Aide mémoire / Notes sur Python 3

Les premières sections de ce document sont reprises dans le support de cours. Les autres sont des notes, légèrement désorganisées, qui sont complétées en cours d'année. Elles sont ensuite intégrées dans les notes de cours si nécessaire.

Rappels des principales constructions

Boucles

Les boucles `for` de Python permettent de parcourir n'importe quel objet *itérable*. C'est le cas de `range(2,10)` qui «contient» les entiers de 2 à 9. C'est aussi le cas de n'importe quelle *séquence*. On pourra ainsi écrire en Python :

```
for i in (3,7,8,12) :
    print(i)
```

ou encore :

```
l=('abc',3,(1,2))
for v in l :
    print(v)
```

Voici d'autres exemples :

```
l=[3.14, "pi" , 42 ]
for a in l :
    print('Objet : ',a)
for k in range(len(l)) :
    print('Elt',k,':',l[k])
k=0
while k<len(l) :
    print('Elt',k,':',l[k])
    k=k+1
```

Boucle while avec else

Cette construction est spécifique à Python :

```
#pseudo code
while truc:
  bloc A
else :
  bloc B
```

Le bloc B est exécuté dès que truc devient (ou est) faux. Il est donc toujours exécuté une fois sauf si on sort de la boucle avec un break

Tests

```
t=(2,6,43,56,73,36,45,23,24)
for v in t :
    if v%3==0 :
        print('La valeur',v,'est multiple de 3')
    else :
        print(v,'n\'est pas divisible par 3')
```

Fonctions

```
def aucarre(u) :
    v=u**2
    return v
```

La fonction `aucarre` possède un argument (paramètre formel u), et renvoie un objet, celui qui est référencé par v au moment de l'exécution de la ligne `return v`.

Utilisation de la fonction :

```
a=5
b=aucarre(a)
print(a,'au carré vaut',b)
```

Types simples

- Doc Python sur les types standards

Entiers

Le type `int`, contrairement à Python 2, permet de représenter les entiers de taille machine (quelques octets) ou les grands entiers (limités uniquement par la taille de la mémoire). Le passage de l'un à l'autre (entiers machines ou grands entiers) est transparent pour l'utilisateur.

Pour indiquer une valeur entière, il est possible d'utiliser différentes bases :

```
42   # en décimal
0o52 # en octal
0x2A # en hexadécimal
0b101010 # en binaire
```

On réalise les opération inverses avec les fonctions `bin`, `oct`, et `hex`.

Les opérations arithmétiques ordinaires sont disponibles sur les entiers :

```
6+9
6-9
6*9 # qui ne fait pas 42
9/6
9//6 # division entière
9%6  # reste
```

Notons qu'en Python 3, l'opérateur de division `/` est un opérateur de **division non entière** contrairement à ce qui se faisait en Python 2 et contrairement ce qui se fait dans de nombreux langages. Même si une division tombe juste, le résultat de l'opération `/` sera un `float`. Il faut donc penser, lorsqu'on souhaite manipuler des entiers, à utiliser l'opérateur `//`.

Opération	**Type retour**	**Description**
x+y	int	Somme
x-y	int	Différence
x*y	int	Produit
x/y	float	Quotient
x//y	int	Quotient entier
x%y	int	Reste de la division entière
-x	int	Opposé de x
x&y	int	Opération *et* sur les bits de x et y
x|y	int	Opération *ou* sur les bits de x et y
x^y	int	Opération *ou exclusif* sur les bits de x et y
x<<y	int	Décalage à gauche de y bits sur x
x>>y	int	Décalage à droite de y bits sur x
~x	int	Inversion des bits de x
abs(x)	int	Valeur absolue
divmod(x, y)	(q,r)	Quotient (q) et reste (r) de la division entière de x par y
float(x)	float	Conversion vers un float
hex(x)	str	Représentation hexadécimale
oct(x)	str	Représentation octale
bin(x)	str	Représentation binaire
int(x)	int	Conversion vers un int (pas de changement donc...)
pow(x, y[, z])	int	x à la puissance y modulo z si z est précisé
repr(x)	str	Représentation *officielle* sous forme de chaîne de caractères
str(x)	str	Conversion en chaîne de caractères

Booléens

Les deux seules valeur du type `bool` sont `True` et `False`. Ces deux valeurs peuvent être entrées litéralement ou peuvent être le résultat d'opérateurs de comparaison, comme >, <, >=, <=, ==(égal), !=(différent).

Les opérateurs dont les opérandes sont des booléens sont les opérateurs logiques `or`, `and`, `not`.

