Programmation avec le langage Python

Version 0.1.273

Xavier Dupré

nov. 12, 2018

Table des matières

CHAPITRE 1

Introduction

Ceci est une relecture du livre que j'ai écrit en 2009 Programmation avec le langage Python [1] et disponible au format PDF [2]. Internet est le média le plus utilisé quant il s'agit d'apprendre à programmer et un module Python est plus pratique pour s'assurer qu'un code Python reste valide lors du passage à une nouvelle version du langage.

Installation

Les exemples de codes sont disponibles sous la forme d'un module python et des *notebooks* (page **??**) accessibles sur le site.

```
pip install teachpyx
```

1. http://www.editions-ellipses.fr/product_info.php?products_id=6891
2. http://www.xavierdupre.fr/site2013/index_documents.html

CHAPITRE 2

Langage, variable et fonctions

2.1 Types et variables du langage python

2.1.1 Variables

Il est impossible d'écrire un programme sans utiliser de variable. Ce terme désigne le fait d'attribuer un nom ou identificateur à des informations : en les nommant, on peut manipuler ces informations beaucoup plus facilement. L'autre avantage est de pouvoir écrire des programmes valables pour des valeurs qui varient : on peut changer la valeur des variables, le programme s'exécutera toujours de la même manière et fera les mêmes types de calculs quelles que soient les valeurs manipulées. Les variables jouent un rôle semblable aux inconnues dans une équation mathématique.

L'ordinateur ne sait pas faire l'addition de plus de deux nombres mais cela suffit à calculer la somme de n premiers nombres entiers. Pour cela, il est nécessaire de créer une variable intermédiaire qu'on appellera par exemple `somme` de manière à conserver le résultat des sommes intermédiaires.

<<<

```
n = 11
somme = 0                    # initialisation : la somme est nulle
for i in range(1, n):        # pour tous les indices de 1 à n exclu
    somme = somme + i        # on ajoute le i ème élément à somme
print(somme)
```

>>>

```
    55
```

Définition D1 : variable

Une variable est caractérisée par :

- **un identificateur :** il peut contenir des lettres, des chiffres, des blancs soulignés mais il ne peut commencer par un chiffre. Minuscules et majuscules sont différenciées. Il est aussi unique.
- **un type :** c'est une information sur le contenu de la variable qui indique à l'interpréteur *python*, la manière de manipuler cette information.

Comme le typage est dynamique en *python*, le type n'est pas précisé explicitement, il est implicitement liée à l'information manipulée. Par exemple, en écrivant, `x=3.4`, on ne précise pas le type de la variable `x` mais il est implicite car x reçoit une valeur réelle : `x` est de type réel ou *float* en *python*. Pour leur première initialisation, une variable reçoit dans la plupart des cas une constante :

Définition D2 : constante

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

Le langage *python* possède un certain nombre de types de variables déjà définis ou types fondamentaux à partir desquels il sera possible de définir ses propres types (voir chapitre *Classes* (page 95)). Au contraire de langages tels que le *C*, il n'est pas besoin de déclarer une variable pour signifier qu'elle existe, il suffit de lui affecter une valeur. Le type de la variable sera défini par le type de la constante qui lui est affectée. Le type d'une variable peut changer, il correspond toujours au type de la dernière affectation.

```
x   = 3.5       # création d'une variable nombre réel appelée x initialisée à 3.5
                # 3.5 est un réel, la variable est de type "float"
sc = "chaîne"   # création d'une variable chaîne de caractères appelée str
                # initialisée à "chaîne", sc est de type "str"
```

Pour tous les exemples qui suivront, le symbole # apparaîtra à maintes reprises. Il marque le début d'un commentaire que la fin de la ligne termine. Autrement dit, un commentaire est une information aidant à la compréhension du programme mais n'en faisant pas partie comme dans l'exemple qui suit.

```
x = 3           # affectation de la valeur entière 3 à la variable x
y = 3.0         # affectation de la valeur réelle 3.0 à la variable y
```

Le *python* impose une instruction par ligne. Il n'est pas possible d'utiliser deux lignes pour écrire une affectation à moins de conclure chaque ligne qui n'est pas la dernière par le symbole \ L'exemple suivant est impossible.

```
x =
    5.5
```

Il devrait être rédigé comme suit :

```
x =  \
    5.5
```

Avec ce symbole, les longues instructions peuvent être écrites sur plusieurs lignes de manière plus lisibles, de sorte qu'elles apparaissent en entier à l'écran. Si le dernier caractère est une virgule, il est implicite.

Les paragraphes suivant énumèrent les types incontournables en *python*. Ils sont classés le plus souvent en deux catégories : types *immuables* ou *modifiables*. Tous les types du langage *python* sont également des objets, c'est pourquoi on retrouve dans ce chapitre certaines formes d'écriture similaires à celles présentées plus tard dans le chapitre concernant les classes (*Classes* (page 95)).

2.1.2 Types immuables (ou immutable)

Définition D3 : type immuable (ou immutable)

Une variable de type immuable ne peut être modifiée. Une opération sur une variable de ce type entraîne nécessairement la création d'une autre variable du même type, même si cette dernière est temporaire.

Autrement dit, la simple instruction `x+=3` qui consiste à ajouter à la variable `x` la valeur `3` crée une seconde variable dont la valeur est celle de `x` augmentée de `3` puis à en recopier le contenu dans celui de la variable `x`. Les nombres sont des types immuables tout comme les chaînes de caractères et les `tuple` qui sont des tableaux d'objets. Il n'est pas possible de modifier une variable de ce type, il faut en recréer une autre du même type qui intègrera la modification.

Type ńărienă" ou None

python propose un type `None` pour signifier qu'une variable ne contient rien. La variable est de type `None` et est égale à `None`.

```
s = None
print(s)     # affiche None
```

Certaines fonctions utilisent cette convention lorsqu'il leur est impossible de retourner un résultat. Ce n'est pas la seule option pour gérer cette impossibilité : il est possible de générer une *exception* (page 155), de retourner une valeur par défaut ou encore de retourner `None`. Il n'y a pas de choix meilleur, il suffit juste de préciser la convention choisie.

Les fonctions sont définies au paragraphe *Fonctions* (page 53), plus simplement, ce sont des mini-programmes : elles permettent de découper un programme long en tâches plus petites. On les distingue des variables car leur nom est suivi d'une liste de constantes ou variables comprises entre parenthèses et séparées par une virgule.

2.1.3 Nombres réels et entiers

Documentation : Numeric Types ¿int, float, complex [3].

3. https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

Il existe deux types de nombres en *python*, les nombres réels `float` et les nombres entiers `int`. L'instruction `x=3` crée une variable de type `int` initialisée à `3` tandis que `y=3.0` crée une variable de type `float` initialisée à `3.0`. Le programme suivant permet de vérifier cela en affichant pour les variables `x` et `y`, leurs valeurs et leurs types respectifs grâce à la fonction `type`.

<<<

```
x = 3
y = 3.0
print("x =", x, type(x))
print("y =", y, type(y))
```

>>>

```
    x = 3 <class 'int'>
    y = 3.0 <class 'float'>
```

La liste des opérateurs qui s'appliquent aux nombres réels et entiers suit. Les trois premiers résultats s'expliquent en utilisant la représentation en base deux. `8 << 1` s'écrit en base deux `100 << 1 = 1000`, ce qui vaut 16 en base décimale : les bits sont décalés vers la droite ce qui équivaut à multiplier par deux. De même, `7 & 2` s'écrit `1011 & 10 = 10`, qui vaut 2 en base décimale. Les opérateurs <<, >>, `|`, `&` sont des opérateurs bit à bit, ils se comprennent à partir de la représentation binaire des nombres entiers.

opérateur	signification	exemple
`<< >>`	décalage à gauche, à droite	`x = 8 << 1`
`|`	opérateur logique `ou` bit à bit	`x = 8 | 1`
`&`	opérateur logique `et` bit à bit	`x = 11 & 2`
`+ -`	addition, soustraction	`x = y + z`
`+= -=`	addition ou soustraction puis affectation	`x += 3`
`* /`	multiplication, division	`x = y * z`
`//`	division entière, le résultat est de type réel si l'un des nombres est réel	`x = y // 3`
`%`	reste d'une division entière (modulo)	`x = y % 3`
`*= /=`	multiplication ou division puis affectation	`x *= 3`
`**`	puissance (entière ou non, racine carrée = ** 0.5)	`x = y ** 3`

Les fonctions `int` et `float` permettent de convertir un nombre quelconque ou une chaîne de caractères respectivement en un entier (arrondi) et en un nombre réel.

<<<

```
x = int(3.5)
y = float(3)
z = int("3")
print("x:", type(x), "   y:", type(y), "   z:", type(z))
```

>>>

```
    x: <class 'int'>    y: <class 'float'>    z: <class 'int'>
```

Il peut arriver que la conversion en un nombre entier ne soit pas directe. Dans l'exemple qui suit, on cherche à convertir une chaîne de caractères (voir *Chaîne de caractères* (page 8)) en entier mais cette chaîne représente un réel. Il faut d'abord la convertir en réel puis en entier, c'est à ce moment que l'arrondi sera effectué.

```
i = int ("3.5")           # provoque une erreur
i = int (float ("3.5"))   # fonctionne
```

Les opérateurs listés par le tableau ci-dessus ont des priorités différentes, triés par ordre croissant. Toutefois, il est conseillé d'avoir recours aux parenthèses pour enlever les doutes : `3 * 2 ** 4 = 3 * (2 ** 4)`. La page Opertor Precedence [4] est plus complète à ce sujet.

python propose l'opérateur `//` pour les divisions entières et c'est une rare exception parmi les languages qui ne possèdent qu'un seul opérateur `/` qui retourne un entier pour une division entière excepté en *python* :

<<<

```
x = 11
y = 2
z = x // y       # le résultat est 5 et non 5.5 car la division est entière
zz = x / y       # le résultat est 5.5

print(z, zz)
```

>>>

```
    5 5.5
```

Pour éviter d'écrire le type `float`, on peut également écrire `11.0` de façon à spécifier explicitement que la valeur `11.0` est réelle et non entière. L'opérateur `//` permet d'effectuer une division entière lorsque les deux nombres à diviser sont réels, le résultat est un entier mais la variable est de type réel si l'un des nombres est de type réel.

Booléen

Les booléens sont le résultat d'opérations logiques et ont deux valeurs possibles : `True` ou `False`. Voici la liste des opérateurs [5] qui s'appliquent aux booléens.

opérateur	signification	exemple
`and or`	et, ou logique	`x = True or False` (résultat = True)
`not`	négation logique	`x = not x`

<<<

```
x = 4 < 5
print(x)          # affiche True
print(not x)      # affiche False
```

>>>

```
    True
    False
```

Voici la liste des opérateurs de comparaisons [6] qui retournent des booléens. Ceux-ci s'applique à tout type, aux entiers, réels, chaînes de caractères, tuples... Une comparaison entre un entier et une chaîne de caractères est syntaxiquement correcte même si le résultat a peu d'intérêt.

opérateur	signification	exemple
`< >`	inférieur, supérieur	`x = 5 < 5`
`<= >=`	inférieur ou égal, supérieur ou égal	`x = 5 <= 5`
`== !=`	égal, différent	`x = 5 == 5`

4. https://docs.python.org/3/reference/expressions.html#operator-precedence
5. https://docs.python.org/3/library/stdtypes.html#boolean-operations-and-or-not
6. https://docs.python.org/3/library/stdtypes.html#comparisons

A l'instar des nombres réels, il est préférable d'utiliser les parenthèses pour éviter les problèmes de priorités d'opérateurs dans des expressions comme : `3 < x and x < 7`. Toutefois, pour cet exemple, *python* accepte l'écriture résumée qui enchaîne des comparaisons : `3 < x and x < 7` est équivalent à `3 < x < 7`. Il existe deux autres mots-clés qui retournent un résultat de type booléen :

opérateur	signification	exemple
`is`	test d'identification	`"3" is str`
`in`	test d'appartenance	`3 in [3, 4, 5]`

Ces deux opérateurs seront utilisés ultérieurement, `in` avec les listes, les dictionnaires, les boucles (paragraphe *Boucle for* (page 44)), `is` lors de l'étude des listes (paragraphe *Copie* (page 23) et des *classes* (page 95)). Bien souvent, les booléens sont utilisés de manière implicite lors de tests (paragraphe test_test) ce qui n'empêche pas de les déclarer explicitement.

```
x = True
y = False
```

2.1.4 Chaîne de caractères

Création d'une chaîne de caractères - str

Définition D4 : chaîne de caractères

Le terme ńăchaîne de caractèresă" ou *string* en anglais signifie une suite finie de caractères, autrement dit, du texte.

Ce texte est compris entre deux guillemets ou deux apostrophes, ces deux symboles sont interchangeables. Le type *python* est str[7]. L'exemple suivant montre comment créer une chaîne de caractères. Il ne faut pas confondre la partie entre guillemets ou apostrophes, qui est une constante, de la variable qui la contient.

<<<

```
t = "string = texte"
print(type(t), t)
t = 'string = texte, initialisation avec apostrophes'
print(type(t), t)

t = "morceau 1" \
    "morceau 2"    # second morceau ajouté au premier par l'ajout du symbole \,
# il ne doit rien y avoir après le symbole \,
# pas d'espace ni de commentaire
print(t)

t = """première ligne
seconde ligne"""   # chaîne de caractères qui s'étend sur deux lignes
print(t)
```

>>>

```
    <class 'str'> string = texte
    <class 'str'> string = texte, initialisation avec apostrophes
    morceau 1morceau 2
    première ligne
        seconde ligne
```

7. https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str

La troisième chaîne de caractères créée lors de ce programme s'étend sur deux lignes. Il est parfois plus commode d'écrire du texte sur deux lignes plutôt que de le laisser caché par les limites de fenêtres d'affichage. *python* offre la possibilité de couper le texte en deux chaînes de caractères recollées à l'aide du symbole \ à condition que ce symbole soit le dernier de la ligne sur laquelle il apparaît. De même, lorsque le texte contient plusieurs lignes, il suffit de les encadrer entre deux symboles `"""` ou `'''` pour que l'interpréteur *python* considère l'ensemble comme une chaîne de caractères et non comme une série d'instructions.

Par défaut, le *python* ne permet pas l'insertion de caractères tels que les accents dans les chaînes de caractères, le paragraphe par_intro_accent_code explique comment résoudre ce problème. De même, pour insérer un guillemet dans une chaîne de caractères encadrée elle-même par des guillemets, il faut le faire précéder du symbole \. La séquence \ est appelée un extra-caractère (voir table extra_caractere) ou un caractère d'échappement.

`"`	guillemet
`'`	apostrophe
`\n`	passage à la ligne
`\\`	insertion du symbole \
`\%`	pourcentage, ce symbole est aussi un caractère spécial
`\t`	tabulation
`\r`	retour à la ligne, peu usité, il a surtout son importance lorsqu'on passe d'un système *Windows* à *Linux* car *Windows* l'ajoute automatiquement à tous ses fichiers textes
...	Lire String and Bytes literals [8].

Liste des extra-caractères les plus couramment utilisés à l'intérieur d'une chaîne de caractères (voir page Lexical analysis [9]).

Il peut être fastidieux d'avoir à doubler tous les symboles \ d'un nom de fichier. Il est plus simple dans ce cas de préfixer la chaîne de caractères par `r` de façon à éviter que l'utilisation du symbole \ ne désigne un caractère spécial. Les deux lignes suivantes sont équivalentes :

```
s = "C:\\Users\\Dupre\\exemple.txt"
s = r"C:\Users\Dupre\exemple.txt"
```

Sans la lettre `"r"`, tous les \ doivent être doublés, dans le cas contraire, *python* peut avoir des effets indésirables selon le caractère qui suit ce symbole.

Manipulation d'une chaîne de caractères

Une chaîne de caractères est semblable à un tableau et certains opérateurs qui s'appliquent aux tableaux s'appliquent également aux chaînes de caractères. Ceux-ci sont regroupés dans la table operation_string. La fonction `str` permet de convertir un nombre, un tableau, un objet (voir chapitre *Classes* (page 95)) en chaîne de caractères afin de pouvoir l'afficher. La fonction `len` retourne la longueur de la chaîne de caractères.

<<<

```
x = 5.567
s = str(x)
print(type(s), s)    # <type 'str'> 5.567
print(len(s))        # affiche 5
```

>>>

```
    <class 'str'> 5.567
    5
```

8. https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals
9. https://docs.python.org/3/reference/lexical_analysis.html

opérateur	signification	exemple
`+`	concaténation de chaînes de caractères	`t = "abc" + "def"`
`+=`	concaténation puis affectation	`t += "abc"`
`in`, `not in`	une chaîne en contient-elle une autre ?	`"ed" in "med"`
`*`	répétition d'une chaîne de caractères	`t = "abc" * 4`
`[n]`	obtention du enième caractère, le premier caractère a pour indice 0	`t = "abc"; print(t[0]) # donne a`
`[i:j]`	obtention des caractères compris entre les indices `i` et `j-1` inclus, le premier caractère a pour indice 0	`t = "abc"; print(t[0:2]) # donne ab`

Il existe d'autres fonctions qui permettent de manipuler les chaînes de caractères.

```
res = s.fonction (...)
```

Où `s` est une chaîne de caractères, `fonction` est le nom de l'opération que l'on veut appliquer à `s`, `res` est le résultat de cette manipulation.

La table string_method présente une liste non exhaustive des fonctions disponibles dont un exemple d'utilisation suit. Cette syntaxe `variable.fonction(arguments)` est celle des classes.

`count( sub[, start[, end]])`	Retourne le nombre d'occurences de la chaîne de caractères `sub`, les paramètres par défaut `start` et `end` permettent de réduire la recherche entre les caractères d'indice `start` et `end` exclu. Par défaut, `start` est nul tandis que `end` correspond à la fin de la chaîne de caractères.
`find( sub[, start[, end]])`	Recherche une chaîne de caractères `sub`, les paramètres par défaut `start` et `end` ont la même signification que ceux de la fonction `count`. Cette fonction retourne -1 si la recherche n'a pas abouti.
`isalpha()`	Retourne `True` si tous les caractères sont des lettres, `False` sinon.
`isdigit()`	Retourne `True` si tous les caractères sont des chiffres, `False` sinon.
`replace( old, new[, count])`	Retourne une copie de la chaîne de caractères en remplaçant toutes les occurrences de la chaîne `old` par `new`. Si le paramètre optionnel `count` est renseigné, alors seules les `count` premières occurrences seront remplacées.
`split( [sep [, maxsplit]])`	Découpe la chaîne de caractères en se servant de la chaîne `sep` comme délimiteur. Si le paramètre `maxsplit` est renseigné, au plus `maxsplit` coupures seront effectuées.
`upper()`	Remplace les minuscules par des majuscules.
`lower()`	Remplace les majuscules par des minuscules.
`join ( li )`	`li` est une liste, cette fonction agglutine tous les éléments d'une liste séparés par `sep` dans l'expression `sep.join ( ["un", "deux"])`.
`startswith(prefix[, start[, end]])`	Teste si la chaîne commence par `prefix`.
`endswith(suffix[, start[, end]])`	Teste si la chaîne se termine par `suffix`.
...	Lire String Methods [10].

<<<

```
st = "langage python"
st = st.upper()                 # mise en lettres majuscules
i = st.find("PYTHON")          # on cherche "PYTHON" dans st
print(i)                        # affiche 8
```

(suite sur la page suivante)

10. https://docs.python.org/3/library/stdtypes.html#string-methods

(suite de la page précédente)

```
print(st.count("PYTHON"))      # affiche 1
print(st.count("PYTHON", 9))   # affiche 0
```

>>>

```
    8
    1
    0
```

L'exemple suivant permet de retourner une chaîne de caractères contenant plusieurs éléments séparés par ";". La chaîne "un;deux;trois" doit devenir "trois;deux;un". On utilise pour cela les fonctionnalités split et join. L'exemple utilise également la fonctionnalité reverse des listes qui seront décrites plus loin dans ce chapitre. Il faut simplement retenir qu'une liste est un tableau. reverse retourne le tableau.

<<<

```
s = "un;deux;trois"
mots = s.split(";")         # mots est égal à ['un', 'deux', 'trois']
mots.reverse()              # retourne la liste, mots devient égal à
#                 ['trois', 'deux', 'un']
s2 = ";".join(mots)         # concaténation des éléments de mots séparés par ";"
print(s2)                    # affiche trois;deux;un
```

>>>

```
    trois;deux;un
```

Formatage d'une chaîne de caractères

Syntaxe %

python (printf-style String Formatting [11]) offre une manière plus concise de former une chaîne de caractères à l'aide de plusieurs types d'informations en évitant la conversion explicite de ces informations (type str) et leur concaténation. Il est particulièrement intéressant pour les nombres réels qu'il est possible d'écrire en imposant un nombre de décimales fixe. Le format est le suivant :

```
".... %c1  ....  %c2 " % (v1,v2)
```

c1 est un code choisi parmi ceux de la table format_print. Il indique le format dans lequel la variable v1 devra être transcrite. Il en est de même pour le code c2 associé à la variable v2. Les codes insérés dans la chaîne de caractères seront remplacés par les variables citées entre parenthèses après le symbole % suivant la fin de la chaîne de caractères. Il doit y avoir autant de codes que de variables, qui peuvent aussi être des constantes.

Voici concrètement l'utilisation de cette syntaxe :

<<<

```
x = 5.5
d = 7
s = "caractères"
res = "un nombre réel %f et un entier %d, une chaîne de %s, \n" \
      "un réel d'abord converti en chaîne de caractères %s" % (
          x, d, s, str(x+4))
print(res)
res = "un nombre réel " + str(x) + " et un entier " + str(d) + \
```

(suite sur la page suivante)

11. https://docs.python.org/3/library/stdtypes.html#old-string-formatting

(suite de la page précédente)

```
      ", une chaîne de " + s + \
      ",\n un réel d'abord converti en chaîne de caractères " + str(x+4)
print(res)
```

>>>

```
    un nombre réel 5.500000 et un entier 7, une chaîne de caractères,
    un réel d'abord converti en chaîne de caractères 9.5
    un nombre réel 5.5 et un entier 7, une chaîne de caractères,
     un réel d'abord converti en chaîne de caractères 9.5
```

La seconde affectation de la variable `res` propose une solution équivalente à la première en utilisant l'opérateur de concaténation `+`. Les deux solutions sont équivalentes, tout dépend des préférences de celui qui écrit le programme. La première option permet néanmoins un formatage plus précis des nombres réels en imposant par exemple un nombre défini de décimal. Le format est le suivant :

```
"%n.df" % x
```

`n` est le nombre de chiffres total et `d` est le nombre de décimales, `f` désigne un format réel indiqué par la présence du symbole `%`.

Exemple :

<<<

```
x = 0.123456789
print(x)              # affiche 0.123456789
print("%1.2f" % x)    # affiche 0.12
print("%06.2f" % x)   # affiche 000.12
```

>>>

```
    0.123456789
    0.12
    000.12
```

Il existe d'autres formats regroupés dans la table format_print. L'aide reste encore le meilleur réflexe car le langage *python* est susceptible d'évoluer et d'ajouter de nouveaux formats.

`d`	entier relatif
`e`	nombre réel au format exponentiel
`f`	nombre réel au format décimal
`g`	nombre réel, format décimal ou exponentiel si la puissance est trop grande ou trop petite
`s`	chaîne de caractères
...	Lire printf-style String Formatting [12].

Méthode format

La méthode format [13] propose plus d'options pour formatter le texte et son usage est de plus en plus fréquent. La méthode interprète les accolades `{}` comme des codes qu'elle remplace avec les valeurs passée en argument. Le type n'importe plus. Quelques exemples :

<<<

12. https://docs.python.org/3/library/stdtypes.html#old-string-formatting
13. https://docs.python.org/3/library/stdtypes.html#str.format

```
print('{0}, {1}, {2}'.format('a', 'b', 'c'))    # le format le plus simple
print('{}, {}, {}'.format('a', 'b', 'c'))       # sans numéro
print('{2}, {1}, {0}'.format('a', 'b', 'c'))    # ordre changé
print('{0}{1}{0}'.format('abra', 'cad'))        # répétition
```

>>>

```
    a, b, c
    a, b, c
    c, b, a
    abracadabra
```

La méthode accepte aussi les paramètres nommées et des expressions.

<<<

```
print('Coordinates: {latitude}, {longitude}'.format(
    latitude='37.24N', longitude='-115.81W'))
coord = (3, 5)
print('X: {0[0]};  Y: {0[1]}'.format(coord))
```

>>>

```
    Coordinates: 37.24N, -115.81W
    X: 3;  Y: 5
```

L'alignement est plus simple :

<<<

```
print('A{:<30}B'.format('left aligned'))
print('A{:>30}B'.format('right aligned'))
print('A{:^30}B'.format('centered'))
print('A{:*^30}B'.format('centered'))
```

>>>

```
    Aleft aligned                  B
    A                 right alignedB
    A           centered           B
    A***********centered**********B
```

Format numérique :

<<<

```
print('{:.2%}'.format(19.0/22.0))
print("int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42))
print("int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42))
print('{:,}'.format(1234567890))
```

>>>

```
    86.36%
    int: 42;  hex: 2a;  oct: 52;  bin: 101010
    int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010
    1,234,567,890
```

Dates :

<<<

```
import datetime
d = datetime.datetime.now()
print('{:%Y-%m-%d %H:%M:%S}'.format(d))
```

>>>

```
    2018-11-12 07:57:36
```

Le site pyformat [14] recense d'autres usages de la méthode format [15] comme l'affichage de chaînes de caractères tronquées.

<<<

```
print('{:10.5}'.format('formatages'))
```

>>>

```
    forma
```

tuple

Définition D5 : tuple

Les tuple sont un tableau d'objets qui peuvent être de tout type. Ils ne sont pas modifiables (les tuple [16] sont *immuables* ou *immutable*).

Un tuple apparaît comme une liste d'objets comprise entre parenthèses et séparés par des virgules. Leur création reprend le même format :

```
x = (4,5)                 # création d'un tuple composé de 2 entiers
x = ("un",1,"deux",2)     # création d'un tuple composé de 2 chaînes de caractères
                          # et de 2 entiers, l'ordre d'écriture est important
x = (3,)                  # création d'un tuple d'un élément, sans la virgule,
                          # le résultat est un entier
```

Ces objets sont des vecteurs d'objets. Etant donné que les chaînes de caractères sont également des tableaux, ces opérations reprennent en partie celles des _string_paragraphe_chaine et décrites par le paragraphe Common Sequence Operations [17].

14. https://pyformat.info/
15. https://docs.python.org/3/library/stdtypes.html#str.format
16. https://docs.python.org/3/library/stdtypes.html#tuple
17. https://docs.python.org/3/library/stdtypes.html#typesseq-common

`x in s`	vrai si `x` est un des éléments de `s`
`x not in s`	réciproque de la ligne précédente
`s + t`	concaténation de `s` et `t`
`s * n`	concatène `n` copies de `s` les unes à la suite des autres
`s[i]`	retourne le ième élément de `s`
`s[i:j]`	retourne un tuple contenant une copie des éléments de `s` d'indices *i* à *j* exclu
`s[i:j:k]`	retourne un tuple contenant une copie des éléments de `s` dont les indices sont compris entre *i* et *j* exclu, ces indices sont espacés de k : $i, i+k, i+2k, i+3k, ...$
`len(s)`	nombre d'éléments de `s`
`min(s)`	plus petit élément de `s`, résultat difficile à prévoir lorsque les types des éléments sont différents
`max(s)`	plus grand élément de `s`
`sum(s)`	retourne la somme de tous les éléments

Les tuples ne sont pas **modifiables** (ou **mutable**), cela signifie qu'il est impossible de modifier un de leurs éléments. Par conséquent, la ligne d'affectation suivante n'est pas correcte :

```
a     = (4,5)
a [0] = 3       # déclenche une erreur d'exécution
```

Le message d'erreur suivant apparaît :

```
File "<pyshell#78>", line 1, in -toplevel-
    a[0]=3
TypeError: object doesn't support item assignment
```

Pour changer cet élément, il est possible de s'y prendre de la manière suivante :

```
a = (4,5)
a = (3,) + a[1:2]  # crée un tuple d'un élément concaténé
                   # avec la partie inchangée de a
```

Nombres complexes

Il existe d'autres types comme le type complex [18] permettant de représenter les nombres complexes. Ce type numérique suit les mêmes règles et fonctionne avec les mêmes opérateurs (excepté les opérateurs de comparaisons) que ceux présentés au paragraphe type_nombre et décrivant les nombres.

<<<

```
print(complex(1, 1))
c = complex(1, 1)
print(c*c)
```

>>>

```
    (1+1j)
    2j
```

Le langage *python* offre la possibilité de créer ses propres types immuables (voir classe_slots_att) mais ils seront définis à partir des types immuables présentés jusqu'ici.

18. https://docs.python.org/3/library/functions.html#complex

bytes

Le type bytes [19] représente un tableau d'octets. Il fonctionne quasiment pareil que le type str [20]. Les opérations qu'on peut faire dessus sont quasiment identiques :

`count( sub[, start[, end]])`	Retourne le nombre d'occurences de la séquence d'octets `sub`, les paramètres par défaut `start` et `end` permettent de réduire la recherche entre les octets d'indice `start` et `end` exclu. Par défaut, `start` est nul tandis que `end` correspond à la fin de la séquence d'octets.
`find( sub[, start[, end]])`	Recherche une séquence d'octets `sub`, les paramètres par défaut `start` et `end` ont la même signification que ceux de la fonction `count`. Cette fonction retourne -1 si la recherche n'a pas abouti.
`replace( old, new[, count])`	Retourne une copie de la séquence d'octets en remplaçant toutes les occurrences de la séquence `old` par `new`. Si le paramètre optionnel `count` est renseigné, alors seules les `count` premières occurrences seront remplacées.
`partition( [sep [, maxsplit]])`	Découpe la séquence d'octets en se servant de la séquence `sep` comme délimiteur. Si le paramètre `maxsplit` est renseigné, au plus `maxsplit` coupures seront effectuées.
`join ( li )`	`li` est une liste, cette fonction agglutine tous les éléments d'une liste séparés par `sep` dans l'expression `sep.join ( ["un", "deux"])`.
`startswith(prefix[, start[, end]])`	Teste si la chaîne commence par `prefix`.
`endswith(suffix[, start[, end]])`	Teste si la chaîne se termine par `suffix`.
...	Lire Bytes and Bytearray Operations [21].

Pour déclarer un tableau de *bytes*, il faut préfixer une chaîne de caractères par **b** :

<<<

```
b = b"345"
print(b, type(b))

b = bytes.fromhex('2Ef0 F1f2  ')
print(b, type(b))
```

>>>

```
    b'345' <class 'bytes'>
    b'.\xf0\xf1\xf2' <class 'bytes'>
```

Le type *bytes* est très utilisé quand il s'agit de convertit une chaîne de caractères d'un encoding [22] à l'autre.

<<<

```
b = "abc".encode("utf-8")
s = b.decode("ascii")
print(b, s)
print(type(b), type(s))
```

>>>

19. https://docs.python.org/3/library/stdtypes.html#bytes
20. https://docs.python.org/3/library/stdtypes.html#string-methods
21. https://docs.python.org/3/library/stdtypes.html#bytes-methods
22. https://fr.wikipedia.org/wiki/Codage_des_caract%C3%A8res

```
    b'abc' abc
    <class 'bytes'> <class 'str'>
```

Les encoding [23] sont utiles dès qu'une chaîne de caractères contient un caractère non anglais (accent, sigle...). Les bytes sont aussi très utilisés pour sérialiser [24] un objet.

2.1.5 Types modifiables ou mutable

Les types modifiables sont des conteneurs (ou containers en anglais) : ils contiennent d'autres objets, que ce soit des nombres, des chaînes de caractères ou des objets de type modifiable. Plutôt que d'avoir dix variables pour désigner dix objets, on en n'utilise qu'une seule qui désigne ces dix objets.

Définition D6 : type modifiable (ou mutable)

Une variable de type modifiable peut être modifiée, elle conserve le même type et le même identificateur. C'est uniquement son contenu qui évolue.

On pourrait penser que les types modifiables sont plus pratiques à l'usage mais ce qu'on gagne en souplesse, on le perd en taille mémoire.

<<<

```
import sys
li = [3, 4, 5, 6, 7]
tu = (3, 4, 5, 6, 7)
print(sys.getsizeof(li), sys.getsizeof(tu))
```

>>>

```
    104 88
```

bytearray

Le type bytearray [25] est la version *mutable* du type *bytes* (page 16).

Liste

Définition et fonctions

Définition D7 : liste

Les listes sont des collections d'objets qui peuvent être de tout type. Elles sont modifiables.

Une liste apparaît comme une succession d'objets compris entre crochets et séparés par des virgules. Leur création reprend le même format :

23. https://fr.wikipedia.org/wiki/Codage_des_caract%C3%A8res
24. https://fr.wikipedia.org/wiki/S%C3%A9rialisation
25. https://docs.python.org/3/library/functions.html#bytearray

```
x = [4,5]                 # création d'une liste composée de deux entiers
x = ["un",1,"deux",2]     # création d'une liste composée de
                          # deux chaînes de caractères
                          # et de deux entiers, l'ordre d'écriture est important
x = [3,]                  # création d'une liste d'un élément, sans la virgule,
                          # le résultat reste une liste
x = [ ]                   # crée une liste vide
x = list ()               # crée une liste vide
y = x [0]                 # accède au premier élément
y = x [-1]                # accède au dernier élément
```

Ces objets sont des listes chaînées d'autres objets de type quelconque (immuable ou modifiable). Il est possible d'effectuer les opérations qui suivent. Ces opérations reprennent celles des *tuple* (page 14) (voir *opération tuple* (page 15)) et incluent d'autres fonctionnalités puisque les listes sont modifiables. Il est donc possible d'insérer, de supprimer des éléments, de les trier. La syntaxe des opérations sur les listes est similaire à celle des opérations qui s'appliquent sur les chaînes de caractères, elles sont présentées par la table suivante.

`x in l`	vrai si `x` est un des éléments de `l`
`x not in l`	réciproque de la ligne précédente
`l + t`	concaténation de `l` et `t`
`l * n`	concatène `n` copies de `l` les unes à la suite des autres
`l[i]`	retourne l'élément i$^text{ème}$ élément de `l`, à la différence des tuples, l'instruction `l [i] = "a"` est valide, elle remplace l'élément `i` par `"a"`. Un indice négatif correspond à la position `len(l)+i`.
`l[i:j]`	retourne une liste contenant les éléments de `l` d'indices *i* à *j* exclu. Il est possible de remplacer cette sous-liste par une autre en utilisant l'affectation `l[i:j] = l2` où `l2` est une autre liste (ou un tuple) de dimension différente ou égale.
`l[i:j:k]`	retourne une liste contenant les éléments de `l` dont les indices sont compris entre *i* et *j* exclu, ces indices sont espacés de *k* : $i, i+k, i+2k, i+3k, ...$ Ici encore, il est possible d'écrire l'affectation suivante : `l[i:j:k] = l2` mais `l2` doit être une liste (ou un tuple) de même dimension que `l[i:j:k]`.
`len(l)`	nombre d'éléments de `l`
`min(l)`	plus petit élément de `l`, résultat difficile à prévoir lorsque les types des éléments sont différents
`max(l)`	plus grand élément de `l`
`sum(l)`	retourne la somme de tous les éléments
`del l [i:j]`	supprime les éléments d'indices entre `i` et `j` exclu. Cette instruction est équivalente à `l [i:j] = []`.
`list (x)`	convertit `x` en une liste quand cela est possible
`l.count (x)`	Retourne le nombre d'occurrences de l'élément `x`. Cette notation suit la syntaxe des classes développée au chapitre *Classes* (page 95). `count` est une méthode de la classe `list`.
`l.index (x)`	Retourne l'indice de la première occurrence de l'élément `x` dans la liste `l`. Si celle-ci n'existe pas, une exception est déclenchée (voir le paragraphe *Exceptions* (page 155))
`l.append (x)`	Ajoute l'élément `x` à la fin de la liste `l`. Si `x` est une liste, cette fonction ajoute la liste `x` en tant qu'élément, au final, la liste `l` ne contiendra qu'un élément de plus.
`l.extend (k)`	Ajoute tous les éléments de la liste `k` à la liste `l`. La liste `l` aura autant d'éléments supplémentaires qu'il y en a dans la liste `k`.
`l.insert(i,x)`	Insère l'élément `x` à la position `i` dans la liste `l`.
`l.remove (x)`	Supprime la première occurrence de l'élément `x` dans la liste `l`. S'il n'y a aucune occurrence de `x`, cette méthode déclenche une exception.
`l.pop ([i])`	Retourne l'élément `l[i]` et le supprime de la liste. Le paramètre `i` est facultatif, s'il n'est pas précisé, c'est le dernier élément qui est retourné puis supprimé de la liste.
`l.reverse (x)`	Retourne la liste, le premier et dernier élément échange leurs places, le second et l'avant dernier, et ainsi de suite.
`l.sort ([key=None, reverse=False])`	Cette fonction trie la liste par ordre croissant. Le paramètre `key` est facultatif, il permet de préciser la fonction qui précise clé de comparaison qui doit être utilisée lors du tri. Si `reverse` est `True`, alors le tri est décroissant. Lire Sorting HOW TO [26].

26. https://docs.python.org/3/howto/sorting.html#sortinghowto

Exemples

L'exemple suivant montre une utilisation de la méthode `sort`.

<<<

```
x = [9, 0, 3, 5, 4, 7, 8]           # définition d'une liste
print(x)                          # affiche cette liste
x.sort()                          # trie la liste par ordre croissant
print(x)                          # affiche la liste triée
```

>>>

```
    [9, 0, 3, 5, 4, 7, 8]
    [0, 3, 4, 5, 7, 8, 9]
```

Pour classer les objets contenus par la liste mais selon un ordre différent, il faut définir une fonction qui détermine un ordre entre deux éléments de la liste. C'est la fonction `compare` de l'exemple suivant.

<<<

```
def compare_key(x):
    return -x


x = [9, 0, 3, 5, 4, 7, 8]
x.sort(key=compare_key)      # trie la liste x à l'aide de la fonction compare
# cela revient à la trier par ordre décroissant
print(x)
```

>>>

```
    [9, 8, 7, 5, 4, 3, 0]
```

L'exemple suivant illustre un exemple dans lequel on essaye d'accéder à l'indice d'un élément qui n'existe pas dans la liste :

```
x = [9,0,3,5,0]
print(x.index(1))    # cherche la position de l'élément 1
```

Comme cet élément n'existe pas, on déclenche ce qu'on appelle une exception qui se traduit par l'affichage d'un message d'erreur. Le message indique le nom de l'exception générée (`ValueError`) ainsi qu'un message d'information permettant en règle générale de connaître l'événement qui en est la cause.

```
Traceback (most recent call last):
  File "c:/temp/temp", line 2, in -toplevel-
    print(x.index(1))
ValueError: list.index(x): x not in list
```

Pour éviter cela, on choisit d'intercepter l'exception (voir paragraphe *Exceptions* (page 155)).

<<<

```
x = [9, 0, 3, 5, 0]
try:
    print(x.index(1))
except ValueError:
    print("1 n'est pas présent dans la liste x")
else:
    print("trouvé")
```

>>>

```
    1 n'est pas présent dans la liste x
```

Fonction range

Les listes sont souvent utilisées dans des boucles ou notamment par l'intermédiaire de la fonction range [27]. Cette fonction retourne un itérateur [28] sur des entiers. Nous verrons les itérateurs plus tard. Disons pour le moment les itérateurs ont l'apparence d'un ensembe mais ce n'en est pas un.

```
range (debut, fin [,marche])
```

Retourne une liste incluant tous les entiers compris entre `debut` et `fin` exclu. Si le paramètre facultatif `marche` est renseigné, la liste contient tous les entiers `n` compris `debut` et `fin` exclu et tels que `n - debut` soit un multiple de `marche`.

Exemple :

<<<

```
print(range(0, 10, 2))
print(list(range(0, 10, 2)))
```

>>>

```
    range(0, 10, 2)
    [0, 2, 4, 6, 8]
```

Cette fonction est souvent utilisée lors de boucles *for* (page 44) pour parcourir tous les éléments d'un tuple, d'une liste, d'un dictionnaire... Le programme suivant permet par exemple de calculer la somme de tous les entiers impairs compris entre 1 et 20 exclu.

```
s = 0
for n in range (1,20,2) :  # ce programme est équivalent à
    s += n                 # s = sum (range(1,20,2))
```

Le programme suivant permet d'afficher tous les éléments d'une liste.

<<<

```
x = ["un", 1, "deux", 2, "trois", 3]
for n in range(0, len(x)):
    print("x[%d] = %s" % (n, x[n]))
```

>>>

```
    x[0] = un
    x[1] = 1
    x[2] = deux
    x[3] = 2
    x[4] = trois
    x[5] = 3
```

27. https://docs.python.org/3/library/functions.html#func-range
28. https://fr.wikipedia.org/wiki/It%C3%A9rateur

Boucles et listes

Il est possible aussi de ne pas se servir des indices comme intermédiaires pour accéder aux éléments d'une liste quand il s'agit d'effectuer un même traitement pour tous les éléments de la liste `x`.

<<<

```
x = ["un", 1, "deux", 2]
for el in x:
    print("la liste inclut : ", el)
```

>>>

```
    la liste inclut :  un
    la liste inclut :  1
    la liste inclut :  deux
    la liste inclut :  2
```

L'instruction `for el in x :` se traduit littéralement par : *pour tous les éléments de la liste, faire....*

Il existe également des notations abrégées lorsqu'on cherche à construire une liste à partir d'une autre. Le programme suivant construit la liste des entiers de 1 à 5 à partir du résultat retourné par la fonction `range`.

<<<

```
y = list()
for i in range(0, 5):
    y.append(i+1)
print(y)
```

>>>

```
    [1, 2, 3, 4, 5]
```

Le langage *python* offre la possibilité de résumer cette écriture en une seule ligne. Cette syntaxe sera reprise au paragraphe *Listes, boucle for, liste en extension* (page 45).

<<<

```
y = [i+1 for i in range(0, 5)]
print(y)
```

>>>

```
    [1, 2, 3, 4, 5]
```

Cette définition de liste peut également inclure des tests ou des boucles imbriquées.

<<<

```
y = [i for i in range(0, 5) if i % 2 == 0]    # sélection les éléments pairs
print(y)                                      # affiche [0,2,4]
z = [i+j for i in range(0, 5)
     for j in range(0, 5)]      # construit tous les nombres i+j possibles
print(z)
```

>>>

```
    [0, 2, 4]
    [0, 1, 2, 3, 4, 1, 2, 3, 4, 5, 2, 3, 4, 5, 6, 3, 4, 5, 6, 7, 4, 5, 6, 7, 8]
```

Collage de séquences, fonction zip

On suppose qu'on dispose de *n* séquences d'éléments (tuple, liste), toutes de longueur *l*. La fonction zip [29] permet de construire une liste de tuples qui est la juxtaposition de toutes ces séquences. Le ième tuple de la liste résultante contiendra tous les ième éléments des séquences juxtaposées. Si les longueurs des séquences sont différentes, la liste résultante aura même taille que la plus courte des séquences.

<<<

```
a = (1, 0, 7, 0, 0, 0)
b = [2, 2, 3, 5, 5, 5]
c = ["un", "deux", "trois", "quatre"]
d = zip(a, b, c)
print(d)
```

>>>

```
    <zip object at 0x7f23d8898b08>
```

Concaténation de chaîne de caractères

Il arrive fréquemment de constuire une chaîne de caractères petits bouts par petits bouts comme le montre le premier exemple ci-dessous. Cette construction peut s'avérer très lente lorsque le résultat est long. Dans ce cas, il est nettement plus rapide d'ajouter chaque morceau dans une liste puis de les concaténer en une seule fois grâce à la méthode join [30]

```
s = ""
while <condition> : s += ...
```

```
s = []
while <condition> : s.append ( ... )
s = "".join (s)
```

Copie

A l'inverse des objets de type immuable, une affectation ne signifie pas une copie. Afin d'éviter certaines opérations superflues et parfois coûteuses en temps de traitement, on doit distinguer la variable de son contenu. Une variable désigne une liste avec un mot (ou identificateur), une affection permet de créer un second mot pour désigner la même liste. Il est alors équivalent de faire des opérations avec le premier mot ou le second comme le montre l'exemple suivant avec les listes `l` et `l2`.

<<<

```
l = [4, 5, 6]
l2 = l
print(l)            # affiche [4,5,6]
print(l2)           # affiche [4,5,6]
l2[1] = "modif"
print(l)            # affiche [4, 'modif', 6]
print(l2)           # affiche [4, 'modif', 6]
```

>>>

29. https://docs.python.org/3/library/functions.html#zip
30. https://docs.python.org/3/library/stdtypes.html#str.join

```
    [4, 5, 6]
    [4, 5, 6]
    [4, 'modif', 6]
    [4, 'modif', 6]
```

Dans cet exemple, il n'est pas utile de créer une seconde variable, dans le suivant, cela permet quelques raccourcis.

