil' object; Ellipsis represents '...' in
      slices.
```

6.5 Où sont installés les modules

Lorsqu'un module est importé, le système va le chercher, dans cet ordre :
- Parmi les modules du système Python (built-in)
- Dans le répertoire courant
- Dans les répertoires listés dans la variable d'environnement $PYTHONPATH
- Dans les répertoires par défaut de l'installation

Remarque : la liste des répertoires est donnée dans la variable `sys.path`, modifiable par le programme.

7 Lectures et écritures de fichiers

La manipulation de fichiers se fait *toujours* en *trois étapes*:
- Ouverture du fichier
- Manipulation du fichier
- Fermeture du fichier

Il faut faire très attention à trois choses :
- Ne pas oublier d'ouvrir le fichier
- Ne pas oublier de fermer le fichier
- Bien gérer les erreurs

Ces dernières sont très fréquentes :
- Fichier inexistant
- Disque plein donc écriture impossible
- Pas les bons droits sur le fichier
- ... (plein de possibilités d'erreurs)

Le mécanisme d'exceptions de Python (voir section 5 est très efficace et très pratique pour récupérer des informations sur les erreurs survenues dans les fonctions de manipulations de fichiers. Il est donc nécessaire de bien gérer les exceptions pouvant être soulevées lors de leurs appels.

7.1 Ouverture et fermeture de fichier

On utilise la fonction `open()`, qui prend deux arguments :
- Le *chemin vers le fichier* (chaîne de caractères)
- Le *mode d'ouverture* (chaîne de caractères)

Le mode d'ouverture précise :
- Si on ouvre en lecture, en écriture, ou en lecture/écriture
- Où on se place initialement dans le fichier : début ou fin
- Si le fichier est tronqué à zéro ou non

Mode	Utilisation
r	Lecture
w	Écriture (créé ou tronqué à 0)
r+	Mise à jour (lecture et écriture)
w+	Tronque le fichier à zéro et l'ouvre en lecture-écriture
a	Ajoute à la fin du fichier
a+	Lit au début du fichier, ajoute à la fin du fichier

Pour résumer les principes des différents modes disponibles :
- Avec `r` ou `w` : on est positionné au *début* du fichier
- Avec `a` : on est positionné à la *fin* du fichier (*append*)
- Avec `w` : on tronque le fichier à 0 ou on le crée
- Pour écrire sans écraser : `a` ou `a+` à la fin, `r+` au début

Exemple : ouverture du fichier `/etc/hosts` en lecture

```
1 fd = open( '/etc/hosts', 'r' )
```

La fonction `open()` retourne un *descripteur de fichier*. C'est sur cette structure que l'on va effectuer les manipulations sur le fichier.

```
1 >>> type( fd )
2 <type 'file'>
3 >>> print fd
4 <open file '/etc/hosts', mode 'r' at
     0x7facf5f3f390>
```

L'ouverture d'un fichier est susceptible de soulever une exception : si les permissions dont nous disposons sur ce fichier ne sont pas suffisantes pour l'ouvrir dans le mode demandé, si le fichier n'existe pas et que le mode demandé ne permet pas de le créer... Il est donc nécessaire de gérer les exceptions lors des ouvertures de fichiers. L'exception soulevée par `open()` es de type `IOError`. On a plus de détails sur l'erreur dans `sys.exc_value`.

Par exemple, si l'on tente d'ouvrir un fichier sur lequel nous n'avons pas les droits en lecture :

```
1 try:
2     fd = open( '/var/log/syslog', 'r' )
3 except:
4     import sys
5     print "Erreur ouverture", sys.exc_type, sys.
         exc_value
```

Une exception est soulevée.

```
1 Erreur ouverture <type 'exceptions.IOError'> [Errno 13] Permission denied:
      '/var/log/syslog'
```

On ferme un fichier avec la procédure `close()` appelée sur le descripteur de fichier :

```
1 fd.close()
```

- On peut fermer une fichier plus d'une fois
- On ne peut pas fermer un fichier qui n'a pas été ouvert
- On doit fermer un fichier quand on en a fini avec lui

Des tentatives d'opérations sur un fichier fermé soulèveront l'exception `ValueError`

7.2 Lecture d'un fichier

On peut lire un fichier de deux façons différentes, selon ce qu'il contient.

Si il s'agit d'un fichier texte : on peut le lire *ligne par ligne* avec la fonction `readline()` sur le descripteur de fichier (ne prend pas d'argument). Cette fonction retourne la chaîne de caractères lue.