L'évaluation des expressions logiques n'est *pas complète*. Elle s'arrête dès que le résultat est connu. En logique, dans l'expression `a and b`, il suffit que `a` soit évalué à `False` pour que le résultat soit `False`. En Python, si `a` est évalué à `False`, le résultat de `a and b` sera `a`. De même si `a` est évalué à `True`, le résultat de `a and b` sera `b`. Ceci permet de construire des expressions riches, mais qui ne sont pas toujours très simples à comprendre. À notre niveau, il convient d'éviter de les écrire, mais il faut être capable de les comprendre.

```
6<10 and 6*9 # Vaut 54 puisque 6<10 est vrai
42==6*7 or 6*9 # Vaut True (valeur de 42==6*7)
```

Les expression qui ne sont ni le résultat de connecteurs logiques, ni le résultat d'opérateurs de comparaison sont évaluées à `True` sauf : `False`, `None`, la valeur `0` ou `0.0`, les collections vides (tuples, dictionnaires, listes, ensembles).

Nombres à virgule flottante

Le type `float` permet d'utiliser la représentation en virgule flottante (Norme IEEE 754). Toute valeur numérique comportant un point décimal ou entrée en notation scientifique sera de type `float`. Le type `float` de Python correspond au codage en double précision[1)] (c'est à dire au type `double` du C):

```
a=7.54
b=7e3
c=0.745e1
```

Le module `math` contient de nombreuses fonctions mathématiques (fonctions trigonométriques, logarithmiques...).

```
import math
a=4.5
math.sin(a)
b=1e-10
math.log10(b)
```

Informations sur les flottants : http://www.cs.berkeley.edu/~wkahan/ieee754status/

Nombres complexes

Python offre la possibilité de manipuler des nombres complexes sans utiliser de module complémentaire.

Un nombre complexe peut être indiqué littéralement en utilisant 'j' :

```
a=4+3j
b=5+0j
```

On peut aussi créer un nombre complexe en utilisant le constructeur `complex` :

```
a=complex(4,3)
b=complex(5,0)
```

Ne pas confondre le `j` complexe et la variable `j`. Ce qui suit ne crée pas de nombre complexe :

```
a=5+j   # Ajoute 5 et le contenu de la variable j
b=5+3*j # Multiplie le contenu de j par 3 et ajoute 5.
```

Pour utiliser les nombres complexes, il faudrait écrire :

```
a=5+1j
b=5+3j
```

Voici d'autres exemples :

```
a=5+4j
a.real
a.imag
abs(a)
```

Le module `cmath` contient des fonctions relatives aux nombres complexes.

Types Collections

Les collections : listes, tuples, objets itérables etc. font la force et l'efficacité des langages interprétés modernes. Savoir les manipuler permet de concevoir des programmes concis, élégants et efficaces [2)].

- Documentation Python sur les types standards

Quelques définitions

Une description complète des types de donnée Python est disponible ici : Python Standard Type Hierarchy

- **Conteneur** ou **Collection** : objet destiné à contenir d'autres objets. Une collection peut être ordonnée (c'est alors une *séquence*) ou non. Les conteneurs standards sont : `list` `tuple` `set` `str` `dict`...
- **Séquence** : collection ordonnée d'éléments indicés par des entiers. Les séquences Python sont par exemple : `list`,`tuple`,`str`. On peut accéder à un élément d'une séquence en précisant entre crochets le numéro d'ordre de l'enregistrement. La numérotation commence à 0. (un

`range` se comporte aussi comme une séquence)
- **Type `list`** : séquence modifiable d'éléments éventuellement hétérogènes
- **Type `tuple`** : séquence non modifiable d'éléments éventuellement hétérogènes.
- **Type `dict`** (dictionnaire ou tableau associatif) : collection non ordonnée modifiable d'éléments éventuellement hétérogènes. Un tableau associatif est un type de données permettant de stocker des couples clé/valeur, avec un accès très rapide à la valeur à partir de la clé. Celle-ci ne peut bien sûr être présente qu'une seule fois dans le tableau. La clé doit être hashable. Le type `dict` fait partie des collections de type **Mapping** qui associent un objet à un autre.
- **Type `set`** : collection non ordonnée d'éléments hashables distincts.
- **Itérable** : capacité, pour une collection d'égrener ses valeurs, par exemple avec une boucle `for` Python. Les objets itérables peuvent être parcourus, mais on ne peut pas nécessairement prendre un élément ou un autre en l'adressant.
- **Modifiable** : capacité pour un objet (pas forcément une collection) de *modifier son contenu sans que l'objet soit recréé*. Ce point est délicat, et il est assez spécifique à Python. Dans le cas d'un conteneur, on peut le qualifier de récursivement (ou complètement) non modifiable s'il est non modifiable et que tous les éléments qu'il contient sont récursivement non modifiables. Les termes anglais pour modifiable/non modifiable sont : *mutable*, *immutable*.
- **Hashable** : Un objet hashable est un objet récursivement non modifiable. Il peut servir de clé dans un dictionnaire. Les éléments d'un `set` doivent être hashables. Dans ce dernier cas, attention : les objets **custom** éventuellement non modifiables ont par défaut une méthode `__hash()__` qui renvoie l'id. Ils peuvent donc être ajoutés à un `set`, mais ça marche mal...