<<<

```
l = [[0, 1], [2, 3]]
l1 = l[0]
l1[0] = "modif"  # ligne équivalente à : l [0][0] = "modif"
print(l, l1)
```

>>>

```
    [['modif', 1], [2, 3]] ['modif', 1]
```

Par conséquent, lorsqu'on affecte une liste à une variable, celle-ci n'est pas recopiée, la liste reçoit seulement un nom de variable. L'affectation est en fait l'association d'un nom avec un objet (voir paragraphe *Copie d'instances* (page 123)). Pour copier une liste, il faut utiliser la fonction copy [31] du module copy [32]

<<<

```
import copy
l = [4, 5, 6]
l2 = copy.copy(l)
print(l)             # affiche [4,5,6]
print(l2)            # affiche [4,5,6]
l2[1] = "modif"
print(l)             # affiche [4,5,6]
print(l2)            # affiche [4, 'modif', 6]
```

>>>

```
    [4, 5, 6]
    [4, 5, 6]
    [4, 5, 6]
    [4, 'modif', 6]
```

Le module copy [33] est une extension interne. Cette syntaxe sera vue au chapitre *Modules* (page **??**). Ce point sera rappelé au paragraphe *Listes et dictionnaires* (page 128). L'opérateur == permet de savoir si deux listes sont égales même si l'une est une copie de l'autre. Le mot-clé is [34] permet de vérifier si deux variables font référence à la même liste ou si l'une est une copie de l'autre comme le montre l'exemple suivant :

<<<

```
import copy
l = [1, 2, 3]
l2 = copy.copy(l)
l3 = l

print(l == l2)  # affiche True
print(l is l2)  # affiche False
print(l is l3)  # affiche True
```

31. https://docs.python.org/3/library/copy.html?highlight=copy#copy.copy
32. https://docs.python.org/3/library/copy.html
33. https://docs.python.org/3/library/copy.html
34. https://docs.python.org/3/library/stdtypes.html#comparisons

>>>

```
    True
    False
    True
```

Fonction ‘‘copy‘‘ et ‘‘deepcopy‘‘

Le comportement de la fonction copy [35] peut surprendre dans le cas où une liste contient d'autres listes. Pour être sûr que chaque élément d'une liste a été correctement recopiée, il faut utiliser la fonction deepcopy [36]. La fonction est plus longue mais elle recopie toutes les listes que ce soit une liste incluse dans une liste elle-même incluse dans une autre liste elle-même incluse...

<<<

```
import copy
l = [[1, 2, 3], [4, 5, 6]]
l2 = copy.copy(l)
l3 = copy.deepcopy(l)
l[0][0] = 1111
print(l)                # affiche [[1111, 2, 3], [4, 5, 6]]
print(l2)               # affiche [[1111, 2, 3], [4, 5, 6]]
print(l3)               # affiche [[1, 2, 3], [4, 5, 6]]
print(l is l2)          # affiche False
print(l[0] is l2[0])  # affiche True
print(l[0] is l3[0])  # affiche False
```

>>>

```
    [[1111, 2, 3], [4, 5, 6]]
    [[1111, 2, 3], [4, 5, 6]]
    [[1, 2, 3], [4, 5, 6]]
    False
    True
    False
```

La fonction deepcopy [37] est plus lente à exécuter car elle prend en compte les références récursives comme celles de l'exemple suivant où deux listes se contiennent l'une l'autre.

<<<

```
l = [1, "a"]
ll = [l, 3]    # ll contient l
l[0] = ll       # l contient ll
print(l)          # affiche [[[...], 3], 'a']
print(ll)         # affiche [[[...], 'a'], 3]

import copy
z = copy.deepcopy(l)
print(z)          # affiche [[[...], 3], 'a']
```

>>>

```
    [[[...], 3], 'a']
    [[[...], 'a'], 3]
    [[[...], 3], 'a']
```

35. https://docs.python.org/3/library/copy.html?highlight=copy#copy.copy
36. https://docs.python.org/3/library/copy.html?highlight=copy#copy.deepcopy
37. https://docs.python.org/3/library/copy.html?highlight=copy#copy.deepcopy

Dictionnaire

Les dictionnaires sont des tableaux plus souples que les listes. Une liste référence les éléments en leur donnant une position : la liste associe à chaque élément une position entière comprise entre 0 et $n-1$ si n est la longueur de la liste. Un dictionnaire permet d'associer à un élément autre chose qu'une position entière : ce peut être un entier, un réel, une chaîne de caractères, un tuple contenant des objets immuables. D'une manière générale, un dictionnaire associe à une valeur ce qu'on appelle une clé de type immuable. Cette clé permettra de retrouver la valeur associée.

L'avantage principal des dictionnaires est la recherche optimisée des clés. Par exemple, on recense les noms des employés d'une entreprise dans une liste. On souhaite ensuite savoir si une personne ayant un nom précisé à l'avance appartient à cette liste. Il faudra alors parcourir la liste jusqu'à trouver ce nom ou parcourir toute la liste si jamais celui-ci ne s'y trouve pas. Dans le cas d'un dictionnaire, cette recherche du nom sera beaucoup plus rapide à écrire et à exécuter.

Définition et fonctions

Définition D8 : dictionnaire

Les dictionnaires sont des listes de couples. Chaque couple contient une clé et une valeur. Chaque valeur est indicée par sa clé. La valeur peut-être de tout type, la clé doit être de type immuable, ce ne peut donc être ni une liste, ni un dictionnaire. Chaque clé comme chaque valeur peut avoir un type différent des autres clés ou valeurs.

Un dictionnaire apparaît comme une succession de couples d'objets comprise entre accolades et séparés par des virgules. La clé et sa valeur sont séparées par le symbole `:`. Leur création reprend le même format :

```
x = { "cle1":"valeur1", "cle2":"valeur2" }
y = { }          # crée un dictionnaire vide
z = dict()       # crée aussi un dictionnaire vide
```

Les indices ne sont plus entiers mais des chaînes de caractères pour cet exemple. Pour associer la valeur à la clé ńăcle1ă", il suffit d'écrire :

<<<

```
x = {"cle1": "valeur1", "cle2": "valeur2"}
print(x["cle1"])
```

>>>

```
    valeur1
```

La plupart des fonctions disponibles pour les listes sont interdites pour les dictionnaires comme la concaténation ou l'opération de multiplication ($\star$). Il n'existe plus non plus d'indices entiers pour repérer les éléments, le seul repère est leur clé. La table suivante dresse la liste des opérations sur les dictionnaires.

`x in d`	vrai si `x` est une des clés de `d`
`x not in d`	réciproque de la ligne précédente
`d[i]`	retourne l'élément associé à la clé `i`
`len(d)`	nombre d'éléments de `d`
`min(d)`	plus petite clé
`max(d)`	plus grande clé
`del d [i]`	supprime l'élément associé à la clé `i`
`list (d)`	retourne une liste contenant toutes les clés du dictionnaire `d`
`dict (x)`	convertit `x` en un dictionnaire si cela est possible, `d` est alors égal à `dict ( d.items () )`
`d.copy ()`	Retourne une copie de `d`
`d.items ()`	Retourne un itérateur sur tous les couples (clé, valeur) inclus dans le dictionnaire.
`d.keys ()`	Retourne un itérateur sur toutes les clés du dictionnaire `d`
`d.values ()`	Retourne un itérateur sur toutes les valeurs du dictionnaire `d`
`d.get (k[,x])`	Retourne `d[k]`, si la clé `k` est manquante, alors la valeur `None` est retournée à moins que le paramètre optionnel `x` soit renseigné, auquel cas, ce sera cette valeur qui sera retourné.
`d.clear ()`	Supprime tous les éléments du dictionnaire.
`d.update(d2)`	Le dictionnaire `d` reçoit le contenu de `d2`.
`d.setdefault(k[,x])`	Définit `d[k]` si la clé `k` existe, sinon, affecte `x` à `d[k]`
`d.pop()`	Retourne un élément et le supprime du dictionnaire.

Contrairement à une liste, un dictionnaire ne peut être trié car sa structure interne est optimisée pour effectuer des recherches rapides parmi les éléments. On peut aussi se demander quel est l'intérêt de la méthode pop [38] qui retourne un élément puis le supprime alors qu'il existe le mot-clé `del`. Cette méthode est simplement plus rapide car elle choisit à chaque fois l'élément le moins coûteux à supprimer, surtout lorsque le dictionnaire est volumineux.

Les itérateurs sont des objets qui permettent de parcourir rapidement un dictionnaire, ils seront décrits en détail au chapitre *Classes* (page 95) sur les classes. Un exemple de leur utilisation est présenté dans le paragraphe suivant.

Exemples

Il n'est pas possible de trier un dictionnaire. L'exemple suivant permet néanmoins d'afficher tous les éléments d'un dictionnaire selon un ordre croissant des clés. Ces exemples font appel aux paragraphes sur les boucles (voir chapitre *Syntaxe du langage Python (boucles, tests, fonctions)* (page 34)).

<<<

```
d = {"un": 1, "zéro": 0, "deux": 2, "trois": 3, "quatre": 4, "cinq": 5,
     "six": 6, "sept": 1, "huit": 8, "neuf": 9, "dix": 10}
key = list(d.keys())
key.sort()
for k in key:
    print(k, d[k])
```

>>>

```
    cinq 5
    deux 2
    dix 10
    huit 8
```

(suite sur la page suivante)

38. https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types

(suite de la page précédente)

```
    neuf 9
    quatre 4
    sept 1
    six 6
    trois 3
    un 1
    zéro 0
```

L'exemple suivant montre un exemple d'utilisation des itérateurs. Il s'agit de construire un dictionnaire inversé pour lequel les valeurs seront les clés et réciproquement.

<<<

```
d = {"un": 1,   "zero": 0, "deux": 2, "trois": 3, "quatre": 4, "cinq": 5,
     "six": 6,  "sept": 1, "huit": 8, "neuf": 9,  "dix": 10}

dinv = {}                          # création d'un dictionnaire vide, on parcout
for key, value in d.items():        # les éléments du dictionnaire comme si
                                    # c'était une liste de 2-uple (clé,valeur)
    dinv[value] = key               # on retourne le dictionnaire

print(dinv)                         # affiche {0: 'zero', 1: 'un', 2: 'deux',
# 3: 'trois', 4: 'quatre', 5: 'cinq', 6: 'six',
# 8: 'huit', 9: 'neuf', 10: 'dix'}
```

>>>

```
    {1: 'sept', 0: 'zero', 2: 'deux', 3: 'trois', 4: 'quatre', 5: 'cinq', 6: 'six',␣
↪8: 'huit', 9: 'neuf', 10: 'dix'}
```

La méthode `items` retourne un itérateur. Cela permet de parcourir les éléments du dicionnaire sans créer de liste intermédiaire. Ceci explique ce qu'affiche le programme suivant :

<<<

```
d = {"un": 1,   "zero": 0, "deux": 2, "trois": 3, "quatre": 4, "cinq": 5,
     "six": 6,  "sept": 1, "huit": 8, "neuf": 9,  "dix": 10}
print(d.items())
print(list(d.items()))
```

>>>

```
    dict_items([('un', 1), ('zero', 0), ('deux', 2), ('trois', 3), ('quatre', 4), (
↪'cinq', 5), ('six', 6), ('sept', 1), ('huit', 8), ('neuf', 9), ('dix', 10)])
    [('un', 1), ('zero', 0), ('deux', 2), ('trois', 3), ('quatre', 4), ('cinq', 5), (
↪'six', 6), ('sept', 1), ('huit', 8), ('neuf', 9), ('dix', 10)]
```

D'une manière générale, il faut éviter d'ajouter ou de supprimer un élément dans une liste ou un dictionnaire qu'on est en train de parcourir au sein d'une boucle `for` ou `while`. Cela peut marcher mais cela peut aussi aboutir à des résultats imprévisibles surtout avec l'utilisation d'itérateurs (fonction `items`, `values`, `keys`). Il est préférable de terminer le parcours de la liste ou du dictionnaire puis de faire les modifications désirées une fois la boucle terminée. Dans le meilleur des cas, l'erreur suivante survient :

```
  File "essai.py", line 6, in <module>
    for k in d :
RuntimeError: dictionary changed size during iteration
```

Copie

A l'instar des listes (voir paragraphe *Copie* (page 23)), les dictionnaires sont des objets et une affectation n'est pas équivalente à une copie comme le montre le programme suivant.

<<<

```
d = {4: 4, 5: 5, 6: 6}
d2 = d
print(d)              # affiche {4: 4, 5: 5, 6: 6}
print(d2)             # affiche {4: 4, 5: 5, 6: 6}
d2[5] = "modif"
print(d)              # affiche {4: 4, 5: 'modif', 6: 6}
print(d2)             # affiche {4: 4, 5: 'modif', 6: 6}
```

>>>

```
    {4: 4, 5: 5, 6: 6}
    {4: 4, 5: 5, 6: 6}
    {4: 4, 5: 'modif', 6: 6}
    {4: 4, 5: 'modif', 6: 6}
```

Lorsqu'on affecte un dictionnaire à une variable, celui-ci n'est pas recopié, le dictionnaire reçoit seulement un nom de variable. L'affectation est en fait l'association d'un nom avec un objet (voir paragraphe *Copie d'instances* (page 123)). Pour copier un dictionnaire, on peut utiliser la méthode copy [39].

<<<

```
d = {4: 4, 5: 5, 6: 6}
d2 = d.copy()
print(d)              # affiche {4: 4, 5: 5, 6: 6}
print(d2)             # affiche {4: 4, 5: 5, 6: 6}
d2[5] = "modif"
print(d)              # affiche {4: 4, 5: 5, 6: 6}
print(d2)             # affiche {4: 4, 5: 'modif', 6: 6}
```

>>>

```
    {4: 4, 5: 5, 6: 6}
    {4: 4, 5: 5, 6: 6}
    {4: 4, 5: 5, 6: 6}
    {4: 4, 5: 'modif', 6: 6}
```

Le mot-clé `is` a la même signification pour les dictionnaires que pour les listes, l'exemple du paragraphe *Copie* (page 23) est aussi valable pour les dictionnaires. Il en est de même pour la remarque concernant la fonction deepcopy [40]. Cette fonction recopie les listes et les dictionnaires.

Clés de type modifiable

Ce paragraphe concerne davantage des utilisateurs avertis qui souhaitent malgré tout utiliser des clés de type modifiable. Dans l'exemple qui suit, la clé d'un dictionnaire est également un dictionnaire et cela provoque une erreur. Il en serait de même si la variable `k` utilisée comme clé était une liste.

```
k = { 1:1}
d = { }
d [k] = 0
```

39. https://docs.python.org/3/library/stdtypes.html?highlight=copy#dict.copy
40. https://docs.python.org/3/library/copy.html?highlight=copy#copy.deepcopy

La sortie :

```
Traceback (most recent call last):
  File "cledict.py", line 3, in <module>
    d [k] = 0
TypeError: dict objects are unhashable
```

Cela ne veut pas dire qu'il faille renoncer à utiliser un dictionnaire ou une liste comme clé. La fonction id [41] permet d'obtenir un entier qui identifie de manière unique tout objet. Le code suivant est parfaitement correct.

```
k = { 1:1}
d = { }
d [id (k)] = 0
```

Toutefois, ce n'est pas parce que deux dictionnaires auront des contenus identiques que leurs identifiants retournés par la fonction `id` seront égaux. C'est ce qui explique l'erreur que provoque la dernière ligne du programme suivant.

```
k = {1:1}
d = { }
d [id (k)] = 0
b = k
print(d[id(b)])  # affiche bien zéro
c = {1:1}
print(d[id(c)])  # provoque une erreur car même si k et c ont des contenus égaux,
                 # ils sont distincts, la clé id(c) n'existe pas dans d
```

Il existe un cas où on peut se passer de la fonction `id` mais il inclut la notion de classe définie au chapitre *Classes* (page 95). L'exemple suivant utilise directement l'instance `k` comme clé. En affichant le dictionnaire `d`, on vérifie que la clé est liée au résultat de l'instruction `id(k)` même si ce n'est pas la clé.

```
class A : pass

k = A ()
d = { }
d [k] = 0
print(d)                    # affiche {<__main__.A object at 0x0120DB90>: 0}
print(id(k), hex(id(k)))    # affiche 18930576, 0x120db90
print(d[id(k)])             # provoque une erreur
```

La fonction hex [42] convertit un entier en notation hexadécimale. Les nombres affichés changent à chaque exécution. Pour conclure, ce dernier exemple montre comment se passer de la fonction `id` dans le cas d'une clé de type dictionnaire.

<<<

```
class A (dict):
    def __hash__(self):
        return id(self)


k = A()
k["t"] = 4
d = {}
d[k] = 0
print(d)          # affiche {{'t': 4}: 0}
```

>>>

41. https://docs.python.org/3/library/functions.html?highlight=id#id
42. https://docs.python.org/3/library/functions.html?highlight=id#hex

```
    {{'t': 4}: 0}
```

Ensemble ou set

Le langage *python* définit également ce qu'on appelle un ensemble. Il est défini par les classes set [43] de type modifiable et la classe frozenset [44] de type immuable. Ils n'acceptent que des types identiques et offrent la plupart des opérations liées aux ensembles comme l'intersection, l'union.

<<<

```
print(set((1, 2, 3)) & set((2, 3, 5)))
# construit l'intersection qui est set([2, 3])
```

>>>

```
    {2, 3}
```

A faire : Compléter le paragraphe sur les set

set, frozen set

2.1.6 Extensions

Fonction `print`, `repr` et conversion en chaîne de caractères

La fonction `print` est déjà apparue dans les exemples présentés ci-dessus, elle permet d'afficher une ou plusieurs variables préalablement définies, séparées par des virgules. Les paragraphes qui suivent donnent quelques exemples d'utilisation. La fonction print [45] permet d'afficher n'importe quelle variable ou objet à l'écran, cet affichage suppose la conversion de cette variable ou objet en une chaîne de caractères. Deux fonctions permettent d'effectuer cette étape sans toutefois afficher le résultat à l'écran.

Point à retenir : la fonction `print` ne change pas le programme, elle affiche à l'écran le résultat d'une variable sans la modifier. Cela revient à écouter un programme avec un stéthoscope pour comprendre comment il fonctionne sans altérer son fonctionnement.

La fonction str [46] (voir paragraphe *Manipulation d'une chaîne de caractères* (page 9) permet de convertir toute variable en chaîne de caractères. Il existe cependant une autre fonction repr [47] qui effectue cette conversion. Dans ce cas, le résultat peut être interprété par la fonction eval [48] qui se charge de la conversion inverse. Pour les types simples comme ceux présentés dans ce chapitre, ces deux fonctions retournent des résultats identiques. Pour l'exemple, `x` désigne n'importe quelle variable.

```
x == eval (repr(x)) # est toujours vrai (True)
x == eval (str (x)) # n'est pas toujours vrai
```

43. https://docs.python.org/3/library/stdtypes.html#set
44. https://docs.python.org/3/library/stdtypes.html#frozenset
45. https://docs.python.org/3/library/functions.html?highlight=id#print
46. https://docs.python.org/3/library/functions.html?highlight=id#func-str
47. https://docs.python.org/3/library/functions.html?highlight=id#repr
48. https://docs.python.org/3/library/functions.html?highlight=id#eval

Fonction `eval`

Comme le suggère le paragraphe précédent, la fonction eval [49] permet d'évaluer une chaîne de caractères ou plutôt de l'interpréter comme si c'était une instruction en *python*. Le petit exemple suivant permet de tester toutes les opérations de calcul possibles entre deux entiers.

<<<

```
x = 32
y = 9
op = "+ - * / % // & | and or << >>".split()
for o in op:
    s = str(x) + " " + o + "  " + str(y)
    print(s, " = ", eval(s))
```

>>>

```
    32 +  9  =  41
    32 -  9  =  23
    32 *  9  =  288
    32 /  9  =  3.5555555555555554
    32 %  9  =  5
    32 //  9  =  3
    32 &  9  =  0
    32 |  9  =  41
    32 and  9  =  9
    32 or  9  =  32
    32 <<  9  =  16384
    32 >>  9  =  0
```

Le programme va créer une chaîne de caractères pour chacune des opérations et celle-ci sera évaluée grâce à la fonction eval [50] comme si c'était une expression numérique. Il faut bien sûr que les variables que l'expression mentionne existent durant son évaluation.

Informations fournies par *python*

Bien que les fonctions ne soient définies que plus tard (paragraphe *Fonctions* (page 53), il peut être intéressant de mentionner la fonction dir [51] qui retourne la liste de toutes les variables créées et accessibles à cet instant du programme. L'exemple suivant :

<<<

```
x = 3
print(dir())
```

>>>

```
    ['__WD__', 'x']
```

Certaines variables - des chaînes des caractères - existent déjà avant même la première instruction. Elles contiennent différentes informations concernant l'environnement dans lequel est exécuté le programme *python* :

49. https://docs.python.org/3/library/functions.html?highlight=id#eval
50. https://docs.python.org/3/library/functions.html?highlight=id#eval
51. https://docs.python.org/3/library/functions.html?highlight=id#dir

`__builtins__`	Ce module contient tous les éléments présents dès le début d'un programme *python*, il contient entre autres les types présentés dans ce chapitre et des fonctions simples comme `range`.
`__doc__`	C'est une chaîne commentant le fichier, c'est une chaîne de caractères insérée aux premières lignes du fichiers et souvent entourée des symboles `"""` (voir chapitre *Modules* (page **??**)).
`__file__`	Contient le nom du fichier où est écrit ce programme.
`__name__`	Contient le nom du module.

La fonction dir [52] est également pratique pour afficher toutes les fonctions d'un module. L'instruction `dir(sys)` affiche la liste des fonctions du module sys [53] (voir chapitre *Modules* (page **??**)).

La fonction `dir()` appelée sans argument donne la liste des fonctions et variables définies à cet endroit du programme. Ce résultat peut varier selon qu'on se trouver dans une fonction, une méthode de classe ou à l'extérieur du programme. L'instruction `dir([])` donne la liste des méthodes qui s'appliquent à une liste.

De la même manière, la fonction type [54] retourne une information concernant le type d'une variable.

<<<

```
x = 3
print(x, type(x))      # affiche 3 <type 'int'>
x = 3.5
print(x, type(x))      # affiche 3.5 <type 'float'>
```

>>>

```
    3 <class 'int'>
    3.5 <class 'float'>
```

Affectations multiples

Il est possible d'effectuer en *python* plusieurs affectations simultanément.

```
x = 5        # affecte 5 à x
y = 6        # affecte 6 à y
x,y = 5,6    # affecte en une seule instruction 5 à x et 6 à y
```

Cette particularité reviendra lorsque les fonctions seront décrites puisqu'il est possible qu'une fonction retourne plusieurs résultats comme la fonction divmod [55] illustrée par le programme suivant.

<<<

```
x, y = divmod(17, 5)
print(x, y)                          # affiche 3 2
print("17 / 5 = 5 * ", x, " + ", y)  # affiche 17 / 5 = 5 *  3  +  2
```

>>>

```
    3 2
    17 / 5 = 5 *  3  +  2
```

Le langage *python* offre la possibilité d'effectuer plusieurs affectations sur la même ligne. Dans l'exemple qui suit, le couple $(5, 5)$ est affecté à la variable `point`, puis le couple `x, y` reçoit les deux valeurs du tuple `point`.

52. https://docs.python.org/3/library/functions.html?highlight=id#dir
53. https://docs.python.org/3/library/sys.html?highlight=sys#module-sys
54. https://docs.python.org/3/library/functions.html?highlight=id#type
55. https://docs.python.org/3/library/functions.html?highlight=divmod#divmod

```
x,y = point = 5,5
```

Hiérarchie des objets

La page modèle de données[56] décrit les différentes catégories d'objets du langage. Des objets de la même classe propose des fonctionnalités similaires.

Objets internes

Les objets objet internes[57] sont à peu près tout ce qui n'existe pas dans un langage compilé. Elles sont propres au langage et laisse transparaître des informations dont l'interpréteur a besoin pour comprendre le programme. Il est déconseillé de s'en servir si jamais on souhaite un jour traduire le même code dans un autre langage.

2.1.7 Commentaires accentués

Les commentaires commencent par le symbole # et se terminent par la fin de la ligne ; ils ne sont pas interprétés, ils n'ont aucune influence sur l'exécution du programme. Lorsque les commentaires incluent des symboles exclusivement français tels que les accents, le compilateur génère l'erreur suivante :

```
SyntaxError: Non-UTF-8 code starting with '\xe9' in file f.py on line 1,
but no encoding declared; see http://python.org/dev/peps/pep-0263/
for details
```

Il est néanmoins possible d'utiliser des accents dans les commentaires à condition d'insérer le commentaire suivant à la première ligne du programme. Il n'est pas nécessaire de retenir cette commande si le programme est écrit dans l'éditeur de texte fourni avec *python* car ce dernier propose automatiquement d'insérer cette ligne. Ce point est abordé au paragraphe par_intro_accent_code. Il faut inclure la placer le texte suivant en première ligne :

```
# -*- coding: utf-8 -*-
```

Ou :

```
# coding: cp1252
```

Ou encore :

```
# coding: latin-1
```

Le premier encoding `utf-8` est le plus communément utilisé dans le monde internet. Le second est utilisé par Windows.

2.2 Syntaxe du langage Python (boucles, tests, fonctions)

56. https://docs.python.org/3/reference/datamodel.html#types
57. https://docs.python.org/3/library/stdtypes.html#internal-objects

Avec les variables, les boucles et les fonctions, on connaît suffisamment d'éléments pour écrire des programmes. Le plus difficile n'est pas forcément de les comprendre mais plutôt d'arriver à découper un algorithme complexe en utilisant ces briques élémentaires. C'est l'objectif des chapitres centrés autour des exercices. Toutefois, même si ce chapitre présente les composants élémentaires du langage, l'aisance qu'on peut acquérir en programmation vient à la fois de la connaissance du langage mais aussi de la connaissance d'algorithmes standards comme celui du tri ou d'une recherche dichotomique. C'est cette connaissance tout autant que la maîtrise d'un langage de programmation qui constitue l'expérience en programmation.

2.2.1 Les trois concepts des algorithmes

Les algorithmes sont composés d'instructions, ce sont des opérations élémentaires que le processeur exécute selon trois schémas :

la séquence	enchaînement des instructions les unes à la suite des autres : passage d'une instruction à la suivante
le saut	passage d'une instruction à une autre qui n'est pas forcément la suivante (c'est une rupture de séquence)
le test	choix entre deux instructions

Le saut n'apparaît plus de manière explicite dans les langages évolués car il est une source fréquente d'erreurs. Il intervient dorénavant de manière implicite au travers des boucles qui combinent un saut et un test. On écrit toujours ceci avec les langages les plus récents :

initialisation de la variable moy à 0
faire pour i allant de 1 à N
    moy reçoit moy + ni
moy reçoit moy / N

Et ceci est traduit par :

ligne 1 : initialisation de la variable moy à 0
ligne 2 : initialisation de la variable i à 1
ligne 3 : moy reçoit moy + ni
ligne 4 : i reçoit i + 1
ligne 5 : si i est inférieur ou égal à N alors aller à la ligne 3
ligne 6 : moy reçoit moy / N

Tout programme peut se résumer à ces trois concepts. Chaque langage les met en place avec sa propre syntaxe et parfois quelques nuances mais il est souvent facile de les reconnaître même dans des langages inconnus. Le calcul d'une somme décrit plus haut et écrit en *python* correspond à l'exemple suivant :

<<<

```
t = [0, 1, 2, 3, 4]
N = 5
moy = 0
for i in range(1, N+1):       # de 1 à N+1 exclu --> de 1 à N inclus
    moy += t[i-1]            # le premier indice est 0 et non 1
moy /= N
print(moy)
```

>>>

```
    2.0
```

Le premier élément de cette syntaxe est constituée de ses mots-clés `for` et `in` et des symboles =, +=, /=, `[`, `]`, `(`, `)`, `:`. La fonction iskeyword[58] permet de savoir si un mot-clé donné fait partie du langage *python*. Même si les modules seront décrits plus tard, la syntaxe suivante reste accessible :

<<<

```
import keyword
print(keyword.iskeyword("for"))      # affiche True
print(keyword.iskeyword("until"))    # affiche False
```

>>>

```
    True
    False
```

58. https://docs.python.org/3/library/keyword.html#keyword.iskeyword

Le programme suivant permet de récupérer la liste des mots-clés [59] du langage :

<<<

```
import keyword
print("\n".join(keyword.kwlist))
```

>>>

```
False
None
True
and
as
assert
async
await
break
class
continue
def
del
elif
else
except
finally
for
from
global
if
import
in
is
lambda
nonlocal
not
or
pass
raise
return
try
while
with
yield
```

A cela s'ajoutent les symboles :

```
+   -   *   **   /   //   %
< > == <= >=   !=
<<   >>   &   |   \   ~   ^
=   +=   -=   *=   /=   //=   %=   **=
|=   &=   <<=   >>=   ~=   ^=
(   )   [   ]   {   }
"   """   '   '''
,   :   .
#
@   @=
```

59. https://docs.python.org/3/reference/lexical_analysis.html#keywords

Les espaces entre les mots-clés et les symboles ont peu d'importance, il peut n'y en avoir aucun comme dix. Les espaces servent à séparer un mot-clé, un nombre d'une variable. Les mots-clés et les symboles définissent la grammaire du langage *python*. Toutes ces règles sont décrites dans un langage un peu particuilier par la page Full Grammar specification [60].

Les fonctions builtin [61] ne font pas partie de la grammaire du langage même si elles sont directement accessibles comme la fonction abs [62] qui retourne la valeur absolue d'un nombre. C'est un choix d'implémentation du programme qui interprète le langage mais absent de la grammaire.

2.2.2 Tests

Définition et syntaxe

Définition D1 : test

Les tests permettent d'exécuter des instructions différentes selon la valeur d'une condition logique.

Syntaxe :

Syntaxe S1 : Tests

```
if condition1 :
   instruction1
   instruction2
   ...
else :
   instruction3
   instruction4
   ...
```

La clause `else` est facultative. Lorsque la condition `condition1` est fausse et qu'il n'y a aucune instruction à exécuter dans ce cas, la clause `else` est inutile. La syntaxe du test devient :

```
if condition1 :
   instruction1
   instruction2
   ...
```

S'il est nécessaire d'enchaîner plusieurs tests d'affilée, il est possible de condenser l'écriture avec le mot-clé `elif` :

```
if condition1 :
   instruction1
   instruction2
   ...
elif condition2 :
   instruction3
   instruction4
   ...
elif condition3 :
   instruction5
   instruction6
```

(suite sur la page suivante)

60. https://docs.python.org/3/reference/grammar.html
61. https://docs.python.org/3/library/functions.html#built-in-functions
62. https://docs.python.org/3/library/functions.html#abs

(suite de la page précédente)

```
    ...
else :
    instruction7
    instruction8
    ...
```

Le décalage des instructions par rapport aux lignes contenant les mots-clés `if`, `elif`, `else` est très important : il fait partie de la syntaxe du langage et s'appelle l¿indentation [63]. Celle-ci permet de grouper les instructions ensemble. Le programme suivant est syntaxiquement correct même si le résultat n'est pas celui désiré.

<<<

```
x = 1
if x > 0:
    signe = 1
    print("le nombre est positif")
else:
    signe = -1
print("le nombre est négatif")  # ligne mal indentée (au sens de l'algorithme)
print("signe = ", signe)
```

>>>

```
    le nombre est positif
    le nombre est négatif
    signe =  1
```

Une ligne est mal indentée : `print("le nombre est négatif")`. Elle ne devrait être exécutée que si la condition `x>0` n'est pas vérifiée. Le fait qu'elle soit alignée avec les premières instructions du programme fait que son exécution n'a plus rien à voir avec cette condition. La programme répond de manière erronée.

Dans certains cas, l'interpréteur *python* ne sait pas à quel bloc attacher une instruction, c'est le cas de l'exemple suivant, la même ligne a été décalée de deux espaces, ce qui est différent de la ligne qui précède et de la ligne qui suit.

```
x = 1
if x > 0:
    signe = 1
    print("le nombre est positif")
else:
    signe = -1
  print("le nombre est négatif")     # ligne mal indentée (au sens de la grammaire)
print("signe = ", signe)
```

L'interpréteur retourne l'erreur suivante :

```
File "test.py", line 7
    print("le nombre est négatif")
                                  ^
IndentationError: unindent does not match any outer indentation level
```

Comparaisons possibles

Les comparaisons possibles entre deux entités sont avant tout numériques mais ces opérateurs peuvent être définis pour tout type (voir *Classes* (page 95)), notamment sur les chaînes de caractères pour lesquelles les opérateurs de comparaison transcrivent l'ordre alphabétique.

63. https://fr.wikipedia.org/wiki/Style_d%27indentation

<, >	inférieur, supérieur
<=, >=	inférieur ou égal, supérieur ou égal
==, !=	égal, différent
is, not is	x is y vérifie que x et y sont égaux, not is, différents, l'opérateur is est différent de l'opérateur ==, il est expliqué au paragraphe *Copie* (page 23)
in, not in	appartient, n'appartient pas

Opérateurs logiques

Il existe trois opérateurs logiques qui combinent entre eux les conditions.

not	négation
and	et logique
or	ou logique

La priorité des opérations numériques est identique à celle rencontrée en mathématiques. L'opérateur puissance vient en premier, la multiplication/division ensuite puis l'addition/soustraction. Ces opérations sont prioritaires sur les opérateurs de comparaisons (>, <, ==, ...) qui sont eux-mêmes sur les opérateurs logiques not, and, or. Il est tout de même conseillé d'ajouter des parenthèses en cas de doute. C'est ce qu décrit la page Operator precedence [64].

Ecriture condensée

Il existe deux écritures condensées de tests. La première consiste à écrire un test et l'unique instruction qui en dépend sur une seule ligne.

if condition : instruction1

else : instruction2

Ce code peut tenir en deux lignes :

```
if condition : instruction1
else : instruction2
```

Le second cas d'écriture condensée concerne les comparaisons enchaînées. Le test `if 3 < x and x < 5 : instruction` peut être condensé par `if 3 < x < 5 : instruction`. Il est ainsi possible de juxtaposer autant de comparaisons que nécessaire : `if 3 < x < y < 5 : instruction`.

Le mot-clé `in` permet également de condenser certains tests lorsque la variable à tester est entière. `if x == 1 or x == 6 or x == 50 :` peut être résumé simplement par `if x in (1,6,50) :` ou `if x in {1,6,50}:` pour les grandes listes.

Exemple

L'exemple suivant associe à la variable `signe` le signe de la variable `x`.

<<<

```
x = -5
if x < 0:
    signe = -1
elif x == 0:
    signe = 0
else:
```

(suite sur la page suivante)

64. https://docs.python.org/3/reference/expressions.html#operator-precedence

(suite de la page précédente)

```
    signe = 1
print(signe)
```

>>>

```
    -1
```

Son écriture condensée lorsqu'il n'y a qu'une instruction à exécuter :

<<<

```
x = -5
if x < 0:
    signe = -1
elif x == 0:
    signe = 0
else:
    signe = 1
print(signe)
```

>>>

```
    -1
```

Le programme suivant saisit une ligne au clavier et dit si c'est ńăouiă" ou ńănonă" qui a été saisi. La fonction input [65] retourne ce qui vient de l'utilisateur :

```
s = input ("dites oui : ")     # voir remarque suivante
if s == "oui" or s [0:1] == "o" or s [0:1] == "O" or s == "1" :
    print "oui"
else:
    print "non"
```

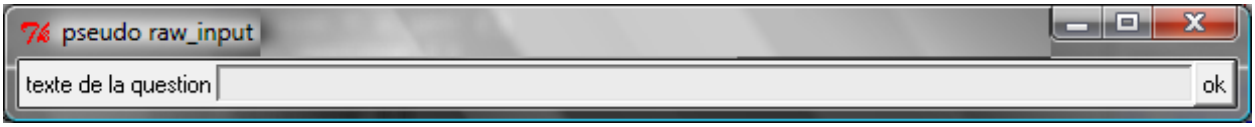
La fonction input [66] invite l'utilisateur d'un programme à saisir une réponse lors de l'exécution du programme. Tant que la touche entrée n'a pas été pressée, l'exécution du programme ne peut continuer. Cette fonction est en réalité peu utilisée. Les interfaces graphiques sont faciles d'accès en *python*, on préfère donc saisir une réponse via une fenêtre plutôt qu'en ligne de commande. L'exemple suivant montre comment remplacer cette fonction à l'aide d'une fenêtre graphique.

```
import tkinter
def question(legende):
    reponse = [""]
    root = tkinter.Tk ()
    root.title("pseudo input")
    tkinter.Label(text=legende).pack(side=tkinter.LEFT)
    s = tkinter.Entry(text="def", width=80)
    s.pack(side=tkinter.LEFT)
    def rget():
        reponse[0] = s.get ()
        root.destroy()
    tkinter.Button(text="ok", command=rget).pack(side=tkinter.LEFT)
    root.mainloop()
    return(reponse[0])

print("réponse ", question("texte de la question"))
```

65. https://docs.python.org/3/library/functions.html#input
66. https://docs.python.org/3/library/functions.html#input

On peut améliorer la fonction `question` en précisant une valeur par défaut par exemple (voir chap_interface à ce sujet). Le programme affiche la fenêtre suivante :



None, True et 1

L'écriture de certains tests peut encore être réduite lorsqu'on cherche à comparer une variable entière, booléenne ou `None` comme le précise la table suivant :

type	test	test équivalent
bool	`if v:`	`if v == True :`
`bool`	`if not v:`	`if v == False :`
`int`	`if v:`	`if v != 0:`
`int`	`if not v :`	`if v == 0 :`
`float`	`if v :`	`if v != 0.0 :`
`float`	`if not v :`	`if v == 0.0 :`
`list, dict, set`	`if v:`	`if v is not None and len(v) > 0:`
`list, dict, set`	`if not v:`	`if v is None or len(v) == 0:`
`object`	`if v :`	`if v is not None:`
`object`	`if not v:`	`if v is None :`

Passer, instruction `pass`

Dans certains cas, aucune instruction ne doit être exécutée même si un test est validé. En *python*, le corps d'un test ne peut être vide, il faut utiliser l'instruction `pass`. Lorsque celle-ci est manquante, *python* affiche un message d'erreur.

Syntaxe S2 : Instruction pass

```
signe = 0
x = 0
if x < 0: signe = -1
elif x == 0:
   pass           # signe est déjà égal à 0
else :
    signe = 1
```

Dans ce cas précis, si l'instruction `pass` est oubliée, l'interpréteur *python* génère l'erreur suivante :

```
File "nopass.py", line 6
    else :
    ^
IndentationError: expected an indented block
```

2.2.3 Boucles

Boucle B1 : test

Les boucles permettent de répéter une séquence d'instructions tant qu'une certaine condition est vérifiée.

Le langage *python* propose deux types de boucles. La boucle `while` suit scrupuleusement la définition précédent. La boucle `for` est une boucle `while` déguisée (voir *Boucle for* (page 44)), elle propose une écriture simplifiée pour répéter la même séquence d'instructions pour tous les éléments d'un ensemble.

Boucle while

L'implémentation d'une boucle de type `while` suit le schéma d'écriture suivant :

Syntaxe S3 : Boucle while

```
while cond :
    instruction 1
    ...
    instruction n
```

Où `cond` est une condition qui détermine la poursuite de la répétition des instructions incluses dans la boucle. Tant que celle-ci est vraie, les instructions 1 à *n* sont exécutées.

Tout comme les tests, l'indentation joue un rôle important. Le décalage des lignes d'un cran vers la droite par rapport à l'instruction `while` permet de les inclure dans la boucle comme le montre l'exemple suivant.

<<<

```
n = 0
while n < 3:
    print("à l'intérieur ", n)
    n += 1
print("à l'extérieur ", n)
```

>>>

```
    à l'intérieur  0
    à l'intérieur  1
    à l'intérieur  2
    à l'extérieur  3
```

Les conditions suivent la même syntaxe que celles définies lors des tests (voir *Comparaisons possibles* (page 39)). A moins d'inclure l'instruction *break* (page 51) qui permet de sortir prématurément d'une boucle, la condition qui régit cette boucle doit nécessairement être modifiée à l'intérieur de celle-ci. Dans le cas contraire, on appelle une telle boucle une boucle infinie[67] puisqu'il est impossible d'en sortir.

L'exemple suivant contient une boucle infinie car le symbole = est manquant dans la dernière instruction. La variable `n` n'est jamais modifiée et la condition `n<3` toujours vraie.

```
n = 0
while n < 3 :
   print(n)
   n + 1         # n n'est jamais modifié, l'instruction correcte serait n += 1
```

67. https://fr.wikipedia.org/wiki/Boucle_infinie

Boucle for

L'implémentation d'une boucle de type `for` suit le schéma d'écriture suivant :

Syntaxe S4 : Boucle for

```
for x in ensemble:
    instruction 1
    ...
    instruction n
```

Où `x` est un élément de l'ensemble `ensemble`. Les instructions 1 à *n* sont exécutées pour chaque élément `x` de l'ensemble `ensemble`. Cet ensemble peut être une chaîne de caractères, un tuple, une liste, un dictionnaire, un set ou tout autre type incluant des itérateurs qui sont présentés au chapitre *Classes* (page 95).

Tout comme les tests, l'indentation est importante. L'exemple suivant affiche tous les éléments d'un tuple à l'aide d'une boucle `for`.

<<<

```
t = (1, 2, 3, 4)
for x in t:        # affiche les nombres 1,2,3,4
    print(x)       # chacun sur une ligne différente
```

>>>

```
    1
    2
    3
    4
```

Lors de l'affichage d'un dictionnaire, les éléments n'apparaissent pas triés ni dans l'ordre dans lequel ils y ont été insérés. L'exemple suivant montre comment afficher les clés et valeurs d'un dictionnaire dans l'ordre croissant des clés.

<<<

```
d = {1: 2, 3: 4, 5: 6, 7: -1, 8: -2}
# affiche le dictionnaire {8: -2, 1: 2, 3: 4, 5: 6, 7: -1}
print(d)
k = list(d.keys())
print(k)                 # affiche les clés [8, 1, 3, 5, 7]
k.sort()
print(k)                 # affiche les clés triées [1, 3, 5, 7, 8]
for x in k:              # affiche les éléments du dictionnaire
    print(x, ":", d[x])  # triés par clés croissantes
```

>>>

```
    {1: 2, 3: 4, 5: 6, 7: -1, 8: -2}
    [1, 3, 5, 7, 8]
    [1, 3, 5, 7, 8]
    1 : 2
    3 : 4
    5 : 6
    7 : -1
    8 : -2
```

Le langage *python* propose néanmoins la fonction `sorted` qui réduit l'exemple suivant en trois lignes :

<<<

```
d = {1: 2, 3: 4, 5: 6, 7: -1, 8: -2}
for x in sorted(d):     # pour les clés dans l'ordre croissant
    print(x, ":", d[x])
```

>>>

```
    1 : 2
    3 : 4
    5 : 6
    7 : -1
    8 : -2
```

La boucle la plus répandue est celle qui parcourt des indices entiers compris entre *0* et *n-1*. On utilise pour cela la boucle `for` et la fonction range [68] comme dans l'exemple qui suit.

<<<

```
sum = 0
N = 10
for n in range(0, N):      # va de 0 à N exclu
    sum += n                # additionne tous les entiers compris entre 0 et N-1
```

>>>

```

```

Ou encore pour une liste quelconque :

<<<

```
li = [4, 5, 3, -6, 7, 9]
sum = 0
for n in range(0, len(li)):  # va de 0 à len(li) exclu
    sum += li[n]              # additionne tous les éléments de li
```

>>>

```

```

Listes, boucle for, liste en extension

Le paragraphe *Boucles et listes* (page 22) a montré comment le mot-clé `for` peut être utilisé pour simplifier la création d'une liste à partir d'une autre. La syntaxe d'une liste en extension [69] suit le schéma suivant :

Syntaxe S5 : Liste en extension

```
[ expression for x in ensemble ]
```

Où `expression` est une expression numérique incluant ou non `x`, la variable de la boucle, `ensemble` est un ensemble d'éléments, tuple, liste, dictionnaire, set ou tout autre chose qui peut être parcouru. Cette syntaxe permet de résumer en une ligne la création de la séquence `y` du programme suivant.