```
1 ligne = fd.readline( )
2 print "ligne lue : ", ligne
```

On peut lire toutes les lignes du fichier d'un coup avec la fonction `readlines()`, qui retourne toutes les lignes dans une liste.

```
1 lignes = fd.readlines( )
2 for l in lignes:
3     print l
```

Si il s'agit d'un *fichier binaire*, on le lit lecture *octet par octet* avec la fonction `read()`, qui prend comme argument le nombre d'octets à lire ou rien pour tout lire d'un coup. Cette fonction retourne une chaîne de caractères ou d'octets. Si on en lit moins (fichier plus court) : on peut utiliser la fonction `len()` sur ce qui est retourné pour avoir la taille réellement lue.

```
1 ret = fd.read( 512 )
2 print "lus : ", len( ret )
```

Si on ne précise pas d'argument, l'intégralité du fichier est lue.

```
1 ret = fd.read( )
2 print "lus : ", len( ret )
```

Quand on arrive à la `fin du fichier`, les fonctions `read()` et `readline()` retournent une chaîne vide :

```
1 line = fd.readline()
2 while line:
3     print line
4     line = fd.readline()
```

Avec `read()` en passant la longueur à lire en argument, si on atteint la fin du fichier on s'arrête là et on obtient la longueur réellement lue avec `len()` :

```
1 buffsize = 512
2 ret = fd.read( buffsize )
3 lu = len( ret )
4 while len( ret ) == buffsize:
5    ret = fd.read( buffsize )
6    lu += len( ret )
7 print lu, "octets lus"
```

7.3 Écriture dans un fichier

On écrit de la même façon du texte ou des données binaires

– Écrire une *chaîne de caractères ou d'octets* : procédure `write()`

```
1 fd.write( "texte" )
```

– Écrire une *liste d'éléments* : procédure `writelines()`

Ce sont des *procédures* : elles ne retournent rien.

7.4 Exceptions soulevées lors des manipulations de fichiers

Attention aux exceptions soulevées

– Très utiles ! Et très courantes

– Système de fichier plein, droits inadaptés, tentative d'écriture sur un descripteur de fichier fermé...

Exemple : on essaye d'écrire sur un fichier qui a été fermé

```
1 fd.close()
2 try:
3     r = fd.read()
4 except:
5     import sys
6     print "Erreur de lecture", sys.exc_type, sys.
          exc_value
```

Donnera :

```
1 Erreur de lecture <type 'exceptions.ValueError'> I/O
      operation on closed file
```

7.5 Autres fonctions utiles sur les fichiers

On peut se *déplacer dans un fichier* en utilisant la procédure `seek()`
- On spécifie le nombre d'octets dont on se déplace
- Optionnel : point de référence du déplacement
  - 0 : par rapport au début du fichier (valeur par défaut)
  - 1 : par rapport à la position actuelle
  - 2 : par rapport à la fin du fichier

```
1 fd.seek( 0 )      # on se place au debut du
      fichier
2 fd.seek( 0, 2 )   # on se place a la fin du
      fichier
3 fd.seek( 512, 1 ) # on avance de 512 octets
4 fd.seek( -64, 1 ) # on recule de 64 octets
```

Une optimisation des systèmes d'exploitation fait que les écritures sur disque sont mises en cache dans des tampons internes Python et du système d'exploitation. La conséquence est que ce qui est écrit dans les fichiers n'est pas forcément mis sur le disque immédiatement. Cela peut poser des problèmes (accès concurrents, programme qui plante...).

On peut forcer le vidage du tampon interne avec `flush()`
- ... mais `flush()` ne force pas l'écriture sur disque :-(
- On utilise alors ensuite une procédure du module `os` : `os.fsync()`, en lui passant le descripteur de fichier.

```
1 fd.flush()
2 os.fsync(fd_out)
```