Type	Ordonné ?	Itérable ?	Modifiable ?
`list`	Oui	Oui	Oui
`tuple`	Oui	Oui	Non
`set`	Non	Oui	Oui
`str`	Oui	Oui	Non
`dict`	Non	Oui	Oui
`bytearray`	Oui	Oui	Oui
`frozenset`	Non	Oui	Non
`bytes`	Oui	Oui	Non

Accéder aux éléments

On peut accéder aux éléments d'une séquence par leurs numéros :

```
>>> l=[1,3,5,7] # création d'une liste
>>> print(l,l[1])
[1, 3, 5, 7] 3
>>> t=(2,4,6,8) # création d'un tuple
>>> print(t,t[2])
(2, 4, 6, 8) 6
>>> s='Supercalifragilistique' # création d'une chaîne
>>> print(s,s[4])
Supercalifragilistique r
```

En Python, il existe un moyen de désigner les éléments d'une séquence en partant de la fin : `s[-1]` est le *dernier* élément de la séquence, `s[-2]` est l'avant-dernier etc... En règle générale, si k>0, `s[-k]` vaut `s[len(s)-k]`.

On peut extraire une tranche de séquence, voire une tranche échantillonée :

```
begin{python}
>>> t=list(range(11)) # 0 1 2 ... 10
>>> t[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> t[3:]
[3, 4, 5, 6, 7, 8, 9, 10]
>>> t[3:6]
[3, 4, 5]
>>> t[1:8]
[1, 2, 3, 4, 5, 6, 7]
>>> t[1:8:2]
[1, 3, 5, 7]
>>> t[::2]
[0, 2, 4, 6, 8, 10]
```

Pour prendre des valeurs n'importe comment (sans loi particulière) :

```
from operator import itemgetter
t=list(range(11))
u=itemgetter(1,3,7)(t)
# (1,3,7)
```

Pour les dictionnaires, les numéros sont remplacés par des clés~:

```
>>> cri={'chien' : 'aboie', 'chat':'miaule', 'caravane':'passe'}
>>> cri
{'chien': 'aboie', 'chat': 'miaule', 'caravane': 'passe'}
>>> cri['chat']
'miaule'
```

Agir sur les collections

Lorsqu'on agit sur un objet (`s`), il y a deux façons de le faire :

1. En affectant `s`, comme dans `s=s[:]+s[5]`. Ceci fonctionne que l'objet soit modifiable ou non. On *extrait* `s[:]` et `s[5]`, et on réalise l'opération +, le résultat étant un nouvel objet (il se peut que l'opération + ne soit pas possible avec les opérandes données...). Puis `s` référencera (affectation) le nouvel objet, l'ancien n'étant plus référencé par `s` (mais il existe peut être encore, référencé par une autre variable). On peut constater que l'objet référencé par `s` n'est plus le même en contrôlant la valeur `id(s)` avant et après l'opération.
2. En appelant une méthode qui agit sur l'objet qui est référencé par `s` comme dans `s.sort()`. Dans ce cas, l'identifiant (`id`) de `s` n'est pas modifié et c'est l'objet qui était désigné par `s` qui est modifé (le type de `s` doit donc être modifiable (on peut trier ainsi une liste, mais pas un tuple)).

Il existe en outre quelques algorithmes implémentés de deux manières :

- modifiant l'objet d'origine

- retournant un nouvel objet modifié

Par exemple, si `t` est une séquence :

- `t.sort()` trie la séquence `t` **sur place** (elle doit être modifiable)
- `sorted(t)` ne modifie pas `t` et renvoie une nouvelle séquence triée (`t` n'a pas besoin d'être modifiable)

On trouve de même : `reversed(t)` et `t.reverse()` etc...