<<<

68. https://docs.python.org/3/library/functions.html#func-range
69. http://sametmax.com/python-love-les-listes-en-intention-partie/

```
y = list()
for i in range(0, 5):
    y.append(i+1)
print(y)                                # affiche [1,2,3,4,5]

y = [i+1 for i in range(0, 5)]      # résume trois lignes du programme précédent
print(y)                                # affiche [1,2,3,4,5]
```

>>>

```
    [1, 2, 3, 4, 5]
    [1, 2, 3, 4, 5]
```

Un autre exemple de cette syntaxe réduite a été présenté au paragraphe *Boucles et listes* (page 22). Cette écriture condensée est bien souvent plus lisible même si tout dépend des préférences de celui qui programme. Elle peut être étendue au dictionnaire.

<<<

```
y = {i: i+1 for i in range(0, 5)}
print(y)
```

>>>

```
    {0: 1, 1: 2, 2: 3, 3: 4, 4: 5}
```

Itérateurs

Toute boucle `for` peut s'appliquer sur un objet muni d'un itérateur tels que les chaînes de caractères, tuples, les listes, les dictionnaires, les ensembles.

<<<

```
d = ["un", "deux", "trois"]
for x in d:
    print(x)          # affichage de tous les éléments de d
```

>>>

```
    un
    deux
    trois
```

Cette syntaxe réduite a déjà été introduite pour les listes et les dictionnaires au chapitre précédent. Il existe une version équivalente avec la boucle `while` utilisant de façon explicite les itérateurs. Il peut être utile de lire le chapitre suivant sur les classes et le chapitre *Exceptions* (page 155) sur les exceptions avant de revenir sur la suite de cette section qui n'est de toutes façons pas essentielle.

L'exemple précédent est convertible en une boucle `while` en faisant apparaître explicitement les itérateurs (voir *Itérateurs* (page 113)). Un itérateur est un objet qui permet de parcourir aisément un ensemble. La fonction `it = iter(e)` permet d'obtenir un itérateur `it` sur l'ensemble `e`. L'appel à l'instruction `it.next()` parcourt du premier élément jusqu'au dernier en retournant la valeur de chacun d'entre eux. Lorsqu'il n'existe plus d'élément, l'exception `StopIteration` est déclenchée (voir *Exceptions* (page 155)). Il suffit de l'intercepter pour mettre fin au parcours.

<<<

```
d = ["un", "deux", "trois"]
it = iter(d)                        # obtient un itérateur sur d
while True:
    try:
        # obtient l'élément suivant, s'il n'existe pas
        x = next(it)
    except StopIteration:
        break                       # déclenche une exception
    print(x)                        # affichage de tous les éléments de d
```

>>>

```
    un
    deux
    trois
```

Plusieurs variables de boucles

Jusqu'à présent, la boucle `for` n'a été utilisée qu'avec une seule variable de boucle, comme dans l'exemple suivant où on parcourt une liste de tuple pour les afficher.

<<<

```
d = [(1, 0, 0), (0, 1, 0), (0, 0, 1)]
for v in d:
    print(v)
```

>>>

```
    (1, 0, 0)
    (0, 1, 0)
    (0, 0, 1)
```

Lorsque les éléments d'un ensemble sont des tuples, des listes, des dictionnaires ou des ensembles composés de taille fixe, il est possible d'utiliser une notation qui rappelle les affectations multiples (voir *Affectations multiples* (page 33)). L'exemple précédent devient dans ce cas :

<<<

```
d = [(1, 0, 0), (0, 1, 0), (0, 0, 1)]
for x, y, z in d:
    print(x, y, z)
```

>>>

```
    1 0 0
    0 1 0
    0 0 1
```

Cette écriture n'est valable que parce que chaque élément de la liste `d` est un tuple composé de trois nombres. Lorsqu'un des éléments est de taille différente à celle des autres, comme dans l'exemple suivant, une erreur survient.

<<<

```
d = [(1, 0, 0), (0, 1, 0, 6), (0, 0, 1)]  # un élément de taille quatre
for x, y, z in d:
    print(x, y, z)
```

>>>

```
    1 0 0
    [runpythonerror]

    Traceback (most recent call last):
        exec(obj, globs, loc)
      File "", line 6, in <module>
      File "", line 4, in run_python_script_139791966096912
    ValueError: too many values to unpack (expected 3)
```

Cette syntaxe est très pratique associée à la fonction `zip` (voir *Collage de séquences, fonction zip* (page 23)). Il est alors possible de parcourir plusieurs séquences (tuple, liste, dictionnaire, ensemble) simultanément.

<<<

```
a = range(0, 5)
b = [x**2 for x in a]
for x, y in zip(a, b):
    print(y), " est le carré de ", x
    # affichage à droite
```

>>>

```
    0
    1
    4
    9
    16
```

Ecriture condensée

Comme pour les tests, lorsque les boucles ne contiennent qu'une seule instruction, il est possible de l'écrire sur la même ligne que celle de la déclaration de la boucle `for` ou `while`.

<<<

```
d = ["un", "deux", "trois"]
for x in d:
    print(x)          # une seule instruction
```

>>>

```
    un
    deux
    trois
```

Il existe peu de cas où la boucle `while` s'écrit sur une ligne car elle inclut nécessairement une instruction permettant de modifier la condition d'arrêt.

<<<

```
d = ["un", "deux", "trois"]
i = 0
while d[i] != "trois":
    i += 1
print("trois a pour position ", i)
```

>>>

```
    trois a pour position  2
```

Pilotage d'une boucle : continue

Pour certains éléments d'une boucle, lorsqu'il n'est pas nécessaire d'exécuter toutes les instructions, il est possible de passer directement à l'élément suivant ou l'itération suivante. Le programme suivant utilise le crible d'Eratosthène [70] pour dénicher tous les nombres premiers compris entre 1 et 99.

Aparté sur le crible d'Eratosthène

Le crible d'Eratosthène est un algorithme permettant de déterminer les nombres premiers. Pour un nombre premier p, il paraît plus simple de considérer tous les entiers de $p-1$ à *1* pour savoir si l'un d'eux divise p. C'est ce qu'on fait lorsqu'on doit vérifier le caractère premier d'un seul nombre. Pour plusieurs nombres à la fois, le crible d'Eratosthène est plus efficace : au lieu de s'intéresser aux diviseurs, on s'intéresse aux multiples d'un nombre. Pour un nombre i, on sait que $2i$, $3i$, ... ne sont pas premiers. On les raye de la liste. On continue avec $i+1$, $2(i+1)$, $3(i+1)$...

<<<

```
d = dict()
for i in range(1, 100):             # d [i] est vrai si i est un nombre premier
    d[i] = True                    # au début, comme on ne sait pas, on suppose
    # que tous les nombres sont premiers
for i in range(2, 100):
                                    # si d [i] est faux,
    if not d[i]:
        continue                    # les multiples de i ont déjà été cochés
        # et peut passer à l'entier suivant
    for j in range(2, 100):
        if i*j < 100:
            d[i*j] = False         # d [i*j] est faux pour tous les multiples de i
            # inférieurs à 100
print("liste des nombres premiers")
for i in d:
    if d[i]:
        print(i)
```

>>>

```
    liste des nombres premiers
    1
    2
    3
    5
    7
    11
    13
    17
    19
    23
    29
    31
    37
    41
    43
    47
    53
```

(suite sur la page suivante)

70. https://fr.wikipedia.org/wiki/Crible_d'%C3%89ratosth%C3%A8ne

(suite de la page précédente)

```
    59
    61
    67
    71
    73
    79
    83
    89
    97
```

Ce programme est équivalent au suivant :

<<<

```
d = dict()
for i in range(1, 100):
    d[i] = True

for i in range(2, 100):
    if d[i]:
        for j in range(2, 100):
            if i*j < 100:
                d[i*j] = False

print("liste des nombres premiers")
for i in d:
    if d[i]:
        print(i)
```

>>>

```
    liste des nombres premiers
    1
    2
    3
    5
    7
    11
    13
    17
    19
    23
    29
    31
    37
    41
    43
    47
    53
    59
    61
    67
    71
    73
    79
    83
    89
    97
```

Le mot-clé `continue` évite de trop nombreuses indentations et rend les programmes plus lisibles.

Pilotage d'une boucle : break

Lors de l'écriture d'une boucle `while`, il n'est pas toujours adéquat de résumer en une seule condition toutes les raisons pour lesquelles il est nécessaire d'arrêter l'exécution de cette boucle. De même, pour une boucle `for`, il n'est pas toujours utile de visiter tous les éléments de l'ensemble à parcourir. C'est le cas par exemple lorsqu'on recherche un élément, une fois qu'il a été trouvé, il n'est pas nécessaire d'aller plus loin. L'instruction `break` permet de quitter l'exécution d'une boucle.

<<<

```
l = [6, 7, 5, 4, 3]
n = 0
c = 5
for x in l:
    if x == c:
        break   # l'élément a été trouvé, on sort de la boucle
    n += 1              # si l'élément a été trouvé, cette instruction
    # n'est pas exécutée
print("l'élément ", c, "est en position ", n)
```

>>>

```
    l'élément  5 est en position  2
```

Si deux boucles sont imbriquées, l'instruction `break` ne sort que de la boucle dans laquelle elle est insérée. L'exemple suivant vérifie si un entier est la somme des carrés de deux entiers compris entre 1 et 20.

<<<

```
ens = range(1, 21)
n = 53
for x in ens:
    for y in ens:
        c = x*x + y*y
        if c == n:
            break
    if c == n:
        break   # cette seconde instruction break est nécessaire
        # pour sortir de la seconde boucle
        # lorsque la solution a été trouvée
if c == n:
    # le symbole \ permet de passer à la ligne sans changer d'instruction
    print(n, " est la somme des carrés de deux entiers :",
          x, "*", x, "+", y, "*", y, "=", n)
else:
    print(n, " n'est pas la somme des carrés de deux entiers")
```

>>>

```
    53  est la somme des carrés de deux entiers : 2 * 2 + 7 * 7 = 53
```

Fin normale d'une boucle : else

Le mot-clé `else` existe aussi pour les boucles et s'utilise en association avec le mot-clé `break`. L'instruction `else` est placée à la fin d'une boucle, indentée au même niveau que `for` ou `while`. Les lignes qui suivent le mot-clé `else` ne sont exécutées que si aucune instruction `break` n'a été rencontrée dans le corps de la boucle. On reprend l'exemple

du paragraphe précédent. On recherche cette fois-ci la valeur 1 qui ne se trouve pas dans la liste `L`. Les lignes suivant le test `if x == c` ne seront jamais exécutées au contraire de la dernière.

<<<

```
L = [6, 7, 5, 4, 3]
n = 0
c = 1
for x in L:
    if x == c:
        print("l'élément ", c, " est en position ", n)
        break
    n += 1
else:
    print("aucun élément ", c, " trouvé")  # affiche aucun élément  1  trouvé
```

>>>

```
    aucun élément  1  trouvé
```

Les lignes dépendant de la clause `else` seront exécutées dans tous les cas où l'exécution de la boucle n'est pas interrompue par une instruction `break` ou une instruction `return`.

Suppression ou ajout d'éléments lors d'une boucle

En parcourant la liste en se servant des indices, il est possible de supprimer une partie de cette liste. Il faut néanmoins faire attention à ce que le code ne produise pas d'erreur comme c'est le cas pour le suivant. La boucle `for` parcourt la liste `list(range(0, len(li)))` qui n'est pas modifiée en même temps que l'instruction `del li[i:i+2]`.

<<<

```
li = list(range(0, 10))
print(li)                 # affiche [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in range(0, len(li)):
    if i == 5:
        del li[i:i+2]
    print(li[i])         # affiche successivement 0, 1, 2, 3, 4, 7, 8, 9 et
    # produit une erreur
print(li)
```

>>>

```
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    0
    1
    2
    3
    4
    7
    8
    9
    [runpythonerror]

    Traceback (most recent call last):
        exec(obj, globs, loc)
      File "", line 11, in <module>
      File "", line 8, in run_python_script_139791967370264
    IndexError: list index out of range
```

Le programme suivant marche parfaitement puisque cette fois-ci la boucle parcourt la liste `li`. En revanche, pour la suppression d'une partie de celle-ci, il est nécessaire de conserver en mémoire l'indice de l'élément visité. C'est le rôle de la variable `i`.

<<<

```
li = list(range(0, 10))
print(li)                # affiche [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
i = 0
for t in li:
    if i == 5:
        del li[i:i+2]
    i = i+1
    print(t)             # affiche successivement 0, 1, 2, 3, 4, 5, 8, 9
print(li)                # affiche [0, 1, 2, 3, 4, 7, 8, 9]
```

>>>

```
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    0
    1
    2
    3
    4
    5
    8
    9
    [0, 1, 2, 3, 4, 7, 8, 9]
```

Le langage *python* offre la possibilité de supprimer des éléments d'une liste alors même qu'on est en train de la parcourir. Le programme qui suit ne marche pas puisque l'instruction `del i` ne supprime pas un élément de la liste mais l'identificateur `i` qui prendra une nouvelle valeur lors du passage suivant dans la boucle.

<<<

```
li = list(range(0, 10))
print(li)                # affiche [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in li:
    if i == 5:
        del i
print(li)                # affiche [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

>>>

```
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

On pourrait construire des exemples similaires dans le cadre de l'ajout d'un élément à la liste. Il est en règle générale déconseillé de modifier une liste, un dictionnaire pendant qu'on le parcourt. Malgré tout, si cela s'avérait indispensable, il convient de faire plus attention dans ce genre de situations.

2.2.4 Fonctions

Les fonctions sont des petits programmes qui effectuent des tâches plus précises que le programme entier. On peut effectivement écrire un programme sans fonction mais ils sont en général illisibles. Une fonction décrit des traitement à faire sur les arguments qu'elle reçoit en supposant qu'ils existent. Utiliser des fonctions implique de découper un algorithme en tâches élémentaires. Le programme final est ainsi plus facile à comprendre. Un autre avantage est de pouvoir plus facilement isoler une erreur s'il s'en produit une : il suffit de tester une à une les fonctions pour déterminer laquelle

retourne un mauvais résultat. L'avantage le plus important intervient lorsqu'on doit effectuer la même chose à deux endroits différentes d'un programme : une seule fonction suffit et elle sera appelée à ces deux endroitsfootnote{Pour les utilisateurs experts : en langage *python*, les fonctions sont également des variables, elles ont un identificateur et une valeur qui est dans ce cas un morceau de code. Cette précision explique certaines syntaxes du chapitre chap_interface sur les interfaces graphiques ou celle introduite en fin de chapitre au paragraphe paragraphe_fonction_variable.

Définition, syntaxe

Définition D2 : fonction

Une fonction est une partie d'un programme - ou sous-programme - qui fonctionne indépendamment du reste du programme. Elle reçoit une liste de paramètres et retourne un résultat. Le corps de la fonction désigne toute instruction du programme qui est exécutée si la fonction est appelée.

Lorsqu'on écrit ses premiers programme, on écrit souvent des fonctions plutôt longues avant de s'apercevoir que certains parties sont identiques ailleurs. On extrait donc la partie répétée pour en faire une fonction. Avec l'habitude, on finit par écrire des fonctions plus petites et réutilisables.

Syntaxe S6 : Déclaration d'une fonction

```
def fonction_nom (par_1, ..., par_n) :
    instruction_1
    ...
    instruction_n
    return res_1, ..., res_n
```

`fonction_nom` est le nom de la fonction, il suit les mêmes règles que le nom des variables. `par_1` à `par_n` sont les noms des paramètres et `res_1` à `res_n` sont les résultats retournés par la fonction. Les instructions associées à une fonction doivent être indentées par rapport au mot-clé `def`.

S'il n'y a aucun résultat, l'instruction `return` est facultative ou peut être utilisée seule sans être suivie par une valeur ou une variable. Cette instruction peut apparaître plusieurs fois dans le code de la fonction mais une seule d'entre elles sera exécutée. A partir de ce moment, toute autre instruction de la fonction sera ignorée. Pour exécuter une fonction ainsi définie, il suffit de suivre la syntaxe suivante :

Syntaxe S7 : Appel d'une fonction

```
x_1, ..., x_n = fonction_nom (valeur_1, valeur_2, ..., valeur_n)
```

Où `fonction_nom` est le nom de la fonction, `valeur_1` à `valeur_n` sont les noms des paramètres, `x_1` à `x_n` reçoivent les résultats retournés par la fonction. Cette affectation est facultative. Si on ne souhaite pas conserver les résultats, on peut donc appeler la fonction comme suit :

```
fonction_nom (valeur_1, valeur_2, ..., valeur_n)
```

Lorsqu'on commence à programmer, il arrive parfois qu'on confonde le rôle des mots-clés `print` et `return`. Il faut se souvenir que l'instruction `print` n'a pas d'impact sur le déroulement du programme. Elle sert juste à visualiser le contenu d'une variable. Sans l'instruction `return`, toute fonction retourne `None`.

Exemple

Le programme suivant utilise deux fonctions. La première convertit des coordonnées cartésiennes en coordonnées polaires. Elle prend deux réels en paramètres et retourne deux autres réels. La seconde fonction affiche les résultats de

la première pour tout couple de valeurs (x, y). Elle ne retourne aucun résultat.

<<<

```
import math


def coordonnees_polaires(x, y):
    rho = math.sqrt(x*x+y*y)    # calcul la racine carrée de x*x+y*y
    theta = math.atan2(y, x)      # calcule l'arc tangente de y/x en tenant
    # compte des signes de x et y
    return rho, theta


def affichage(x, y):
    r, t = coordonnees_polaires(x, y)
    print("cartésien (%f,%f) --> polaire (%f,%f degrés)"
          % (x, y, r, math.degrees(t)))


affichage(1, 1)
affichage(0.5, 1)
affichage(-0.5, 1)
affichage(-0.5, -1)
affichage(0.5, -1)
```

>>>

```
    cartésien (1.000000,1.000000) --> polaire (1.414214,45.000000 degrés)
    cartésien (0.500000,1.000000) --> polaire (1.118034,63.434949 degrés)
    cartésien (-0.500000,1.000000) --> polaire (1.118034,116.565051 degrés)
    cartésien (-0.500000,-1.000000) --> polaire (1.118034,-116.565051 degrés)
    cartésien (0.500000,-1.000000) --> polaire (1.118034,-63.434949 degrés)
```

Paramètres avec des valeurs par défaut

Lorsqu'une fonction est souvent appelée avec les mêmes valeurs pour ses paramètres, il est possible de spécifier pour ceux-ci une valeur par défaut.

Syntaxe S8 : Valeurs par défaut

```
def fonction_nom (param_1, param_2 = valeur_2, ..., param_n = valeur_n):
    ...
```

Où `fonction_nom` est le nom de la fonction. `param_1` à `param_n` sont les noms des paramètres, `valeur_2` à `valeur_n` sont les valeurs par défaut des paramètres `param_2` à `param_n`. La seule contrainte lors de cette définition est que si une valeur par défaut est spécifiée pour un paramètre, alors tous ceux qui suivent devront eux aussi avoir une valeur par défaut.

Exemple : %

<<<

```
def commander_carte_orange(nom, prenom, paiement="carte", nombre=1, zone=2):
    print("nom : ", nom)
    print("prénom : ", prenom)
    print("paiement : ", paiement)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print("nombre : ", nombre)
    print("zone :", zone)


commander_carte_orange("Dupré", "Xavier", "chèque")
# les autres paramètres nombre et zone auront pour valeur
# leurs valeurs par défaut
```

>>>

```
    nom :  Dupré
    prénom :  Xavier
    paiement :  chèque
    nombre :  1
    zone : 2
```

Il est impossible qu'un paramètre sans valeur par défaut associée se situe après un paramètre dont une valeur par défaut est précisée. Le programme suivant ne pourra être exécuté.

```
def commander_carte_orange (nom, prenom, paiement="carte", nombre=1, zone):
    print("nom : ", nom)
    # ...
```

Il déclenche l'erreur suivante :

```
File "problem_zone.py", line 1
    def commander_carte_orange (nom, prenom, paiement = "carte", nombre = 1, zone):
SyntaxError: non-default argument follows default argument
```

Les valeurs par défaut de type modifiable (liste, dictionnaire, ensemble, classes) peuvent introduire des erreurs inattendues dans les programmes comme le montre l'exemple suivant :

<<<

```
def fonction(l=[0, 0]):
    l[0] += 1
    return l


print(fonction())         # affiche [1,0] : résultat attendu
print(fonction())         # affiche [2,0] : résultat surprenant
print(fonction([0, 0]))    # affiche [1,0] : résultat attendu
```

>>>

```
    [1, 0]
    [2, 0]
    [1, 0]
```

L'explication provient du fait que la valeur par défaut est une liste qui n'est pas recréée à chaque appel : c'est la même liste à chaque fois que la fonction est appelée sans paramètre. Pour remédier à cela, il faudrait écrire :

<<<

```
import copy


def fonction(l=[0, 0]):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    l = copy.copy(l)
    l[0] += 1
    return l
```

>>>

```

```

L'exercice ex_hypercube propose un exemple plus complet, voire retors.

Ordre des paramètres

Le paragraphe *Définition, syntaxe* (page 54) a présenté la syntaxe d'appel a une fonction. Lors de l'appel, le nom des paramètres n'intervient plus, supposant que chaque paramètre reçoit pour valeur celle qui a la même position que lui lors de l'appel à la fonction. Il est toutefois possible de changer cet ordre en précisant quel paramètre doit recevoir quelle valeur.

```
x_1, ..., x_n = fonction_nom (param_1 = valeur_1, ..., param_n = valeur_n)
```

Où `fonction_nom` est le nom de la fonction, `param_1` à `param_n` sont les noms des paramètres, `valeur_1` à `valeur_n` sont les valeurs que reçoivent ces paramètres. Avec cette syntaxe, l'ordre d'écriture n'importe pas. La valeur `valeur_i` sera toujours attribuée à `param_i`. Les variables `x_1` à `x_n` reçoivent les résultats retournés par la fonction. L'ordre des résultats ne peut pas être changé. S'il y a plusieurs résultats retournés, il est impossible de choisir lesquels conserver : soit tous, soit aucun.

Exemple :

<<<

```
def identite(nom, prenom):
    print("nom : ", nom, " prénom : ", prenom)


identite("Xavier", "Dupré")                  # nom :  Xavier prénom :  Dupré
identite(prenom="Xavier", nom="Dupré")  # nom :  Dupré  prénom :  Xavier
```

>>>

```
    nom :  Xavier  prénom :  Dupré
    nom :  Dupré  prénom :  Xavier
```

Cette possibilité est intéressante surtout lorsqu'il y a de nombreux paramètres par défaut et que seule la valeur d'un des derniers paramètres doit être changée.

<<<

```
def commander_carte_orange(paiement="carte", nombre=1, zone=2):
    print("paiement : ", paiement)
    print("nombre : ", nombre)
    print("zone :", zone)


commander_carte_orange(zone=5)  # seule la valeur par défaut
# du paramètre zone sera changée
```

>>>

```
    paiement :  carte
    nombre :  1
    zone : 5
```

Surcharge de fonction

Contrairement à d'autres langages, *python* n'autorise pas la surcharge de fonction. Autrement dit, il n'est pas possible que plusieurs fonctions portent le même nom même si chacune d'entre elles a un nombre différent de paramètres.

<<<

```
def fonction(a, b):
    return a + b


def fonction(a, b, c):
    return a + b + c


print(fonction(5, 6))
print(fonction(5, 6, 7))
```

>>>

```
    [runpythonerror]

    Traceback (most recent call last):
        exec(obj, globs, loc)
      File "", line 11, in <module>
      File "", line 9, in run_python_script_139791966268656
    TypeError: fonction() missing 1 required positional argument: 'c'
```

Le petit programme précédent est syntaxiquement correct mais son exécution génère une erreur parce que la seconde définition de la fonction `fonction` efface la première.

Commentaires

Le langage *python* propose une fonction `help` qui retourne pour chaque fonction un commentaire ou mode d'emploi qui indique comment se servir de cette fonction. L'exemple suivant affiche le commentaire associé à la fonction `round`.

```
>>> help (round)

Help on built-in function round:

round(...)
    round(number[, ndigits]) -> floating point number

    Round a number to a given precision in decimal digits (default 0 digits).
    This always returns a floating point number.  Precision may be negative.
```

Lorsqu'on utilise cette fonction `help` sur la fonction `coordonnees_polaires` définie dans l'exemple du paragraphe précédent, le message affiché n'est pas des plus explicites.

```
>>> help (coordonnees_polaires)

Help on function coordonnees_polaires in module __main__:

coordonnees_polaires(x, y)
```

Pour changer ce message, il suffit d'ajouter en première ligne du code de la fonction une chaîne de caractères.

<<<

```
import math


def coordonnees_polaires(x, y):
    """
    convertit des coordonnées cartésiennes en coordonnées polaires
    (x,y) --> (pho,theta)
    """
    rho = math.sqrt(x*x+y*y)
    theta = math.atan2(y, x)
    return rho, theta


help(coordonnees_polaires)
```

>>>

```
    Help on function coordonnees_polaires in module pyquickhelper.sphinxext.sphinx_
↪runpython_extension:

    coordonnees_polaires(x, y)
        convertit des coordonnées cartésiennes en coordonnées polaires
        (x,y) --> (pho,theta)
```

Le programme affiche alors un message d'aide nettement plus explicite. Il est conseillé d'écrire ce commentaire pour toute nouvelle fonction avant même que son corps ne soit écrit. L'expérience montre qu'on oublie souvent de l'écrire après.

Paramètres modifiables

Les paramètres de types immuables et modifiables se comportent de manières différentes à l'intérieur d'une fonction. Ces paramètres sont manipulés dans le corps de la fonction, voire modifiés parfois. Selon le type du paramètre, ces modifications ont des répercussions à l'extérieur de la fonction.

Les types immuables ne peuvent être modifiés et cela reste vrai. Lorsqu'une fonction accepte un paramètre de type immuable, elle ne reçoit qu'une copie de sa valeur. Elle peut donc modifier ce paramètre sans que la variable ou la valeur utilisée lors de l'appel de la fonction n'en soit affectée. On appelle ceci un passage de paramètre par valeur. A l'opposé, toute modification d'une variable d'un type modifiable à l'intérieur d'une fonction est répercutée à la variable qui a été utilisée lors de l'appel de cette fonction. On appelle ce second type de passage un passage par adresse.

L'exemple suivant utilise une fonction `somme_n_premier_terme` qui modifie ces deux paramètres. Le premier `n`, est immuable, sa modification n'a aucune incidence sur la variable `nb`. En revanche, le premier élément du paramètre `liste` reçoit la valeur 0. Le premier élément de la liste `l` n'a plus la même valeur après l'appel de la fonction `somme_n_premier_terme` que celle qu'il avait avant.

<<<

```
def somme_n_premier_terme(n, liste):
    """calcul la somme des n premiers termes d'une liste"""
    somme = 0
    for i in liste:
        somme += i
        n -= 1              # modification de n (type immuable)
        if n <= 0:
            break
    liste[0] = 0            # modification de liste (type modifiable)
    return somme


l = [1, 2, 3, 4]
nb = 3
# affiche   avant la fonction  3 [1, 2, 3, 4]
print("avant la fonction ", nb, l)
s = somme_n_premier_terme(nb, l)
# affiche   après la fonction  3 [0, 2, 3, 4]
print("après la fonction ", nb, l)
print("somme : ", s)                # affiche   somme :  6
```

>>>

```
    avant la fonction  3 [1, 2, 3, 4]
    après la fonction  3 [0, 2, 3, 4]
    somme :  6
```

La liste `l` est modifiée à l'intérieur de la fonction `somme_n_premier_terme` comme l'affichage suivant le montre. En fait, à l'intérieur de la fonction, la liste `l` est désignée par l'identificateur `liste`, c'est la même liste. La variable `nb` est d'un type immuable. Sa valeur a été recopiée dans le paramètre `n` de la fonction `somme_n_premier_terme`. Toute modification de `n` à l'intérieur de cette fonction n'a aucune répercussion à l'extérieur de la fonction.

Passage par adresse

Dans l'exemple précédent, il faut faire distinguer le fait que la liste passée en paramètre ne soit que modifiée et non changée. L'exemple suivant inclut une fonction qui affecte une nouvelle valeur au paramètre `liste` sans pour autant modifier la liste envoyée en paramètre.

```
def fonction (liste):
    liste = []

liste = [0,1,2]
print(liste)        # affiche [0,1,2]
fonction(liste)
print(liste)        # affiche [0,1,2]
```

Il faut considérer dans ce programme que la fonction `fonction` reçoit un paramètre appelé `liste` mais utilise tout de suite cet identificateur pour l'associer à un contenu différent. L'identificateur `liste` est en quelque sorte passé du statut de paramètre à celui de variable locale. La fonction associe une valeur à `liste` - ici, une liste vide - sans toucher à la valeur que cet identificateur désignait précédemment.

Le programme qui suit est différent du précédent mais produit les mêmes effets. Ceci s'explique par le fait que le mot-clé `del` ne supprime pas le contenu d'une variable mais seulement son identificateur. Le langage *python* détecte ensuite qu'un objet n'est plus désigné par aucun identificateur pour le supprimer. Cette remarque est à rapprocher de celles du paragraphe *Copie d'instances* (page 123).

<<<

```
def fonction(liste):
    del liste


liste = [0, 1, 2]
print(liste)        # affiche [0,1,2]
fonction(liste)
print(liste)        # affiche [0,1,2]
```

>>>

```
    [0, 1, 2]
    [0, 1, 2]
```

Le programme qui suit permet cette fois-ci de vider la liste `liste` passée en paramètre à la fonction `fonction`. La seule instruction de cette fonction modifie vraiment le contenu désigné par l'identificateur `liste` et cela se vérifie après l'exécution de cette fonction.

<<<

```
def fonction(liste):
    del liste[0:len(liste)]  # on peut aussi écrire : liste[:] = []


liste = [0, 1, 2]
print(liste)        # affiche [0,1,2]
fonction(liste)
print(liste)        # affiche []
```

>>>

```
    [0, 1, 2]
    []
```

Fonction récursive

Définition D3 : fonction récursive

Une fonction récursive est une fonction qui s'appelle elle-même.

La fonction récursive la plus fréquemment citée en exemple est la fonction factorielle. Celle-ci met en évidence les deux composantes d'une fonction récursive, la récursion proprement dite et la condition d'arrêt.

```
def factorielle(n):
    if n == 0: return 1
    else: return n * factorielle(n-1)
```

La dernière ligne de la fonction `factorielle` est la récursion tandis que la précédente est la condition d'arrêt, sans laquelle la fonction ne cesserait de s'appeler, empêchant le programme de terminer son exécution. Si celle-ci est mal spécifiée ou absente, l'interpréteur *python* affiche une suite ininterrompue de messages. *python* n'autorise pas plus de 1000 appels récursifs : `factorielle(999)` provoque nécessairement une erreur d'exécution même si la condition d'arrêt est bien spécifiée.

```
Traceback (most recent call last):
  File "fact.py", line 5, in <module>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    factorielle(999)
  File "fact.py", line 3, in factorielle
    else : return n * factorielle(n-1)
  File "fact.py", line 3, in factorielle
    else : return n * factorielle(n-1)
  ...
```

La liste des messages d'erreurs est aussi longue qu'il y a eu d'appels à la fonction récursive. Dans ce cas, il faut transformer cette fonction en une fonction non récursive équivalente, ce qui est toujours possible.

```
def factorielle_non_recursive(n):
    r = 1
    for i in range (2, n+1) :
        r *= i
    return r
```

Portée des variables, des paramètres

Lorsqu'on définit une variable, elle n'est pas utilisable partout dans le programme. Par exemple, elle n'est pas utilisable avant d'avoir été déclarée au moyen d'une affectation. Le court programme suivant déclenche une erreur.

<<<

```
print(x)    # déclenche une erreur
```

>>>

```
    [runpythonerror]

    Traceback (most recent call last):
        exec(obj, globs, loc)
      File "", line 4, in <module>
      File "", line 3, in run_python_script_139791967370840
    NameError: name 'x' is not defined
```

Il est également impossible d'utiliser une variable à l'extérieur d'une fonction où elle a été déclarée. Plusieurs fonctions peuvent ainsi utiliser le même nom de variable sans qu'à aucun moment, il n'y ait confusion. Le programme suivant déclenche une erreur identique à celle reproduite ci-dessus.

<<<

```
def portee_variable(x):
    var = x
    print(var)


portee_variable(3)
print(var)          # déclenche une erreur car var est déclarée dans
# la fonction portee_variable
```

>>>

```
    3
    [runpythonerror]

    Traceback (most recent call last):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        exec(obj, globs, loc)
      File "", line 10, in <module>
      File "", line 8, in run_python_script_139791963543808
    NameError: name 'var' is not defined
```

Définition D4 : portée d'un variable

La portée d'une variable associée à un identificateur recouvre la portion du programme à l'intérieur de laquelle ce même identificateur la désigne. Ceci implique que, dans cette portion de code, aucune autre variable, aucune autre fonction, aucune autre classe, ne peut porter le même identificateur.

Une variable n'a donc d'existence que dans la fonction dans laquelle elle est déclarée. On appelle ce type de variable une variable locale. Par défaut, toute variable utilisée dans une fonction est une variable locale.

Définition D5 : variable locale

Une variable locale est une variable dont la portée est réduite à une fonction.

Par opposition aux variables locales, on définit les variables globales qui sont déclarées à l'extérieur de toute fonction.

Définition D6 : variable globale

Une variable globale est une variable dont la portée est l'ensemble du programme.

L'exemple suivant mélange variable locale et variable globale. L'identificateur `n` est utilisé à la fois pour désigner une variable globale égale à 1 et une variable locale égale à 1. A l'intérieur de la fonction, `n` désigne la variable locale égale à 2. A l'extérieur de la fonction, `n` désigne la variable globale égale à 1.

<<<

```
n = 1                    # déclaration d'une variable globale


def locale_globale():
    n = 2                # déclaration d'une variable locale
    print(n)             # affiche le contenu de la variable locale


print(n)                 # affiche 1
locale_globale()         # affiche 2
print(n)                 # affiche 1
```

>>>

```
    1
    2
    1
```

Il est possible de faire référence aux variables globales dans une fonction par l'intermédiaire du mot-clé `global`. Celui-ci indique à la fonction que l'identificateur `n` n'est plus une variable locale mais désigne une variable globale déjà déclarée.

<<<

```
n = 1                    # déclaration d'une variable globale


def locale_globale():
    global n             # cette ligne indique que n désigne la variable globale
    n = 2                # change le contenu de la variable globale
    print(n)             # affiche le contenu de la variable globale


print(n)                 # affiche 1
locale_globale()         # affiche 2
print(n)                 # affiche 2
```

>>>

```
    1
    2
    1
```

Cette possibilité est à éviter le plus possible car on peut considérer que `locale_globale` est en fait une fonction avec un paramètre caché. La fonction `locale_globale` n'est plus indépendante des autres fonctions puisqu'elle modifie une des données du programme.

Portée des fonctions

Le langage *python* considère les fonctions également comme des variables d'un type particulier. La portée des fonctions obéit aux mêmes règles que celles des variables. Une fonction ne peut être appelée que si elle a été définie avant son appel.

<<<

```
def factorielle(n):
    # ...
    return 1


print(type(factorielle))  # affiche <type 'function'>
```

>>>

```
    <class 'function'>
```

Comme il est possible de déclarer des variables locales, il est également possible de définir des fonctions locales ou fonctions imbriquées. Une fonction locale n'est appelable qu'à l'intérieur de la fonction dans laquelle elle est définie. Dans l'exemple suivant, la fonction `affiche_pair` inclut une fonction locale qui n'est appelable que par cette fonction `affiche_pair`.

<<<

```
def affiche_pair():
    def fonction_locale(i):            # fonction locale ou imbriquée
        if i % 2 == 0:
            return True
        else:
            return False
    for i in range(0, 10):
        if fonction_locale(i):
```

(suite sur la page suivante)

(suite de la page précédente)

```
            print(i)


affiche_pair()
fonction_locale(5)      # l'appel à cette fonction locale
# déclenche une erreur d'exécution
```

>>>

```
    0
    2
    4
    6
    8
    [runpythonerror]

    Traceback (most recent call last):
        exec(obj, globs, loc)
      File "", line 14, in <module>
      File "", line 12, in run_python_script_139791968447584
    NameError: name 'fonction_locale' is not defined
```

A l'intérieur d'une fonction locale, le mot-clé `global` désigne toujours les variables globales du programme et non les variables de la fonction dans laquelle cette sous-fonction est définie.

Nombre de paramètres variable

Il est possible de définir des fonctions qui prennent un nombre indéterminé de paramètres, lorsque celui-ci n'est pas connu à l'avance. Hormis les paramètres transmis selon le mode présenté dans les paragraphes précédents, des informations peuvent être ajoutées à cette liste lors de l'appel de la fonction, ces informations sont regroupées soit dans une liste de valeurs, soit dans une liste de couples (identificateur, valeur). La déclaration d'une telle fonction obéit à la syntaxe suivante :

Syntaxe S9 : Nombre indéfini de paramètres

```
def fonction (param_1, ..., param_n, *liste, **dictionnaire) :
```

Où `fonction` est un nom de fonction, `param_1` à `param_n` sont des paramètres de la fonction, `liste` est le nom de la liste qui doit recevoir la liste des valeurs seules envoyées à la fonction et qui suivent les paramètres (plus précisément, c'est un tuple), `dictionnaire` reçoit la liste des couples (identificateur, valeur). L'appel à cette fonction suit quant à lui la syntaxe suivante :

```
fonction (valeur_1, ..., valeur_n, \
          liste_valeur_1, ..., liste_valeur_p, \
          nom_1 = v_1, ..., nom_q = v_q)
```

Où `fonction` est un nom de fonction, `valeur_1` à `valeur_n` sont les valeurs associées aux paramètres `param_1` à `param_n`, `liste_valeur_1` à `liste_valeur_p` formeront la liste `liste`, les couples `nom_1 : v_1` à `nom_q : v_q` formeront le dictionnaire `dictionnaire`.

Exemple :

<<<

```
def fonction(p, *l, **d):
    print("p = ", p)
    print("liste (tuple) l :", l)
    print("dictionnaire d :", d)


fonction(1, 2, 3, a=5, b=6)  # 1 est associé au paramètre p
# 2 et 3 sont insérés dans la liste l
# a=5 et b=6 sont insérés dans le dictionnaire d
```

>>>

```
    p =  1
    liste (tuple) l : (2, 3)
    dictionnaire d : {'a': 5, 'b': 6}
```

A l'instar des paramètres par défaut, la seule contrainte de cette écriture est la nécessité de respecter l'ordre dans lequel les informations doivent apparaître. Lors de l'appel, les valeurs sans précision de nom de paramètre seront placées dans une liste (ici le tuple `l`). Les valeurs associées à un nom de paramètre seront placées dans un dictionnaire (ici `d`). Les valeurs par défaut sont obligatoirement placées après les paramètres non nommés explicitement.

Une fonction qui accepte des paramètres en nombre variable peut à son tour appeler une autre fonction acceptant des paramètres en nombre variable. Il faut pour cela se servir du symbole `*` afin de transmettre à `fonction` les valeurs reçues par `fonction2`.

<<<

```
def fonction(p, *l, **d):
    print("p = ", p)
    print("liste l :", l)
    print("dictionnaire d :", d)


def fonction2(p, *l, **d):
    l += (4,)               # on ajoute une valeur au tuple
    d["c"] = 5             # on ajoute un couple (paramètre,valeur)
    fonction(p, *l, **d)  # ne pas oublier le symbole *


fonction2(1, 2, 3, a=5, b=6)
```

>>>

```
    p =  1
    liste l : (2, 3, 4)
    dictionnaire d : {'a': 5, 'b': 6, 'c': 5}
```

Ecriture simplifiée pour des fonctions simples : lambda

Lorsque le code d'une fonction tient en une ligne et est le résultat d'une expression, il est possible de condenser son écriture à l'aide du mot-clé `lambda`.

```
nom_fonction = lambda param_1, ..., param_n : expression
```

`nom_fonction` est le nom de la fonction, `param_1` à `param_n` sont les paramètres de cette fonction (ils peuvent également recevoir des valeurs par défaut), `expression` est l'expression retournée par la fonction.

L'exemple suivant utilise cette écriture pour définir la fonction `min` retournant le plus petit entre deux nombres positifs.

<<<

```
def min(x, y): return (abs(x+y) - abs(x-y))/2


print(min(1, 2))       # affiche 1
print(min(5, 4))       # affiche 4
```

>>>

```
    1.0
    4.0
```

Cette écriture correspond à l'écriture non condensée suivante :

<<<

```
def min(x, y):
    return (abs(x+y) - abs(x-y))/2


print(min(1, 2))       # affiche 1
print(min(5, 4))       # affiche 4
```

>>>

```
    1.0
    4.0
```

La fonction `lambda` considère le contexte de fonction qui la contient comme son contexte. Il est possible de créer des fonctions `lambda` mais celle-ci utiliseront le contexte dans l'état où il est au moment de son exécution et non au moment de sa création.

<<<

```
fs = []
for a in range(0, 10):
    def f(x): return x + a
    fs.append(f)
for f in fs:
    print(f(1))    # le programme affiche 10 fois 10 de suite
    # car la variable a vaut dix à la fin de la boucle
```

>>>

```
    10
    10
    10
    10
    10
    10
    10
    10
    10
    10
```

Pour que le programme affiche les entiers de 1 à 10, il faut préciser à la fonction `lambda` une variable `y` égale à `a` au moment de la création de la fonction et qui sera intégrée au contexte de la fonction `lambda` :

<<<

```
fs = []
for a in range(0, 10):
    f = lambda x, y=a: x + y    # ligne changée
    fs.append(f)
for f in fs:
    print(f(1))
```

>>>

```
    1
    2
    3
    4
    5
    6
    7
    8
    9
    10
```

Fonctions générateur

Le mot-clé `yield` est un peu à part. Utilisé à l'intérieur d'une fonction, il permet d'interrompre le cours de son exécution à un endroit précis de sorte qu'au prochain appel de cette fonction, celle-ci reprendra le cours de son exécution exactement au même endroit avec des variables locales inchangées. Le mot-clé `return` ne doit pas être utilisé. Ces fonctions ou générateurs [71] sont utilisées en couple avec le mot-clé `for` pour simuler un ensemble. L'exemple suivant implémente une fonction `fonction_yield` qui simule l'ensemble des entiers compris entre 0 et *n* exclu

<<<

```
def fonction_yield(n):
    i = 0
    while i < n-1:
        print("yield 1")  # affichage : pour voir ce que fait le programme
        yield i           # arrête la fonction qui reprendra
        i = i+1           # à la ligne suivante lors du prochain appel
    print("yield 2")      # affichage : pour voir ce que fait le programme
    yield i               # arrête la fonction qui ne reprendra pas
    # lors du prochain appel car le code de la fonction
    # prend fin ici


for a in fonction_yield(2):
    print(a)                # affiche tous les éléments que retourne la
    # fonction fonction_yield, elle simule la liste
    # [0,1]
print("-----------------------------------------------")
for a in fonction_yield(3):
    print(a)                # nouvel appel, l'exécution reprend
    # au début de la fonction,
    # affiche tous les éléments que retourne la
    # fonction fonction_yield, elle simule la liste
    # [0,1,2]
```

>>>

71. https://docs.python.org/3/glossary.html#term-generator

```
    yield 1
    0
    yield 2
    1
    ------------------------------------------------
    yield 1
    0
    yield 1
    1
    yield 2
    2
```

Le programme affiche tous les entiers compris entre~0 et 4 inclus ainsi que le texte `"yield 1"` ou `"yield 2"` selon l'instruction `yield` qui a retourné le résultat. Lorsque la fonction a finalement terminé son exécution, le prochain appel agit comme si c'était la première fois qu'on l'appelait.

Identificateur appelable

La fonction `callable` retourne un booléen permettant de savoir si un identificateur est une fonction (voir *Classes* (page 95)), de savoir par conséquent si tel identificateur est appelable comme une fonction.

<<<

```
x = 5


def y():
    return None


print(callable(x))  # affiche False car x est une variable
print(callable(y))  # affiche True car y est une fonction
```

>>>

```
    False
    True
```

Compilation dynamique (eval)

Cette fonction a déjà été abordée lors des paragraphes :*Fonction print, repr et conversion en chaîne de caractères* (page 31) ou *Fonction eval* (page 32). Elle évalue toute chaîne de caractères contenant une expression écrite avec la syntaxe du langage *python*. Cette expression peut utiliser toute variable ou toute fonction accessible au moment où est appelée la fonction `eval`.

<<<

```
x = 3
y = 4
print(eval("x*x+y*y+2*x*y"))  # affiche 49
print((x+y)**2)               # affiche 49
```

>>>

```
    49
    49
```

Si l'expression envoyée à la fonction `eval` inclut une variable non définie, l'interpréteur *python* génère une erreur comme le montre l'exemple suivant.

<<<

```
x = 3
y = 4
print(eval("x*x+y*y+2*x*y+z"))
```

>>>

```
    [runpythonerror]

    Traceback (most recent call last):
        exec(obj, globs, loc)
      File "", line 6, in <module>
      File "", line 5, in run_python_script_139791967369256
      File "<string>", line 1, in <module>
    NameError: name 'z' is not defined
```

La variable `z` n'est pas définie et l'expression n'est pas évaluable. L'erreur se produit dans une chaîne de caractères traduite en programme informatique, c'est pourquoi l'interpréteur ne peut pas situer l'erreur dans un fichier. L'erreur ne se produit dans aucun fichier, cette chaîne de caractères pourrait être définie dans un autre.

Compilation dynamique (compile, exec)

Plus complète que la fonction eval [72], la fonction compile [73] permet d'ajouter une ou plusieurs fonctions au programme, celle-ci étant définie par une chaîne de caractères. Le code est d'abord compilé (fonction `compile`) puis incorporé au programme (fonction exec [74]) comme le montre l'exemple suivant.

<<<

```
import math
str = """def coordonnees_polaires(x,y):
    rho     = math.sqrt(x*x+y*y)
    theta   = math.atan2 (y,x)
    return rho, theta"""          # fonction définie par une chaîne de caractères

obj = compile(str, "", "exec")      # fonction compilée
exec(obj)                           # fonction incorporée au programme
# affiche (1.4142135623730951, 0.78539816339744828)
print(coordonnees_polaires(1, 1))
```

>>>

```
    (1.4142135623730951, 0.7853981633974483)
```

La fonction `compile` prend en fait trois arguments. Le premier est la chaîne de caractères contenant le code à compiler. Le second paramètre (`""` dans l'exemple) contient un nom de fichier dans lequel seront placées les erreurs de compilation. Le troisième paramètre est une chaîne de caractères à choisir parmi ńăexecă" ou ńăevală". Selon ce choix, ce sera la fonction `exec` ou `eval` qui devra être utilisée pour agréger le résultat de la fonction `compile` au programme. L'exemple suivant donne un exemple d'utilisation de la fonction `compile` avec la fonction `eval`.

<<<

72. https://docs.python.org/3/library/functions.html?highlight=eval#eval
73. https://docs.python.org/3/library/functions.html?highlight=eval#compile
74. https://docs.python.org/3/library/functions.html?highlight=eval#exec

```
import math
str = """math.sqrt(x*x+y*y)"""  # expression définie par une chaîne de caractères

obj = compile(str, "", "eval")    # expression compilée
x = 1
y = 2
print(eval(obj))                # résultat de l'expression
```

>>>

```
    2.23606797749979
```

2.2.5 Indentation

L'indentation est synonyme de décalage. Pour toute boucle, test, fonction, et plus tard, toute définition de classe, le fait d'indenter ou décaler les lignes permet de définir une dépendance d'un bloc de lignes par rapport à un autre. Les lignes indentées par rapport à une boucle `for` dépendent de celle-ci puisqu'elle seront exécutées à chaque passage dans la boucle. Les lignes indentées par rapport au mot-clé `def` sont considérées comme faisant partie du corps de la fonction.

IndentationError [75] est l'erreur que l'interpréteur *python* retourne en cas de mauvaise indentation (voir *tests* (page 39)).

Contrairement à d'autres langages comme le C [76] ou PERL [77], *python* n'utilise pas de délimiteurs pour regrouper les lignes. L'indentation, souvent présentée comme un moyen de rendre les programmes plus lisibles, est ici intégrée à la syntaxe du langage. Il n'y a pas non plus de délimiteurs entre deux instructions autre qu'un passage à la ligne. Le caractère \ placé à la fin d'une ligne permet de continuer l'écriture d'une instruction à la ligne suivante.

2.2.6 Fonctions usuelles

Certaines fonctions sont communes aux dictionnaires et aux listes, elles sont également définis pour de nombreux objets présents dans les extensions du langages. Quelque soit le contexte, le résultat attendu à la même signification. Les plus courantes sont présentées *plus bas* (page 73).

La fonction map [78] permet d'écrire des boucles de façon simplifiée. Elle est utile dans le cas où on souhaite appliquer la même fonction à tous les éléments d'un ensemble. Par exemple les deux dernières lignes du programme suivant sont équivalentes.

<<<

```
def est_pair(n):
    return n % 2 == 0


l = [0, 3, 4, 4, 5, 6]
print([est_pair(i) for i in l])  # affiche [0, 1, 0, 0, 1, 0]
print(map(est_pair, l))
print(list(map(est_pair, l)))       # affiche [0, 1, 0, 0, 1, 0]
```

>>>

```
    [True, False, True, True, False, True]
    <map object at 0x7f23da0f6b38>
    [True, False, True, True, False, True]
```

75. https://docs.python.org/3/library/exceptions.html?highlight=indentationerror#IndentationError
76. https://fr.wikipedia.org/wiki/C_(langage)
77. https://fr.wikipedia.org/wiki/Perl_(langage)
78. https://docs.python.org/3/library/functions.html?highlight=map#map

La fonction map [79] retourne un itérateur et non un ensemble. Cela explique le second résultat du programme précédent. Pour obtenir les résultats, il faut explicitement parcourir l'ensemble des résultats. C'est ce que fait la dernière instruction. La fonction map [80] est une *fonction générateur* (page 68). Elle peut aider à simplifier l'écriture lorsque plusieurs listes sont impliquées. Ici encore, les deux dernières lignes sont équivalentes.

<<<

```
def addition(x, y):
    return x + y


li = [0, 3, 4, 4, 5, 6]
mo = [1, 3, 4, 5, 6, 8]
print([addition(li[i], mo[i]) for i in range(0, len(li))])
print(list(map(addition, li, mo)))   # affiche [1, 6, 8, 9, 11, 14]
```

>>>

```
    [1, 6, 8, 9, 11, 14]
    [1, 6, 8, 9, 11, 14]
```

Il est possible de substituer d'utiliser la fonction `map` pour obtenir l'équivalent de la fonction zip [81].

<<<

```
li = [0, 3, 4, 4, 5, 6]
mo = [1, 3, 4, 5, 6, 8]
print(list(map((lambda x, y: (x, y)), li, mo)))
print(list(zip(li, mo)))
```

>>>

```
    [(0, 1), (3, 3), (4, 4), (4, 5), (5, 6), (6, 8)]
    [(0, 1), (3, 3), (4, 4), (4, 5), (5, 6), (6, 8)]
```

Comme pour les dictionnaires, la fonction sorted [82] permet de parcourir les éléments d'une liste de façon ordonnée. Les deux exemples qui suivent sont presque équivalents. Dans le second, la liste `li` demeure inchangée alors qu'elle est triée dans le premier programme.

<<<

```
li = [4, 5, 3, -6, 7, 9]

for n in sorted(li):     # on parcourt la liste li
    print(n)             # de façon triée
print(li)                # la liste li n'est pas triée

li.sort()                # la liste est triée
for n in li:
    print(n)
```

>>>

```
    -6
    3
```

(suite sur la page suivante)

79. https://docs.python.org/3/library/functions.html?highlight=map#map
80. https://docs.python.org/3/library/functions.html?highlight=map#map
81. https://docs.python.org/3/library/functions.html?highlight=map#zip
82. https://docs.python.org/3/library/functions.html?highlight=map#sorted

(suite de la page précédente)

```
    4
    5
    7
    9
    [4, 5, 3, -6, 7, 9]
    -6
    3
    4
    5
    7
    9
```

La fonction enumerate [83] permet d'éviter l'emploi de la fonction range [84] lorsqu'on souhaite parcourir une liste alors que l'indice et l'élément sont nécessaires.

<<<

```
li = [4, 5, 3, -6, 7, 9]

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

print("--")

for i, v in enumerate(li):
    print(i, v)
```

>>>

```
    0 4
    1 5
    2 3
    3 -6
    4 7
    5 9
    --
    0 4
    1 5
    2 3
    3 -6
    4 7
    5 9
```

Voici la liste non exhaustive de fonctions définies par le langage *python* sans qu'aucune extension ne soit nécessaire.

83. https://docs.python.org/3/library/functions.html?highlight=map#enumerate
84. https://docs.python.org/3/library/functions.html?highlight=map#func-range

`abs (x)`	Retourne la valeur absolue de `x`.
`callable (x)`	Dit si la variable `x` peut être appelée.
`chr (i)`	Retourne le caractère associé au code numérique `i`.
`cmp (x,y)`	Compare `x` et `y`, retourne -1 si `x<y`, 0 en cas d'égalité, 1 sinon.
`dir (x)`	Retourne l'ensemble des méthodes associées à `x` qui peut être un objet, un module, un variable, ...
`enumerate(x)`	Parcourt un ensemble itérable (voir paragraphe fonction_sorted_enumerate.
`help(x)`	Retourne l'aide associée à `x`.
`id(x)`	Retourne un identifiant unique associé à l'objet `x`. Le mot-clé `is` est relié à cet identifiant.
`isinstance(x, classe)`	Dit si l'objet `x` est de type `classe` (voir le chapitre *Classes* (page 95)).
`issubclass(cl1, cl2)`	Dit si la classe `cl1` hérite de la classe `cl2` (voir le chapitre *Classes* (page 95)).
`len(l)`	Retourne la longueur de `l`.
`map(f,l1,l2,...)`	Applique la fonction `f` sur les listes `l1`, `l2`...
`max(l)`	Retourne le plus grand élément de `l`.
`min(l)`	Retourne le plus petit élément de `l`.
`ord(s)`	Fonction réciproque de `chr`.
`range(i,j[,k])`	Construit la liste des entiers de `i` à `j`. Si `k` est précisé, va de `k` en `k` à partir de `i`.
`reload(module)`	Recharge un module (voir *Modules* (page **??**)).
`repr(o)`	Retourne une chaîne de caractères qui représente l'objet `o`.
`round(x[,n])`	Arrondi `x` à `n` décimales près ou aucune si `n` n'est pas précisé.
`sorted(x [, cmp[, key[, reverse]]])`	Tri un ensemble itérable (voir paragraphe fonction_sorted_enumerate)
`str(o)`	Retourne une chaîne de caractères qui représente l'objet `o`.
`sum(l)`	Retourne la somme de l'ensemble `l`.
`type(o)`	Retourne le type de la variable `o`.
`zip(l1,l2,...)`	Construit une liste de tuples au lieu d'un tuple de listes.

2.2.7 Constructions classiques

Il fait aller à *Constructions classiques* (page 74).

Ces paragraphes qui suivent décrivent des schémas qu'on retrouve dans les programmes dans de nombreuses situations. Ce sont des combinaisons simples d'une ou deux boucles, d'un test, d'une liste, d'un dictionnaire.

2.3 Constructions classiques

2.3.1 Constructions classiques

1 *calcul d'une somme* (page **??**)
2 *calcul de la somme des dix premiers entiers au carré* (page **??**)
3 *comptage* (page **??**)
4 *conversion d'un vecteur en une matrice* (page **??**)

5 *conversion d'une chaîne de caractère en datetime* (page **??**)
6 *conversion d'une chaîne de caractère en matrice* (page **??**)
7 *conversion d'une matrice en chaîne de caractères* (page **??**)
8 *conversion d'une matrice en un vecteur* (page **??**)
9 *fonction comme paramètre* (page **??**)
10 *minimum avec position* (page **??**)
11 *recherche avec index* (page **??**)
12 *recherche dichotomique* (page **??**)
13 *tri, garder les positions initiales* (page **??**)

---

calcul d'une somme

Le calcul d'une somme fait toujours intervenir une boucle car le langage Python [85] ne peut faire des additions qu'avec deux nombres. Le schéma est toujours le même : initialisation et boucle.

<<<

```
li = [0, 434, 43, 6456]
s = 0                          # initialisation
for l in li:                   # boucle
    s += l                      # addition
print(s)
```

>>>

```
    6933
```

Ce code est équivalent à la fonction sum [86]. Dans ce cas où la somme intègre le résultat d'une fonction (au sens mathématique) et non les éléments d'une liste, il faudrait écrire :

<<<

```
def fonction(x):
    return x


li = [0, 434, 43, 6456]
s = 0
for l in li:
    s += fonction(l)
print(s)
```

>>>

```
    6933
```

Et ces deux lignes pourraient être résumées en une seule grâce à l'une de ces instructions :

<<<

```
def fonction(x):
    return x
```

(suite sur la page suivante)

85. http://www.python.org/
86. https://docs.python.org/3/library/functions.html#sum

(suite de la page précédente)

```
li = [0, 434, 43, 6456]
s1 = sum([fonction(l) for l in li])
s2 = sum(fonction(l) for l in li)
s3 = sum(map(fonction, li))
print(s1, s2, s3)
```

>>>

```
    6933 6933 6933
```

L'avantage des deux dernières instructions est qu'elles évitent la création d'une liste intermédiaire, c'est un point à prendre en compte si la liste sur laquelle opère la somme est volumineuse.

(entrée originale : construction_classique.py :docstring of teachpyx.examples.construction_classique.somme, line 6)

---

calcul de la somme des dix premiers entiers au carré

Ce calcul simple peut s'écrire de diffèrentes manières.

```
s = 0
for i in range(1,11):
    s += i**2
```

D'une façon abrégée :

```
s = sum ( [ i**2 for i in range(1,11) ] )
```

(entrée originale : classiques.py :docstring of teachpyx.examples.classiques.dix_entiers_carre, line 18)

---

comptage

On souhaite ici compter le nombre d'occurrences de chaque élément d'un tableau. Par exemple, on pourrait connaître par ce moyen la popularité d'un mot dans un discours politique ou l'étendue du vocabulaire utilisé. L'exemple suivant compte les mots d'une liste de mots.

<<<

```
li = ["un", "deux", "un", "trois"]
d = {}
for l in li:
    if l not in d:
        d[l] = 1
    else:
        d[l] += 1
print(d)    # affiche {'un': 2, 'trois': 1, 'deux': 1}
```

>>>

```
    {'un': 2, 'deux': 1, 'trois': 1}
```

La structure la plus appropriée ici est un dictionnaire puisqu'on cherche à associer une valeur à un élément d'une liste qui peut être de tout type. Si la liste contient des éléments de type modifiable comme une liste, il faudrait convertir ceux-ci en un type immuable comme une chaîne de caractères. L'exemple suivant illustre ce cas en comptant les occurrences des lignes d'une matrice.

<<<

```
mat = [[1, 1, 1], [2, 2, 2], [1, 1, 1]]
d = {}
for l in mat:
    k = str(l)     # k = tuple (l) lorsque cela est possible
    if k not in d:
        d[k] = 1
    else:
        d[k] += 1
print(d)    # affiche {'[1, 1, 1]': 2, '[2, 2, 2]': 1}
```

>>>

```
    {'[1, 1, 1]': 2, '[2, 2, 2]': 1}
```

Les listes ne peuvent pas être les clés du dictionnaire : Why Lists Can't Be Dictionary Keys [87].

On peut également vouloir non pas compter le nombre d'occurrence mais mémoriser les positions des éléments tous identiques. On doit utiliser un dictionnaire de listes :

<<<

```
li = ["un", "deux", "un", "trois"]
d = {}
for i, v in enumerate(li):
    if v not in d:
        d[v] = [i]
    else:
        d[v].append(i)
print(d)    # affiche {'un': [0, 2], 'trois': [3], 'deux': [1]}
```

>>>

```
    {'un': [0, 2], 'deux': [1], 'trois': [3]}
```

S'il suffit juste de compter, l'écriture la plus simple est :

<<<

```
r = {}
li = ["un", "deux", "un", "trois"]
for x in li:
    r[x] = r.get(x, 0) + 1
print(r)
```

>>>

```
    {'un': 2, 'deux': 1, 'trois': 1}
```

(entrée originale : construction_classique.py :docstring of teachpyx.examples.construction_classique.compte, line 6)

---

conversion d'un vecteur en une matrice

Dans un langage comme le *C++*, il arrive fréquemment qu'une matrice ne soit pas représentée par une liste de listes mais par une seule liste car cette représentation est plus efficace. Il faut donc convertir un indice en deux indices ligne et colonne. Il faut bien sûr que le nombre de colonnes sur chaque ligne soit constant. Le premier programme convertit une liste de listes en une seule liste.

<<<

87. https://wiki.python.org/moin/DictionaryKeys

```
ncol = 2
vect = [0, 1, 2, 3, 4, 5]
mat = [vect[i*ncol: (i+1)*ncol] for i in range(0, len(vect)//ncol)]
print(mat)
```

>>>

```
    [[0, 1], [2, 3], [4, 5]]
```

(entrée originale : construction_classique.py :docstring of teachpyx.examples.construction_classique.vect2mat, line 8)

---

conversion d'une chaîne de caractère en datetime

C'est le genre de fonction qu'on n'utilise pas souvent mais qu'on peine à retrouver lorsqu'on en a besoin. Il faut utiliser la fonction strftime [88].

```
import datetime
dt = datetime.datetime.strptime ("16/01/2014", "%d/%m/%Y")
```

(entrée originale : classiques.py :docstring of teachpyx.examples.classiques.str2date, line 7)

---

conversion d'une chaîne de caractère en matrice

Les quelques lignes qui suivent permettent de décomposer une chaîne de caractères en matrice. Chaque ligne et chaque colonne sont séparées par des séparateurs différents. Ce procédé intervient souvent lorsqu'on récupère des informations depuis un fichier texte lui-même provenant d'un tableur.

<<<

```
s = "case11;case12;case13|case21;case22;case23"
# décomposition en matrice
ligne = s.split("|")                        # lignes
mat = [l.split(";") for l in ligne]  # colonnes

print(mat)
```

>>>

```
    [['case11', 'case12', 'case13'], ['case21', 'case22', 'case23']]
```

Comme cette opération est très fréquente lorsqu'on travaille avec les données, on ne l'implémente plus soi-même. On préfère utiliser un module comme pandas [89] qui est plus robuste et considère plus de cas. Pour écrire, utilise la méthode to_csv [90], pour lire, la fonction read_csv [91]. On peut également directement enregistrer au format Excel read_excel [92] et écrire dans ce même format to_excel [93].

(entrée originale : construction_classique.py :docstring of teachpyx.examples.construction_classique.text2mat, line 9)

---

conversion d'une matrice en chaîne de caractères

<<<

88. https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior
89. http://pandas.pydata.org/
90. http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_csv.html
91. http://pandas.pydata.org/pandas-docs/stable/generated/pandas.io.parsers.read_csv.html
92. http://pandas.pydata.org/pandas-docs/stable/generated/pandas.io.excel.read_excel.html
93. http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_excel.html

```
mat = [['case11', 'case12', 'case13'], ['case21', 'case22', 'case23']]
ligne = [";".join(l) for l in mat]     # colonnes
s = "|".join(ligne)                    # lignes

print(s)
```

>>>

```
    case11;case12;case13|case21;case22;case23
```

(entrée originale : construction_classique.py :docstring of teachpyx.examples.construction_classique.mat2text, line 9)

---

conversion d'une matrice en un vecteur

Dans un langage comme le *C++*, il arrive fréquemment qu'une matrice ne soit pas représentée par une liste de listes mais par une seule liste car cette représentation est plus efficace. Il faut donc convertir un indice en deux indices ligne et colonne. Il faut bien sûr que le nombre de colonnes sur chaque ligne soit constant. Le premier programme convertit une liste de listes en une seule liste.

<<<

```
mat = [[0, 1, 2], [3, 4, 5]]
lin = [i * len(mat[i]) + j
       for i in range(0, len(mat))
       for j in range(0, len(mat[i]))]
print(lin)
```

>>>

```
    [0, 1, 2, 3, 4, 5]
```

Vous pouvez aussi utiliser des fonctions telles que reduce [94].

<<<

```
from functools import reduce
mat = [[0, 1, 2], [3, 4, 5]]
lin = reduce(lambda x, y: x+y, mat)
print(lin)
```

>>>

```
    [0, 1, 2, 3, 4, 5]
```

(entrée originale : construction_classique.py :docstring of teachpyx.examples.construction_classique.mat2vect, line 7)

---

fonction comme paramètre

Une fonction peut aussi recevoir en paramètre une autre fonction. L'exemple suivant inclut la fonction `calcul_n_valeur` qui prend comme paramètres `l` et `f`. Cette fonction calcule pour toutes les valeurs `x` de la liste `l` la valeur `f(x)`. `fonction_carre` ou `fonction_cube` sont passées en paramètres à la fonction `calcul_n_valeur` qui les exécute.

<<<

94. https://docs.python.org/3/library/functools.html?highlight=reduce#module-functools

```
def fonction_carre(x):
    return x*x


def fonction_cube(x):
    return x*x*x


def calcul_n_valeur(l, f):
    res = [f(i) for i in l]
    return res


l = [0, 1, 2, 3]
print(l)   # affiche [0, 1, 2, 3]

l1 = calcul_n_valeur(l, fonction_carre)
print(l1)  # affiche [0, 1, 4, 9]

l2 = calcul_n_valeur(l, fonction_cube)
print(l2)  # affiche [0, 1, 8, 27]
```

>>>

```
    [0, 1, 2, 3]
    [0, 1, 4, 9]
    [0, 1, 8, 27]
```

(entrée originale : construction_classique.py :docstring of teachpyx.examples.construction_classique.integrale, line 10)

---

minimum avec position

La fonction min[95] retourne le minium d'un tableau mais pas sa position. Le premier réflexe est alors de recoder le parcours de la liste tout en conservant la position du minimum.

<<<

```
li = [0, 434, 43, 6436, 5]
m = 0
for i in range(0, len(li)):
    if li[m] < li[i]:
        m = i
print(m)
```

>>>

```
    3
```

Mais il existe une astuce pour obtenir la position sans avoir à le reprogrammer.

<<<

```
li = [0, 434, 43, 6436, 5]
k = [(v, i) for i, v in enumerate(li)]
m = min(k)
print(m)
```

95. https://docs.python.org/3/library/functions.html#min

>>>

```
    (0, 0)
```

La fonction `min` choisit l'élément minimum d'un tableau dont les éléments sont des couples (élément du premier tableau, sa position). Le minimum est choisi en comparant les éléments, et la position départegera les exaequo.

(entrée originale : construction_classique.py :docstring of teachpyx.examples.construction_classique.minindex, line 7)

---

recherche avec index

Lorsqu'on cherche un élément dans un tableau, on cherche plus souvent sa position que le fait que le tableau contient cet élément.

<<<

```
def recherche(li, c):
    for i, v in enumerate(li):
        if v == c:
            return i
    return -1


li = [45, 32, 43, 56]
print(recherche(li, 43))  # affiche 2
```

>>>

```
    2
```

En python, il existe un fonction simple qui permet de faire ça :

```
print(li.index(43))  # affiche 2
```

Lorsque l'élément n'y est pas, on retourne souvent la position `-1` qui ne peut être prise par aucun élément :

```
if c in li: return li.index(c)
else: return -1
```

Même si ce bout de code parcourt deux fois le tableau (une fois déterminer sa présence, une seconde fois pour sa position), ce code est souvent plus rapide que la première version et la probabilité d'y faire une erreur plus faible.

(entrée originale : construction_classique.py :docstring of teachpyx.examples.construction_classique.recherche, line 7)

---

recherche dichotomique

La recherche dichotomique [96] est plus rapide qu'une recherche classique mais elle suppose que celle-ci s'effectue dans un ensemble trié. L'idée est de couper en deux l'intervalle de recherche à chaque itération. Comme l'ensemble est trié, en comparant l'élément cherché à l'élément central, on peut éliminer une partie de l'ensemble : la moitié inférieure ou supérieure.

<<<

96. http://fr.wikipedia.org/wiki/Dichotomie

```
def recherche_dichotomique(li, c):
    a, b = 0, len(li)-1
    while a <= b:
        m = (a + b)//2
        if c == li[m]:
            return m
        elif c < li[m]:
            b = m-1
        else:
            a = m+1
    return -1


print(recherche_dichotomique([0, 2, 5, 7, 8], 7))
```

>>>

```
    3
```

(entrée originale : construction_classique.py :docstring of teachpyx.examples.construction_classique.recherche_dichotomique, line 7)

tri, garder les positions initiales

Le tri est une opération fréquente. On n'a pas toujours le temps de programmer le tri le plus efficace comme un tri quicksort[97] et un tri plus simple suffit la plupart du temps. Le tri suivant consiste à recherche le plus petit élément puis à échanger sa place avec le premier élément du tableau du tableau. On recommence la même procédure à partir de la seconde position, puis la troisième et ainsi de suite jusqu'à la fin du tableau.

<<<

```
li = [5, 6, 4, 3, 8, 2]

for i in range(0, len(li)):
    # recherche du minimum entre i et len (li) exclu
    pos = i
    for j in range(i+1, len(li)):
        if li[j] < li[pos]:
            pos = j
    # échange
    ech = li[pos]
    li[pos] = li[i]
    li[i] = ech

print(li)
```

>>>

```
    [2, 3, 4, 5, 6, 8]
```

La fonction sorted[98] trie également une liste mais selon un algorithme plus efficace que celui-ci (voir Timsort[99]). On est parfois amené à reprogrammer un tri parce qu'on veut conserver la position des éléments dans le tableau non trié. Cela arrive quand on souhaite trier un tableau et appliquer la même transformation à un second tableau. Il est toujours

97. http://fr.wikipedia.org/wiki/Tri_rapide
98. https://docs.python.org/3/library/functions.html#sorted
99. http://en.wikipedia.org/wiki/Timsort

préférable de ne pas reprogrammer un tri (moins d'erreur). Il suffit d'appliquer la même idée que pour la fonction *minindex* (page **??**).

<<<

```
tab = ["zero", "un", "deux"]                          # tableau à trier
pos = sorted((t, i) for i, t in enumerate(tab))   # tableau de couples
print(pos)                 # affiche [('deux', 2), ('un', 1), ('zero', 0)]
```

>>>

```
    [('deux', 2), ('un', 1), ('zero', 0)]
```

Si cette écriture est trop succincte, on peut la décomposer en :

<<<

```
tab = ["zero", "un", "deux"]
tab_position = [(t, i) for i, t in enumerate(tab)]
tab_position.sort()
print(tab_position)
```

>>>

```
    [('deux', 2), ('un', 1), ('zero', 0)]
```

(entrée originale : construction_classique.py :docstring of teachpyx.examples.construction_classique.triindex, line 6)

2.3.2 Constructions négatives

1 *Eviter d'effectuer le même appel deux fois* (page **??**)

2 *Modifier un dictionnaire en le parcourant* (page **??**)

Eviter d'effectuer le même appel deux fois

Dans cette fonction on calcule la variance d'une série d'observations.

```
def moyenne(serie):
    return sum(serie) / len(serie)

def variance_a_eviter(serie):
    s = 0
    for obs in serie :
        s += (obs-moyenne(serie))**2
    return s / len(serie)
```

La fonction `variance_a_eviter` appelle la fonction `moyenne` à chaque passage dans la boucle. Or, rien ne change d'un passage à l'autre. Il vaut mieux stocker le résultat dans une variable :

(entrée originale : classiques.py :docstring of teachpyx.examples.classiques.repetition_a_eviter, line 3)

Modifier un dictionnaire en le parcourant

Il faut éviter de modifier un container lorsqu'on le parcourt. Lorsqu'on supprime un élément d'un dictionnaire, la structure de celui-ci s'en trouve modifiée et affecte la boucle qui le parcourt. La boucle parcourt toujours l'ancienne structure du dictionnaire, celle qui existait au début au début de la boucle.

```
d = { k: k for k in range(10) }
for k, v in d.items():
    if k == 4 :
        del d[k]
```

En Python, cela produit l'erreur qui suit mais d'autres langages ne préviennent pas (C++) et cela aboutit à une erreur qui intervient plus tard dans le code (comme une valeur numérique inattendue).

```
Traceback (most recent call last):
File "session1.py", line 176, in <module>
    l = liste_modifie_dans_la_boucle()
File "session1.py", line 169, in liste_modifie_dans_la_boucle
    for k,v in d.items():
RuntimeError: dictionary changed size during iteration
```

Il faut pour éviter cela stocker les éléments qu'on veut modifier pour les supprimer ensuite.

```
d = { k:k for k in l }
rem = [ ]
for k,v in d.items():
    if k == 4 :
        rem.append(k)
for r in rem :
    del d[r]
```

Même si Python autorise cela pour les listes, il est conseillé de s'en abstenir ainsi que pour tout type d'objets qui en contient d'autres. C'est une habitude qui vous servira pour la plupart des autres langages.

(entrée originale : classiques.py :docstring of teachpyx.examples.classiques.dictionnaire_modifie_dans_la_boucle, line 3)

2.4 Dates

2.4.1 datetime

Le module datetime [100] fournit une classe datetime [101] qui permet de faire des opérations et des comparaisons sur les dates et les heures. L'exemple suivant calcule l'âge d'une personne née le 11 août 1975.

<<<

```
import datetime
naissance = datetime.datetime(1975, 11, 8, 10, 0, 0)
jour = naissance.now()  # obtient l'heure et la date actuelle
print(jour)             # affiche quelque chose comme 2017-05-22 11:24:36.312000
age = jour - naissance  # calcule une différence
print(age)              # affiche 12614 days, 1:25:10.712000
```

>>>

100. https://docs.python.org/3/library/datetime.html?highlight=datetime#module-datetime
101. https://docs.python.org/3/library/datetime.html?highlight=datetime#datetime.datetime

```
    2018-11-12 07:57:23.874516
    15709 days, 21:57:23.874516
```

L'objet datetime [102] autorise les soustractions et les comparaisons entre deux dates. Une soustraction retourne un objet de type timedelta [103] qui correspond à une durée qu'on peut multiplier par un réel ou ajouter à un objet de même type ou à un objet de type datetime [104]. L'utilisation de ce type d'objet évite de se pencher sur tous les problèmes de conversion.

Le module calendar [105] est assez pratique pour construire des calendriers. Le programme ci-dessous affiche une liste de t-uples incluant le jour et le jour de la semaine du mois d'août 1975. Dans cette liste, on y trouve le t-uple `(11,0)` qui signifie que le 11 août 1975 était un lundi. Cela permet de récupérer le jour de la semaine d'une date de naissance.

<<<

```
import calendar
c = calendar.Calendar()
for d in c.itermonthdays2(1975, 8):
    print(d)
```

>>>

```
    (0, 0)
    (0, 1)
    (0, 2)
    (0, 3)
    (1, 4)
    (2, 5)
    (3, 6)
    (4, 0)
    (5, 1)
    (6, 2)
    (7, 3)
    (8, 4)
    (9, 5)
    (10, 6)
    (11, 0)
    (12, 1)
    (13, 2)
    (14, 3)
    (15, 4)
    (16, 5)
    (17, 6)
    (18, 0)
    (19, 1)
    (20, 2)
    (21, 3)
    (22, 4)
    (23, 5)
    (24, 6)
    (25, 0)
    (26, 1)
    (27, 2)
    (28, 3)
    (29, 4)
```

(suite sur la page suivante)

102. https://docs.python.org/3/library/datetime.html?highlight=datetime#datetime.datetime
103. https://docs.python.org/3/library/datetime.html?highlight=timedelta#datetime.timedelta
104. https://docs.python.org/3/library/datetime.html?highlight=datetime#datetime.datetime
105. https://docs.python.org/3/library/calendar.html?highlight=calendar#module-calendar

(suite de la page précédente)

```
    (30, 5)
    (31, 6)
```

2.4.2 Autres formats de date

La date d'un événement est constamment utilisée, la date de modification d'un fichier par exemple pour détecter que celui-ci a été modifié depuis sa synchronisation avec un site web par exemple. On appelle ces dates associées à des événements des timestamp [106]. Ce sont des nombres entiers qui expriment le nombre de secondes qui se sont écoulées depuis une certaine origine qui dépend du système d'exploitation. C'est un nombre réel : la partie entière est un nombre de seconde, la partie décimale donne les millisecondes. La valeur n'a pas de sens exploitable à moins de la convertir en un format compréhensible. C'est ce que fait la fonction fromtimestamp [107].

<<<

```
import time
ts = time.time()
print(ts)

import datetime
st = datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d %H:%M:%S')
print(st)
```

>>>

```
    1542005843.9746695
    2018-11-12 07:57:23
```

2.5 Encoding et jeux de caractères

2.5.1 Jeux de caractères

La langue anglaise est la langue dominante en ce qui concerne l'informatique mais cela n'empêche pas que des programmes anglais manipulent du japonais même si le nombre de caractères est beaucoup plus grand. Les caractères les plus fréquent peuvent être représentés par un octet, soit au plus 256 caractères. Lorsque la langue en contient plus, il faut utiliser plusieurs octets pour un caractères. C'est à ça ce que servent les jeux de caractères : ils définissent la façon de passer d'une suite d'octets à une séquence de caractères et réciproquement. Pour être plus précis, le jeu de caractère désigne l'ensemble de caractères dont le programme a besoin, l'encodage décrit la manière dont on passe d'une séquence de caractères français, japonais, anglais à une séquence d'octets qui est la seule information manipulée par un ordinateur. Le décodage définit le passage inverse. Les langues latines n'ont besoin que d'un octet pour coder un caractère, les langues asiatiques en ont besoin de plusieurs. Il n'existe pas qu'un seul jeu de caractères lorsqu'on programme. Ils interviennent à plusieurs endroits différents :

— Le jeu de caractères utilisé par l'éditeur de texte pour afficher le programme.

106. https://fr.wikipedia.org/wiki/Horodatage
107. https://docs.python.org/3/library/datetime.html?highlight=fromtimestamp#datetime.datetime.fromtimestamp

— Le jeu de caractères du programme, par défaut ascii [108] mais il peut être changé en insérant une première ligne de commentaire (exemple : `# -*- coding: utf-8 -*-`). Les chaînes de caractères du programme sont codées avec ce jeu de caractères. Ce jeu devrait être identique à celui utilisé par l'éditeur de texte afin d'éviter les erreurs.
— Le jeu de caractères de la sortie, utilisé pour chaque instruction `print`, il est désigné par le code cp1252 [109] sur un système *Windows*.
— Le jeu de caractères dans lequel les chaînes de caractères sont manipulées. Un jeu standard qui permet de représenter toutes les langues est le jeu de caractères utf-8 [110]. Il peut être différent pour chaque variable.
— Le jeu de caractères d'un fichier texte. Il peut être différent pour chaque fichier.

Depuis la version 3 de *Python*, toutes les chaînes de caractères sont au format unicode [111]. C'est à dire que le jeu de caractères est le même pour toutes les chaînes de caractères. Pour en changer, il faut *encoder* le texte avec un jeu spécifique et la fonction encode [112]. Il deviendra une chaîne d'octets ou bytes [113]. La transformation inverse s'effectue avec la méthode decode [114].

<<<

```
st = "eé"
print(type(st))
print(len(st))

sb = st.encode("latin-1")
print(type(sb))
print(len(sb))
```

>>>

```
    <class 'str'>
    2
    <class 'bytes'>
    2
```

L'exemple précédent montre que la fonction len [115] retourne le nombre de caractères mais cela ne correspond pas au nombre d'octets que cette chaîne occupe en mémoire.

<<<

```
st = "eé"
print(type(st))
print(len(st))

sb = st.encode("utf-16")
print(type(sb))
print(len(sb))
```

>>>

```
    <class 'str'>
    2
    <class 'bytes'>
    6
```

Un autre jeu, une autre longueur.

108. https://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange
109. https://fr.wikipedia.org/wiki/Windows-1252
110. https://fr.wikipedia.org/wiki/UTF-8
111. https://fr.wikipedia.org/wiki/Unicode
112. https://docs.python.org/3/library/stdtypes.html?highlight=encode#str.encode
113. https://docs.python.org/3/library/functions.html?highlight=bytes#bytes
114. https://docs.python.org/3/library/stdtypes.html?highlight=encode#bytes.decode
115. https://docs.python.org/3/library/functions.html?highlight=len#len

2.5.2 Fichiers

Comme *Python* utilise un jeu unique, il suffit de faire attention où on récupère les chaînes de caractères et au moment où on les transfère. C'est ce qu'on appelle les entrées sorties comme lire ou écrire un fichier. Par défaut un fichier est écrit avec le jeu de caractères du système d'exploitation. Pour un préciser un autre, il faut spécifier le paramètre *encoding*. Toutes les chaînes de caractères seront lues et converties au format *unicode* depuis ou vers l'encoding spécifié.

```
with open("essai.txt", "r", "utf-8") as f:
    text = f.read()
```

2.5.3 Méthodes

`encode( [enc[,err]])`	Cette fonction permet de passer d'un jeu de caractères, celui de la variable, au jeu de caractères précisé par *enc* à moins que ce ne soit le jeu de caractères par défaut. Le paramètre *err* permet de préciser comment gérer les erreurs, doit-on interrompre le programme (valeur `'strict'` ou les ignorer (valeur `'ignore'`). La documentation *Python* recense toutes les valeurs possibles pour ces deux paramètres aux adresses
`decode([enc[, err]]`	Cette fonction est la fonction inverse de la fonction `encode`. Avec les mêmes paramètres, elle effectue la transformation inverse.

2.5.4 Encodings par défaut

Le programme suivant permet d'obtenir le jeu de caractères par défaut et celui du système d'exploitation.

<<<

```
import sys
import locale
import platform
print(sys.platform)
print(platform.architecture)
print(sys.getdefaultencoding())
print(locale.getdefaultlocale())
```

>>>

```
    linux
    <function architecture at 0x7f23f2bb6ae8>
    utf-8
    ('en_US', 'UTF-8')
```

Les problèmes d'encoding surviennent parfois car on précise rarement l'encoding du programme *Python* ni le programmeur ne contrôle pas facilement celui de la sortie (`print`). Ces deux paramètres changent selon les éditeurs ou les systèmes d'exploitations.

2.6 FAQ

— *FAQ* (page 89)

2.6.1 FAQ

A quoi sert un ‘‘StringIO‘‘ ?

La plupart du temps, lorsqu’on récupère des données, elles sont sur le disque dur de votre ordinateur dans un fichier texte. Lorsqu’on souhaite automatiser un processur qu’on répète souvent avec ce fichier, on écrit une fonction qui prend le nom du fichier en entrée.

```
def processus_quotidien(nom_fichier) :
    # on compte les lignes
    nb = 0
    with open(nom_fichier,"r") as f :
        for line in f :
            nb += 1
    return nb
```

Et puis un jour, les données ne sont plus dans un fichier mais sur Internet. Le plus simple dans ce cas est de recopier ces données sur disque dur et d’appeler la même fonction. Simple. Un autre les données qu’on doit télécharger font plusieurs gigaoctets. Tout télécharger prend du temps pour finir pour s’apercevoir qu’elles sont corrompues. On a perdu plusieurs heures pour rien. On aurait bien voulu que la fonction `processus_quotidien` commence à traiter les données dès le début du téléchargement.

Pour cela, on a inventé la notion de **stream** ou **flux** qui sert d’interface entre la fonction qui traite les données et la source des données. Le flux lire les données depuis n’importe quel source (fichier, internet, mémoire), la fonction qui les traite n’a pas besoin d’en connaître la provenance.

StringIO [116] est un flux qui considère la mémoire comme source de données.

```
def processus_quotidien(data_stream):
    # on compte toujours les lignes
    nb = 0
    for line in data_stream :
        nb += 1
    return nb
```

La fonction `processus_quotidien` fonctionne pour des données en mémoire et sur un fichier.

```
fichier = __file__
f = open(fichier,"r")
nb = processus_quotidien(f)
print(nb)

text = "ligne1
```

116. https://docs.python.org/3/library/io.html#io.StringIO

(entrée originale : faq_python.py :docstring of teachpyx.faq.faq_python.stringio, line 6)

Pourquoi l'installation de pandas (ou numpy) ne marche pas sous Windows avec pip ?

Python est un langage très lent et c'est pourquoi la plupart des modules de calculs numériques incluent des parties implémentées en langage C++. numpy [117], pandas [118], matplotlib [119], scipy [120], scikit-learn [121], ...

Sous Linux, le compilateur est intégré au système et l'installation de ces modules via l'instruction `pip install <module>` met implicitement le compilateur à contribution. Sous Windows, il n'existe pas de compilateur C++ par défaut à moins de l'installer. Il faut faire attention alors d'utiliser exactement le même que celui utilisé pour compiler Python (voir Compiling Python on Windows [122]).

C'est pour cela qu'on préfère utiliser des distributions comme Anaconda [123] qui propose par défaut une version de Python accompagnée des modules les plus utilisés. Elle propose également une façon simple d'installer des modules précompilés avec l'instruction

```
conda install <module_compile>
```

L'autre option est d'utilser le site Unofficial Windows Binaries for Python Extension Packages [124] qui propose des versions compilées sous Windows d'un grand nombre de modules. Il faut télécharger le fichier *.whl* puis l'installer avec l'instruction `pip install <fichier.whl>`. La différence entre les deux ooptions tient aux environnements virtuels, voir Python virtual environments [125].

(entrée originale : faq_python.py :docstring of teachpyx.faq.faq_python.information_about_package, line 17)

Python n'accepte pas les accents

Le langage Python a été conçu en langage anglais. Dès qu'on on ajoute un caractère qui ne fait pas partie de l'alphabet anglais (ponctuation comprise), il déclenche une erreur :

```
File "faq_cvxopt.py", line 3
SyntaxError: Non-UTF-8 code starting with 'è' in file faq_cvxopt.py on line 4, but no␣
↪encoding declared;
    see http://python.org/dev/peps/pep-0263/ for details
```

Pour la résoudre, il faut dire à l'interpréteur que des caractères non anglais peuvent apparaître et écrire sur la première ligne du programme :

```
# -*- coding: latin-1 -*-
```

Ou pour tout caractère y compris chinois :

```
# -*- coding: utf-8 -*-
```

Si vous utilisez l'éditeur SciTE [126] sous Windows, après avoir ajouté cette ligne avec l'encoding *utf-8*, il est conseillé de fermer le fichier puis de le réouvrir. SciTE le traitera différemment.

117. http://www.numpy.org/
118. http://pandas.pydata.org/
119. http://matplotlib.org/
120. http://www.scipy.org/
121. http://scikit-learn.org/stable/
122. https://docs.python.org/3/using/windows.html#compiling-python-on-windows
123. http://continuum.io/downloads#py34
124. http://www.lfd.uci.edu/~gohlke/pythonlibs/
125. http://astropy.readthedocs.org/en/stable/development/workflow/virtual_pythons.html
126. http://www.scintilla.org/SciTE.html

L'encodage ''utf-8'' est la norme sur Internet. C'est pourquoi il est préférable d'utiliser celui-ci pour partager son code via une page Web.

(entrée originale : classiques.py :docstring of teachpyx.examples.classiques.commentaire_accentues, line 3)

Qu'est-ce qu'un type immuable ou immutable ?

Une variable de type *immuable* ne peut être modifiée. Cela concerne principalement :

— `int`, `float`, `str`, `tuple`

Si une variable est de type *immuable*, lorsqu'on effectue une opération, on créé implicitement une copie de l'objet.

Les dictionnaires et les listes sont *modifiables* (ou *mutable*). Pour une variable de ce type, lorsqu'on écrit `a = b`, `a` et `b` désigne le même objet même si ce sont deux noms différentes. C'est le même emplacement mémoire accessible paur deux moyens (deux identifiants).

Par exemple

```
a  = (2,3)
b  = a
a += (4,5)
print( a == b ) # --> False
print(a,b)      # --> (2, 3, 4, 5) (2, 3)

a  = [2,3]
b  = a
a += [4,5]
print( a == b ) # --> True
print(a,b)      # --> [2, 3, 4, 5] [2, 3, 4, 5]
```

Dans le premier cas, le type (`tuple`) est _immutable_, l'opérateur += cache implicitement une copie. Dans le second cas, le type (`list`) est _mutable_, l'opérateur += évite la copie car la variable peut être modifiée. Même si `b=a` est exécutée avant l'instruction suivante, elle n'a **pas** pour effet de conserver l'état de `a` avant l'ajout d'élément. Un autre exemple

```
a  = [1, 2]
b  = a
a [0] = -1
print(a)        # --> [-1, 2]
print(b)        # --> [-1, 2]
```

Pour copier une liste, il faut expliciter la demander

```
a  = [1, 2]
b  = list(a)
a [0] = -1
print(a)        # --> [-1, 2]
print(b)        # --> [1, 2]
```

La page Immutable Sequence Types [127] détaille un peu plus le type qui sont *mutable* et ceux qui sont *immutable*. Parmi les types standards :

— **mutable**
  — bool [128]
  — int [129], float [130], complex [131]

127. https://docs.python.org/3/library/stdtypes.html?highlight=immutable#immutable-sequence-types
128. https://docs.python.org/3/library/functions.html#bool
129. https://docs.python.org/3/library/functions.html#int
130. https://docs.python.org/3/library/functions.html#float
131. https://docs.python.org/3/library/functions.html#complex

    - str [132], bytes [133]
    - None [134]
    - tuple [135], frozenset [136]
- **immutable, par défaut tous les autres types dont :**
    - list [137]
    - dict [138]
    - set [139]
    - bytearray [140]

Une instance de classe est mutable. Il est possible de la rendre immutable par quelques astuces :

- __slots__ [141]
- How to Create Immutable Classes in Python [142]
- Ways to make a class immutable in Python [143]
- freeze [144]

Enfin, pour les objects qui s'imbriquent les uns dans les autres, une liste de listes, une classe qui incluent des dictionnaires et des listes, on distingue une copie simple d'une copie intégrale (**deepcopy**). Dans le cas d'une liste de listes, la copie simple recopie uniquement la première liste

```
import copy
l1 = [ [0,1], [2,3] ]
l2 = copy.copy(l1)
l1 [0][0] = '##'
print(l1,l2)         # --> [['##', 1], [2, 3]] [['##', 1], [2, 3]]

l1 [0] = [10,10]
print(l1,l2)         # --> [[10, 10], [2, 3]] [['##', 1], [2, 3]]
```

La copie intégrale recopie également les objets inclus

```
import copy
l1 = [ [0,1], [2,3] ]
l2 = copy.deepcopy(l1)
l1 [0][0] = '##'
print(l1,l2)         # --> [['##', 1], [2, 3]] [[0, 1], [2, 3]]
```

Les deux fonctions s'appliquent à tout object Python : module copy [145].