Attention, `reversed(t)` renvoie un itérateur, qui est évalué le plus tard possible :

```
t=[1,2,3]
u=reversed(t)
t[1]=10
v=list(u)
# v contient 3, 10, 1
```

Copier une collection

La copie des types simples ne pose pas de problème particulier. Il faut en revanche être prudent avec les collections. On peut copier une collection ainsi :

```
a=[1,2,[5,6,7]]
b=list(a)
```

Mais c'est une copie *superficielle* (*shallow copy*). Si un des éléments est modifiable (si c'est une liste par exemple), ça peut être un problème. Le test suivant permet de comprendre ce qui se passe :

```
a=[1,[1,2]]
print (id(a), id(a[0]), id(a[1]) )
b=list(a)
print (id(b), id(b[0]), id(b[1]) )
print(a,b)
b[0]=42
print(a,b)
b[1][0]=54
print(a,b)
```

Pour obtenir un comportement différent, on dispose d'un module `copy` qui offre une copie en profondeur (*deep copy*) :

```
import copy
a=[1,2,[5,6,7]]
b=copy.deepcopy(a)
b[2][0]=3.2
```

```
print(a,b)
```

Opérations sur les collections

Opérations sur les séquences

Les manipulations suivantes sont possibles sur les séquences en général, donc sur les listes, les tuples et les chaînes en particulier.

Méthode	Type retour	Description
`list(sequence)`	list	Renvoie une liste contenant les objets de `sequence`
x==y	bool	Retourne `True` si x et y contiennent les même objets (au sens de == pour chaque paire d'objets)
x>=y	bool	Vrai si x est avant y dans l'*ordre lexicographique* ou si x==y. Faux sinon.
x<=y	bool	Vrai si x est après y dans l'*ordre lexicographique* ou si x==y. Faux sinon.
x>y	bool	Vrai si x est avant y dans l'*ordre lexicographique*. Faux sinon
x<y	bool	Vrai si x est après y dans l'*ordre lexicographique*. Faux sinon
x!=y	bool	Vrai si x et y sont de longueur différente ou si un item de x est différent d'un item de y. Faux sinon.
`x[i]`	item	Retourne l'item en position `i` de x. Si `i` est strictement négatif, vaut `x[len(x)+i]`
`x[[i]:[j]]`	list	Retourne la tranche d'items de la position `i` à la position `j-1`. Si `i` n'est pas précisé, il est pris égal à 0. Si `j` n'est pas précisé, il est pris égal à `len(x)`. Les indices négatifs ont la même signification que ci-dessus.
`iter(x)`	iterator	Retourne un itérateur sur x
`repr(x)`	str	Représentation *officielle* de x sous forme de chaîne de caractères

Opérations sur les listes

Méthode	Type retour	Description
`list()`	list	Renvoie une liste vide
`[item[,item]+]`	list	Liste contenant les *items* donnés entre crochets
`del x[i]`		Efface l'objet en position `i` de x
`x[i]=e`		Affecte l'item e à la position `i` de x. L'item `i` doit déjà exister (la liste n'ets pas étirée).
x+=y		Modifie la liste x en lui ajoutant les éléments de y
x*=i		Modifie la liste x pour qu'elle contienne `i` copies concaténée de la liste originale.
`x.append(e)`	None	Modifie x en lui ajoutant e comme dernire élément.
`x.count(e)`	int	Retourne le nombre d'apparitions de e dans x. Les apparitions sont détectées en utilisant ==.
`x.extend(iter)`	None	Modifie x en lui ajoutant les éléments de l'itérable `iter`.

Méthode	**Type retour**	**Description**
`x.index(e,[i,[j]])`	int	Retourne l'indice du premier élément de x égale à e (au sens de ==), situé entre les indices `i` et `j-1`. L-ve l'exception `ValueError` si un tel élément n'existe pas.
`x.insert(i, e)`	None	Modifie x en insérant e de tel sorte qu'il se trouve à l'indice `i`.
`x.pop([index])`	item	Supprime (modifie donc x) et renvoie l'élément en position `index`. Si `index` n'est pas précisé, il est pris par défaut égal à `len(x)-1`.
`x.remove(e)`	None	Modifie x en supprimant la première occurence de e. Lève l'exception `ValueError` si e n'est pas trouvé.
`x.reverse()`	None	Modifie x en renversant l'ordre de ses éléments.
`x.sort()`	None	Modifie x pour qu'il soit trié en utilisant la méthode de comparaison `__cmp__` de ses éléments. La méthode accepte deux paramètres optionels : key et `reverse`. Si `reverse` vaut `True`, le tri est fait à l'envers (décroissant). Le paramètre key est une fonction qui prend en paramètre un élément et renvoie la valeur qui sera utilisée réellement pour le tri.