(entrée originale : faq_python.py :docstring of teachpyx.faq.faq_python.same_variable, line 8)

Quel est l'entier le plus grand ?

La version 3 du langage Python a supprimé la constante `sys.maxint` qui définissait l'entier le plus grand (voir What's New In Python 3.0 [146]). De ce fait la fonction getrandbit [147] retourne un entier aussi grand que l'on veut.

132. https://docs.python.org/3/library/functions.html#func-str
133. https://docs.python.org/3/library/functions.html#bytes
134. https://docs.python.org/3/library/constants.html?highlight=none#None
135. https://docs.python.org/3/library/functions.html#func-tuple
136. https://docs.python.org/3/library/functions.html#func-frozenset
137. https://docs.python.org/3/library/functions.html#func-list
138. https://docs.python.org/3/library/functions.html#func-dict
139. https://docs.python.org/3/library/functions.html#func-set
140. https://docs.python.org/3/library/functions.html#bytearray
141. https://docs.python.org/3/reference/datamodel.html?highlight=_slots__#object.__slots__
142. http://www.blog.pythonlibrary.org/2014/01/17/how-to-create-immutable-classes-in-python/
143. http://stackoverflow.com/questions/4996815/ways-to-make-a-class-immutable-in-python
144. https://freeze.readthedocs.org/en/latest/
145. https://docs.python.org/3/library/copy.html
146. https://docs.python.org/3.1/whatsnew/3.0.html#integers
147. https://docs.python.org/3/library/random.html#random.getrandbits

```
import random,sys
x = random.getrandbits(2048)
print(type(x),x)
```

Qui affiche

```
<class 'int'>␣
↪28821592245571075131654830983838148370214474845580101472119213042190172126736565496812698627920 29.
↪..
```

Les calculs en nombre réels se font toujours avec huit octets de précision. Au delà, il faut utiliser la librairie gmpy2 [148]. Il est également recommandé d'utiliser cette librairie pour les grands nombres entiers (entre 20 et 40 chiffres). La librairie est plus rapide que l'implémentation du langage Python (voir Overview of gmpy2 [149]).

(entrée originale : faq_python.py :docstring of teachpyx.faq.faq_python.entier_grande_taille, line 1)

Quelle est la différence entre / et // - division ?

Le résultat de la division avec l'opérateur `/` est toujours réel : la division de deux entiers `1/2` donne `0.5`. Le résultat de la division avec l'opérateur `//` est toujours entier. Il correspond au quotient de la division.

```
div1 = 1/2
div2 = 4/2
div3 = 1//2
div4 = 1.0//2.0
print(div1,div2,div3,div4) # affiche (0.5, 2.0, 0, 0)
```

Le reste d'une division entière est obtenue avec l'opérateur `%`.

```
print ( 5 % 2 )  # affiche 1
```

C'est uniquement vrai pour les version Python 3.x. Pour les versions 2.x, les opérateurs `/` et `//` avaient des comportements différents (voir What's New In Python 3.0 [150]).

(entrée originale : faq_python.py :docstring of teachpyx.faq.faq_python.difference_div, line 1)

Quelle est la différence entre return et print ?

La fonction `print` sert à afficher un résultat sur la sortie standard. Elle peut être utilisée à tout moment mais elle n'a pas d'impact sur le déroulement programme. Le mot-clé `return` n'est utilisé que dans une fonction. Lorsque le programme rencontre une instruction commençant par `return`, il quitte la fonction et transmet le résultat à l'instruction qui a appelé la fonction. La fonction `print` ne modifie pas votre algorithme. La fonction `return` spécifie le résultat de votre fonction : elle modifie l'algorithme.

(entrée originale : classiques.py :docstring of teachpyx.examples.classiques.dix_entiers_carre, line 5)

Récupérer le nom du jour à partir d'une date

```
import datetime
dt = datetime.datetime(2016, 1, 1)
print(dt.strftime("%A"))
```

148. http://gmpy2.readthedocs.org/en/latest/
149. https://gmpy2.readthedocs.org/en/latest/overview.html
150. https://docs.python.org/3/whatsnew/3.0.html#integers

(entrée originale : faq_python.py :docstring of teachpyx.faq.faq_python.get_day_name, line 6)

Récupérer le nom du mois à partir d'une date

```
import datetime
dt = datetime.datetime(2016, 1, 1)
print(dt.strftime("%B"))
```

(entrée originale : faq_python.py :docstring of teachpyx.faq.faq_python.get_month_name, line 6)

Tabulations ou espace ?

Il est préférable de ne pas utiliser les tabulations et de les remplacer par des espaces. Lorsqu'on passe d'un Editeur à un autre, les espaces ne bougent pas. Les tabulations sont plus ou moins grandes visuellement. L'essentiel est de ne pas mélanger. Dans SciTE [151], il faut aller dans le menu Options / Change Indentation Settings... Tous les éditeurs ont une option similaire.

(entrée originale : faq_python.py :docstring of teachpyx.faq.faq_python.entier_grande_taille, line 27)

151. http://www.scintilla.org/SciTE.html

CHAPITRE 3

Classes et programmation objet

3.1 Classes

Imaginons qu'une banque détienne un fichier contenant des informations sur ses clients et qu'il soit impossible pour un client d'avoir accès directement à ces informations. Toutefois, il lui est en théorie possible de demander à son banquier quelles sont les informations le concernant détenues par sa banque. Il est en théorie également possible de les rectifier s'il estime qu'elles sont incorrectes.

On peut comparer cette banque à un objet qui possède des informations et des moyens permettant de lire ces informations et de les modifier. Vu de l'extérieur, cette banque cache son fonctionnement interne et les informations dont elle dispose, mais propose des services à ses utilisateurs.

On peut considérer la banque comme un *objet* au sens informatique. Ce terme désigne une entité possédant des données et des méthodes permettant de les manipuler. Plus concrètement, une classe est un assemblage de variables appelées *attributs* et de fonctions appelées *méthodes*. L'ensemble des propriétés associées aux classes est regroupé sous la désignation de *programmation objet*.

3.1.1 Présentation des classes : méthodes et attributs

Définition, déclaration

Définition D1 : classe

Une classe est un ensemble incluant des variables ou *attributs* et des fonctions ou *méthodes*. Les attributs sont des variables accessibles depuis toute méthode de la classe où elles sont définies. En *python*, les classes sont des types modifiables.

Syntaxe S1 : Déclaration d'une classe

```
class nom_classe :
    # corps de la classe
    # ...
```

Le corps d'une classe peut être vide, inclure des variables ou attributs, des fonctions ou méthodes. Il est en tout cas indenté de façon à indiquer à l'interpréteur *python* les lignes qui forment le corps de la classe.

Les classes sont l'unique moyen en langage *python* de définir de nouveaux types propres à celui qui programme. Il n'existe pas de type ńămatriceă" ou de type ńăgrapheă" en langage *python* qui soit prédéfini. Il est néanmoins possible de les définir au moyen des classes. Une matrice est par exemple un objet qui inclut les attributs suivant : le nombre de lignes, le nombre de colonnes, les coefficients de la matrice. Cette matrice inclut aussi des méthodes comme des opérations entre deux matrices telles que l'addition, la soustraction, la multiplication ou des opérations sur elle-même comme l'inversion, la transposition, la diagonalisation.

Cette liste n'est pas exhaustive, elle illustre ce que peut être une classe ńămatriceă" - représentation informatique d'un objet ńămatriceă" -, un type complexe incluant des informations de types variés (entier pour les dimensions, réels pour les coefficients), et des méthodes propres à cet objet, capables de manipuler ces informations.

Il est tout-à-fait possible de se passer des classes pour rédiger un programme informatique. Leur utilisation améliore néanmoins sa présentation et la compréhension qu'on peut en avoir. Bien souvent, ceux qui passent d'un langage uniquement fonctionnel à un langage objet ne font pas marche arrière. L¿**instantiation** se fait selon le schéma suivant :

Syntaxe S2 : Instanciation d'une classe

```
cl = nom_classe()
```

La création d'une variable de type objet est identique à celle des types standards du langage *python* : elle passe par une simple affectation. On appelle aussi `cl` une *instance* de la classe `nom_classe`.

Cette syntaxe est identique à la syntaxe d'appel d'une fonction. La création d'une instance peut également faire intervenir des paramètres (voir paragraphe *Opérateurs* (page 107)). Le terme *instance* va de paire avec le terme *classe* :

Définition D2 : instantiaion

Une instance d'une classe `C` désigne une variable de type `C`. Le terme instance ne s'applique qu'aux variables dont le type est une classe.

L'exemple suivant permet de définir une classe vide. Le mot-clé `pass` permet de préciser que le corps de la classe ne contient rien.

<<<

```
class classe_vide:
    pass
```

>>>

```

```

Il est tout de même possible de définir une instance de la classe `classe_vide` simplement par l'instruction suivante :

<<<

```
class classe_vide:
    pass


cl = classe_vide()
```

>>>

Dans l'exemple précédent, la variable `cl` n'est pas de type `exemple_classe` mais de type `instance` comme le montre la ligne suivante :

<<<

```
class classe_vide:
    pass


cl = classe_vide()
print(type(cl))     # affiche <type 'instance'>
```

>>>

```
    <class 'pyquickhelper.sphinxext.sphinx_runpython_extension.run_python_script_
↪139791945532688.<locals>.classe_vide'>
```

Pour savoir si une variable est une instance d'une classe donnée, il faut utiliser la fonction `isinstance` :

<<<

```
class classe_vide:
    pass


cl = classe_vide()
print(type(cl))                     # affiche <type 'instance'>
print(isinstance(cl, classe_vide))  # affiche True
```

>>>

```
    <class 'pyquickhelper.sphinxext.sphinx_runpython_extension.run_python_script_
↪139791943455656.<locals>.classe_vide'>
    True
```

Méthodes

Définition D3 : méthode

Les méthodes sont des fonctions qui sont associées de manière explicite à une classe. Elles ont comme particularité un accès privilégié aux données de la classe elle-même.

Ces données ou *attributs* sont définis plus loin. Les méthodes sont en fait des fonctions pour lesquelles la liste des paramètres contient obligatoirement un paramètre explicite qui est l'instance de la classe à laquelle cette méthode est associée. Ce paramètre est le moyen d'accéder aux données de la classe.

Syntaxe S3 : Déclaration d'une méthode

```
class nom_classe :
    def nom_methode(self, param_1, ..., param_n):
        # corps de la méthode...
```

A part le premier paramètre qui doit de préférence s'appeler `self`, la syntaxe de définition d'une méthode ressemble en tout point à celle d'une fonction. Le corps de la méthode est indenté par rapport à la déclaration de la méthode, elle-même indentée par rapport à la déclaration de la classe. Comme une fonction, une méthode suppose que les arguments qu'elle reçoit existe, y compris `self`. On écrit la méthode en supposant qu'un object existe qu'on nomme `self`. L'appel à cette méthode obéit à la syntaxe qui suit :

Syntaxe S4 : Appel d'une méthode

```
cl = nom_classe()     # variable de type nom_classe
t  = cl.nom_methode (valeur_1, ..., valeur_n)
```

L'appel d'une méthode nécessite tout d'abord la création d'une variable. Une fois cette variable créée, il suffit d'ajouter le symbole ńă.ă" pour exécuter la méthode. Le paramètre `self` est ici implicitement remplacé par `cl` lors de l'appel.

L'exemple suivant simule le tirage de nombres aléatoires à partir d'une suite définie par récurrence $u_{n+1} = (u_n * A) mod B$ où A et B sont des entiers très grands. Cette suite n'est pas aléatoire mais son comportement imite celui d'une suite aléatoire. Le terme u_n est dans cet exemple contenu dans la variable globale `rnd`.

<<<

```
rnd = 42


class exemple_classe:
    def methode1(self, n):
        """simule la génération d'un nombre aléatoire
           compris entre 0 et n-1 inclus"""
        global rnd
        rnd = 397204094 * rnd % 2147483647
        return int(rnd % n)


nb = exemple_classe()
l1 = [nb.methode1(100) for i in range(0, 10)]
print(l1)   # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]

nb2 = exemple_classe()
l2 = [nb2.methode1(100) for i in range(0, 10)]
print(l2)   # affiche [46, 42, 89, 66, 48, 12, 61, 84, 71, 41]
```

>>>

```
    [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
    [46, 42, 89, 66, 48, 12, 61, 84, 71, 41]
```

Deux instances `nb` et `nb2` de la classe `exemple_classe` sont créées, chacune d'elles est utilisée pour générer aléatoirement dix nombres entiers compris entre 0 et 99 inclus. Les deux listes sont différentes puisque l'instance `nb2` utilise la variable globale `rnd` précédemment modifiée par l'appel `nb.methode1(100)`.

Les méthodes sont des fonctions insérées à l'intérieur d'une classe. La syntaxe de la déclaration d'une méthode est identique à celle d'une fonction en tenant compte du premier paramètre qui doit impérativement être `self`. Les paramètres par défaut, l'ordre des paramètres, les nombres variables de paramètres présentés au paragraphe *Fonctions* (page 53) sont des extensions tout autant applicables aux méthodes qu'aux fonctions.

Attributs

Définition D4 : attribut

Les attributs sont des variables qui sont associées de manière explicite à une classe. Les attributs de la classe se comportent comme des variables globales pour toutes les méthodes de cette classe.

Une classe permet en quelque sorte de regrouper ensemble des informations liées. Elles n'ont de sens qu'ensemble et les méthodes manipulent ces données liées. C'est le cas pour un segment qui est toujours défini par ces deux extrémités qui ne vont pas l'une sans l'autre.

Syntaxe S5 : Déclaration d'un attribut

```
class nom_classe :
    def nom_methode (self, param_1, ..., param_n) :
        self.nom_attribut = valeur
```

Le paramètre `self` n'est pas un mot-clé même si le premier paramètre est le plus souvent appelé `self`. Il désigne l'instance de la classe sur laquelle va s'appliquer la méthode. La déclaration d'une méthode inclut toujours un paramètre `self` de sorte que `self.nom_attribut` désigne un attribut de la classe. `nom_attribut` seul désignerait une variable locale sans aucun rapport avec un attribut portant le même nom. Les attributs peuvent être déclarés à l'intérieur de n'importe quelle méthode, voire à l'extérieur de la classe elle-même.

L'endroit où est déclaré un attribut a peu d'importance pourvu qu'il le soit avant sa première utilisation. Dans l'exemple qui suit, la méthode `methode1` utilise l'attribut `rnd` sans qu'il ait été créé.

<<<

```
class exemple_classe:
    def methode1(self, n):
        """simule la génération d'un nombre aléatoire
           compris entre 0 et n-1 inclus"""
        self.rnd = 397204094 * self.rnd % 2147483647
        return int(self.rnd % n)


nb = exemple_classe()
li = [nb.methode1(100) for i in range(0, 10)]
print(li)
```

>>>

```
    [runpythonerror]

    Traceback (most recent call last):
        exec(obj, globs, loc)
      File "", line 13, in <module>
      File "", line 11, in run_python_script_139791941458608
      File "", line 11, in <listcomp>
      File "", line 7, in methode1
    AttributeError: 'exemple_classe' object has no attribute 'rnd'
```

Cet exemple déclenche donc une erreur (ou exception) signifiant que l'attribut `rnd` n'a pas été créé.

Pour remédier à ce problème, il existe plusieurs endroits où il est possible de créer l'attribut `rnd`. Il est possible de créer l'attribut à l'intérieur de la méthode `methode1`. Mais le programme n'a plus le même sens puisqu'à chaque appel de la méthode `methode1`, l'attribut `rnd` reçoit la valeur 42. La liste de nombres aléatoires contient dix fois la même valeur.

<<<

```
class exemple_classe:
    def methode1(self, n):
        """simule la génération d'un nombre aléatoire
           compris entre 0 et n-1 inclus"""
        self.rnd = 42  # déclaration à l'intérieur de la méthode,
        # doit être précédé du mot-clé self
        self.rnd = 397204094 * self.rnd % 2147483647
        return int(self.rnd % n)


nb = exemple_classe()
li = [nb.methode1(100) for i in range(0, 10)]
print(li)  # affiche [19, 19, 19, 19, 19, 19, 19, 19, 19, 19]
```

>>>

```
    [19, 19, 19, 19, 19, 19, 19, 19, 19, 19]
```

Il est possible de créer l'attribut `rnd` à l'extérieur de la classe. Cette écriture devrait toutefois être évitée puisque la méthode `methode1` ne peut pas être appelée sans que l'attribut `rnd` ait été ajouté.

<<<

```
class exemple_classe:
    def methode1(self, n):
        """simule la génération d'un nombre aléatoire
           compris entre 0 et n-1 inclus"""
        self.rnd = 397204094 * self.rnd % 2147483647
        return int(self.rnd % n)


nb = exemple_classe()
nb.rnd = 42              # déclaration à l'extérieur de la classe,
# indispensable pour utiliser la méthode methode1
li = [nb.methode1(100) for i in range(0, 10)]
print(li)  # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

>>>

```
    [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

Ceux qui découvrent la programmation se posent toujours la question de l'utilité de ce nouveau concept qui ne permet pas de faire des choses différentes, tout au plus de les faire mieux. La finalité des classes apparaît avec le concept d¿*Héritage* (page 134). L'article illustre une façon de passer progressivent des fonctions aux classes de fonctions : *C'est obligé les classes ?* (page **??**).

3.1.2 Constructeur

L'endroit le plus approprié pour déclarer un attribut est à l'intérieur d'une méthode appelée le *constructeur*. S'il est défini, il est implicitement exécuté lors de la création de chaque instance. Le constructeur d'une classe se présente comme une méthode et suit la même syntaxe à ceci près que son nom est imposé : `__init__`. Hormis le premier paramètre, invariablement `self`, il n'existe pas de contrainte concernant la liste des paramètres excepté que le constructeur ne doit pas retourner de résultat.

Syntaxe S6 : Déclaration d'un constructeur

```
class nom_classe :
    def __init__(self, param_1, ..., param_n):
        # code du constructeur
```

nom_classe est une classe, __init__ est son constructeur, sa syntaxe est la même que celle d'une méthode sauf que le constructeur ne peut employer l'instruction return. La modification des paramètres du constructeur implique également la modification de la syntaxe de création d'une instance de cette classe.

Syntaxe S7 : Appel d'un constructeur

```
x = nom_classe (valeur_1,...,valeur_n)
```

nom_classe est une classe, valeur_1 à valeur_n sont les valeurs associées aux paramètres param_1 à param_n du constructeur.

L'exemple suivant montre deux classes pour lesquelles un constructeur a été défini. La première n'ajoute aucun paramètre, la création d'une instance ne nécessite pas de paramètre supplémentaire. La seconde classe ajoute deux paramètres a et b. Lors de la création d'une instance de la classe classe2, il faut ajouter deux valeurs.

<<<

```
class classe1:
    def __init__(self):
        # pas de paramètre supplémentaire
        print("constructeur de la classe classe1")
        self.n = 1  # ajout de l'attribut n


x = classe1()       # affiche constructeur de la classe classe1
print(x.n)          # affiche 1


class classe2:
    def __init__(self, a, b):
        # deux paramètres supplémentaires
        print("constructeur de la classe classe2")
        self.n = (a+b)/2  # ajout de l'attribut n


x = classe2(5, 9)  # affiche constructeur de la classe classe2
print(x.n)          # affiche 7
```

>>>

```
    constructeur de la classe classe1
    1
    constructeur de la classe classe2
    7.0
```

Le constructeur autorise autant de paramètres qu'on souhaite lors de la création d'une instance et celle-ci suit la même syntaxe qu'une fonction. La création d'une instance pourrait être considérée comme l'appel à une fonction à ceci près que le type du résultat est une instance de classe.

En utilisant un constructeur, l'exemple du paragraphe précédent simulant une suite de variable aléatoire permet d'obtenir une classe autonome qui ne fait pas appel à une variable globale ni à une déclaration d'attribut extérieur à la classe.

<<<

```
class exemple_classe:
    def __init__(self):  # constructeur
        self.rnd = 42     # on crée l'attribut rnd, identique pour chaque instance
        # --> les suites générées auront toutes le même début

    def methode1(self, n):
        self.rnd = 397204094 * self.rnd % 2147483647
        return int(self.rnd % n)


nb = exemple_classe()
l1 = [nb.methode1(100) for i in range(0, 10)]
print(l1)   # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]

nb2 = exemple_classe()
l2 = [nb2.methode1(100) for i in range(0, 10)]
print(l2)   # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

>>>

```
    [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
    [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

De la même manière qu'il existe un constructeur exécuté à chaque création d'instance, il existe un destructeur exécuté à chaque destruction d'instance. Il suffit pour cela de redéfinir la méthode `__del__`. A l'inverse d'autres langages comme le *C++*, cet opérateur est peu utilisé car le *python* nettoie automatiquement les objets qui ne sont plus utilisés ou plus référencés par une variable.

3.1.3 Apport du langage python

Liste des attributs

Chaque attribut d'une instance de classe est inséré dans un dictionnaire appelé __dict__ [152], attribut implicitement présent dès la création d'une instance.

<<<

```
class exemple_classe:
    def __init__(self):
        self.rnd = 42

    def methode1(self, n):
        self.rnd = 397204094 * self.rnd % 2147483647
        return int(self.rnd % n)


nb = exemple_classe()
print(nb.__dict__)      # affiche {'rnd': 42}
```

>>>

```
    {'rnd': 42}
```

Ce dictionnaire offre aussi la possibilité de tester si un attribut existe ou non. Dans un des exemples du paragraphe précédent, l'attribut `rnd` était créé dans la méthode `methode1`, sa valeur était alors initialisée à chaque appel et la

152. https://docs.python.org/3/library/stdtypes.html?highlight=__dict__#object.__dict__

fonction retournait sans cesse la même valeur. En testant l'existence de l'attribut `rnd`, il est possible de le créer dans la méthode `methode1` au premier appel sans que les appels suivants ne réinitialisent sa valeur à 42.

<<<

```
class exemple_classe:
    def methode1(self, n):
        if "rnd" not in self.__dict__:  # l'attribut existe-t-il ?
            self.rnd = 42                # création de l'attribut
            self.__dict__["rnd"] = 42   # autre écriture possible
        self.rnd = 397204094 * self.rnd % 2147483647
        return int(self.rnd % n)


nb = exemple_classe()
li = [nb.methode1(100) for i in range(0, 10)]
print(li)  # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

>>>

```
    [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

Attributs implicites

Certains attributs sont créés de manière implicite lors de la création d'une instance. Ils contiennent des informations sur l'instance.

`__module__`	Contient le nom du module dans lequel est incluse la classe (voir chapitre *Modules* (page **??**)).
`__class__`	Contient le nom de la classe de l'instance. Ce nom est précédé du nom du module suivi d'un point.
`__dict__`	Contient la liste des attributs de l'instance (voir paragraphe *Liste des attributs* (page 102).
`__doc__`	Contient un commentaire associé à la classe (voir paragraphe *Commentaires, aide* (page 104).

L'attribut `__class__` contient lui même d'autres d'attributs :

`__doc__`	Contient un commentaire associé à la classe (voir paragraphe *Commentaires, aide* (page 104).
`__dict__`	Contient la liste des attributs statiques (définis hors d'une méthode) et des méthodes (voir paragraphe *Attributs statiques* (page 119).
`__name__`	Contient le nom de l'instance.
`__bases__`	Contient les classes dont la classe de l'instance hérite (voir paragraphe *Héritage* (page 134).

<<<

```
class classe_vide:
    pass


cl = classe_vide()
print(cl.__module__)             # affiche __main__
print(cl.__class__)              # affiche __main__.classe_vide ()
```

(suite sur la page suivante)

(suite de la page précédente)

```
print(cl.__dict__)                # affiche {}
print(cl.__doc__)                 # affiche None  (voir paragraphe suivant)
print(cl.__class__.__doc__)       # affiche None
print(cl.__class__.__dict__)      # affiche {'__module__': '__main__',
#           '__doc__': None}
print(cl.__class__.__name__)      # affiche classe_vide
print(cl.__class__.__bases__)     # affiche ()
```

>>>

```
    pyquickhelper.sphinxext.sphinx_runpython_extension
    <class 'pyquickhelper.sphinxext.sphinx_runpython_extension.run_python_script_
↪139791947388608.<locals>.classe_vide'>
    {}
    None
    None
    {'__module__': 'pyquickhelper.sphinxext.sphinx_runpython_extension', '__dict__':
↪<attribute '__dict__' of 'classe_vide' objects>, '__weakref__': <attribute '__
↪weakref__' of 'classe_vide' objects>, '__doc__': None}
    classe_vide
    (<class 'object'>,)
```

Commentaires, aide

Comme les fonctions et les méthodes, des commentaires peuvent être associés à une classe, ils sont affichés grâce à la fonction `help`. Cette dernière présente le commentaire associé à la classe, la liste des méthodes ainsi que chacun des commentaires qui leur sont associés. Ce commentaire est affecté à l'attribut implicite `__doc__`. L'appel à la fonction `help` rassemble le commentaire de toutes les méthodes, le résultat suit le programme ci-dessous.

<<<

```
class exemple_classe:
    """simule une suite de nombres aléatoires"""

    def __init__(self):
        """constructeur : initialisation de la première valeur"""
        self.rnd = 42

    def methode1(self, n):
        """simule la génération d'un nombre aléatoire
        compris entre 0 et n-1 inclus"""
        self.rnd = 397204094 * self.rnd % 2147483647
        return int(self.rnd % n)


nb = exemple_classe()
help(exemple_classe)      # appelle l'aide associée à la classe
```

>>>

```
    Help on class exemple_classe in module pyquickhelper.sphinxext.sphinx_runpython_
↪extension:

    class exemple_classe(builtins.object)
     |  simule une suite de nombres aléatoires
     |
```

(suite sur la page suivante)

(suite de la page précédente)

```
     |  Methods defined here:
     |
     |  __init__(self)
     |      constructeur : initialisation de la première valeur
     |
     |  methode1(self, n)
     |      simule la génération d'un nombre aléatoire
     |      compris entre 0 et n-1 inclus
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
```

Pour obtenir seulement le commentaire associé à la classe, il suffit d'écrire l'une des trois lignes suivantes :

<<<

```
class exemple_classe:
    """simule une suite de nombres aléatoires"""
    pass


nb = exemple_classe()

print(exemple_classe.__doc__)   # affiche simule une suite de nombres aléatoires
print(nb.__doc__)               # affiche simule une suite de nombres aléatoires
print(nb.__class__.__doc__)     # affiche simule une suite de nombres aléatoires
```

>>>

```
    simule une suite de nombres aléatoires
    simule une suite de nombres aléatoires
    simule une suite de nombres aléatoires
```

La fonction `help` permet d'accéder à l'aide associée à une fonction, une classe. Il existe des outils qui permettent de collecter tous ces commentaires pour construire une documentation au format *HTML* à l'aide d'outils comme pydoc [153] Ces outils sont souvent assez simples d'utilisation. Le plus utilisé est sphinx [154].

La fonction dir [155] permet aussi d'obtenir des informations sur la classe. Cette fonction appliquée à la classe ou à une instance retourne l'ensemble de la liste des attributs et des méthodes. L'exemple suivant utilise la fonction dir [156] avant et après l'appel de la méthode `meth`. Etant donné que cette méthode ajoute un attribut, la fonction dir [157] retourne une liste plus longue après l'appel.

<<<

```
class essai_class:
    def meth(self):
        x = 6
```

(suite sur la page suivante)

153. https://docs.python.org/3/library/pydoc.html
154. http://www.sphinx-doc.org/en/
155. https://docs.python.org/3/library/functions.html?highlight=dir#dir
156. https://docs.python.org/3/library/functions.html?highlight=dir#dir
157. https://docs.python.org/3/library/functions.html?highlight=dir#dir

(suite de la page précédente)

```
        self.y = 7


a = essai_class()
print(dir(a))              # affiche ['__doc__', '__module__', 'meth']
a.meth()
print(dir(a))              # affiche ['__doc__', '__module__', 'meth', 'y']
print(dir(essai_class))    # affiche ['__doc__', '__module__', 'meth']
```

>>>

```
    ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__
↪format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_
↪subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '_
↪_reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__
↪', '__weakref__', 'meth']
    ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__
↪format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_
↪subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '_
↪_reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__
↪', '__weakref__', 'meth', 'y']
    ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__
↪format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_
↪subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '_
↪_reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__
↪', '__weakref__', 'meth']
```

La fonction dir [158] appliquée à la classe elle-même retourne une liste qui inclut les méthodes et les attributs déjà déclarés. Elle n'inclut pas ceux qui sont déclarés dans une méthode jamais exécutée jusqu'à présent.

Classe incluse

Parfois, il arrive qu'une classe soit exclusivement utilisée en couple avec une autre, c'est par exemple le cas des itérateurs (voir paragraphe *Itérateurs* (page 113). Il est alors possible d'inclure dans la déclaration d'une classe celle d'une sous-classe.

L'exemple qui suit contient la classe `ensemble_element`. C'est un ensemble de points en trois dimensions (classe `element`) qui n'est utilisé que par cette classe. Déclarer la classe `element` à l'intérieur de la classe `ensemble_element` est un moyen de signifier ce lien.

<<<

```
class ensemble_element:

    class element:
        def __init__(self):
            self.x, self.y, self.z = 0, 0, 0

    def __init__(self):
        self.all = [ensemble_element.element() for i in range(0, 3)]

    def barycentre(self):
        b = ensemble_element.element()
        for el in self.all:
            b.x += el.x
```

(suite sur la page suivante)

158. https://docs.python.org/3/library/functions.html?highlight=dir#dir

(suite de la page précédente)

```
            b.y += el.y
            b.z += el.z
        b.x /= len(self.all)
        b.y /= len(self.all)
        b.z /= len(self.all)
        return b


f = ensemble_element()
f.all[0].x, f.all[0].y, f.all[0].z = 4.5, 1.5, 1.5
b = f.barycentre()
print(b.x, b.y, b.z)  # affiche 1.5 0.5 0.5
```

>>>

```
    1.5 0.5 0.5
```

Pour créer une instance de la classe `element`, il faut faire précéder son nom de la classe où elle est déclarée : `b = ensemble_element.element()` comme c'est le cas dans la méthode `barycentre` par exemple.

3.1.4 Opérateurs

Les opérateurs sont des symboles du langages comme +, −, +=, ... Au travers des opérateurs, il est possible de donner un sens à une syntaxe comme celle de l'exemple suivant :

<<<

```
class nouvelle_classe:
    pass


x = nouvelle_classe() + nouvelle_classe()
```

>>>

```
    [runpythonerror]

    Traceback (most recent call last):
        exec(obj, globs, loc)
      File "", line 6, in <module>
      File "", line 5, in run_python_script_139791947700192
    TypeError: unsupported operand type(s) for +: 'nouvelle_classe' and 'nouvelle_
↪classe'
```

L'addition n'est pas le seul symbole concerné, le langage *python* permet de donner un sens à tous les opérateurs numériques et d'autres reliés à des fonctions du langage comme `len` ou `max`. Le programme suivant contient une classe définissant un nombre complexe. La méthode `ajoute` définit ce qu'est une addition entre nombres complexes.

<<<

```
import math


class nombre_complexe:
    def __init__(self, a=0, b=0):
        self.a, self.b = a, b
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def get_module(self):
        return math.sqrt(self.a * self.a + self.b * self.b)

    def ajoute(self, c):
        return nombre_complexe(self.a + c.a, self.b + c.b)


c1 = nombre_complexe(0, 1)
c2 = nombre_complexe(1, 0)

c = c1.ajoute(c2)          # c = c1 + c2
print(c.a, c.b)
```

>>>

```
    1 1
```

Toutefois, on aimerait bien écrire simplement `c = c1 + c2` au lieu de `c = c1.ajoute(c2)` car cette syntaxe est plus facile à lire et surtout plus intuitive. Le langage *python* offre cette possibilité. Il existe en effet des méthodes *clés* dont l'implémentation définit ce qui doit être fait dans le cas d'une addition, d'une comparaison, d'un affichage, ... A l'instar du constructeur, toutes ces méthodes clés, qu'on appelle des *opérateurs*, sont encadrées par deux blancs soulignés, leur déclaration suit invariablement le même schéma. Voici celui de l'opérateur `__add__` qui décrit ce qu'il faut faire pour une addition.

<<<

```
class nom_class:
    def __add__(self, autre):
        # corps de l'opérateur
        return ...   # nom_classe
```

>>>

```

```

`nom_classe` est une classe. L'opérateur `__add__` définit l'addition entre l'instance `self` et l'instance `autre` et retourne une instance de la classe `nom_classe`.

Le programme suivant reprend le précédent de manière à ce que l'addition de deux nombres complexes soit dorénavant une syntaxe correcte.

<<<

```
import math


class nombre_complexe:
    def __init__(self, a=0, b=0):
        self.a, self.b = a, b

    def get_module(self):
        return math.sqrt(self.a * self.a + self.b * self.b)

    def __add__(self, c):
        return nombre_complexe(self.a + c.a, self.b + c.b)
```

(suite sur la page suivante)

(suite de la page précédente)

```
c1 = nombre_complexe(0, 1)
c2 = nombre_complexe(1, 0)
c = c1 + c2           # cette expression est maintenant syntaxiquement correcte
c = c1.__add__(c2)   # même ligne que la précédente mais écrite explicitement
print(c.a, c.b)
```

>>>

```
    1 1
```

L'avant dernière ligne appelant la méthode `__add__` transcrit de façon explicite ce que le langage *python* fait lorsqu'il rencontre un opérateur + qui s'applique à des classes. Plus précisément, `c1` et `c2` pourraient être de classes différentes, l'expression serait encore valide du moment que la classe dont dépend `c1` a redéfini la méthode `__add__`. Chaque opérateur possède sa méthode-clé associée. L'opérateur +=, différent de + est associé à la méthode-clé `__iadd__`.

```
class nom_class :
    def __iadd__ (self, autre) :
        # corps de l'opérateur
        return self
```

`nom_classe` est une classe. L'opérateur `__iadd__` définit l'addition entre l'instance `self` et l'instance `autre`. L'instance `self` est modifiée pour recevoir le résultat. L'opérateur retourne invariablement l'instance modifiée `self`. On étoffe la classe `nombre_complexe` à l'aide de l'opérateur `__iadd__`.

<<<

```
import math


class nombre_complexe:
    def __init__(self, a=0, b=0):
        self.a, self.b = a, b

    def get_module(self):
        return math.sqrt(self.a * self.a + self.b * self.b)

    def __add__(self, c):
        return nombre_complexe(self.a + c.a, self.b + c.b)

    def __iadd__(self, c):
        self.a += c.a
        self.b += c.b
        return self


c1 = nombre_complexe(0, 1)
c2 = nombre_complexe(1, 0)
c1 += c2           # utilisation de l'opérateur +=
c1.__iadd__(c2)    # c'est la transcription explicite de la ligne précédente
print(c1.a, c1.b)
```

>>>

```
    2 1
```

Un autre opérateur souvent utilisé est `__str__` qui permet de redéfinir l'affichage d'un objet lors d'un appel à l'instruction `print`.

Syntaxe S8 : Déclaration de l'opérateur __str__

```
class nom_class :
    def __str__ (self) :
        # corps de l'opérateur
        return...
```

nom_classe est une classe. L'opérateur __str__ construit une chaîne de caractères qu'il retourne comme résultat de façon à être affiché. L'exemple suivant reprend la classe nombre_complexe pour que l'instruction print affiche un nombre complexe sous la forme $a + ib$.

<<<

```
class nombre_complexe:
    def __init__(self, a=0, b=0):
        self.a, self.b = a, b

    def __add__(self, c):
        return nombre_complexe(self.a + c.a, self.b + c.b)

    def __str__(self):
        if self.b == 0:
            return "%f" % (self.a)
        elif self.b > 0:
            return "%f + %f i" % (self.a, self.b)
        else:
            return "%f - %f i" % (self.a, -self.b)


c1 = nombre_complexe(0, 1)
c2 = nombre_complexe(1, 0)
c3 = c1 + c2
print(c3)       # affiche 1.000000 + 1.000000 i
```

>>>

```
    1.000000 + 1.000000 i
```

Il existe de nombreux opérateurs qu'il est possible de définir. La table operateur_classe présente les plus utilisés. Parmi ceux-là, on peut s'attarder sur les opérateurs __getitem__ et __setitem__, ils redéfinissent l'opérateur [] permettant d'accéder à un élément d'une liste ou d'un dictionnaire. Le premier, __getitem__ est utilisé lors d'un calcul, un affichage. Le second, __setitem__, est utilisé pour affecter une valeur.

L'exemple suivant définit un point de l'espace avec trois coordonnées. Il redéfinit ou *surcharge* les opérateurs __getitem__ et __setitem__ de manière à pouvoir accéder aux coordonnées de la classe point_espace qui définit un point dans l'espace. En règle générale, lorsque les indices ne sont pas corrects, ces deux opérateurs lèvent l'exception IndexError (voir le chapitre *Exceptions* (page 155).

<<<

```
class point_espace:
    def __init__(self, x, y, z):
        self._x, self._y, self._z = x, y, z

    def __getitem__(self, i):
        if i == 0:
            return self._x
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if i == 1:
            return self._y
        if i == 2:
            return self._z
        # pour tous les autres cas --> erreur
        raise IndexError("indice impossible, 0,1,2 autorisés")

    def __setitem__(self, i, x):
        if i == 0:
            self._x = x
        elif i == 1:
            self._y = y
        elif i == 2:
            self._z = z
        # pour tous les autres cas --> erreur
        raise IndexError("indice impossible, 0,1,2 autorisés")

    def __str__(self):
        return "(%f,%f,%f)" % (self._x, self._y, self._z)


a = point_espace(1, -2, 3)

print(a)                      # affiche (1.000000,-2.000000,3.000000)
a[1] = -3                    # (__setitem__) affecte -3 à a.y
print("abscisse : ", a[0])   # (__getitem__) affiche abscisse :  1
print("ordonnée : ", a[1])   # (__getitem__) affiche ordonnée :  -3
print("altitude : ", a[2])   # (__getitem__) affiche altitude :  3
```

>>>

```
    (1.000000,-2.000000,3.000000)
    [runpythonerror]

    Traceback (most recent call last):
        exec(obj, globs, loc)
      File "", line 31, in <module>
      File "", line 27, in run_python_script_139791947601240
      File "", line 16, in __setitem__
    NameError: name 'y' is not defined
```

Par le biais de l'exception `IndexError`, les expressions `a[i]` avec `i != 0,1,2` sont impossibles et arrêtent le programme par un message comme celui qui suit obtenu après l'interprétation de `print(a[4])` :

`__cmp__(self,x)`	Retourne un entier égale à -1, 0, 1, chacune de ces valeurs étant associés respectivement à : `self < x`, `self == x`, `self > x`. Cet opérateur est appelé par la fonction `cmp`.
`__str__(self)`	Convertit un objet en une chaîne de caractère qui sera affichée par la fonction `print` ou obtenu avec la fonction `str`.
`__contains__(self,x)`	Retourne `True` ou `False` selon que `x` appartient à `self`. Le mot-clé `in` renvoie à cet opérateur. En d'autres termes, `if x in obj:` appelle `obj.__contains__(x)`.
`__len__(self)`	Retourne le nombre d'élément de `self`. Cet opérateur est appelé par la fonction `len`.
`__abs__(self)`	Cet opérateur est appelé par la fonction `abs`.
`__getitem__(self,i)`	Cet opérateur est appelé lorsqu'on cherche à accéder à un élément de l'objet `self` d'indice `i` comme si c'était une liste. Si l'indice `i` est incorrect, l'exception `IndexError` doit être levée.
`__setitem__(self,i,v)`	Cet opérateur est appelé lorsqu'on cherche à affecter une valeur `v` à un élément de l'objet `self` d'indice `i` comme si c'était une liste ou un dictionnaire. Si l'indice `i` est incorrect, l'exception `IndexError`.
`__delitem__(self,i)`	Cet opérateur est appelé lorsqu'on cherche à supprimer l'élément de l'objet `self` d'indice `i` comme si c'était une liste ou un dictionnaire. Si l'indice `i` est incorrect, l'exception `IndexError` doit être levée.
`__int__(self)`, `__float__(self)`, `__complex__(self)`	Ces opérateurs implémente la conversion de l'instance `self` en entier, réel ou complexe.
`__add__(self,x)`, `__div__(self,x)`, `__mul__(self,x)` `__sub__(self,x)`, `__pow__(self,x)`, `__lshift__(self, x)`, `__rshift__(self, x)`	Opérateurs appelés pour les opérations +, /, *, -, **, <, <
`__iadd__(self,x)`, `__idiv__(self,x)`, `__imul__(self,x)`, `__isub__(self,x)`, `__ipow__(self,x)`, `__ilshift__(self, x)` `__irshift__(self, x)`	Opérateurs appelés pour les opérations +=, /=, *=, -=, **=, <<=, >>=

La liste complète est accessible à Operators [159]. Le langage Python [160] autorise une opération peu commune aux autres langages : des opérateurs retournés [161]. Cela permet de donner un sens à une expression du type `4 + instance d'un objet`. Le type entier ne définit pas cette opération et elle devrait normalement échoué. Comme elle n'exsite pas, il est possible de définir un opérateur retourné qui prend le relais dans ce cas. Il est recommandé d'en faire un usage modéré car c'est quelque chose peu répandu dans les langages de programmation.

<<<

```
class RightSide:

    def __init__(self, v):
        self.v = v
```

(suite sur la page suivante)

159. https://docs.python.org/3/library/operator.html
160. http://www.python.org/
161. https://docs.python.org/3/reference/datamodel.html#object.__radd__

(suite de la page précédente)

```
    def __str__(self):
        return "RS({})".format(self.v)

    def __add__(self, v):
        return RightSide('9999999999')


class LeftSide:

    def __init__(self, v):
        self.v = v

    def __str__(self):
        return "LS({})".format(self.v)

    def __add__(self, o):
        return LeftSide(self.v + o)

    def __radd__(self, o):
        return RightSide(self.v + o)

    def __lshift__(self, o):
        return self.__add__(o)

    def __rlshift__(self, o):
        return self.__radd__(o)


print(LeftSide(3) + 4)
print(4 + LeftSide(3))
print('---')
print(LeftSide(3) << 4)
print(4 << LeftSide(3))
print('---')
print(RightSide(4) + LeftSide(3))
```

>>>

```
    LS(7)
    RS(7)
    ---
    LS(7)
    RS(7)
    ---
    RS(9999999999)
```

3.1.5 Itérateurs

L'opérateur `__iter__` permet de définir ce qu'on appelle un itérateur. C'est un objet qui permet d'en explorer un autre, comme une liste ou un dictionnaire. Un itérateur est un objet qui désigne un élément d'un ensemble à parcourir et qui connaît l'élément suivant à visiter. Il doit pour cela contenir une référence à l'objet qu'il doit explorer et inclure une méthode `__next__` qui retourne l'élément suivant ou lève une exception si l'élément actuel est le dernier.

Par exemple, on cherche à explorer tous les éléments d'un objet de type `point_espace` défini au paragraphe précédent. Cette exploration doit s'effectuer au moyen d'une boucle `for`.

<<<

```
class point_espace:
    def __init__(self, x, y, z):
        self._x, self._y, self._z = x, y, z

    def __iter__(self):
        yield self._x
        yield self._y
        yield self._z


a = point_espace(1, -2, 3)

for x in a:
    print(x)       # affiche successivement 1,-2,3
```

>>>

```
    1
    -2
    3
```

Cette boucle cache en fait l'utilisation d'un itérateur qui apparaît explicitement dans l'exemple suivant équivalent au précédent (voir paragraphe *Itérateurs* (page 46)).