Opérations sur les itérables

Les opérations qui suivent concernent tous les itérables, donc en particulier les tuples et les listes.

Méthode	**Description**
`x+y`	Concaténation
`x*i`	Retourne un **nouvel** itérable contenant `i` copies de x
`e in x`	Retourne `True` si e est dans x et `False` sinon
`all(x)`	Vaut `True` si chaque élément de x est évalué à `True`
`any(x)`	Vaut `True` s'il y a au moins un élément de x évalué à `True`
`enumerate(x,start)`	Retourne une séquence de tuples de la forme `(index,item)` énumérant les éléments de x avec leur position. `start` est un offset appliqué à la position initiale (0).
`len(x)`	Nombre d'éléments de x
`max(x,key)`	Retourne le plus grand élémént de x. L'argument key a le même rôle que pour la fonction `sorted`
`min(x,key)`	Retourne le plus petitélémént de x. L'argument key a le même rôle que pour la fonction `sorted()`
`reversed(x)`	Retourne un nouvel itérateur contenant les même éléments que x mais en sens inverse.
`sorted(x,key,reverse)`	Retourne une liste contenant les mêmes éléments que x, mais triés en utilisant la méthode de comparaison `__cmp__` de ses éléments. Si `reverse` vaut `True`, le tri est fait à l'envers (décroissant). Le paramètre key est une fonction qui prend en paramètre un élément et renvoie la valeur qui sera utilisée réellement pour le tri.
`sum(x,start)`	Retourne la somme des éléments de x augmentée de `start` (qui vaut 0 par défaut)
`zip(x1,…,xN)`	Retourne un nouvel itérateur contenant des tuples de *N* élément. Chaque élément est pris dans un des itérateurs. Exemple : `zip(('a','b','c'),(1,2,3))` vaudra `(('a',1),('b',2),('c',3))`

Opérations sur les dictionnaires

Méthode	Description
`dict()`	Retourne un dictionnaire
`dict(mapping)`	Construit un dictionnaire à partir d'un objet de type mapping (clé/valeur)
`dict(seq)`	Construit un dictionnaire à partit d'une séquence. Remplace le code `D = {}; for k, v in seq: D[k] = v`
`dict(**kwargs)`	Construit un dictionnaire à partir de paires nom=valeur : `dict(alpha=1,beta="omega"`
`k in D`	Vaut `True` si `k` est une clé de `D` et faux sinon
`del D[k]`	Efface la clé `k` du dictionnaire
`D1==D2`	Retourne `True` si les dictionaires ont les même clés faisant référence aux mêmes valeurs
`D[k]`	Renvoie la valeur associée à la clé `k` dans `D`. Si la clé `k` n'existe pas,lève l'exception `KeyError`
`iter(D)`	Renvoie un itérateur sur le dictionnaire
`len(D)`	Renvoie le nombre de clés de `D`
`D1!=D2`	Vaut `Vrai` si `D1` et `D2` ont des clés différentes ou que certaines clés identiques sont associées à des valeurs différentes
`repr(D)`	Retourne la représentation de `D`
`D[k]=e`	Associe la valeur `e` à la clé `k`
`D.clear()`	Vide `D` de ses clés
`D.copy()`	Retourne une copie non récursive de `D`
`D.get(k[,e])`	Retourne `D[k]` si `k` est une clé de `D` et `e` sinon (par défaut, `e` vaut `None`
`D.items()`	Retourne une **vue** du dictionnaire. Souvent utilisé sous la forme : `for k,v in D.items():…`

Opérations sur les ensembles

Méthode	Description
`set(iter)`	Retourne un ensemble, contenant les éléments de l'itérable
`k in E`	Vaut `True` si `k` est dans l'ensemble et faux sinon
`E.pop()`	Efface un élément au hasard et le renvoie
`E | F`	Ensemble union de deux ensembles
`E ^ F`	Ensemble des éléments qui sont dans `E` ou dans `F`, mais pas dans `E` et `F` (différence symétrique)
`E & F`	Ensemble intersection de `E` et `F`
`E - F`	Ensemble des éléments qui sont dans `E` mais pas dans `F`
`E <= F`	Vaut `Vrai` si `E` est un sous ensemble de `F`

D'autres constructions sont possibles avec les ensembles. Pour en avoir un aperçu : `help(set)`

Type modifiable / non modifiable

En Python, un objet peut être *modifiable* ou non.

Dans les documentations en anglais, on trouve systématiquement les termes : *mutable*,

immutable pour discriminer les deux familles de types en Python. Dans les documentations en français, il n'y a pas de règle, on verra : muable, mutable, immuable, variable, invariable, modifiable... Dans ce document, nous utiliserons les termes : modifiable et non modifiable (ou à la rigueur, dans un instant d'égarement, muable et immuable).

Type	Modifiable/Non modifiable
int	Non modifiable
float	Non modifiable
tuple	Non modifiable
str	Non modifiable
list	Modifiable
dict	Modifiable
set	Modifiable

On peut considérer qu'un objet est non modifiable s'il ne possède aucune méthode qui le modifie. Par exemple, une liste dispose de la méthode `append` qui rajoute un élément à la liste. L'existence même de cette méthode rend les listes modifiables. Il n'existe pas de telle méthode pour les tuples ou les chaînes, qui sont donc **non modifiables**

Les chaînes ne sont pas modifiables

Une conséquence notable du fait que le type `str` est non modifiable est qu'on ne peut pas modifier une tranche de chaîne de caractères. Ainsi, modifier la première lettre d'un mot nécessite de créer une **copie** du mot :

```
s='Paon'
t='F'+s[1:]
print(s,t)
```

Voici un autre exemple :

```
s='Nicher'
t=s
s=s[:-1]
print(s,t)
```

Le code qui précède affichera : `Niche Nicher`.

Entrées/Sorties Clavier et écran

Vos programme peuvent permettre la saisie au clavier avec la fonction `input` (Python 3) :

```
a=input('Entrez votre nom')
print(a,type(a))
```

La fonction `input` renvoie une chaîne de caractères, mais on peut la convertir au passage :

```
a=int(input('Entrez votre age'))
print(a,type(a))
```

L'affichage de chaînes formatées peut se faire de plusieurs façons différentes :

La plus simple et la moins souple :

```
>>> a=43.34456
>>> b=12
>>> print("Un entier",b,"et un float",a)
Un entier 12 et un float 43.34456
```

La nouvelle méthode : `format` :

```
>>> a=43.34456
>>> b=12
>>> print("Un entier {} et un float {}".format(b,a))
Un entier 12 et un float 43.34456
>>> print("Un entier {0} et un float {1}".format(b,a))
Un entier 12 et un float 43.34456
>>> print("Un entier {1} et un float {0}".format(a,b))
Un entier 12 et un float 43.34456
>>> print("Un entier {0:03d} et un float {1:.2f}".format(b,a))
Un entier 012 et un float 43.34
```

C'est cette dernière méthode, utilisant `format` qui est préconisée. Les différentes options de formatage sont disponible ici : Format String Syntax

---

Compléments

Structure de programme propre

Lors des TDs, essentiellement, nous avons écrit des fonction et procédures que nous avons testées dans le shell. Parfois, nous avons écrit un module, comportant quelques fonctions, et un script final,

comme ceci :

```
def suivant(a) :
   """ renvoie le terme suivant dans la suite de collatz """
   if a%2==0 : return a//2
   return 3*a+1

def suite(a,n) :
   """ Renvoie n termes de la suite de collatz commençant par a """
   lst=[a]
   while len(a)<n : lst.append(suvant(lst[-1]))
   return lst

# Script principal
print(suivant(5))
l=suite(5,20)
print(l)
```

Lors de l'élaboration d'un programme complet, il est cependant conseillé de grouper le script principal dans une fonction à part, qui peut porter n'importe quel nom, mais qu'on appelle souvent `main` en référence à la fonction principale en C. Ceci évite que les variables utilisée dans le script soient visibles depuis les fonction (rendant ainsi le programme plus sifficile à vérifier si on utilise à dessein ou par erreur une de ces variables dans les fonctions) :

```
def suivant(a) :
   """ renvoie le terme suivant dans la suite de collatz """
   if a%2==0 : return a//2
   return 3*a+1

def suite(a,n) :
   """ Renvoie n termes de la suite de collatz commençant par a """
   lst=[a]
   while len(a)<n : lst.append(suvant(lst[-1]))
   return lst

def main() :
    print(suivant(5))
    l=suite(5,20)
    print(l)
```

Lorsqu'on importe ce module dans le Shell, rien ne se passe. Il faut **provoquer** l'exécution de la fonction principale en entrant dans le shell :

```
>>> main()
...
```

Il est néanmoins possible de rendre cette exécution automatique. On peut de plus différnencier les cas où le module est *importé* dans un autre module, et le cas où le module est exécuté directement. En effet, dans le premier cas, on souhaite sans doute utiliser les fonctions du module, mais pas la fonction principale. Alors que dans le second cas, on aimerait que l'exécution soit automatique.