<<<

```
class point_espace:
    def __init__(self, x, y, z):
        self._x, self._y, self._z = x, y, z

    def __iter__(self):
        yield self._x
        yield self._y
        yield self._z


a = point_espace(1, -2, 3)
it = iter(a)
while True:
    try:
        print(next(it))
    except StopIteration:
        break
```

>>>

```
    1
    -2
    3
```

Afin que cet extrait de programme fonctionne, il faut définir un itérateur pour la classe `point_espace`. Cet itérateur doit inclure la méthode `__next__`. La classe `point_espace` doit quant à elle définir l'opérateur `__iter__` pour retourner l'itérateur qui permettra de l'explorer.

<<<

```
class point_espace:
    def __init__(self, x, y, z):
        self._x, self._y, self._z = x, y, z

    def __str__(self):
        return "(%f,%f,%f)" % (self._x, self._y, self._z)

    def __getitem__(self, i):
        if i == 0:
            return self._x
        if i == 1:
            return self._y
        if i == 2:
            return self._z
        # pour tous les autres cas --> erreur
        raise IndexError("indice impossible, 0,1,2 autorisés")

    class class_iter:
        """cette classe définit un itérateur pour point_espace"""

        def __init__(self, ins):
            """initialisation, self._ins permet de savoir quelle
               instance de point_espace on explore,
               self._n mémorise l'indice de l'élément exploré"""
            self._n = 0
            self._ins = ins

        def __iter__(self):   # le langage impose cette méthode
            return self           # dans certaines configurations

        def __next__(self):
            """retourne l'élément d'indice self._n et passe à l'élément suivant"""
            if self._n <= 2:
                v = self._ins[self._n]
                self._n += 1
                return v
            else:
                # si cet élément n'existe pas, lève une exception
                raise StopIteration

    def __iter__(self):
        """opérateur de la classe point_espace, retourne un itérateur
           permettant de l'explorer"""
        return point_espace.class_iter(self)


a = point_espace(1, -2, 3)
for x in a:
    print(x)       # affiche successivement 1,-2,3
```

>>>

```
    1
    -2
    3
```

Cette syntaxe peut paraître fastidieuse mais elle montre de manière explicite le fonctionnement des itérateurs. Cette construction est plus proche de ce que d'autres langages objets proposent. *python* offre néanmoins une syntaxe plus

courte avec le mot-clé `yield` qui permet d'éviter la création de la classe `class_iter`. Le code de la méthode `__iter__` change mais les dernières lignes du programme précédent qui affichent successivement les éléments de `point_espace` sont toujours valides.

<<<

```
class point_espace:
    def __init__(self, x, y, z):
        self._x, self._y, self._z = x, y, z

    def __str__(self):
        return "(%f,%f,%f)" % (self._x, self._y, self._z)

    def __getitem__(self, i):
        if i == 0:
            return self._x
        if i == 1:
            return self._y
        if i == 2:
            return self._z
        # pour tous les autres cas --> erreur
        raise IndexError("indice impossible, 0,1,2 autorisés")

    def __iter__(self):
        """itérateur avec yield (ou générateur)"""
        _n = 0
        while _n <= 2:
            yield self.__getitem__(_n)
            _n += 1


a = point_espace(1, -2, 3)
for x in a:
    print(x)       # affiche successivement 1,-2,3
```

>>>

```
    1
    -2
    3
```

3.1.6 Méthodes, attributs statiques et ajout de méthodes

Méthode statique

Définition D5 : méthode statique

Les méthodes statiques sont des méthodes qui peuvent être appelées même si aucune instance de la classe où elles sont définies n'a été créée.

L'exemple suivant définit une classe avec une seule méthode. Comme toutes les méthodes présentées jusqu'à présent, elle inclut le paramètre `self` qui correspond à l'instance pour laquelle elle est appelée.

<<<

```
class essai_class:
    def methode(self):
        print("méthode non statique")


x = essai_class()
x.methode()
```

>>>

```
    méthode non statique
```

Une méthode statique ne nécessite pas qu'une instance soit créée pour être appelée. C'est donc une méthode n'ayant pas besoin du paramètre `self`.

Syntaxe S9 : Déclaration d'une méthode statique

```
class nom_class :
    @staticmethod
    def nom_methode(params, ...) :
        # corps de la méthode
        ...
```

`nom_classe` est une classe, `nom_methode` est une méthode statique. Il faut pourtant ajouter la ligne suivante pour indiquer à la classe que cette méthode est bien statique à l'aide du mot-clé staticmethod [162]. Le programme précédent est modifié pour inclure une méthode statique. La méthode `methode` ne nécessite aucune création d'instance pour être appelée.

<<<

```
class essai_class:
    @staticmethod
    def methode():
        print("méthode statique")


essai_class.methode()
```

>>>

```
    méthode statique
```

Il est également possible de déclarer une fonction statique à l'extérieur d'une classe puis de l'ajouter en tant que méthode statique à cette classe. Le programme suivant déclare une fonction `methode` puis indique à la classe `essai_class` que la fonction est aussi une méthode statique de sa classe (avant-dernière ligne de l'exemple).

<<<

```
def methode():
    print("méthode statique")


class essai_class:
    pass
```

(suite sur la page suivante)

162. https://docs.python.org/3/library/functions.html?highlight=staticmethod#staticmethod

(suite de la page précédente)

```
essai_class.methode = staticmethod(methode)
essai_class.methode()
```

>>>

```
    méthode statique
```

Toutefois, il est conseillé de placer l'instruction qui contient `staticmethod` à l'intérieur de la classe. Elle n'y sera exécutée qu'une seule fois comme le montre l'exemple suivant :

<<<

```
def methode():
    print("méthode statique")


class classe_vide:
    print("création d'une instance de la classe classe_vide")
    methode = staticmethod(methode)


cl = classe_vide()      # affiche création d'une instance de la classe essai_class
ck = classe_vide()      # n'affiche rien
```

>>>

```
    création d'une instance de la classe classe_vide
```

Les méthodes statiques sont souvent employées pour créer des instances spécifiques d'une classe.

<<<

```
class Couleur:
    def __init__(self, r, v, b):
        self.r, self.v, self.b = r, v, b

    def __str__(self):
        return str((self.r, self.v, self.b))

    @staticmethod
    def blanc():
        return Couleur(255, 255, 255)

    @staticmethod
    def noir():
        return Couleur(0, 0, 0)


c = Couleur.blanc()
print(c)                # affiche (255, 255, 255)
c = Couleur.noir()
print(c)                # affiche (0, 0, 0)
```

>>>

```
    (255, 255, 255)
    (0, 0, 0)
```

Les méthodes sont des fonctions spécifiques à une classe sans être spécifique à une isntance.

Attributs statiques

Définition D6 : attribut statique

Les attributs statiques sont des attributs qui peuvent être utilisés même si aucune instance de la classe où ils sont définis n'a été créée. Ces attributs sont partagés par toutes les instances.

Syntaxe S10 : Déclaration d'un attribut statique

```
class nom_class :
    attribut_statique = valeur
    def nom_methode (self,params, ...):
        nom_class.attribut_statique2 = valeur2
    @staticmethod
    def nom_methode_st (params, ...) :
        nom_class.attribut_statique3 = valeur3
```

`nom_classe` est une classe, `nom_methode` est une méthode non statique, `nom_methode_st` est une méthode statique. Les trois paramètres `attribut_statique`, `attribut_statique2`, `attribut_statique3` sont statiques, soit parce qu'ils sont déclarés en dehors d'une méthode, soit parce que leur déclaration fait intervenir le nom de la classe.

Pour le programme suivant, la méthode `meth` n'utilise pas `self.x` mais `essai_class.x`. L'attribut `x` est alors un attribut statique, partagé par toutes les instances. C'est pourquoi dans l'exemple qui suit l'instruction `z.meth()` affiche la valeur `6` puisque l'appel `y.meth()` a incrémenté la variable statique `x`.

<<<

```
class essai_class:
    x = 5

    def meth(self):
        print(essai_class.x)
        essai_class.x += 1


y = essai_class()
z = essai_class()
y.meth()    # affiche 5
z.meth()    # affiche 6
```

>>>

```
    5
    6
```

Même si un attribut est statique, il peut être utilisé avec la syntaxe `self.attribut_statique` dans une méthode non statique à condition qu'un attribut non statique ne porte pas le même nom. Si tel est pourtant le cas, certaines confusions peuvent apparaître :

<<<

```
class exemple_classe:
    rnd = 42

    def incremente_rnd(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        self.rnd += 1
        return self.rnd


cl = exemple_classe()

print(cl.__dict__)                          # affiche {}
print(cl.__class__.__dict__["rnd"])     # affiche 42
cl.incremente_rnd()
print(cl.__dict__)                          # affiche {'rnd': 43}
print(cl.__class__.__dict__["rnd"])     # affiche 42
```

>>>

```
    {}
    42
    {'rnd': 43}
    42
```

Dans ce cas, ce sont en fait deux attributs qui sont créés. Le premier est un attribut statique créé avec la seconde ligne de l'exemple `rnd=42`. Le second attribut n'est pas statique et apparaît dès la première exécution de l'instruction `self.rnd+=1` comme le montre son apparition dans l'attribut `__dict__` qui ne recense pas les attributs statiques.

Ajout de méthodes

Ce point décrit une fonctionnalité du langage *python* rarement utilisée. Il offre la possibilité d'ajouter une méthode à une classe alors même que cette fonction est définie à l'extérieur de la déclaration de la classe. Cette fonction doit obligatoirement accepter un premier paramètre qui recevra l'instance de la classe. La syntaxe utilise le mot-clé classmethod [163].

<<<

```
def nom_methode(cls):
    # code de la fonction
    pass


class nom_classe:
    # code de la classe
    nom_methode = classmethod(nom_methode)          # syntaxe 1


nom_classe.nom_methode = classmethod(nom_methode)   # syntaxe 2
```

>>>

```

```

`nom_classe` est une classe, `nom_methode` est une méthode, `nom_methode` est une fonction qui est par la suite considérée comme une méthode de la classe `nom_methode` grâce à l'une ou l'autre des deux instructions incluant le mot-clé `classmethod`. Dans l'exemple qui suit, cette syntaxe est utilisée pour inclure trois méthodes à la classe `essai_class` selon que la méthode est déclarée et affectée à cette classe à l'intérieur ou à l'extérieur du corps de `essai_class`.

<<<

163. https://docs.python.org/3/library/functions.html?highlight=classmethod#classmethod

```
def meth3(cls):
    print("ok meth3", cls.x)


def meth4(cls):
    print("ok meth4", cls.x)


class essai_classe:
    x = 5

    def meth(self):
        print("ok meth", self.x)

    def meth2(cls):
        print("ok meth2", cls.x)

    meth3 = classmethod(meth3)


x = essai_classe()
x.meth()                                # affiche ok meth 5
x.meth2()                               # affiche ok meth2 5
x.meth3()                               # affiche ok meth3 5

essai_classe.meth4 = classmethod(meth4)
x.meth4()                               # affiche ok meth4 5
```

>>>

```
    ok meth 5
    ok meth2 5
    ok meth3 5
    ok meth4 5
```

Propriétés

Cette fonctionnalité est également peu utilisée, elle permet des raccourcis d'écriture. Les propriétés permettent de faire croire à l'utilisateur d'une instance de classe qu'il utilise une variable alors qu'il utilise en réalité une ou plusieurs méthodes. A chaque fois que le programmeur utilise ce faux attribut, il appelle une méthode qui calcule sa valeur. A chaque fois que le programmeur cherche à modifier la valeur de ce faux attribut, il appelle une autre méthode qui modifie l'instance.

Syntaxe S11 : Déclaration d'une propriété

```
class nom_classe :
    nom_propriete = property (fget, fset, fdel, doc)
```

La documentation de la fonction property [164] propose une autre écriture plus intuitive.

Syntaxe S12 : Déclaration d'une propriété (2)

164. https://docs.python.org/3/library/functions.html#property

```
class nom_classe :

    @property
    def fget_variable(self):
        return self.variable

    @variable.setter
    def fset_variable(self, v):
        self.variable = v
```

Au sein de ces trois lignes, `nom_classe` est une classe, `nom_propriete` est le nom de la propriété, `fget` est la méthode qui doit retourner la valeur du pseudo-attribut `nom_propriete`, `fset` est la méthode qui doit modifier la valeur du pseudo-attribut `nom_propriete`, `fdel` est la méthode qui doit détruire le pseudo-attribut `nom_propriete`, `doc` est un commentaire qui apparaîtra lors de l'appel de la fonction `help(nom_class)` ou `help(nom_class.nom_propriete)`.

Pour illustrer l'utilisation des propriétés, on part d'une classe `nombre_complexe` qui ne contient que les parties réelle et imaginaire. Le module désigne ici le module d'un nombre complexe qui est égal à sa norme. On le note $|a+ib| = \sqrt{a^2+b^2}$. On fait appel à une méthode qui calcule ce module. Lorsqu'on cherche à modifier ce module, on fait appel à une autre méthode qui multiplie les parties réelle et imaginaire par un nombre réel positif de manière à ce que le nombre complexe ait le module demandé. On procède de même pour la propriété `arg`.

La propriété `conj` retourne quant à elle le conjugué du nombre complexe mais la réciproque n'est pas prévue. On ne peut affecter une valeur à `conj`.

<<<

```
import math


class nombre_complexe(object):            # voir remarque après l'exemple
    def __init__(self, a=0, b=0):
        self.a = a
        self.b = b

    def __str__(self):
        if self.b == 0:
            return "%f" % (self.a)
        elif self.b > 0:
            return "%f + %f i" % (self.a, self.b)
        else:
            return "%f - %f i" % (self.a, -self.b)

    def get_module(self):
        return math.sqrt(self.a * self.a + self.b * self.b)

    def set_module(self, m):
        r = self.get_module()
        if r == 0:
            self.a = m
            self.b = 0
        else:
            d = m / r
            self.a *= d
            self.b *= d

    def get_argument(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        r = self.get_module()
        if r == 0:
            return 0
        else:
            return math.atan2(self.b / r, self.a / r)

    def set_argument(self, arg):
        m = self.get_module()
        self.a = m * math.cos(arg)
        self.b = m * math.sin(arg)

    def get_conjugue(self):
        return nombre_complexe(self.a, -self.b)

    module = property(fget=get_module,   fset=set_module,   doc="module")
    arg = property(fget=get_argument, fset=set_argument, doc="argument")
    conj = property(fget=get_conjugue,                   doc="conjugué")


c = nombre_complexe(0.5, math.sqrt(3)/2)
print("c = ",         c)          # affiche c =  0.500000 + 0.866025 i
print("module = ",    c.module)   # affiche module =  1.0
print("argument = ",  c.arg)      # affiche argument =  1.0471975512

c = nombre_complexe()
c.module = 1
c.arg = math.pi * 2 / 3
print("c = ",         c)          # affiche c =  -0.500000 + 0.866025 i
print("module = ",    c.module)   # affiche module =  1.0
print("argument = ",  c.arg)      # affiche argument =  2.09439510239
print("conjugué = ",  c.conj)     # affiche conjugué =  -0.500000 - 0.866025 i
```

>>>

```
    c =  0.500000 + 0.866025 i
    module =  0.9999999999999999
    argument =  1.0471975511965976
    c =  -0.500000 + 0.866025 i
    module =  0.9999999999999999
    argument =  2.0943951023931953
    conjugué =  -0.500000 - 0.866025 i
```

La propriété `conj` ne possède pas de fonction qui permet de la modifier. Par conséquent, l'instruction `c.conj = nombre_complexe(0,0)` produit une erreur. Etant donné qu'une propriété porte déjà le nom de `conj`, aucun attribut du même nom ne peut être ajouté à la classe `nombre_complexe`.

Afin que la propriété fonctionne correctement, il est nécessaire que la classe hérite de la classe `object` ou une de ses descendantes (voir également *Héritage* (page 134)).

3.1.7 Copie d'instances

Copie d'instance de classe simple

Aussi étrange que cela puisse paraître, le signe = ne permet pas de recopier une instance de classe. Il permet d'obtenir deux noms différents pour désigner le même objet. Dans l'exemple qui suit, la ligne `nb2 = nb` ne fait pas de copie de l'instance `nb`, elle permet d'obtenir un second nom `nb2` pour l'instance `nb`. Vu de l'extérieur, la ligne `nb2.rnd =`

0 paraît modifier à la fois les objets nb et nb2 puisque les lignes print(nb.rnd) et print(nb2.rnd) affichent la même chose. En réalité, nb et nb2 désignent le même objet.

<<<

```
class exemple_classe:
    def __init__(self):
        self.rnd = 42

    def methode1(self, n):
        self.rnd = 397204094 * self.rnd % 2147483647
        return int(self.rnd % n)


nb = exemple_classe()
nb2 = nb
print(nb.rnd)         # affiche 42
print(nb2.rnd)        # affiche 42

nb2.rnd = 0

print(nb2.rnd)        # affiche 0, comme prévu
print(nb.rnd)         # affiche 0, si nb et nb2 étaient des objets différents,
# cette ligne devrait afficher 42
```

>>>

```
    42
    42
    0
    0
```

Pour créer une copie de l'instance nb, il faut le dire explicitement en utilisant la fonction copy du module copy (voir le chapitre *Modules* (page **??**)).

```
import copy
nom_copy = copy.copy(nom_instance)
```

nom_instance est une instance à copier, nom_copy est le nom désignant la copie. L'exemple suivant applique cette copie sur la classe exemple_classe générant des nombres aléatoires.

<<<

```
class exemple_classe:
    def __init__(self):
        self.rnd = 42

    def methode1(self, n):
        self.rnd = 397204094 * self.rnd % 2147483647
        return int(self.rnd % n)


nb = exemple_classe()

import copy             # pour utiliser le module copy
nb2 = copy.copy(nb)     # copie explicite

print(nb.rnd)     # affiche 42
print(nb2.rnd)    # affiche 42
```

(suite sur la page suivante)

(suite de la page précédente)

```
nb2.rnd = 0

print(nb2.rnd)    # affiche 0
print(nb.rnd)     # affiche 42
```

>>>

```
    42
    42
    0
    42
```

Le symbole égalité ne fait donc pas de copie, ceci signifie qu'une même instance de classe peut porter plusieurs noms.

<<<

```
m = [0, 1]
m2 = m
del m2     # supprime l'identificateur mais pas la liste
print(m)   # affiche [0, 1]
```

>>>

```
    [0, 1]
```

La suppression d'un objet n'est effective que s'il ne reste aucune variable le référençant. L'exemple suivant le montre.

<<<

```
class CreationDestruction (object):

    def __init__(self):
        print("constructeur")

    def __new__(self):
        print("__new__")
        return object.__new__(self)

    def __del__(self):
        print("__del__")


print("a")
m = CreationDestruction()
print("b")
m2 = m
print("c")
del m
print("d")
del m2
print("e")
```

>>>

```
    a
    __new__
    constructeur
```

(suite sur la page suivante)

(suite de la page précédente)

```
    b
    c
    d
    __del__
    e
```

Le destructeur est appelé autant de fois que le constructeur. Il est appelé lorsque plus aucun identificateur n'est relié à l'objet. Cette configuration survient lors de l'exemple précédent car le mot-clé `del` a détruit tous les identificateurs `m` et `m2` qui étaient reliés au même objet.

Copie d'instance de classes incluant d'autres classes

La fonction `copy` n'est pas suffisante lorsqu'une classe inclut des attributs qui sont eux-mêmes des classes incluant des attributs. Dans l'exemple qui suit, la classe `exemple_classe` inclut un attribut de type `classe_incluse` qui contient un attribut `attr`. Lors de la copie à l'aide de l'instruction `nb2 = copy.copy(nb)`, l'attribut `inclus` n'est pas copié, c'est l'instruction `nb2.inclus = nb.inclus` qui est exécutée. On se retrouve donc avec deux noms qui désignent encore le même objet.

<<<

```
class classe_incluse:
    def __init__(self):
        self.attr = 3


class exemple_classe:
    def __init__(self):
        self.inclus = classe_incluse()
        self.rnd = 42


nb = exemple_classe()

import copy              # pour utiliser le module copy
nb2 = copy.copy(nb)      # copie explicite

print(nb.inclus.attr)    # affiche 3
print(nb2.inclus.attr)   # affiche 3

nb2.inclus.attr = 0

print(nb.inclus.attr)    # affiche 0 (on voudrait 3 ici)
print(nb2.inclus.attr)   # affiche 0
```

>>>

```
    3
    3
    0
    0
```

Pour effectivement copier les attributs dont le type est une classe, la première option - la plus simple - est de remplacer la fonction `copy` par la fonction `deepcopy`. Le comportement de cette fonction dans le cas des classes est le même que dans le cas des listes comme l'explique la remarque copy_deepopy_remarque_. La seconde solution, rarement utilisée, est d'utiliser l'opérateur `__copy__` et ainsi écrire le code associé à la copie des attributs de la classe.

Syntaxe S13 : Déclaration de l'opérateur __copy__

```
class nom_classe :
    def __copy__ () :
        copie = nom_classe(...)
        # ...
        return copie
```

---

nom_classe est le nom d'une classe. La méthode __copy__ doit retourner une instance de la classe nom_classe, dans cet exemple, cette instance a pour nom copie.

L'exemple suivant montre un exemple d'implémentation de la classe __copy__. Cette méthode crée d'abord une autre instance copie de la classe exemple_classe puis initialise un par un ses membres. L'attribut rnd est recopié grâce à une affectation car c'est un nombre. L'attribut inclus est recopié grâce à la fonction copy du module copy car c'est une instance de classe. Après la copie, on vérifie bien que modifier l'attribut inclus.attr de l'instance nb ne modifie pas l'attribut inclus.attr de l'instance nb2.

<<<

```
import copy


class classe_incluse:
    def __init__(self): self.attr = 3


class exemple_classe:
    def __init__(self):
        self.inclus = classe_incluse()
        self.rnd = 42

    def __copy__(self):
        copie = exemple_classe()
        copie.rnd = self.rnd
        copie.inclus = copy.copy(self.inclus)
        return copie


nb = exemple_classe()

nb2 = copy.copy(nb)     # copie explicite,
# utilise l'opérateur __copy__,
# cette ligne est équivalente à
# nb2 = nb.__copy__()

print(nb.rnd)           # affiche 42
print(nb2.rnd)          # affiche 42
print(nb.inclus.attr)   # affiche 3
print(nb2.inclus.attr)  # affiche 3

nb.inclus.attr = 0
nb.rnd = 1

print(nb.rnd)           # affiche 1
print(nb2.rnd)          # affiche 42
print(nb.inclus.attr)   # affiche 0
print(nb2.inclus.attr)  # affiche 3 (c'est le résultat souhaité)
```

>>>

```
    42
    42
    3
    3
    1
    42
    0
    3
```

On peut se demander pourquoi l'affectation n'est pas équivalente à une copie. Cela tient au fait que l'affectation en langage *python* est sans cesse utilisée pour affecter le résultat d'une fonction à une variable. Lorsque ce résultat est de taille conséquente, une copie peut prendre du temps. Il est préférable que le résultat de la fonction reçoive le nom prévu pour le résultat.

<<<

```
def fonction_liste():
    return list(range(4, 7))  # retourne la liste [4,5,6]


li = fonction_liste()         # la liste [4,5,6] n'est pas recopiée,
# l'identificateur l lui est affecté
print(li)
```

>>>

```
    [4, 5, 6]
```

Lorsqu'une fonction retourne un résultat mais que celui-ci n'est pas attribué à un nom de variable. Le langage *python* détecte automatiquement que ce résultat n'est plus lié à aucune variable. Il est détruit automatiquement. *python* implémente un mécanisme de garbage collector [165].

<<<

```
def fonction_liste():
    return list(range(4, 7))


fonction_liste()  # la liste [4,5,6] n'est pas recopiée,
# elle n'est pas non plus attribuée à une variable,
# elle est alors détruite automatiquement par le langage Python
```

>>>

```

```

Listes et dictionnaires

Les listes et les dictionnaires sont des types modifiables et aussi des classes. Par conséquent, l'affectation et la copie ont un comportement identique à celui des classes.

<<<

```
l1 = [4, 5, 6]
l2 = l1
print(l1)       # affiche [4, 5, 6]
```

(suite sur la page suivante)

165. https://fr.wikipedia.org/wiki/Ramasse-miettes_(informatique)

(suite de la page précédente)

```
print(l2)        # affiche [4, 5, 6]
l2[1] = 10
print(l1)        # affiche [4, 10, 6]
print(l2)        # affiche [4, 10, 6]
```

>>>

```
    [4, 5, 6]
    [4, 5, 6]
    [4, 10, 6]
    [4, 10, 6]
```

Pour effectuer une copie, il faut écrire le code suivant :

<<<

```
l1 = [4, 5, 6]
import copy
l2 = copy.copy(l1)
print(l1)        # affiche [4, 5, 6]
print(l2)        # affiche [4, 5, 6]
l2[1] = 10
print(l1)        # affiche [4, 5, 6]
print(l2)        # affiche [4, 10, 6]
```

>>>

```
    [4, 5, 6]
    [4, 5, 6]
    [4, 5, 6]
    [4, 10, 6]
```

La fonction copy [166] ne suffit pourtant pas lorsque l'objet à copier est par exemple une liste incluant d'autres objets. Elle copiera la liste et ne fera pas de copie des objets eux-mêmes.

<<<

```
import copy
l0 = [[i] for i in range(0, 3)]
l1 = copy.copy(l0)
print(l0, "  -  ", l1)     # affiche [[0], [1], [2]]   -   [[0], [1], [2]]
l1[0][0] = 6
print(l0, "  -  ", l1)     # affiche [[6], [1], [2]]   -   [[6], [1], [2]]
```

>>>

```
    [[0], [1], [2]]   -   [[0], [1], [2]]
    [[6], [1], [2]]   -   [[6], [1], [2]]
```

Il n'est pas possible de modifier la méthode `__copy__` d'un objet de type liste. Il existe néanmoins la fonction deepcopy [167] qui permet de faire une copie à la fois de la liste et des objets qu'elle contient.

<<<

```
import copy
l0 = [[i] for i in range(0, 3)]
```

(suite sur la page suivante)

166. https://docs.python.org/3/library/copy.html?highlight=copy#copy.copy
167. https://docs.python.org/3/library/copy.html?highlight=copy#copy.deepcopy

(suite de la page précédente)

```
ll = copy.deepcopy(l0)
print(l0, "  -  ", ll)     # affiche [[0], [1], [2]]   -   [[0], [1], [2]]
ll[0][0] = 6
print(l0, "  -  ", ll)     # affiche [[0], [1], [2]]   -   [[6], [1], [2]]
```

>>>

```
    [[0], [1], [2]]   -   [[0], [1], [2]]
    [[0], [1], [2]]   -   [[6], [1], [2]]
```

copy et deepcopy

La fonction `copy` effectue une copie d'un objet, la fonction `deepcopy` effectue une copie d'un objet et de ceux qu'il contient. La fonction `copy` est associée à la méthode `__copy__` tandis que la fonction `deepcopy` est associée à la méthode `__deepcopy__`. Il est rare que l'une de ces deux méthodes doivent être redéfinies. L'intérêt de ce paragraphe est plus de montrer le mécanisme que cache la fonction `deepcopy` qui est la raison pour laquelle il existe deux fonctions de copie et non une seule.

```
import copy
memo = {}
nom_copy = copy.deepcopy (nom_instance [,memo])
```

`nom_instance` est une instance à copier, `nom_copy` est le nom désignant la copie. `memo` est un paramètre facultatif : s'il est envoyé à la fonction `deepcopy`, il contiendra alors la liste de toutes les copies d'objet effectuées lors de cet appel.

```
class nom_classe :
    def __deepcopy__ (self,memo) :
        copie = copy.copy (self)
        # ...
        return copie
```

`nom_classe` est le nom d'une classe. La méthode `__deepcopy__` doit retourner une instance de la classe `nom_classe`, dans cet exemple, cette instance a pour nom `copie`. Le paramètre `memo` permet de conserver la liste des copies effectuées à condition d'appeler `deepcopy` avec un dictionnaire en paramètre.

Le programme suivant reprend le second programme du paragraphe *Copie d'instance de classes incluant d'autres classes* (page 126) et modifie la classe `classe_incluse` pour distinguer copie et copie profonde. Il peut être utile de lire le paragraphe *Clés de type modifiable* (page 29) pour comprendre pourquoi un dictionnaire utilisant comme clé une instance de classe est possible.

<<<

```
import copy


class classe_incluse:
    def __init__(self):
        self.attr = 3


class exemple_classe:
    def __init__(self):
        self.inclus = classe_incluse()
        self.rnd = 42
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def __copy__(self):
        copie = exemple_classe()
        copie.rnd = self.rnd
        return copie

    def __deepcopy__(self, memo):
        if self in memo:
            return memo[self]
        copie = copy.copy(self)
        memo[self] = copie     # mémorise la copie de self qui est copie
        copie.inclus = copy.deepcopy(self.inclus, memo)
        return copie


nb = exemple_classe()

nb2 = copy.deepcopy(nb)    # copie explicite à tous niveaux,
# utilise l'opérateur __copy__,
# cette ligne est équivalente à
# nb2 = nb.__deepcopy__()

print(nb.rnd)            # affiche 42
print(nb2.rnd)           # affiche 42
print(nb.inclus.attr)    # affiche 3
print(nb2.inclus.attr)   # affiche 3

nb.inclus.attr = 0
nb.rnd = 1

print(nb.rnd)            # affiche 1
print(nb2.rnd)           # affiche 42
print(nb.inclus.attr)    # affiche 0
print(nb2.inclus.attr)   # affiche 3  # résultat souhaité
```

>>>

```
    42
    42
    3
    3
    1
    42
    0
    3
```

On peut se demander quel est l'intérêt de la méthode `__deepcopy__` et surtout du paramètre `memo` modifié par la ligne `memo[self] = copie`. Ce détail est important lorsqu'un objet inclut un attribut égal à lui-même ou inclut un objet qui fait référence à l'objet de départ comme dans l'exemple qui suit.

<<<

```
import copy


class Objet1:
    def __init__(self, i):
        self.i = i
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def __str__(self):
        return "o1 " + str(self.i) + " : " + str(self.o2.i)


class Objet2:
    def __init__(self, i, o):
        self.i = i
        self.o1 = o
        o.o2 = self

    def __str__(self):
        return "o2 " + str(self.i) + " : " + str(self.o1.i)

    def __deepcopy__(self, memo):
        return Objet2(self.i, self.o1)


o1 = Objet1(1)
o2 = Objet2(2, o1)
print(o1)  # affiche o1 1 : 2
print(o2)  # affiche o2 2 : 1

o3 = copy.deepcopy(o2)
o3.i = 4
print(o1)  # affiche o1 1 : 4    --> on voudrait 2
print(o2)  # affiche o2 2 : 1
print(o3)  # affiche o2 4 : 1
```

>>>

```
    o1 1 : 2
    o2 2 : 1
    o1 1 : 4
    o2 2 : 1
    o2 4 : 1
```

On modifie le programme comme suit pour obtenir une recopie d'instances de classes qui pointent les unes sur vers les autres. Le paramètre `memo` sert à savoir si la copie de l'objet a déjà été effectuée ou non. Si non, on fait une copie, si oui, on retourne la copie précédemment effectuée et conservée dans `memo`.

<<<

```
import copy


class Objet1:
    def __init__(self, i):
        self.i = i

    def __str__(self):
        return "o1 " + str(self.i) + " : " + str(self.o2.i)

    def __deepcopy__(self, memo={}):
        if self in memo:
            return memo[self]
        r = Objet1(self.i)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        memo[self] = r
        r.o2 = copy.deepcopy(self.o2, memo)
        return r


class Objet2:
    def __init__(self, i, o):
        self.i = i
        self.o1 = o
        o.o2 = self

    def __str__(self):
        return "o2 " + str(self.i) + " : " + str(self.o1.i)

    def __deepcopy__(self, memo={}):
        if self in memo:
            return memo[self]
        r = Objet2(self.i, self.o1)
        memo[self] = r
        r.o1 = copy.deepcopy(self.o1, memo)
        return r


o1 = Objet1(1)
o2 = Objet2(2, o1)

print(o1)  # affiche o1 1 : 2
print(o2)  # affiche o2 2 : 1

o3 = copy.deepcopy(o2)
o3.i = 4
print(o1)  # affiche o1 1 : 2    --> on a 2 cette fois-ci
print(o2)  # affiche o2 2 : 1
print(o3)  # affiche o2 4 : 1
```

>>>

```
    o1 1 : 2
    o2 2 : 1
    o1 1 : 2
    o2 2 : 1
    o2 4 : 1
```

Ces problématiques se rencontrent souvent lorsqu'on aborde le problème de la sérialisation d'un objet qui consiste à enregistrer tout objet dans un fichier, même si cet objet inclut des références à des objets qui font référence à lui-même. C'est ce qu'on appelle des références circulaires. L'enregistrement d'un tel objet avec des références circulaires et sa relecture depuis un fichier se résolvent avec les mêmes artifices que ceux proposés ici pour la copie. L'utilisation des opérateurs `__copy__` et `__deepcopy__` est peu fréquente. Les fonctions `copy` et `deepcopy` du module `copy` suffisent dans la plupart des cas.

3.1.8 Attributs figés

Il arrive parfois qu'une classe contienne peu d'informations et soit utilisée pour créer un très grand nombre d'instances. Les paragraphes précédents ont montré que l'utilisation des attributs était assez souple. Il est toujours possible d'ajouter un attribut à n'importe quelle instance. En contrepartie, chaque instance conserve en mémoire un dictionnaire `__dict__` qui recense tous les attributs qui lui sont associés. Pour une classe susceptible d'être fréquemment instan-

ciée comme un point dans l'espace (voir paragraphe exemple_point_xyz), chaque instance n'a pas besoin d'avoir une liste variable d'attributs. Le langage *python* offre la possibilité de figer cette liste.

Syntaxe S14 : Déclaration d'attributs figés

```
class nom_classe (object) :
    __slots__ = "attribut_1", ..., "attribut_n"
```

`nom_classe` est le nom de la classe, elle doit hériter de `object` ou d'une classe qui en hérite elle-même (voir paragraphe *Héritage* (page 134)). Il faut ensuite ajouter au début du corps de la classe la ligne `__slots__= "attribut_1", ..., "attribut_n"` où `attribut_1` à `attribut_n` sont les noms des attributs de la classe. Aucun autre ne sera accepté.

L'exemple suivant utilise cette syntaxe pour définir un point avec seulement trois attributs `_x`, `_y`, `_z`.

<<<

```
class point_espace(object):
    __slots__ = "_x", "_y", "_z"

    def __init__(self, x, y, z):
        self._x, self._y, self._z = x, y, z

    def __str__(self):
        return "(%f,%f,%f)" % (self._x, self._y, self._z)


a = point_espace(1, -2, 3)
print(a)
```

>>>

```
    (1.000000,-2.000000,3.000000)
```

Etant donné que la liste des attributs est figée, l'instruction `a.j = 6` qui ajoute un attribut `j` à l'instance `a` déclenche une exception. La même erreur se déclenche si on cherche à ajouter cet attribut depuis une méthode (`self.j=6`). L'attribut `__dict__` n'existe pas non plus, par conséquent, l'expression `a.__dict__` génère la même exception. La présence de l'instruction `__slots__ = ...` n'a aucun incidence sur les attributs statiques.

3.1.9 Héritage

L'héritage est un des grands avantages de la programmation objet. Il permet de créer une classe à partir d'une autre en ajoutant des attributs, en modifiant ou en ajoutant des méthodes. En quelque sorte, on peut modifier des méthodes d'une classe tout en conservant la possibilité d'utiliser les anciennes versions.

Exemple autour de pièces de monnaie

On désire réaliser une expérience à l'aide d'une pièce de monnaie. On effectue cent tirages successifs et on compte le nombre de fois où la face pile tombe. Le programme suivant implémente cette expérience sans utiliser la programmation objet.

<<<

```
import random  # extension interne incluant des fonctions
# simulant des nombres aléatoires,
# random.randint (a,b) --> retourne un nombre entier entre a et b
```

(suite sur la page suivante)

(suite de la page précédente)

```
# cette ligne doit être ajoutée à tous les exemples suivant
# même si elle n'y figure plus


def cent_tirages():
    s = 0
    for i in range(0, 100):
        s += random.randint(0, 1)
    return s


print(cent_tirages())
```

>>>

```
    48
```

On désire maintenant réaliser cette même expérience pour une pièce truquée pour laquelle la face pile sort avec une probabilité de `0,7`. Une solution consiste à réécrire la fonction `cent_tirages` pour la pièce truquée.

<<<

```
import random


def cent_tirages():
    s = 0
    for i in range(0, 100):
        t = random.randint(0, 10)
        if t >= 3:
            s += 1
    return s


print(cent_tirages())
```

>>>

```
    77
```

Toutefois cette solution n'est pas satisfaisante car il faudrait réécrire cette fonction pour chaque pièce différente pour laquelle on voudrait réaliser cette expérience. Une autre solution consiste donc à passer en paramètre de la fonction `cent_tirages` une fonction qui reproduit le comportement d'une pièce, qu'elle soit normale ou truquée.

<<<

```
import random


def piece_normale():
    return random.randint(0, 1)


def piece_truquee():
    t = random.randint(0, 10)
    if t >= 3:
        return 1
    else:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return 0


def cent_tirages(piece):
    s = 0
    for i in range(0, 100):
        s += piece()
    return s


print(cent_tirages(piece_normale))
print(cent_tirages(piece_truquee))
```

>>>

```
    52
    75
```

Mais cette solution possède toujours un inconvénient car les fonctions associées à chaque pièce n'acceptent aucun paramètre. Il n'est pas possible de définir une pièce qui est normale si la face *pile* vient de sortir et qui devient truquée si la face *face* vient de sortir. On choisit alors de représenter une pièce normale par une classe.

<<<

```
import random


class piece_normale:
    def tirage(self):
        return random.randint(0, 1)

    def cent_tirages(self):
        s = 0
        for i in range(0, 100):
            s += self.tirage()
        return s


p = piece_normale()
print(p.cent_tirages())
```

>>>

```
    53
```

On peut aisément recopier et adapter ce code pour la pièce truquée.

<<<

```
import random


class piece_normale:
    def tirage(self):
        return random.randint(0, 1)

    def cent_tirages(self):
        s = 0
```

(suite sur la page suivante)

(suite de la page précédente)

```
        for i in range(0, 100):
            s += self.tirage()
        return s


class piece_truquee:
    def tirage(self):
        t = random.randint(0, 10)
        if t >= 3:
            return 1
        else:
            return 0

    def cent_tirages(self):
        s = 0
        for i in range(0, 100):
            s += self.tirage()
        return s


p = piece_normale()
print(p.cent_tirages())
p2 = piece_truquee()
print(p2.cent_tirages())
```

>>>

```
    49
    72
```

Toutefois, pour les deux classes `piece_normale` et `piece_truquee`, la méthode `cent_tirage` est exactement la même. Il serait préférable de ne pas répéter ce code puisque si nous devions modifier la première - un nombre de tirages différent par exemple -, il faudrait également modifier la seconde. La solution passe par l'héritage. On va définir la classe `piece_truquee` à partir de la classe `piece_normale` en remplaçant seulement la méthode `tirage` puisqu'elle est la seule à changer.

On indique à la classe `piece_truquee` qu'elle hérite - ou dérive - de la classe `piece_normale` en mettant `piece_normale` entre parenthèses sur la ligne de la déclaration de la classe `piece_truquee`. Comme la méthode `cent_tirages` ne change pas, elle n'a pas besoin d'apparaître dans la définition de la nouvelle classe même si cette méthode est aussi applicable à une instance de la classe `piece_truquee`.

<<<

```
import random


class piece_normale:
    def tirage(self):
        return random.randint(0, 1)

    def cent_tirages(self):
        s = 0
        for i in range(0, 100):
            s += self.tirage()
        return s
```

(suite sur la page suivante)

(suite de la page précédente)

```
class piece_truquee (piece_normale):
    def tirage(self):
        t = random.randint(0, 10)
        if t >= 3:
            return 1
        else:
            return 0


p = piece_normale()
print(p.cent_tirages())
p2 = piece_truquee()
print(p2.cent_tirages())
```

>>>

```
    53
    69
```

Enfin, on peut définir une pièce très truquée qui devient truquée si *face* vient de sortir et qui redevient normale si *pile* vient de sortir. Cette pièce très truquée sera implémentée par la classe `piece_tres_truquee`. Elle doit contenir un attribut `avant` qui conserve la valeur du précédent tirage. Elle doit redéfinir la méthode `tirage` pour être une pièce normale ou truquée selon la valeur de l'attribut `avant`. Pour éviter de réécrire des méthodes déjà écrites, la méthode `tirage` de la classe `piece_tres_truquee` doit appeler la méthode `tirage` de la classe `piece_truquee` ou celle de la classe `piece_normale` selon la valeur de l'attribut `avant`.

<<<

```
import random


class piece_normale:
    def tirage(self):
        return random.randint(0, 1)

    def cent_tirages(self):
        s = 0
        for i in range(0, 100):
            s += self.tirage()
        return s


class piece_truquee (piece_normale):
    def tirage(self):
        t = random.randint(0, 10)
        if t >= 3:
            return 1
        else:
            return 0


class piece_tres_truquee (piece_truquee):
    def __init__(self):
        # création de l'attribut avant
        self.avant = 0

    def tirage(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if self.avant == 0:
            # appel de la méthode tirage de la classe piece_truquee
            self.avant = piece_truquee.tirage(self)
        else:
            # appel de la méthode tirage de la classe piece_normale
            self.avant = piece_normale.tirage(self)
        return self.avant


p = piece_normale()
print("normale ", p.cent_tirages())
p2 = piece_truquee()
print("truquee ", p2.cent_tirages())
p3 = piece_tres_truquee()
print("tres truquee ", p3.cent_tirages())
```

>>>

```
    normale  43
    truquee  77
    tres truquee  62
```

L'héritage propose donc une manière élégante d'organiser un programme. Il rend possible la modification des classes d'un programme sans pour autant les altérer.

Définition D7 : héritage

On dit qu'une classe B hérite d'une autre classe A si la déclaration de B inclut les attributs et les méthodes de la classe A.

La surcharge est un autre concept qui va de pair avec l'héritage. Elle consiste à redéfinir des méthodes déjà définies chez l'ancêtre. Cela permet de modifier le comportement de méthodes bien que celles-ci soient utilisées par d'autres méthodes dont la définition reste inchangée.

Définition D8 : surcharge

Lorsqu'une classe *B* hérite de la classe *A* et redéfinit une méthode de la classe *A* portant le même nom, on dit qu'elle surcharge cette méthode. S'il n'est pas explicitement précisé qu'on fait appel à une méthode d'une classe donnée, c'est toujours la méthode surchargée qui est exécutée.

Syntaxe

L'héritage obéit à la syntaxe suivante.

Syntaxe S15 : Héritage

```
class nom_classe (nom_ancetre) :
    # corps de la classe
    # ...
```

`nom_classe` désigne le nom d'une classe qui hérite ou dérive d'une autre classe `nom_ancetre`. Celle-ci `nom_ancetre` doit être une classe déjà définie.

L'utilisation de la fonction `help` permet de connaître tous les ancêtres d'une classe. On applique cette fonction à la classe `piece_tres_truquee` définie au paragraphe précédent.

```
help (piece_tres_truquee)
```

On obtient le résultat suivant :

```
Help on class piece_tres_truquee in module __main__:

class piece_tres_truquee(piece_truquee)
 |  Method resolution order:
 |      piece_tres_truquee
 |      piece_truquee
 |      piece_normale
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |
 |  tirage(self)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from piece_normale:
 |
 |  cent_tirages(self)
```

La rubrique Method Resolution Order[168] indique la liste des héritages successifs qui ont mené à la classe `piece_tres_truquee`. Cette rubrique indique aussi que, lorsqu'on appelle une méthode de la classe `piece_tres_truquee`, si elle n'est pas redéfinie dans cette classe, le langage *python* la cherchera chez l'ancêtre direct, ici, la classe `piece_truquee`. Si elle ne s'y trouve toujours pas, *python* ira la chercher aux niveaux précédents jusqu'à ce qu'il la trouve.

L'attribut `__bases__` d'une classe (voir paragraphe *Attributs implicites* (page 103)) contient le (ou les ancêtres, voir paragraphe *Héritage multiple* (page 144)). Il suffit d'interroger cet attribut pour savoir si une classe hérite d'une autre comme le montre l'exemple suivant.