Voici une façon de procéder, simple et claire, qui permet d'attendre tous ces objectifs :

```
def suivant(a) :
   """ renvoie le terme suivant dans la suite de collatz """
   if a%2==0 : return a//2
   return 3*a+1

def suite(a,n) :
   """ Renvoie n termes de la suite de collatz commençant par a """
   lst=[a]
   while len(lst)<n : lst.append(suivant(lst[-1]))
   return lst

def main() :
    print(suivant(5))
    l=suite(5,20)
    print(l)

if __name__=='__main__' : main()
```

Le test de la dernière ligne n'est vrai que si le module est exécuté directement, et non importé.

Si on désire néanmoins (il peut y avoir de bonnes raisons pour ça) utiliser des variables globales au module, on prendra soin de les préfixer par un tiret bas, ce qui les rendra invisible si on importe le module :

```
_coef=3
def suivant(a) :
   """ renvoie le terme suivant dans la suite de collatz
   Utilise la variable globale _coef
   """

   if a%2==0 : return a//2
   return _coef*a+1

def suite(a,n) :
   """ Renvoie n termes de la suite de collatz commençant par a """
   lst=[a]
   while len(lst)<n : lst.append(suivant(lst[-1]))
   return lst

def main() :
    global _coef
    _coef=3
    print(suite(5,20))
    _coef=5
    print(suite(5,20))

if __name__=='__main__' : main()
```

Notez que nous avons été contraints d'indiquer la ligne `global _coef` dans `main()`. Voyez ce qui

se passe si on ne le fait pas. C'est certes une contrainte, mais qu'on **souhaite** avoir car elle évite des erreurs difficile à détecter.

N'importez pas tous les noms d'un module : Évitez à tout prix les : `from math import *`, car on ne pourra plus alors obtenir l'aide spécifique de votre module (elle sera noyée dans l'aide du module de math). Faites toujours : `import math` ou, dans les cas extrêmes : `from math import sin`

Expérimenter la portée des variables, utilisation du debugger

Le petit programme suivant, à utiliser avec un *debugger* permet de mieux comprendre la notion de portée, de variables locales et de variables globales.

```
def additionne(u,v) :
    val=u+v
    return val

a=5
b=6
c=42
print(a,b)
d=additionne(a,b)
print(d)
```

Dans **Wing IDE**, sélectionnez la ligne a=5 et pressez la touche *F9*. Un point rouge apparaît dans la marge (c'est un point d'arrêt). Vous pouvez lancer l'exécution du programme en mode débuggage avec la touche *F5*. L'exécution stoppe sur le premier point d'arrêt rencontré.

Puis, vous pouvez exécuter le programme ligne à ligne avec *F7*, et même **entrer** dans les fonctions avec la touche *F6*. Essayez ces possibilités en notant bien la ligne qui s'exécute (elle s'affiche en surbrillance). Vous devez pouvoir exécuter le programme ligne à ligne en choisissant d'entrer ou non dans la fonction `additionne`.

Pour utiliser le debugger avec **Idle**, aller dans le menu *Debug/Debugger*. **Cochez toutes les cases**, puis exécutez le programme. Vous pouvez exécuter pas à pas à cliquant le bouton *Step* du débugger. DAns la fenêtre de l'éditeur, vous verrez la ligne en cours d'exécution s'afficher en surbrillance. Noez, en bas de la fenêtre de débuggage, les noms des variables locales, globales, et les valeurs des objets qu'elles référencent. Avec Idle, il faut commence

Lorsque vous maîtrisez la manière d'exécuter pas à pas le programme, relancez le débuggage (avec Idle, ouvrez aussi l'onglet *Pile de données*). Exécutez pas à pas en entrant dans la fonction

`additionne`. Notez quelles sont les variables *locales* : ce sont les variables visibles à ce moment de l'exécution. Voyez à quel moment les variables apparaissent et disparaissent de la liste des variables locales : vous observez ainsi leur portée.

Renommez la variable `val` en `a`, et constatez qu'il y a maintenant deux variables a distinctes.