<<<

```
class piece_normale:
    pass


class piece_truquee(piece_normale):
    pass


class piece_tres_truquee(piece_truquee):
    pass


for l in piece_tres_truquee.__bases__:
    print(l)    # affiche __main__.piece_truquee
print(piece_normale in piece_tres_truquee.__bases__)  # affiche False
print(piece_truquee in piece_tres_truquee.__bases__)  # affiche True
```

>>>

168. https://www.python.org/download/releases/2.3/mro/

```
    <class 'pyquickhelper.sphinxext.sphinx_runpython_extension.run_python_script_
↪139791940545848.<locals>.piece_truquee'>
    False
    True
```

La fonction issubclass [169] permet d'obtenir un résultat équivalent. `issubclass(A,B)` indique si la classe `A` hérite directement ou indirectement de la classe `B`. Le paragraphe *Fonctions issubclass et isinstance* (page 146) revient sur cette fonction.

<<<

```
class piece_normale:
    pass


class piece_truquee(piece_normale):
    pass


class piece_tres_truquee(piece_truquee):
    pass


print(issubclass(piece_tres_truquee, piece_normale))  # affiche True
print(issubclass(piece_truquee, piece_normale))       # affiche True
```

>>>

```
    True
    True
```

Dans les exemples précédents, `piece_normale` ne dérive d'aucune autre classe. Toutefois, le langage *python* propose une classe d'objets dont héritent toutes les autres classes définies par le langage : c'est la classe `object`. Les paragraphes précédents ont montré qu'elle offrait certains avantages (voir paragraphe *Propriétés* (page 121) sur les propriétés ou encore paragraphe *Attributs figés* (page 133) sur les attributs non liés).

Le paragraphe précédent a montré qu'il était parfois utile d'appeler dans une méthode une autre méthode appartenant explicitement à l'ancêtre direct de cette classe ou à un de ses ancêtres. La syntaxe est la suivante.

Syntaxe S16 : Surcharge de méthodes héritées

```
class nom_classe (nom_ancetre) :
    def nom_autre_methode (self, ...) :
        # ...
    def nom_methode (self, ...) :
        nom_ancetre.nom_methode (self, ...)
            # appel de la méthode définie chez l'ancêtre
        nom_ancetre.nom_autre_methode (self, ...)
            # appel d'une autre méthode définie chez l'ancêtre
        self.nom_autre_methode (...)
            # appel d'une méthode surchargée
```

`nom_classe` désigne le nom d'une classe, `nom_ancetre` est le nom de la classe dont `nom_classe` hérite ou dérive. `nom_methode` est une méthode surchargée qui appelle la méthode portant le même nom mais définie dans la classe `nom_ancetre` ou un de ses ancêtres. `nom_autre_methode` est une autre méthode. La mé-

169. https://docs.python.org/3/library/functions.html?highlight=issubclass#issubclass

thode nom_methode de la classe nom_classe peut faire explicitement appel à une méthode définie chez l'ancêtre nom_ancetre même si elle est également surchargée ou faire appel à la méthode surchargée.

Ces appels sont très fréquents en ce qui concerne les constructeurs qui appellent le constructeur de l'ancêtre. Il est même conseillé de le faire à chaque fois.

<<<

```
class A:
    def __init__(self):
        self.x = 0


class B (A):
    def __init__(self):
        A.__init__(self)
        self.y = 0
```

>>>

```

```

Contrairement aux méthodes, la surcharge d'attributs n'est pas possible. Si un ancêtre possède un attribut d'identificateur a, les classes dérivées le possèdent aussi et ne peuvent en déclarer un autre du même nom. Cela tient au fait que quelque soit la méthode utilisée, celle de l'ancêtre ou celle d'une classe dérivée, c'est le même dictionnaire d'attributs __dict__ qui est utilisé. En revanche, si la classe ancêtre déclare un attribut dans son constructeur, il ne faut pas oublier de l'appeler dans le constructeur de la classe fille afin que cette attribut existe pour la classe fille.

<<<

```
class ancetre:
    def __init__(self):
        self.a = 5

    def __str__(self):
        return "a = " + str(self.a)


class fille (ancetre):
    def __init__(self):
        ancetre.__init__(self)      # cette ligne est importante
        # car sans elle, l'attribut a n'existe pas
        self.a += 1

    def __str__(self):
        s = "a = " + str(self.a)
        return s


x = ancetre()
print(x)          # affiche a = 5
y = fille()
print(y)          # affiche a = 6
```

>>>

```
    a = 5
    a = 6
```

Sens de l'héritage

Il n'est pas toujours évident de concevoir le sens d'un héritage. En mathématique, le carré est un rectangle dont les côtés sont égaux. A priori, une classe `carre` doit dériver d'une classe `rectangle`.

<<<

```
class rectangle:
    def __init__(self, a, b):
        self.a, self.b = a, b

    def __str__(self):
        return "rectangle " + str(self.a) + " x " + str(self.b)


class carre (rectangle):
    def __init__(self, a):
        rectangle.__init__(self, a, a)


r = rectangle(3, 4)
print(r)  # affiche rectangle 3 x 4
c = carre(5)
print(c)  # affiche rectangle 5 x 5
```

>>>

```
    rectangle 3 x 4
    rectangle 5 x 5
```

Toutefois, on peut aussi considérer que la classe `carre` contient une information redondante puisqu'elle possède deux attributs qui seront toujours égaux. On peut se demander s'il n'est pas préférable que la classe `rectangle` hérite de la classe `carre`.

<<<

```
class carre:
    def __init__(self, a):
        self.a = a

    def __str__(self):
        return "carre " + str(self.a)


class rectangle (carre):
    def __init__(self, a, b):
        carre.__init__(self, a)
        self.b = b

    def __str__(self):
        return "rectangle " + str(self.a) + " x " + str(self.b)


r = rectangle(3, 4)
print(r)  # affiche rectangle 3 x 4
c = carre(5)
print(c)  # affiche carre 5
```

>>>

```
    rectangle 3 x 4
    carre 5
```

Cette seconde version minimise l'information à mémoriser puisque la classe `carre` ne contient qu'un seul attribut et non deux comme dans l'exemple précédent. Néanmoins, il a fallu surcharger l'opérateur `__str__` afin d'afficher la nouvelle information.

Il n'y a pas de meilleur choix parmi ces deux solutions proposées. La première solution va dans le sens des propriétés croissantes, les méthodes implémentées pour les classes de bases restent vraies pour les suivantes. La seconde solution va dans le sens des attributs croissants, des méthodes implémentées pour les classes de bases doivent souvent être adaptées pour les héritiers. En contrepartie, il n'y a pas d'information redondante.

Ce problème d'héritage ne se pose pas à chaque fois. Dans l'exemple du paragraphe *Exemple autour de pièces de monnaie* (page 134) autour des pièces truquées, il y a moins d'ambiguïté sur le sens de l'héritage. Celui-ci est guidé par le problème à résoudre qui s'avère plus simple à concevoir dans le sens d'un héritage d'une pièce normale vers une pièce truquée.

Dans le cas des classes `carre` et `rectangle`, il n'est pas possible de déterminer la meilleure solution tant que leur usage ultérieur n'est pas connu. Ce problème revient également lorsqu'on définit des matrices et des vecteurs. Un vecteur est une matrice d'une seul colonne, il ne possède qu'une seule dimension au lieu de deux pour une matrice. L'exercice exeercie_heritage_multiple revient sur ce dilemme.

Héritage multiple

Jusqu'à présent, tous les exemples d'héritages entre classes n'ont fait intervenir que deux classes, la classe ancêtre dont hérite la classe descendante. L'héritage multiple part du principe qu'il peut y avoir plusieurs ancêtres pour une même classe. La classe descendante hérite dans ce cas de tous les attributs et méthodes de tous ses ancêtres.

Dans l'exemple qui suit, la classe `C` hérite des classes `A` et `B`. Elle hérite donc des méthodes de `carre` et `cube`. Chacune des classes `A` et `B` contient un constructeur qui initialise l'attribut `a`. Le constructeur de la classe `C` appelle le constructeur de la classe `A` pour initialiser cet attribut.

<<<

```
class A:
    def __init__(self):
        self.a = 5

    def carre(self):
        return self.a ** 2


class B:
    def __init__(self):
        self.a = 6

    def cube(self):
        return self.a ** 3


class C (A, B):
    def __init__(self):
        A.__init__(self)


x = C()
print(x.carre())    # affiche 25
print(x.cube())     # affiche 125
```

>>>

```
    25
    125
```

Mais ces héritages multiples peuvent parfois apporter quelques ambiguïtés comme le cas où au moins deux ancêtres possèdent une méthode du même nom. Dans l'exemple qui suit, la classe `C` hérite toujours des classes `A` et `B`. Ces deux classes possèdent une méthode `calcul`. La classe `C`, qui hérite des deux, possède aussi une méthode `calcul` qui, par défaut, sera celle de la classe `A`.

<<<

```
class A:
    def __init__(self):
        self.a = 5

    def calcul(self):
        return self.a ** 2


class B:
    def __init__(self):
        self.a = 6

    def calcul(self):
        return self.a ** 3


class C (A, B):
    def __init__(self):
        A.__init__(self)


x = C()
print(x.calcul())  # affiche 25
```

>>>

```
    25
```

Cette information est disponible via la fonction `help` appliquée à la classe `C`. C'est dans ce genre de situations que l'information apportée par la section Method Resolution Order [170] est importante.

```
class C(A, B)
 |  Method resolution order:
 |      C
 |      A
 |      B
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |
 |  calcul(self)
```

Pour préciser que la méthode `calcul` de la classe `C` doit appeler la méthode `calcul` de la classe `B` et non `A`, il faut l'écrire explicitement en surchargeant cette méthode.

<<<

170. https://www.python.org/download/releases/2.3/mro/

```
class A:
    def __init__(self):
        self.a = 5

    def calcul(self):
        return self.a ** 2


class B:
    def __init__(self):
        self.a = 6

    def calcul(self):
        return self.a ** 3


class C (A, B):
    def __init__(self):
        A.__init__(self)

    def calcul(self):
        return B.calcul(self)


x = C()
print(x.calcul())  # affiche 125
```

>>>

```
    125
```

L'exemple précédent est un cas particulier où il n'est pas utile d'appeler les constructeurs des deux classes dont la classe `C` hérite mais c'est un cas particulier. Le constructeur de la classe `C` devrait être ainsi :

```
class C (A,B) :
    def __init__ (self):
        A.__init__(self)
        B.__init__ (self)
```

Fonctions issubclass et isinstance

La fonction issubclass [171] permet de savoir si une classe hérite d'une autre.

```
issubclass (B, A)
```

Le résultat de cette fonction est vrai si la classe `B` hérite de la classe `A`, le résultat est faux dans tous les autres cas. La fonction prend comme argument des classes et non des instances de classes.

L'exemple qui suit utilise cette fonction dont le résultat est vrai même pour des classes qui n'héritent pas directement l'une de l'autre.

<<<

```
class A (object):
    pass
```

(suite sur la page suivante)

171. https://docs.python.org/3/library/functions.html?highlight=issubclass#issubclass

(suite de la page précédente)

```
class B (A):
    pass


class C (B):
    pass


print(issubclass(A, B))      # affiche False
print(issubclass(B, A))      # affiche True
print(issubclass(A, C))      # affiche False
print(issubclass(C, A))      # affiche True
```

>>>

```
    False
    True
    False
    True
```

Lorsqu'on souhaite appliquer la fonction à une instance de classe, il faut faire appel à l'attribut `__class__`. En reprenant les classes définies par l'exemple précédant cela donne :

<<<

```
class A (object):
    pass


class B (A):
    pass


class C (B):
    pass


a = A()
b = B()

print(issubclass(a.__class__, B))      # affiche False
print(issubclass(b.__class__, A))      # affiche True
print(issubclass(a.__class__, A))      # affiche True
```

>>>

```
    False
    True
    True
```

La fonction isinstance [172] permet de savoir si une instance de classe est d'une type donné. Elle est équivalente à la fonction `issubclass` à ceci près qu'elle prend comme argument une instance et une classe. L'exemple précédent devient avec la fonction `isinstance` :

<<<

172. https://docs.python.org/3/library/functions.html?highlight=isinstance#isinstance

```
class A (object):
    pass


class B (A):
    pass


class C (B):
    pass


a = A()
b = B()
print(isinstance(a, B))      # affiche False
print(isinstance(b, A))      # affiche True
print(isinstance(a, A))      # affiche True
```

>>>

```
    False
    True
    True
```

L'utilisation des fonctions `issubclass` et `isinstance` n'est pas très fréquente mais elle permet par exemple d'écrire une fonction qui peut prendre en entrée des types variables.

<<<

```
def fonction_somme_list(ens):
    r = "list "
    for e in ens:
        r += e
    return r


def fonction_somme_dict(ens):
    r = "dict "
    for k, v in ens.items():
        r += v
    return r


def fonction_somme(ens):
    if isinstance(ens, dict):
        return fonction_somme_dict(ens)
    elif isinstance(ens, list):
        return fonction_somme_list(ens)
    else:
        return "erreur"


li = ["un", "deux", "trois"]
di = {1: "un", 2: "deux", 3: "trois"}
tu = ("un", "deux", "trois")

print(fonction_somme(li))   # affiche list undeuxtrois
print(fonction_somme(di))   # affiche dict undeuxtrois
print(fonction_somme(tu))   # affiche erreur
```

>>>

```
    list undeuxtrois
    dict undeuxtrois
    erreur
```

L'avantage est d'avoir une seule fonction capable de s'adapter à différents type de variables, y compris des types créés par un programmeur en utilisant les classes.

3.1.10 Compilation de classes

La compilation de classe fonctionne de manière similaire à celle de la compilation de fonctions (voir paragraphe *Compilation dynamique (compile, exec)* (page 70)). Il s'agit de définir une classe sous forme de chaîne de caractères puis d'appeler la fonction `compile` pour ajouter cette classe au programme et s'en servir. L'exemple suivant reprend deux classes décrites au paragraphe *Sens de l'héritage* (page 143). Elles sont incluses dans une chaîne de caractères, compilées puis incluses au programme (fonction `exec`).

<<<

```
s = """class carre :
    def __init__( self, a) : self.a = a
    def __str__ (self)     : return "carre " + str (self.a)

class rectangle (carre):
    def __init__(self,a,b) :
        carre.__init__(self, a)
        self.b = b
    def __str__ (self) :
        return "rectangle " + str(self.a) +  " x " +  str (self.b)
"""

obj = compile(s, "", "exec")        # code à compiler
exec(obj)                           # classes incorporées au programme

r = rectangle(3, 4)
print(r)  # affiche rectangle 3 x 4
c = carre(5)
print(c)  # affiche carre 5
```

>>>

```
    [runpythonerror]
    Traceback (most recent call last):
      File "<stdin>", line 17, in <module>
      File "", line 5
        class rectangle (carre):
                                ^
    IndentationError: unindent does not match any outer indentation level
```

Comme toute fonction, la fonction `compile` génère une exception lorsque la chaîne de caractères contient une erreur. Le programme qui suit essaye de compiler une chaîne de caractères confondant `self` et `seilf`.

<<<

```
"""erreur de compilation incluses dans le code inséré dans la
chaîne de caractère s"""
s = """class carre :
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def __init__( self, a) :
        seilf.a = a   # erreur de compilation
    def __str__ (self) :
        return "carre " + str (self.a) """

obj = compile(s, "variable s", "exec")    # code à compiler
exec(obj)                                 # classes incorporées au programme

c = carre(5)
print(c)      # affiche carre 5
```

>>>

```
    [runpythonerror]
    Traceback (most recent call last):
      File "<stdin>", line 16, in <module>
      File "variable s", line 3, in __init__
    NameError: name 'seilf' is not defined
```

Le message d'erreur est le même que pour un programme ne faisant pas appel à la fonction `compile` à ceci près que le fichier où a lieu l'erreur est `variable s` qui est le second paramètre envoyé à la fonction `compile`. Le numéro de ligne fait référence à la troisième ligne de la chaîne de caractères `s` et non à la troisième ligne du programme.

<<<

```
class fromage:

    def __init__(self, p, c, o):
        self.poids = p
        self.couleur = c
        self.odeur = o

    def decouper(self, nb):
        l = []
        for i in range(0, nb):
            f = fromage(self.poids/nb,
                        self.couleur, self.odeur)
            l.append(f)
        return l

    def __str__(self):
        s = "poids : " + str(self.poids)
        s += " couleur : " + str(self.couleur)
        s += " odeur : " + str(self.odeur)
        return s

    def __add__(self, f):
        print("ajout fromage")
        poids = self.poids + f.poids
        couleur = [0, 0, 0]
        for i in range(0, 3):
            couleur[i] = (self.couleur[i] * self.poids
                          + f.couleur[i] * f.poids) / poids
        odeur = (self.odeur * self.poids +
                 f.odeur * f.poids) / poids
        couleur = (couleur[0], couleur[1], couleur[2])
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return fromage(poids, couleur, odeur)


class gruyere(fromage):
    def __init__(self, p):
        fromage.__init__(self, p, c=(150, 150, 0), o=0.1)

    def __str__(self):
        s = fromage.__str__(self)
        s = "gruyère, " + s
        return s

    def __add__(self, f):
        print("ajout gruyère")
        if not isinstance(f, gruyere):
            return fromage.__add__(self, f)
        else:
            r = gruyere(self.poids + f.poids)
            return r


# ------------------------------------------
fr = fromage(5.0, (255, 0, 0), 0.5)
fr2 = fromage(10.0, (0, 255, 0), 1)
fr3 = fr + fr2
print(fr)
print(fr2)
print(fr3)
print("----------------------")
g = gruyere(3.0)
g2 = gruyere(7.0)
g3 = g + g2
print(g)
print(g2)
print(g3)
print("----------------------")
print(fr2 + g)
```

>>>

```
    ajout fromage
    poids : 5.0 couleur : (255, 0, 0) odeur : 0.5
    poids : 10.0 couleur : (0, 255, 0) odeur : 1
    poids : 15.0 couleur : (85.0, 170.0, 0.0) odeur : 0.8333333333333334
    ----------------------
    ajout gruyère
    gruyère, poids : 3.0 couleur : (150, 150, 0) odeur : 0.1
    gruyère, poids : 7.0 couleur : (150, 150, 0) odeur : 0.1
    gruyère, poids : 10.0 couleur : (150, 150, 0) odeur : 0.1
    ----------------------
    ajout fromage
    poids : 13.0 couleur : (34.61538461538461, 230.76923076923077, 0.0) odeur : 0.
↪7923076923076924
```

3.1.11 Constructions classiques

Héritage

Le premier exemple est classique puisqu'il reprend le programme du paragraphe paragraphe_fonction_variable pour le réécrire avec des classes et éviter de passer des fonctions comme paramètre d'une autre fonction. La première classe définit le module des suivantes. La méthode `calcul` n'accepte qu'un seul paramètre `x` mais pourrait également prendre en compte des constantes si celles-ci sont renseignées via le constructeur de la classe. C'est l'avantage de cette solution déjà illustrée par les pièces normales et truquées.

<<<

```
class Fonction:
    def calcul(self, x):
        pass

    def calcul_n_valeur(self, l):
        res = [self.calcul(i) for i in l]
        return res


class Carre (Fonction):
    def calcul(self, x):
        return x*x


class Cube (Fonction):
    def calcul(self, x):
        return x*x*x


li = [0, 1, 2, 3]
print(li)   # affiche [0, 1, 2, 3]

l1 = Carre().calcul_n_valeur(li)  # l1 vaut [0, 1, 4, 9]
l2 = Cube().calcul_n_valeur(li)   # l2 vaut [0, 1, 8, 27]
print(l1)
print(l2)
```

>>>

```
    [0, 1, 2, 3]
    [0, 1, 4, 9]
    [0, 1, 8, 27]
```

La version suivante mélange héritage et méthodes envoyées comme paramètre à une fonction. Il est préférable d'éviter cette construction même si elle est très utilisée par les interfaces graphiques. Elle n'est pas toujours transposable dans tous les langages de programmation tandis que le programme précédent aura un équivalent dans tous les langages objet.

<<<

```
class Fonction:
    def calcul(self, x):
        pass


class Carre(Fonction):
    def calcul(self, x):
        return x*x
```

(suite sur la page suivante)

(suite de la page précédente)

```
class Cube(Fonction):
    def calcul(self, x):
        return x*x*x


def calcul_n_valeur(l, f):
    res = [f(i) for i in l]
    return res


l = [0, 1, 2, 3]
l1 = calcul_n_valeur(l, Carre().calcul)  # l1 vaut [0, 1, 4, 9]
l2 = calcul_n_valeur(l, Cube().calcul)  # l2 vaut [0, 1, 8, 27]
print(l1)
print(l2)
```

>>>

```
    [0, 1, 4, 9]
    [0, 1, 8, 27]
```

Deux lignées d'héritages

C'est une configuration qui arrive fréquemment lorsqu'on a d'un côté différentes structures pour les mêmes données et de l'autre différents algorithmes pour le même objectif. Par exemple, une matrice peut être définie comme une liste de listes ou un dictionnaire de tuples (voir exercice exercice_dame_dico_matrice). La multiplication de deux matrices peut être une multiplication classique ou celle de Strassen [173]. L'algorithme de multiplication de deux matrices de Strassen est plus rapide qu'une multiplication classique pour de grandes matrices. Son coût est en $O(n^{\frac{\ln 7}{\ln 2}}) \sim O(n^{2,807})$ au lieu de $O(n^3)$.

La question est comment modéliser ces deux structures et ces deux multiplications sachant que les quatre paires *structure - algorithme* doivent fonctionner. On pourrait simplement créer deux classes faisant référence aux deux structures différentes et à l'intérieur de chacune d'entre elles, avoir deux méthodes de multiplication. Néanmoins, si une nouvelle structure ou un nouvel algorithme apparaît, la mise à jour peut être fastidieuse.

Il est conseillé dans ce cas d'avoir quatre classes et de définir une interface d'échanges communes. L'algorithme de multiplication ne doit pas savoir quelle structure il manipule : il doit y accéder par des méthodes. De cette manière, c'est la classe qui indique l'algorithme choisi et non une méthode. Ajouter un troisième algorithme ou une troisième structure revient à ajouter une classe : l'interface d'échange ne change pas. Le programme pourrait suivre le schéma qui suit.

```
class Matrice :
    def __init__ (self,lin,col,coef):
        self.lin, self.col = lin, col

    # interface d'échange
    def get_lin () : return self.lin
    def get_col () : return self.col
    def __getitem__(self,i,j): pass
    def __setitem__(self,i,j,v): pass
    def get_submat(self, i1,j1,i2,j2): pass
    def set_submat(self, i1,j1,mat): pass
    # fin de l'interface d'échange
```

(suite sur la page suivante)

173. https://fr.wikipedia.org/wiki/Algorithme_de_Strassen

(suite de la page précédente)

```
    def trace (self) :
        t = 0
        for i in range(0, self.lin):
            t += self (i,i)
        return t

class MatriceList (Matrice) :
    def __init__ (self,lin,col,coef):
        Matrice.__init__ (self, \
                      lin, col, coef)
        #...

    def __getitem__ (self, i,j) : #...
    def __setitem__ (self, i,j, v) : #...
    def get_submat(self, i1,j1,i2,j2): #...
    def set_submat(self, i1,j1,mat): #...

class MatriceDict (Matrice) :
    def __init__ (self,lin,col,coef):
        Matrice.__init__ (self, \
                       lin, col, coef)
        #...

    def __getitem__ (self, i,j) : #...
    def __setitem__ (self, i,j, v) : #...
    def get_submat(self, i1,j1,i2,j2): #...
    def set_submat(self, i1,j1,mat): #...
```

Illustration :

```
class Produit :
    def calcul (self, mat1, mat2):
        pass

class ProduitClassique (Produit) :
    def calcul (self, mat1, mat2):
        #...
        return

class ProduitStrassen (Produit) :
    def calcul (self, mat1,mat2):
        #...
        return
```

Cette construction autorise même la multiplication de matrices de structures différentes. Très répandue, cette architecture est souvent plus coûteuse au moment de la conception car il faut bien penser l'interface d'échange mais elle l'est beaucoup moins par la suite. Il existe d'autres assemblages de classes assez fréquents, regroupés sous le terme de Design Patterns [174]. Pour peu que ceux-ci soient connus de celui qui conçoit un programme, sa relecture et sa compréhension en sont facilitées si les commentaires font mention du *pattern* utilisé.

174. https://fr.wikipedia.org/wiki/Patron_de_conception

CHAPITRE 4

Exceptions

4.1 Exceptions

Le petit programme suivant déclenche une erreur parce qu'il effectue une division par zéro.

```
x = 0
y = 1.0 / x
```

Il déclenche une erreur ou ce qu'on appelle une *exception* :

```
Traceback (most recent call last):
  File "cours.py", line 2, in ?
    y = 1.0 / x
ZeroDivisionError: float division
```

Le mécanisme des exceptions permet au programme de ńărattraperă" les erreurs, de détecter qu'une erreur s'est produite et d'agir en conséquence afin que le programme ne s'arrête pas.

4.1.1 Principe des exceptions

Attraper toutes les erreurs

Une exception est un objet qui indique que le programme ne peut continuer son exécution. Le type de l'exception donne une indication sur le type de l'erreur rencontrée. L'exception contient généralement un message plus détaillé. Toutes les exceptions hérite du type Exception [175].

On décide par exemple qu'on veut rattraper toutes les erreurs du programme et afficher un message d'erreur. Le programme suivant appelle la fonction qui retourne l'inverse d'un nombre.

```
def inverse (x):
    y = 1.0 / x
    return y

a = inverse(2)
print(a)
b = inverse(0)
print(b)
```

Lorsque `x == 0`, le programme effectue une division par zéro et déclenche une erreur. L'interpréteur Python affiche ce qu'on appelle la *pile d'appels* ou pile d'exécution [176]. La pile d'appel permet d'obtenir la liste de toutes les fonctions pour remonter jusqu'à celle où l'erreur s'est produite.

```
Traceback (most recent call last):
  File "cours.py", line 8, in ?
    b = inverse (0)
  File "cours.py", line 3, in inverse
    y = 1.0 / x
ZeroDivisionError: float division
```

Afin de rattraper l'erreur, on insère le code susceptible de produire une erreur entre les mots clés try [177] et except [178].

<<<

```
def inverse(x):
    y = 1.0 / x
    return y


try:
    a = inverse(2)
    print(a)
    b = inverse(0)   # déclenche une exception
    print(b)
except:
    print("le programme a déclenché une erreur")
```

>>>

```
    0.5
    le programme a déclenché une erreur
```

Le programme essaye d'exécuter les quatre instructions incluses entre les instructions `try` et `except`. Si une erreur se produit, le programme exécute alors les lignes qui suivent l'instruction `except`. L'erreur se produit en fait à l'intérieur

175. https://docs.python.org/3/library/exceptions.html#Exception
176. https://fr.wikipedia.org/wiki/Pile_d%27ex%C3%A9cution
177. https://docs.python.org/3/reference/compound_stmts.html#try
178. https://docs.python.org/3/reference/compound_stmts.html#except

de la fonction mais celle-ci est appelée à l'intérieur d'un code ńăprotégéă" contre les erreurs. Ceci explique les lignes affichées par le programme. Il est aussi possible d'ajouter une clause qui sert de préfixe à une liste d'instructions qui ne sera exécutée que si aucune exception n'est déclenchée.

<<<

```
def inverse(x):
    y = 1.0 / x
    return y


try:
    print(inverse(2))  # pas d'erreur
    print(inverse(1))  # pas d'erreur non plus
except:
    print("le programme a déclenché une erreur")
else:
    print("tout s'est bien passé")
```

>>>

```
    0.5
    1.0
    tout s'est bien passé
```

Ce dernier programme attrape l'erreur et affiche un message. Ce programme ne s'arrête jamais, il ne *plante* jamais. Pour résumer, la syntaxe suivante permet d'attraper toutes les erreurs qui se produisent pendant l'exécution d'une partie du programme. Cette syntaxe permet en quelque sorte de protéger cette partie du programme contre les erreurs.

```
try:
    # ... instructions à protéger
except:
    # ... que faire en cas d'erreur
else:
    # ... que faire lorsque aucune erreur n'est apparue
```

Toute erreur déclenchée alors que le programme exécute les instructions qui suivent le mot-clé `try` déclenche immédiatement l'exécution des lignes qui suivent le mot-clé `except`. Dans le cas contraire, le programme se poursuit avec l'exécution des lignes qui suivent le mot-clé `else`. Cette dernière partie est facultative, la clause `else` peut ou non être présente. Le bout de code prévoit ce qu'il faut faire dans n'importe quel cas.

Lorsqu'une section de code est protégée contre les exceptions, son exécution s'arrête à la première erreur d'exécution. Le reste du code n'est pas exécuté. Par exemple, dès la première erreur qui correspond au calcul d'une puissance non entière d'un nombre négatif, l'exécution du programme suivant est dirigée vers l'instruction qui suit le mot-clé `except`.

```
def inverse (x):
    y = 1.0 / x
    return y

try:
    print((-2.1) ** 3.1)   # première erreur
    print(inverse(2))
    print(inverse(0))      # cette ligne produirait une erreur
                           # mais le programme n'arrive jamais jusqu'ici
except:
    print("le programme a déclenché une erreur")
```

Obtenir le type d'erreur, attraper un type d'exception

Parfois, plusieurs types d'erreurs peuvent être déclenchés à l'intérieur d'une portion de code protégée. Pour avoir une information sur ce type, il est possible de récupérer une variable de type Exception [179].

<<<

```
def inverse(x):
    y = 1.0 / x
    return y


try:
    print(inverse(2))
    print(inverse(0))
except Exception as exc:
    print("exception de type ", exc.__class__)
    # affiche exception de type  exceptions.ZeroDivisionError
    print("message", exc)
    # affiche le message associé à l'exception
```

>>>

```
    0.5
    exception de type  <class 'ZeroDivisionError'>
    message float division by zero
```

Le programme précédent récupère une exception sous la forme d'une variable appelée . Cette variable est en fait une instance d'une classe d'erreur, `__class__` correspond au nom de cette classe. A l'aide de la fonction `isinstance`, il est possible d'exécuter des traitements différents selon le type d'erreur.

<<<

```
def inverse(x):
    y = 1.0 / x
    return y


try:
    print((-2.1) ** 3.1)   # première erreur
    print(inverse(2))
    print(inverse(0))      # seconde erreur
except Exception as exc:
    if isinstance(exc, ZeroDivisionError):
        print("division par zéro")
    else:
        print("erreur insoupçonnée :", exc.__class__)
        print("message", exc)
```

>>>

```
    (-9.48606594010979-3.0822096637057887j)
    0.5
    division par zéro
```

L'exemple précédent affiche le message qui suit parce que la première erreur intervient lors du calcul de `(-2.1) ** 3.1`. Une autre syntaxe plus simple permet d'attraper un type d'exception donné en accolant au mot-clé `except` le type de l'exception qu'on désire attraper. L'exemple précédent est équivalent au suivant mais syntaxiquement différent.

179. https://docs.python.org/3/library/exceptions.html

<<<

```
def inverse(x):
    y = 1.0 / x
    return y


try:
    print((-2.1) ** 3.1)
    print(inverse(2))
    print(inverse(0))
except ZeroDivisionError:
    print("division par zéro")
except Exception as exc:
    print("erreur insoupçonnée :", exc.__class__)
    print("message ", exc)
```

>>>

```
    (-9.48606594010979-3.0822096637057887j)
    0.5
    division par zéro
```

Cette syntaxe obéit au schéma qui suit.

Syntaxe S1 : Attraper une exception

```
try:
    # ... instructions à protéger
except type_exception_1:
    # ... que faire en cas d'erreur de type type_exception_1
except (type_exception_i, type_exception_j):
    # ... que faire en cas d'erreur de type type_exception_i ou type_exception_j
except type_exception_n:
    # ... que faire en cas d'erreur de type type_exception_n
except:
    # ... que faire en cas d'erreur d'un type différent de tous
    #     les précédents types
else:
    # ... que faire lorsque une erreur aucune erreur n'est apparue
```

Toute erreur déclenchée alors que le programme exécute les instructions qui suivent le mot-clé `try` déclenche immédiatement l'exécution des lignes qui suivent un mot-clé `except`. Le programme compare le type d'exception aux types `type_exception_1` à `type_exception_n`. S'il existe une correspondance alors ce sont les instructions de la clause `except` associée qui seront exécutées et uniquement ces instructions. La dernière clause `except` est facultative, elle est utile lorsque aucun type de ceux prévus ne correspond à l'exception générée. La clause `else` est aussi facultative. Si la dernière clause `except` n'est pas spécifiée et que l'exception déclenchée ne correspond à aucune de celle listée plus haut, le programme s'arrête sur cette erreur à moins que celle-ci ne soit attrapée plus tard.

Le langage Python propose une liste d¿exceptions standards [180]. Lorsqu'une erreur ne correspond pas à l'une de ces exceptions, il est possible de créer une exception propre à un certain type d'erreur. Lorsqu'une fonction ou une méthode déclenche une exception non standard, généralement, le commentaire qui lui est associé l'indique. Quelques types d'exception courantes documentée dans la section Concrete exceptions [181]. Certaines surviennent car le programme est mal écrit et l'interpréteur ne peut le comprendre :

180. https://docs.python.org/3/library/exceptions.html#base-classes
181. https://docs.python.org/3/library/exceptions.html#concrete-exceptions

— **IndentationError** : L'interpréteur ne peut interpréter une partie du programme à cause d'un problème d'indentation. Il n'est pas possible d'exécuter un programme mal indenté mais cette erreur peut se produire lors de l'utilisation de la fonction compile [182].
— **SyntaxError** : Le programme a un problème de syntaxe comme une parenthèse en trop ou ou en moins.

Les deux suivantes surviennent lorsqu'on se trompe dans l'orthographe d'une variable, une fonction, un module :

— **AttributeError** : Une référence à un attribut inexistant ou une affectation a échoué.
— **ImportError** : Cette erreur survient lorsqu'on cherche à importer un module qui n'existe pas, son nom est mal orthographié ou il n'est pas installé.
— **NameError** : On utilise une variable, une fonction, une classe qui n'existe pas.

Les erreurs très fréquentes, erreur d'indices, de types :

— **IndexError** : On utilise un index erroné pour accéder à un élément d'une liste, d'un dictionnaire ou de tout autre tableau.
— **KeyError** : Une clé est utilisée pour accéder à un élément d'un dictionnaire dont elle ne fait pas partie.
— **TypeError** : Erreur de type, une fonction est appliquée sur un objet qu'elle n'est pas censée manipuler.
— **ValueError** : Cette exception survient lorsqu'une valeur est inappropriée pour une certaine opération, par exemple, l'obtention du logarithme d'un nombre négatif.

Les erreurs qui surviennent lorsqu'on travaille avec des fichiers :

— **OSError** : Une opération concernant les entrées/sorties (Input/Output) a échoué. Cette erreur survient par exemple lorsqu'on cherche à lire un fichier qui n'existe pas.
— **UnicodeError** : Erreur de conversion d'un encodage [183] de texte à un autre. C'est une erreur qui survient régulièrement quand on travaille avec des langues qui ont des accents (non anglophones).

Lancer une exception

Lorsqu'une fonction détecte une erreur, il lui est possible de déclencher une exception par l'intermédiaire du mot-clé `raise`. La fonction `inverse` compare `x` à `0` et déclenche l'exception `ValueError` si `x` est nul. Cette exception est attrapée plus bas.

<<<

```
def inverse(x):
    if x == 0:
        raise ValueError
    y = 1.0 / x
    return y


try:
    print(inverse(0))  # erreur
except ValueError:
    print("erreur de type ValueError")
```

>>>

```
    erreur de type ValueError
```

Il est parfois utile d'associer un message à une exception afin que l'utilisateur ne soit pas perdu. Le programme qui suit est identique au précédent à ceci près qu'il associe à l'exception `ValueError` qui précise l'erreur et mentionne la fonction où elle s'est produite. Le message est ensuite intercepté plus bas.

<<<

182. https://docs.python.org/3/library/functions.html?highlight=compile#compile
183. https://fr.wikipedia.org/wiki/Codage_des_caract%C3%A8res

```
def inverse(x):
    if x == 0:
        raise ValueError("valeur nulle interdite, fonction inverse")
    y = 1.0 / x
    return y


try:
    print(inverse(0))  # erreur
except ValueError as exc:
    print("erreur, message :", exc)
```

```
>>>
    erreur, message : valeur nulle interdite, fonction inverse
```

Le déclenchement d'une exception suit la syntaxe suivante.

Syntaxe S2 : Lever une exception

```
raise exception_type(message)
```

Cette instruction lance l'exception `exception_type` associée au message `message`. Le message est facultatif, lorsqu'il n'y en a pas, la syntaxe se résume à `raise exception_type`.

Et pour attraper cette exception et le message qui lui est associé, il faut utiliser la syntaxe décrite au paragraphe précédent.

Héritage et exception

L'instruction `help(ZeroDivisionError)` retourne l'aide associée à l'exception `ZeroDivisionError`. Celle-ci indique que l'exception `ZeroDivisionError` est en fait un cas particulier de l'exception `ArithmeticError`, elle-même un cas particulier de `StandardError`.

```
class ZeroDivisionError(ArithmeticError)
    |  Second argument to a division or modulo operation was zero.
    |
    |  Method resolution order:
    |      ZeroDivisionError
    |      ArithmeticError
    |      StandardError
    |      Exception
```

Toutes les exceptions sont des cas particuliers de l'exception de type `Exception`. C'est pourquoi l'instruction `except Exception:` attrape toutes les exceptions. L'instruction `except ArithmeticError:` attrape toutes les erreurs de type `ArithmeticError`, ce qui inclut les erreurs de type `ZeroDivisionError`. Autrement dit, toute exception de type `ZeroDivisionError` est attrapée par les instructions suivantes :

```
except ZeroDivisionError:
except ArithmeticError:
except StandardError:
except Exception:
```

Plus précisément, chaque exception est une classe qui dérive directement ou indirectement de la classe `Exception`. L'instruction `except ArithmeticError :` par exemple attrape toutes les exceptions de type `ArithmeticError` et toutes celles qui en dérivent comme la classe `ZeroDivisionError`.

Instructions `try`, `except` imbriquées

Comme pour les boucles, il est possible d'imbriquer les portions protégées de code les unes dans les autres. Dans l'exemple qui suit, la première erreur est l'appel à une fonction non définie, ce qui déclenche l'exception `NameError`.

<<<

```
def inverse(x):
    y = 1.0 / x
    return y


try:
    try:
        print(inverses(0))   # fonction inexistante --> exception NameError
        print(inverse(0))    # division par zéro --> ZeroDivisionError
    except NameError:
        print("appel à une fonction non définie")
except ZeroDivisionError as exc:
    print("erreur", exc)
```

>>>

```
    appel à une fonction non définie
```

En revanche, dans le second exemple, les deux lignes `print(inverse(0))` et `print(inverses(0))` ont été permutées. La première exception déclenchée est la division par zéro. La première clause `except` n'interceptera pas cette erreur puisqu'elle n'est pas du type recherché.

<<<

```
def inverse(x):
    y = 1.0 / x
    return y


try:
    try:
        print(inverse(0))    # division par zéro --> ZeroDivisionError
        print(inverses(0))   # fonction inexistante --> exception NameError
    except NameError:
        print("appel à une fonction non définie")
except ZeroDivisionError as exc:
    print("erreur", exc)
```

>>>

```
    erreur float division by zero
```

Une autre imbrication possible est l'appel à une fonction qui inclut déjà une partie de code protégée. L'exemple suivant appelle la fonction `inverse` qui intercepte les exceptions de type `ZeroDivisionError` pour retourner une grande valeur lorsque `x=0`. La seconde exception générée survient lors de l'appel à la fonction `inverses` qui déclenche l'exception `NameError`, elle aussi interceptée.

<<<

```
def inverse(x):
    try:
        y = 1.0 / x
    except ZeroDivisionError as exc:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        print("erreur ", exc)
        if x > 0:
            return 1000000000
        else:
            return -1000000000
    return y


try:
    print(inverse(0))    # division par zéro    --> la fonction inverse sait gérer
    print(inverses(0))   # fonction inexistante --> exception NameError
except NameError:
    print("appel à une fonction non définie")
```

>>>

```
    erreur  float division by zero
    -1000000000
    appel à une fonction non définie
```

4.1.2 Définir ses propres exceptions

Dériver une classe d'exception

Pour définir sa propre exception, il faut créer une classe qui dérive d'une classe d'exception existante par exemple, la classe `Exception`. L'exemple suivant crée une exception `AucunChiffre` qui est lancée par la fonction `conversion` lorsque la chaîne de caractères qu'elle doit convertir ne contient pas que des chiffres.

<<<

```
class AucunChiffre(Exception):
    """
    chaîne de caractères contenant aussi autre chose que des chiffres
    """
    pass


def conversion(s):
    """
    conversion d'une chaîne de caractères en entier
    """
    if not s.isdigit():
        raise AucunChiffre(s)
    return int(s)


try:
    s = "123a"
    print(s, " = ", conversion(s))
except AucunChiffre as exc:
    # on affiche ici le commentaire associé à la classe d'exception
    # et le message associé
    print(AucunChiffre.__doc__, " : ", exc)
```

>>>

```
            chaîne de caractères contenant aussi autre chose que des chiffres
              :  123a
```

En redéfinissant l'opérateur `__str__` d'une exception, il est possible d'afficher des messages plus explicites avec la seule instruction `print`.

```
class AucunChiffre(Exception):
    """
    chaîne de caractères contenant aussi autre chose que des chiffres
    """
    def __str__ (self):
        return "{0} {1}".format(self.__doc__, Exception.__str__(self))
```

Personnalisation d'une classe d'exception

Il est parfois utile qu'une exception contienne davantage d'informations qu'un simple message. L'exemple suivant reprend l'exemple du paragraphe précédent. L'exception `AucunChiffre` inclut cette fois-ci un paramètre supplémentaire contenant le nom de la fonction où l'erreur a été déclenchée.

La classe `AucunChiffre` possède dorénavant un constructeur qui doit recevoir deux paramètres : une valeur et un nom de fonction. L'exception est levée à l'aide de l'instruction `raise AucunChiffre(s, "conversion")` qui regroupe dans un T-uple les paramètres à envoyer à l'exception.

L'opérateur `__str__` a été modifié de façon à ajouter ces deux informations dans le message associé à l'exception. Ainsi, l'instruction `print(exc)` présente à l'avant dernière ligne de cet exemple affiche un message plus complet.

<<<

```
class AucunChiffre(Exception):
    """
    chaîne de caractères contenant aussi autre chose que des chiffres
    """

    def __init__(self, s, f=""):
        Exception.__init__(self, s)
        self.s = s
        self.f = f

    def __str__(self):
        return "exception AucunChiffre, depuis la fonction {0} avec le paramètre {1}".
↪format(self.f, self.s)


def conversion(s):
    """
    conversion d'une chaîne de caractères en entier
    """
    if not s.isdigit():
        raise AucunChiffre(s, "conversion")
    return int(s)


try:
    s = "123a"
    i = conversion(s)
    print(s, " = ", i)
```

(suite sur la page suivante)

(suite de la page précédente)

```
except AucunChiffre as exc:
    print(exc)
    print("fonction : ", exc.f)
```

```
>>>
```

```
    exception AucunChiffre, depuis la fonction conversion avec le paramètre 123a
    fonction :  conversion
```

Etant donné que le programme déclenche une exception dans la section de code protégée, les deux derniers affichages sont les seuls exécutés correctement. Ils produisent les deux lignes qui suivent. %

4.1.3 Exemples d'utilisation des exceptions

Les itérateurs

Les itérateurs sont des outils qui permettent de parcourir des objets qui sont des ensembles, comme une liste, un dictionnaire. Ils fonctionnent toujours de la même manière. L'exemple déjà présenté au chapitre *Itérateurs* (page 113) et repris en partie ici définit une classe contenant trois coordonnées, ainsi qu'un itérateur permettant de parcourir ces trois coordonnées. Arrivée à la troisième itération, l'exception StopIteration [184] est déclenchée. Cette exception indique à une boucle `for` de s'arrêter.

```
class point_espace:

    # ...

    class class_iter:
        def __init__(self, ins):
            self._n   = 0
            self._ins = ins
        def __iter__(self) :
            return self
        def next(self):
            if self._n <= 2:
                v = self._ins[self._n]
                self._n += 1
                return v
            else:
                raise StopIteration

    def __iter__(self):
        return point_espace.class_iter(self)
```

Cet exemple montre seulement que les exceptions n'interviennent pas seulement lors d'erreurs mais font parfois partie intégrante d'un algorithme.

Exception ou valeur aberrante

Sans exception, une solution pour indiquer un cas de mauvaise utilisation d'une fonction est de retourner une valeur aberrante. Retourner `-1` pour une fonction dont le résultat est nécessairement positif est une valeur aberrante. Cette convention permet de signifier à celui qui appelle la fonction que son appel n'a pu être traité correctement. Dans l'exemple qui suit, la fonction `racine_carree` retourne un couple de résultats, `True` ou `False` pour savoir si le calcul est possible, suivi du résultat qui n'a un sens que si `True` est retournée en première valeur.

184. https://docs.python.org/3/library/exceptions.html#StopIteration

<<<

```
def racine_carree(x):
    if x < 0:
        return False, 0
    else:
        return True, x ** 0.5


print(racine_carree(-1))   # (False, 0)
print(racine_carree(1))    # (True, 1.0)
```

>>>

```
    (False, 0)
    (True, 1.0)
```

Plutôt que de compliquer le programme avec deux résultats ou une valeur aberrante, on préfère souvent déclencher une exception, ici, `ValueError`. La plupart du temps, cette exception n'est pas déclenchée. Il est donc superflu de retourner un couple plutôt qu'une seule valeur.

```
def racine_carree(x) :
    if x < 0:
        raise ValueError("valeur négative")
    return x ** 0.5

print(racine_carree(-1))   # déclenche une exception
print(racine_carree(1))
```

Le piège des exceptions

Ce paragraphe évoque certains problèmes lorsqu'une exception est levée. L'exemple utilise les fichiers décrits au chapitre *Fichiers* (page 177). Lorsqu'une exception est levée à l'intérieur d'une fonction, l'exécution de celle-ci s'interrompt. Si l'exception est attrapée, le programme continue sans problème ; les objets momentanément créés seront détruits par le garbage collector [185]. Il faut pourtant faire attention dans le cas par exemple où l'exception est levée alors qu'un fichier est ouvert : il ne sera pas fermé.

```
for i in range(0, 5):
    try :
        x, y = i-1, i-2
        print("{}/{}".format(x, y))
        f = open("essai.txt", "a")
        f.write("{}/{}=".format(x, y))
        f.write(str((float (x)/y)) + "\n" )      # exception si y == 0
        f.close()
    except Exception as e:
        print("erreur avec i = ", i, ",", e, f.closed)
```

Les écritures dans le fichier se font en mode ajout `"a"`, le fichier `"essai.txt"` contiendra tout ce qui aura été écrit.

185. https://docs.python.org/3/library/gc.html

affichage	fichier
-1/-2 0/-1 1/0 erreur avec i = 2 , float division by zero False 2/1 3/2	-1/-2=0.5 0/-1=-0.0 1/0=2/1=2.0 3/2=1.5

La troisième ligne du fichier est tronquée puisque l'erreur est intervenue juste avant l'affichage. On voit aussi que `f.closed` est faux. Cela signifie que le fichier n'est pas fermé. Pour se prémunir contre les exceptions lorsqu'on écrit un fichier, il faut utiliser le mot clé with [186] :

```
for i in range(0, 5):
    try :
        x, y = i-1, i-2
        print("{}/{}".format(x, y))
        with open("essai.txt", "a") as f:
            f.write("{}/{}=".format(x, y))
            f.write(str((float (x)/y)) + "\n" )      # exception si y == 0
    except Exception as e:
        print("erreur avec i = ", i, ",", e, f.closed)
```

Pour en savoir un peu plus : Les context managers et le mot clé with en Python [187].

4.2 Usage

4.2.1 Pile d'appel

La pile d'appel [188] (ou *pile d'exécution* ou *call stack*) mémorise les appels de fonctions. La premièrel ligne est le point d'entrée [189] du programme. La suivante est la seconde fonction appelée. Si celle-ci en appelle une autre, une autre ligne est ajoutée et celle-ci demeure jusqu'à ce qu'elle est terminée son exécution. A chaque instant, la dernière ligne est la fonction en train de s'exécuter, les lignes précédentes définissent le chemin que l'ordinateur a suivi pour arriver jusque là.

<<<

```
import traceback
import sys


def foncA():
    print("foncA begin")
    foncB()
    print("foncA end")


def foncB():
```

(suite sur la page suivante)

186. https://www.python.org/dev/peps/pep-0343/
187. http://sametmax.com/les-context-managers-et-le-mot-cle-with-en-python/
188. https://fr.wikipedia.org/wiki/Pile_d%27ex%C3%A9cution
189. https://fr.wikipedia.org/wiki/Point_d%27entr%C3%A9e

(suite de la page précédente)

```
    print("foncB begin")
    foncC()
    print("foncB end")


def foncC():
    print("foncC begin")
    try:
        raise Exception("erreur volontaire")
    except Exception:
        print("Erreur")
        print("\n".join(traceback.format_stack()))
    print("foncC end")


foncA()
```

>>>

```
    foncA begin
    foncB begin
    foncC begin
    Erreur
      File "/var/lib/jenkins/workspace/teachpyx/teachpyx_UT_37_std/_venv/lib/python3.
↪7/site-packages/pyquickhelper/helpgen/process_sphinx_cmd.py", line 23, in <module>
        main()

      File "/var/lib/jenkins/workspace/teachpyx/teachpyx_UT_37_std/_venv/lib/python3.
↪7/site-packages/pyquickhelper/helpgen/process_sphinx_cmd.py", line 19, in main
        run_sphinx_build(sys.argv)

      File "/var/lib/jenkins/workspace/teachpyx/teachpyx_UT_37_std/_venv/lib/python3.
↪7/site-packages/pyquickhelper/helpgen/process_sphinx_cmd.py", line 15, in run_
↪sphinx_build
        build_main(argv=argv[1:])

      File "/usr/local/lib/python3.7/site-packages/sphinx/cmd/build.py", line 319, in␣
↪main
        return build_main(argv)

      File "/usr/local/lib/python3.7/site-packages/sphinx/cmd/build.py", line 304, in␣
↪build_main
        app.build(args.force_all, filenames)

      File "/usr/local/lib/python3.7/site-packages/sphinx/application.py", line 341,␣
↪in build
        self.builder.build_update()

      File "/usr/local/lib/python3.7/site-packages/sphinx/builders/__init__.py", line␣
↪342, in build_update
        self.build(['__all__'], to_build)

      File "/usr/local/lib/python3.7/site-packages/sphinx/builders/__init__.py", line␣
↪360, in build
        updated_docnames = set(self.read())

      File "/usr/local/lib/python3.7/site-packages/sphinx/builders/__init__.py", line␣
↪468, in read
```

(suite sur la page suivante)

(suite de la page précédente)

```
        self._read_serial(docnames)

      File "/usr/local/lib/python3.7/site-packages/sphinx/builders/__init__.py", line␣
↪490, in _read_serial
        self.read_doc(docname)

      File "/usr/local/lib/python3.7/site-packages/sphinx/builders/__init__.py", line␣
↪534, in read_doc
        doctree = read_doc(self.app, self.env, self.env.doc2path(docname))

      File "/usr/local/lib/python3.7/site-packages/sphinx/io.py", line 318, in read_
↪doc
        pub.publish()

      File "/usr/local/lib/python3.7/site-packages/docutils/core.py", line 217, in␣
↪publish
        self.settings)

      File "/usr/local/lib/python3.7/site-packages/docutils/readers/__init__.py",␣
↪line 72, in read
        self.parse()

      File "/usr/local/lib/python3.7/site-packages/docutils/readers/__init__.py",␣
↪line 78, in parse
        self.parser.parse(self.input, document)

      File "/usr/local/lib/python3.7/site-packages/sphinx/parsers.py", line 88, in␣
↪parse
        self.statemachine.run(inputstring, document, inliner=self.inliner)

      File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 171, in run
        input_source=document['source'])

      File "/usr/local/lib/python3.7/site-packages/docutils/statemachine.py", line␣
↪239, in run
        context, state, transitions)

      File "/usr/local/lib/python3.7/site-packages/docutils/statemachine.py", line␣
↪460, in check_line
        return method(match, context, next_state)

      File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 2988, in text
        self.section(title.lstrip(), source, style, lineno + 1, messages)

      File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 327, in section
        self.new_subsection(title, lineno, messages)

      File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 395, in new_subsection
        node=section_node, match_titles=True)

      File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 282, in nested_parse
        node=node, match_titles=match_titles)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 196, in run
      results = StateMachineWS.run(self, input_lines, input_offset)

    File "/usr/local/lib/python3.7/site-packages/docutils/statemachine.py", line␣
↪239, in run
      context, state, transitions)

    File "/usr/local/lib/python3.7/site-packages/docutils/statemachine.py", line␣
↪460, in check_line
      return method(match, context, next_state)

    File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 2753, in underline
      self.section(title, source, style, lineno - 1, messages)

    File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 327, in section
      self.new_subsection(title, lineno, messages)

    File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 395, in new_subsection
      node=section_node, match_titles=True)

    File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 282, in nested_parse
      node=node, match_titles=match_titles)

    File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 196, in run
      results = StateMachineWS.run(self, input_lines, input_offset)

    File "/usr/local/lib/python3.7/site-packages/docutils/statemachine.py", line␣
↪239, in run
      context, state, transitions)

    File "/usr/local/lib/python3.7/site-packages/docutils/statemachine.py", line␣
↪460, in check_line
      return method(match, context, next_state)

    File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 2326, in explicit_markup
      nodelist, blank_finish = self.explicit_construct(match)

    File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 2338, in explicit_construct
      return method(self, expmatch)

    File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 2081, in directive
      directive_class, match, type_name, option_presets)

    File "/usr/local/lib/python3.7/site-packages/docutils/parsers/rst/states.py",␣
↪line 2130, in run_directive
      result = directive_instance.run()
```

(suite sur la page suivante)

(suite de la page précédente)

```
      File "/var/lib/jenkins/workspace/teachpyx/teachpyx_UT_37_std/_venv/lib/python3.
↪7/site-packages/pyquickhelper/sphinxext/sphinx_runpython_extension.py", line 480,␣
↪in run
        chdir=cs_source_dir if p['current'] else None)

      File "/var/lib/jenkins/workspace/teachpyx/teachpyx_UT_37_std/_venv/lib/python3.
↪7/site-packages/pyquickhelper/sphinxext/sphinx_runpython_extension.py", line 196,␣
↪in run_python_script
        exec(obj, globs, loc)

      File "", line 26, in <module>

      File "", line 25, in run_python_script_139791938759416

      File "", line 8, in foncA

      File "", line 13, in foncB

      File "", line 22, in foncC

    foncC end
    foncB end
    foncA end
```

Récupération de la pile d'appel

Le module traceback [190] permet de récupérer la pile d'appels lorsqu'une exception survient.

```
def raise_exception():
    raise Exception("an error was raised")

try:
    insidefe()
except:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print("".join(traceback.format_tb(exc_traceback)))
```

Le programme affiche :

```
File "test_faq_exception.py", line 57, in raise_exception
  raise Exception("an error was raised")
```

Il est possible de récupérer la liste des appels de fonctions avec la fonction extract_tb [191]. Cette information est précieuse pour écrire un test qui vérifie qu'une erreur s'est bien produite à un endroit particulier, de détecter les cas particuliers comme les boucles infinies ou d'améliorer un message d'erreur en cas de besoin (lire How do I write Flask's excellent debug log message to a file in production ? [192]).

Message d'erreur plus explicite

Lorsqu'une erreur se produit dans une librairie de Python, le message ne mentionne aucune information à propos du code qui l'a provoquée.

190. https://docs.python.org/3/library/traceback.html
191. https://docs.python.org/3/library/traceback.html#traceback.extract_tb
192. http://stackoverflow.com/questions/14037975/how-do-i-write-flasks-excellent-debug-log-message-to-a-file-in-production

```
import math
ensemble = [1, 0, 2]
s = 0
for e in ensemble:
    s += math.log(e)
```

```
Traceback (most recent call last):
  File "i.py", line 5, in <module>
    s += math.log(e)
ValueError: math domain error
```

Typiquement dans ce cas précis, on ne sait pas quel est l'indice de l'élément qui a provoqué l'erreur. On utilise alors un mécanisme qui permet d'ajouter une erreur sans perdre les informations l'exception original

```
import math
ensemble = [1, 0, 2]
s = 0
for i, e in enumerate(ensemble):
    try:
        s += math.log(e)
    except Exception as exc:
        raise Exception("Issue with element {0}".format(i)) from exc
```

La dernière partie de la dernière ligne est importante : `from exc`. Le langage garde ainsi la trace de la première exception.

```
Traceback (most recent call last):
  File "i.py", line 6, in <module>
    s += math.log(e)
ValueError: math domain error

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "i.py", line 8, in <module>
    raise Exception("Issue with element {0}".format(i)) from exc
Exception: Issue with element 1
```

4.2.2 Conventions

Erreur ou code d'erreur

A faire : terminer la section Erreur ou code d'erreur

parler aussi de coûts d'une exception, libération des ressources

4.3 Warning

Les warning [193] ne sont pas des erreurs mais des soupçons d'erreurs. Le programme peut continuer mais il est possible qu'il s'arrête un peu plus tard et la cause pourrait être un *warning* déclenché un peu plus tôt.

193. https://docs.python.org/3/library/warnings.html

Un des *warning* les plus utilisé est le DeprecationWarning [194] qui indique que le code utilisé va disparaître sous cette forme lors des prochaines release. scikit-learn [195] suit la règle suivante à chaque fois qu'une API change, un *warning* subsiste pendant deux release pour une fonction appelée à disparaître. Le *warning* apparaît à chaque fois qu'elle est exécutée, puis la fonction est finalement supprimée. Tout code s'appuyant encore sur cette fonction provoquera une erreur.

4.3.1 Générer un warning

Le module warnings [196] permet de lancer un *warning* comme ceci :

<<<

```
import warnings
warnings.warn("Warning lancé !")
```

>>>

```
    /var/lib/jenkins/workspace/teachpyx/teachpyx_UT_37_std/_venv/lib/python3.7/site-
↪packages/pyquickhelper/sphinxext/sphinx_runpython_extension.py:4: UserWarning:␣
↪Warning lancé !
      @brief Defines runpython directives.
```

Il est préférable d'ailleurs de spécifier un type précis de *warning* qui indique à l'utilisateur à quel type d'erreur il s'expose sans avoir à lire le message, voire plus tard de les trier.

<<<

```
import warnings
warnings.warn("Warning d'un certain lancé !", UserWarning)
```

>>>

```
    /var/lib/jenkins/workspace/teachpyx/teachpyx_UT_37_std/_venv/lib/python3.7/site-
↪packages/pyquickhelper/sphinxext/sphinx_runpython_extension.py:4: UserWarning:␣
↪Warning d'un certain lancé !
      @brief Defines runpython directives.
```

4.3.2 Intercepter un warning

Les *warning* sont parfois très agaçants car il s'insère dans les sorties du programme qui deviennent moins lisibles. Il serait préférable de les corriger mais ils surviennent parfois dans un module qui n'a pas pris en compte l'évolution d'une de ses dépendances. Il est difficile de corriger cette erreur immédiatement à moins de modifier le code du module installé, ce qui n'est souvent pas souhaitable voire impossible si ce module est écrit en C++ [197]. Le plus simple reste de les intercepter.

<<<

```
import warnings


def fxn():
    warnings.warn("deprecated", DeprecationWarning)
```

(suite sur la page suivante)

194. https://docs.python.org/3/library/warnings.html#warning-categories
195. http://scikit-learn.org/
196. https://docs.python.org/3/library/warnings.html
197. https://fr.wikipedia.org/wiki/C%2B%2B

(suite de la page précédente)

```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

>>>

```
```

On peut également intercepter un type particulier de *warning*, c'est le rôle de la classe catch_warnings [198] :

<<<

```
import warnings


def fxn():
    warnings.warn("deprecated", DeprecationWarning)


print("Boucle 1")
with warnings.catch_warnings():
    warnings.simplefilter("ignore", DeprecationWarning)
    fxn()

print("Boucle 2")
with warnings.catch_warnings():
    warnings.simplefilter("ignore", UserWarning)
    fxn()

print("Boucle 3")
with warnings.catch_warnings():
    warnings.simplefilter("ignore", DeprecationWarning)
    warnings.simplefilter("ignore", UserWarning)
    fxn()
```

>>>

```
    Boucle 1
    Boucle 2
    /var/lib/jenkins/workspace/teachpyx/teachpyx_UT_37_std/_venv/lib/python3.7/site-
↪packages/pyquickhelper/sphinxext/sphinx_runpython_extension.py:6:␣
↪DeprecationWarning: deprecated
      """
    Boucle 3
```

Il est parfois utile de mémoriser les *warning* généré par un programme, c'est nécessaire principalement lorsqu'on écrit des tests unitaires.

<<<

```
import warnings


def fxn():
```

(suite sur la page suivante)

198. https://docs.python.org/3/library/warnings.html#warnings.catch_warnings

(suite de la page précédente)

```
    warnings.warn("deprecated", DeprecationWarning)


with warnings.catch_warnings(record=True) as ws:
    warnings.simplefilter("always")
    fxn()

    print("nombre de warnings :", len(ws))

    for i, w in enumerate(ws):
        print("warning {0} : {1}".format(i, w))
```

>>>

```
    nombre de warnings : 1
    warning 0 : {message : DeprecationWarning('deprecated'), category :
↪'DeprecationWarning', filename : '/var/lib/jenkins/workspace/teachpyx/teachpyx_UT_
↪37_std/_venv/lib/python3.7/site-packages/pyquickhelper/sphinxext/sphinx_runpython_
↪extension.py', lineno : 6, line : None}
```

4.3.3 Warning personnalisé

Comme pour les exceptions, il est possible de définir ses propres *warning* en héritant d'un *warning* en particulier.

<<<

```
import warnings


class MonWarning(UserWarning):
    pass


warnings.warn("mon warning", MonWarning)
```

>>>

```
    /var/lib/jenkins/workspace/teachpyx/teachpyx_UT_37_std/_venv/lib/python3.7/site-
↪packages/pyquickhelper/sphinxext/sphinx_runpython_extension.py:8: MonWarning: mon␣
↪warning
      import os
```

CHAPITRE 5

Entrées Sorties

5.1 Fichiers

Lorsqu'un programme termine son exécution, toutes les informations stockées dans des variables sont perdues. Un moyen de les conserver est de les enregistrer dans un fichier sur disque dur. A l'intérieur de celui-ci, ces informations peuvent apparaître sous un format texte qui est lisible par n'importe quel éditeur de texte, dans un format compressé, ou sous un autre format connu par le concepteur du programme. On appelle ce dernier type un format binaire [199], il est adéquat lorsque les données à conserver sont très nombreuses ou lorsqu'on désire que celles-ci ne puissent pas être lues par une autre application que le programme lui-même. En d'autres termes, le format binaire est illisible excepté pour celui qui l'a conçu.

Ce chapitre abordera pour commencer les formats texte, binaire et compressé (*zip*) directement manipulable depuis *python*. Les manipulations de fichiers suivront pour terminer sur les expressions régulières qui sont très utilisées pour

199. https://fr.wikipedia.org/wiki/Fichier_binaire

effectuer des recherches textuelles. A l'issue de ce chapitre, on peut envisager la recherche à l'intérieur de tous les documents textes présents sur l'ordinateur, de dates particulières, de tous les numéros de téléphones commençant par 06... En utilisant des modules tels que reportlab [200] ou encore openpyxl [201], il serait possible d'étendre cette fonctionnalité aux fichiers de type pdf [202] et aux fichiers Excel [203].

5.1.1 Format texte

Les fichiers texte [204] sont les plus simples : ce sont des suites de caractères. Le format HTML [205] et XML [206] font partie de cette catégorie. Ils servent autant à conserver des informations qu'à en échanger comme par exemple transmettre une matrice à Excel [207].

Ce format, même s'il est simple, implique une certaine organisation dans la façon de conserver les données afin de pouvoir les récupérer. Le cas le plus fréquent est l'enregistrement d'une matrice : on choisira d'écrire les nombres les uns à la suite des autres en choisissant un séparateur de colonnes et un séparateur de lignes. Ce point sera abordé à la fin de cette section. Les fichiers texte que les programmes informatiques manipulent sont souvent structurées.

Ecriture

La première étape est l'écriture. Les informations sont toujours écrites sous forme de chaînes de caractères et toujours ajoutées à la fin du fichier qui s'allonge jusqu'à ce que toutes les informations y soient écrites. L'écriture s'effectue toujours selon le même schéma.

1. création ou ouverture du fichier,
2. écriture,
3. fermeture.

Lors de l'ouverture, le fichier dans lequel seront écrites les informations est créé s'il n'existe pas ou nettoyé s'il existe déjà. La fermeture permet à d'autres programmes de lire ce que vous avez placé dans ce fichier. Sans cette dernière étape, il sera impossible d'y accéder à nouveau pour le lire ou y écrire à nouveau. A l'intérieur d'un programme informatique, écrire dans un fichier suit toujours le même schéma :

```
f = open ("nom-fichier", "w")    # ouverture

f.write ( s )    # écriture de la chaîne de caractères  s
f.write ( s2 )   # écriture de la chaîne de caractères  s2
...

f.close ()  # fermeture
```

Les étapes d'ouverture et de fermeture sont toujours présentes en ce qui concerne les fichiers. Il s'agit d'indiquer au système d'exploitation que le programme souhaite accéder à un fichier et interdire à tout autre programme l'accès à ce fichier. Un autre programme qui souhaiterait créer un fichier du même nom ne le pourrait pas tant que l'étape de fermeture n'est pas exécutée. En revanche, il pourrait tout à fait le lire car la lecture ne perturbe pas l'écriture.

Il faut donc toujours fermer le fichier à la fin pour indiquer que le fichier est accessible pour un autre usage. Un fichier est une ressource [208]. Et comme ce schéma se répète toujours, le langage *Python* a prévu une syntaxe avec le mot-clé with [209].

200. https://pypi.python.org/pypi/reportlab
201. https://pypi.python.org/pypi/openpyxl
202. https://fr.wikipedia.org/wiki/Portable_Document_Format
203. https://fr.wikipedia.org/wiki/Microsoft_Excel
204. https://fr.wikipedia.org/wiki/Fichier_texte
205. https://fr.wikipedia.org/wiki/Hypertext_Markup_Language
206. https://fr.wikipedia.org/wiki/Extensible_Markup_Language
207. https://fr.wikipedia.org/wiki/Microsoft_Excel
208. https://fr.wikipedia.org/wiki/Ressource_(informatique)
209. https://docs.python.org/3/reference/compound_stmts.html#the-with-statement

```
with open ("nom-fichier", "w") as f:    # ouverture

    f.write ( s )    # écriture de la chaîne de caractères  s
    f.write ( s2 )   # écriture de la chaîne de caractères  s2
    ...
```

L'instruction `f.close()` est implicite et automatiquement exécutée dès que le programme sort de la section *with*. Lorsque que le programme se termine, même s'il reste des fichiers ńăouvertsă" pour lesquels la méthode `close` n'a pas été appelée, ils seront automatiquement fermés.

Certains caractères sont fort utiles lors de l'écriture de fichiers texte afin d'organiser les données. Le symbole `;` est très utilisé comme séparateur de colonnes pour une matrice, on utilise également le passage à la ligne ou la tabulation. Comme ce ne sont pas des caractères ńăvisiblesă", ils ont des codes :

— `\n` : passage à la ligne
— `\t` : tabulation, indique un passage à la colonne suivante dans le format tsv [210] (Tabulation-separated values).

Il existe peu de manières différentes de conserver une matrice dans un fichier, le programme ressemble dans presque tous les cas à celui qui suit :

```
mat =  ... # matrice de type liste de listes
f = open ("mat.txt", "w")
for i in range (0,len (mat)) :                  # la fonction join est aussi
    for j in range (0, len (mat [i])) :         # fréquemment utilisée
        f.write ( str (mat [i][j]) + "\t")      # pour assembler les chaînes
    f.write ("\n")                              # un une seule et réduire le
f.close ()                                      # nombre d'appels à f.write
```

Ou encore :

```
mat =  ... # matrice de type liste de listes
with open ("mat.txt", "w") as f:
    for i in range (0,len (mat)) :
        for j in range (0, len (mat [i])) :
            f.write ( str (mat [i][j]) + "\t")
        f.write ("\n")
```

La fonction open [211] accepte deux paramètres, le premier est le nom du fichier, le second définit le mode d'ouverture : `"w"` pour écrire (**w**rite), ńăaă" pour écrire et ajouter (**a**ppend), ńără" pour lire (**r**ead). Ceci signifie que la fonction `open` sert à ouvrir un fichier quelque soit l'utilisation qu'on en fait.

A la première écriture dans un fichier (premier appel à la fonction `write`, la taille du fichier créée est souvent nulle. L'écriture dans un fichier n'est pas immédiate, le langage *python* attend d'avoir reçu beaucoup d'informations avant de les écrire physiquement sur le disque dur. Les informations sont placées dans un tampon ou buffer [212]. Lorsque le tampon est plein, il est écrit sur disque dur. Pour éviter ce délai, il faut soit fermer puis réouvrir le fichier soit appeler la méthode flush [213] qui ne prend aucun paramètre. Ce mécanisme vise à réduire le nombre d'accès au disque dur car selon les technologies, il n'est pas nécessairement beaucoup plus long d'y écrire un caractère plutôt que 1000 en une fois.

Ecriture en mode ńăajoută"

Lorsqu'on écrit des informations dans un fichier, deux cas se présentent. Le premier consiste à ne pas tenir compte du précédent contenu de ce fichier lors de son ouverture pour écriture et à l'écraser. C'est le cas traité par le précédent paragraphe. Le second cas consiste à ajouter toute nouvelle information à celles déjà présentes lors de l'ouverture du fichier. Ce second cas est presque identique au suivant hormis la première ligne qui change :

210. https://fr.wikipedia.org/wiki/Tabulation-separated_values
211. https://docs.python.org/3/library/functions.html?highlight=open#open
212. https://en.wikipedia.org/wiki/Data_buffer
213. https://docs.python.org/3.6/library/io.html#io.IOBase.flush

```
with open ("nom-fichier", "a") as f:    # ouverture en mode ajout, mode "a"
    ...
```

Pour comprendre la différence entre ces deux modes d'ouverture, voici deux programmes. Celui de gauche n'utilise pas le mode ajout tandis que celui de droite l'utilise lors de la seconde ouverture.

Premier programme

```
with open ("essai.txt", "w") as f:
    f.write (" premiere fois ")
    f.close ()

with f = open ("essai.txt", "w") as f:
    f.write (" seconde fois ")
    f.close ()
```

Second programme

```
with open ("essai.txt", "w") as f:
    f.write (" premiere fois ")
    f.close ()

with f = open ("essai.txt", "a") as f:  ###
    f.write (" seconde fois ")
    f.close ()
```

Le premier programme crée un fichier `"essai.txt"` qui ne contient que les informations écrites lors de la seconde phase d'écriture, soit `seconde fois`. Le second utilise le mode ajout lors de la seconde ouverture. Le fichier `"essai.txt"`, même s'il existait avant l'exécution de ce programme, est effacé puis rempli avec l'information `premiere fois`. Lors de la seconde ouverture, en mode ajout, une seconde chaîne de caractères est ajoutée. le fichier `"essai.txt"`, après l'exécution du programme contient donc le message : `premiere fois seconde fois`.

Un des moyens pour comprendre ou suivre l'évolution d'un programme est d'écrire des informations dans un fichier ouvert en mode ajout qui est ouvert et fermé sans cesse. Ce sont des fichiers de *traces* ou de log [214]. Ils sont souvent utilisés pour vérifier des calculs complexes. Ils permettent par exemple de comparer deux versions différentes d'un programme pour trouver à quel endroit ils commencent à diverger.

Lecture

La lecture d'un fichier permet de retrouver les informations stockées grâce à une étape préalable d'écriture. Elle se déroule selon le même principe, à savoir :

1. ouverture du fichier en mode lecture,
2. lecture,
3. fermeture.

Une différence apparaît cependant lors de la lecture d'un fichier : celle-ci s'effectue ligne par ligne alors que l'écriture ne suit pas forcément un découpage en ligne. Les instructions à écrire pour lire un fichier diffèrent rarement du schéma qui suit où seule la ligne indiquée par `(*)` change en fonction ce qu'il faut faire avec les informations lues.

```
with f = open ("essai.txt", "r") as f:  # ouverture du fichier en mode lecture
    for ligne in f :                    # pour toutes les lignes du fichier
        print ligne                     # on affiche la ligne (*)
    # f.close ()                        # on ferme le fichier, ce qui est implicite avec with
```

214. https://fr.wikipedia.org/wiki/Historique_(informatique)

Pour des fichiers qui ne sont pas trop gros (< 100000 lignes), il est possible d'utiliser la méthode readlines [215] qui récupère toutes les lignes d'un fichier texte en une seule fois. Le programme suivant donne le même résultat que le précédent.

```
with open ("essai.txt", "r") as f: # ouverture du fichier en mode lecture
    l = f.readlines ()             # lecture de toutes les lignes, placées dans une␣
↪liste

for s in l:
    print(s)                       # on affiche les lignes à l'écran (*)
```

Lorsque le programme précédent lit une ligne dans un fichier, le résultat lu inclut le ou les caractères (`\n`, `\r` - sous Windows seulement) qui marquent la fin d'une ligne. C'est pour cela que la lecture est parfois suivie d'une étape de nettoyage.

```
with open ("essai.txt", "r") as f: # ouverture du fichier en mode lecture
    l = f.readlines ()             # lecture de toutes les lignes, placées dans une␣
↪liste

# contiendra la liste des lignes nettoyées
l_net = [ s.strip ("\n\r") for s in l ]
```

Les informations peuvent être structurées de façon plus élaborée dans un fichier texte, c'est le cas des formats HTML [216] et XML [217]. Pour ce type de format plus complexe, il est déconseillé de concevoir soi-même un programme capable de les lire, il existe presque toujours un module qui permette de le faire. C'est le cas du module html.parser [218] ou xml [219]. De plus, les modules sont régulièrement mis à jour et suivent l'évolution des formats qu'ils décryptent.

Un fichier texte est le moyen le plus simple d'échanger des matrices ou des données avec un tableur et il n'est pas besoin de modules dans ce cas. Lorsqu'on enregistre une feuille de calcul sous format texte, le fichier obtenu est organisé en colonnes : sur une même ligne, les informations sont disposées en colonne délimitées par un séparateur qui est souvent une tabulation (`\t`) ou un point virgule comme dans l'exemple suivant :

```
nom  ; prénom ; livre
Hugo  ; Victor  ; Les misérables
Kessel ; Joseph  ; Le lion
Woolf ; Virginia  ; Mrs Dalloway
Calvino ; Italo  ; Le baron perché
```

Pour lire ce fichier, il est nécessaire de scinder chaque ligne en une liste de chaînes de caractères, on utilise pour cela la méthode split [220] des chaînes de caractères.

```
mat = []                           # création d'une liste vide,
with open ("essai.txt", "r") as f: # ouverture du fichier en mode lecture
    for li in f :                  # pour toutes les lignes du fichier
        s = li.strip ("\n\r")      # on enlève les caractères de fin de ligne
        l = s.split (";")          # on découpe en colonnes
        mat.append (l)             # on ajoute la ligne à la matrice
```

Ce format de fichier texte est appelé CSV [221] (Comma Separated Value), il peut être relu depuis un programme *python* comme le montre l'exemple précédent, par *Excel* en précisant que le format du fichier est le format *CSV* et par toutes les applications ou langages traitant de données. Pour les valeurs numériques, il ne faut pas oublier de convertir en caractères lors de l'écriture et de convertir en nombres lors de la lecture.

215. https://docs.python.org/3/library/io.html?highlight=readlines#io.IOBase.readlines
216. https://fr.wikipedia.org/wiki/Hypertext_Markup_Language
217. https://fr.wikipedia.org/wiki/Extensible_Markup_Language
218. https://docs.python.org/3/library/html.parser.html
219. https://docs.python.org/3/library/xml.html?highlight=xml#module-xml
220. https://docs.python.org/3/library/stdtypes.html?highlight=split#str.split
221. https://fr.wikipedia.org/wiki/Comma-separated_values

Les nombres réels s'écrivent en anglais avec un point pour séparer la partie entière de la partie décimale. En français, il s'agit d'une virgule. Il est possible que, lors de la conversion d'une matrice, il faille remplacer les points par des virgules et réciproquement pour éviter les problèmes de conversion.

Encoding et les accents

Par défaut, un fichier n'accepte pas d'enregistrer des accents, uniquement les acaractères ascii [222]. C'est pourquoi il faut presque tout le temps utiliser le paramètre *encoding* de la fonction open [223] que ce soit pour écrire ou lire.

```
with open("fichier.txt", "r", encoding="utf-8") as f:
    texte = f.read()
```

L'encoding *utf-8* est une façon de représenter les caractères, les caractères ascii sur un octet, les autres sur deux ou trois octets. Cet encoding est le plus fréquent sur internet.

5.1.2 Fichiers zip

Les fichiers zip [224] sont très répandus de nos jours et constituent un standard de compression facile d'accès quelque soit l'ordinateur et son système d'exploitation. Le langage *python* propose quelques fonctions pour compresser et décompresser ces fichiers par l'intermédiaire du module zipfile [225]. Le format de compression *zip* est un des plus répandus bien qu'il ne soit pas le plus performant. D'autres formats proposent de meilleurs taux de compression sur les fichiers textes existent comme 7-zip [226]. Ce format n'est pas seulement utilisé pour compresser mais aussi comme un moyen de regrouper plusieurs fichiers en un seul.

Lecture

L'exemple suivant permet par exemple d'obtenir la liste des fichiers inclus dans un fichier *zip* :

```
import zipfile
with zipfile.ZipFile ("exemplezip.zip", "r") as fz:
    for info in fz.infolist () :
        print(info.filename, info.date_time, info.file_size)
```

Les fichiers compressés ne sont pas forcément des fichiers textes mais de tout format. Le programme suivant extrait un fichier parmi ceux qui ont été compressés puis affiche son contenu (on suppose que le fichier lu est au format texte donc lisible).

```
import zipfile
with zipfile.ZipFile ("exemplezip.zip", "r") as fz:
    data = fz.read ("informatique/testzip.py")
print(data)
```

On retrouve dans ce cas les étapes d'ouverture et de fermeture même si la première est implicitement inclus dans le constructeur de la classe ZipFile [227].

Ecriture

Pour créer un fichier *zip*, le procédé ressemble à la création de n'importe quel fichier. La seule différence provient du fait qu'il est possible de stocker le fichier à compresser sous un autre nom à l'intérieur du fichier *zip*, ce qui explique

222. https://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange
223. https://docs.python.org/3/library/functions.html?highlight=open#open
224. https://fr.wikipedia.org/wiki/ZIP_(format_de_fichier)
225. https://docs.python.org/3/library/zipfile.html
226. http://www.7-zip.org/
227. https://docs.python.org/3/library/zipfile.html#zipfile.ZipFile

les deux premiers arguments de la méthode write [228]. Le troisième paramètre indique si le fichier doit être compressé ZIP_DEFLATED [229] ou non ZIP_STORED [230].

```
import zipfile
with zipfile.ZipFile ("test.zip", "w") as f:
    file.write ("fichier.txt", "nom_fichier_dans_zip.txt", zipfile.ZIP_DEFLATED)
```

Une utilisation possible de ce procédé serait l'envoi automatique d'un mail contenant un fichier *zip* en pièce jointe. Une requête comme *python* précédant le nom de votre serveur de mail permettra, via un moteur de recherche, de trouver des exemples sur Internet.

Selon les serveurs de mails, le programme permettant d'envoyer automatiquement un mail en *python* peut varier. L'exemple suivant permet d'envoyer un email automatiquement via un serveur de mails, il montre aussi comment attacher des pièces jointes. Il faut bien sûr être autorisé à se connecter. De plus, il est possible que l'exécution de ce programme ne soit pas toujours couronnée de succès si le mail est envoyé plusieurs fois à répétition, ce comportement est en effet proche de celui d'un spammeur.

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.base import MIMEBase
from email.mime.text import MIMEText
from email.utils import formatdate
from email import encoders
import os

def envoyer_mail (aqui, sujet, contenu, files = []):
    de = "email de l'auteur"
    msg = MIMEMultipart()
    msg['From'] = de
    msg['To'] = aqui
    msg['Date'] = formatdate (localtime = True)
    msg['Subject'] = sujet

    msg.attach(MIMEText(contenu))
    for file in files:
        part = MIMEBase('application', 'octet-stream')
        with open(file,'rb') as f:
            content = f.read()
        part.set_payload(content)
        encoders.encode_base64(part)
        part.add_header('Content-Disposition',  \
                        'attachment; filename="%s"' % os.path.basename(file))
        msg.attach(part)

    smtp = smtplib.SMTP("smtp.gmail.com", 587)
    smtp.ehlo()
    smtp.starttls()
    smtp.ehlo()
    smtp.login("login", "mot_de_passe")

    smtp.sendmail(de, aqui, msg.as_string())
    smtp.close()

envoyer_mail("destinataire", "sujet","contenu", ["mail.py"])
```

228. https://docs.python.org/3/library/zipfile.html#zipfile.ZipFile.write
229. https://docs.python.org/3/library/zipfile.html#zipfile.ZIP_DEFLATED
230. https://docs.python.org/3/library/zipfile.html#zipfile.ZIP_STORED

5.1.3 Manipulation de fichiers

Il arrive fréquemment de copier, recopier, déplacer, effacer des fichiers. Lorsqu'il s'agit de quelques fichiers, le faire manuellement ne pose pas de problème. Lorsqu'il s'agit de traiter plusieurs centaines de fichiers, il est préférable d'écrire un programme qui s'occupe de le faire automatiquement. Cela peut être la création automatique d'un fichier *zip* incluant tous les fichiers modifiés durant la journée ou la réorganisation de fichiers musicaux au format *mp3* à l'aide de modules complémentaires tel que mutagen [231].

Pour ceux qui ne sont pas familiers des systèmes d'exploitation, il faut noter que *Windows* ne fait pas de différences entre les majuscules et les minuscules à l'intérieur d'un nom de fichier. Les systèmes *Linux* et *Mac OSX* font cette différence. Ceci explique que certains programmes aient des comportements différents selon le système d'exploitation sur lequel ils sont exécutés ou encore que certains liens Internet vers des fichiers ne débouchent sur rien car ils ont été saisis avec des différences au niveau des minuscules majuscules.

Gestion des noms de chemins

Le module os.path [232] propose plusieurs fonctions très utiles qui permettent entre autres de tester l'existence d'un fichier, d'un répertoire, de récupérer diverses informations comme sa date de création, sa taille... La liste qui suit est loin d'être exhaustive mais elle donne une idée de ce qu'il est possible de faire.

abspath(path) [233]	Retourne le chemin absolu d'un fichier ou d'un répertoire.
commonprefix(list) [234]	Retourne le plus grand préfixe commun à un ensemble de chemins.
dirname(path) [235]	Retourne le nom du répertoire.
exists(path) [236]	Dit si un chemin est valide ou non.
getatime(path) [237]	date de la dernière modification
getmtime(path) [238]	date de la dernière modification
getctime(path) [239]	date de la création
getsize(file) [240]	Retourne la taille d'un fichier.
isabs(path) [241]	Retourne `True` si le chemin est un chemin absolu.
isfile(path) [242]	Retourne `True` si le chemin fait référence à un fichier.
isdir(path) [243]	Retourne `True` si le chemin fait référence à un répertoire.
join(p1, p2, ...) [244]	Construit un nom de chemin étant donné une liste de répertoires.
split(path) [245]	Découpe un chemin, isole le nom du fichier ou le dernier répertoire des autres répertoires.
splitext(path) [246]	Découpe un chemin en nom + extension.

231. https://pypi.python.org/pypi/mutagen
232. https://docs.python.org/3/library/os.path.html
233. https://docs.python.org/3/library/os.path.html#os.path.abspath
234. https://docs.python.org/3/library/os.path.html#os.path.commonprefix
235. https://docs.python.org/3/library/os.path.html#os.path.dirname
236. https://docs.python.org/3/library/os.path.html#os.path.exists
237. https://docs.python.org/3/library/os.path.html#os.path.getatime
238. https://docs.python.org/3/library/os.path.html#os.path.getmtime
239. https://docs.python.org/3/library/os.path.html#os.path.getctime
240. https://docs.python.org/3/library/os.path.html#os.path.getsize
241. https://docs.python.org/3/library/os.path.html#os.path.isabs
242. https://docs.python.org/3/library/os.path.html#os.path.isfile
243. https://docs.python.org/3/library/os.path.html#os.path.isdir
244. https://docs.python.org/3/library/os.path.html#os.path.join
245. https://docs.python.org/3/library/os.path.html#os.path.split
246. https://docs.python.org/3/library/os.path.html#os.path.splitext

Copie, suppression

copy (f1,f2) [247]	Copie le fichier `f1` vers `f2`.
chdir (p) [248]	Change le répertoire courant, cette fonction peut être importante lorsqu'on utilise la fonction system [249] du module os [250] pour lancer une instruction en ligne de commande ou lorsqu'on écrit un fichier sans préciser le nom du répertoire, le fichier sera écrit dans ce répertoire courant qui est par défaut le répertoire où est situé le programme *python*. C'est à partir du répertoire courant que sont définis les chemins relatifs.
getcwd () [251]	Retourne le répertoire courant, voir la fonction `chdir`.
mkdir (p) [252]	Crée le répertoire `p`. \ hline
makedirs (p) [253]	Crée le répertoire `p` et tous les répertoires des niveaux supérieurs s'ils n'existent pas. Dans le cas du répertoire `d:/base/repfinal`, crée d'abord `d:/base` s'il n'existe pas, puis `d:/base/repfinal`.
remove (f) [254]	Supprime un fichier.
rename (f1,f2) [255]	Renomme un fichier
rmdir (p) [256]	Supprime un répertoire

Liste de fichiers

La fonction listdir [257] permet de retourner les listes des éléments inclus dans un répertoire (fichiers et sous-répertoires). Le module glob [258] propose une fonction plus intéressante qui permet de retourner la liste des éléments d'un répertoire en appliquant un filtre. Le programme suivant permet par exemple de retourner la liste des fichiers et des répertoires inclus dans un répertoire.

<<<

```
import glob
import os.path


def liste_fichier_repertoire(folder, filter):
    # résultats
    file, fold = [], []

    # recherche des fichiers obéissant au filtre
    res = glob.glob(folder + "\\" + filter)

    # on inclut les sous-répertoires qui n'auraient pas été
    # sélectionnés par le filtre
    rep = glob.glob(folder + "\\*")
    for r in rep:
        if r not in res and os.path.isdir(r):
```

(suite sur la page suivante)

247. https://docs.python.org/3/library/shutil.html?highlight=shutil#shutil.copy
248. https://docs.python.org/3/library/os.html?highlight=chdir#os.chdir
249. https://docs.python.org/3/library/os.html?highlight=chdir#os.system
250. https://docs.python.org/3/library/os.html?highlight=chdir#os.chdir
251. https://docs.python.org/3/library/os.html?highlight=chdir#os.getcwd
252. https://docs.python.org/3/library/os.html?highlight=chdir#os.mkdir
253. https://docs.python.org/3/library/os.html?highlight=chdir#os.makedirs
254. https://docs.python.org/3/library/os.html?highlight=chdir#os.remove
255. https://docs.python.org/3/library/os.html?highlight=chdir#os.rename
256. https://docs.python.org/3/library/os.html?highlight=chdir#os.rmdir
257. https://docs.python.org/3/library/os.html?highlight=chdir#os.listdir
258. https://docs.python.org/3/library/glob.html?highlight=glob#module-glob

(suite de la page précédente)

```
            res.append(r)

    # on ajoute fichiers et répertoires aux résultats
    for r in res:
        path = r
        if os.path.isfile(path):
            # un fichier, rien à faire à part l'ajouter
            file.append(path)
        else:
            # sous-répertoire : on appelle à nouveau la fonction
            # pour retourner la liste des fichiers inclus
            fold.append(path)
            fi, fo = liste_fichier_repertoire(path, filter)
            file.extend(fi)  # on étend la liste des fichiers
            fold.extend(fo)  # on étend la liste des répertoires
    # fin
    return file, fold


folder = r"."
filter = "*.rst"
file, fold = liste_fichier_repertoire(folder, filter)

for i, f in enumerate(file):
    print("fichier ", f)
    if i >= 10:
        break
for i, f in enumerate(fold):
    print("répertoire ", f)
    if i >= 10:
        break
```

>>>

```
```

Le programme repose sur l'utilisation d'une fonction récursive qui explore d'abord le premier répertoire. Elle se contente d'ajouter à une liste les fichiers qu'elle découvre puis cette fonction s'appelle elle-même sur le premier sous-répertoire qu'elle rencontre. La fonction walk [259] permet d'obtenir la liste des fichiers et des sous-répertoire. Cette fonction parcourt automatiquement les sous-répertoires inclus, le programme est plus court mais elle ne prend pas en compte le filtre qui peut être alors pris en compte grâce aux expressions régulières (voir regex_label_chap).

<<<

```
import os


def liste_fichier_repertoire(folder):
    file, rep = [], []
    for r, d, f in os.walk(folder):
        for a in d:
            rep.append(r + "/" + a)
        for a in f:
            file.append(r + "/" + a)
    return file, rep
```

(suite sur la page suivante)

259. https://docs.python.org/3/library/os.html?highlight=walk#os.walk