Constructions courantes

Ajout d'éléménts dans une liste (**modification** de l'objet liste)

```
l1=[1,2,3]
l2=[4,5,6]
l1.append(12)
# => l1 = [1,2,3,12]
l2.insert(0,34)  # Insère 34 en position 0
# => l2 = [34,4,5,6]
l2.insert(3,42) # Insère 42 en position 3
# => l2 = [34,4,5,42,6]
l1.extend(l2)
# => l1 = [1,2,3,12,34,4,5,42,6]
```

Effacement d'un élément d'une liste

Effacement d'un élément d'une liste à partir de l'indice (**modification** de l'objet liste), ou *effacement* d'une variable [3] :

```
l1=[1,2,3,4,5]
del l1[2]
# => l1 = [1,2,4,5]
a=45
del a
print(a)
# => NameError : name 'a' is not defined
```

Effacement d'un élément d'une liste à partir de la valeur :

```
l1=[1,2,3,66,23,66,5]
l1.remove(66)
# => l1 = [1,2,3,23,66,5]
```

Transformation chaîne <-> liste

La méthode `split` découpe une chaîne en morceaux, chaque morceau est placé dans une liste.

```
s="""Gloire à qui n'ayant pas d'idéal sacro-saint
Se borne à ne pas trop emmerder ses voisins"""
```

```
l=s.split()
# l => ['Gloire', 'à', 'qui', "n'ayant", 'pas', "d'idéal", 'sacro-saint',
'Se', 'borne' ...]
l1=s.split("o")
# => l1=['Gl', "ire à qui n'ayant pas d'idéal sacr", '-saint\nSe b',...]
l2=l1.split("nt")
# => l2=["Gloire à qui n'aya", " pas d'idéal sacro-sai", '\nSe bor..',...]
```

La méthode `join` recolle les chaînes stockées dans une liste en une seule liste :

```
lst=['Et','gloire','à','don','Juan']
s=" ".join(lst)
# s='Et gloire à don Juan'
s1="-".join(lst)
# s1='Et-gloire-à-don-Juan'
```

On peut trouver curieux que `join` soit une méthode qui s'applique à la chaîne séparateur, et non pas à la liste. C'est ainsi.

Trier une liste

Trier une liste, en la modifiant (c'est l'objet liste qui est modifié (tri sur place))

```
l=[1,4,2,3,5]
l.sort()
# => l = [1, 2, 3, 4, 5]
```

On peut trier et obtenir un **nouvelle** liste, sans modifier la liste d'origine (ça fonctionne donc aussi avec un tuple) :

```
l=[1,4,2,3,5]
l1=sorted(l)
# => l1 = [1, 2, 3, 4, 5]
# => l  = [1, 4, 2, 3, 5]
```

Si les éléments de la liste ne sont pas des nombres, on peut trier aussi, si les objets sont comparables. Les séquences sont généralement triés en prenant comme clé leur premier élément :

```
l=[(1,2),(5,2),(2,8)]
l1=sorted(l)
# => l1=[(1, 2), (2, 8), (5, 2)]
```

Des éléments complémentaires sont disponibles ici : howto/sorting.html

Les motifs les plus courants

Un programme fait souvent intervenir des séquences plus ou moins habituelles de plusieurs instructions. C'est ce que nous appelon ici des *motifs*. Nous allons en détailler quelques uns.

Dans ce qui suit, les noms indiqués entre <...> sont à remplacer par des fonctions, des variables, ou des valeurs, selon le contexte. Il s'agit des parties *variables* des motifs.

Accumulateur

```
<acc>=<val>
for <element> in <sequence> :
  <acc>=<acc> <operation> <element>
```

Exemple d'application du motif accumulateur : calculer la somme des éléments d'un tuple (en Python, la fonction `sum` fait déjà ce travail).

```
s=0
for e in (1,3,5,7,2,5,4) :
    s=s+e
```

Adaptez le motif au calcul du produit des éléments d'une liste.

La fonction `reduce` du module `functools` permet d'appliquer ce motif encore plus simplement en Python.

```
functools.reduce(fonction, sequence[, initial]) -> value
```

La `fonction` remplace `<operation>`, et initial, la valeur `<val>` :

```
def somme(a,b) : return a+b
s=functools.reduce(somme,(1,3,5,7,2,5,4),0)
```

Il est possible de se passer de la définition prélable de la fonction `somme` en utilisant la directive `lambda`.

```
s=functools.reduce(lambda x,y : x+y,(1,3,5,7,2,5,4),0)
```

Motif Essais-Erreurs à répétition

```
fin=False
while not fin :
    <var>=input("...")
    if <test de fin sur var> :
        fin=True
    elif <test1 sur var> :
        <intructions>
    elif <test2 dur var> :
        <instructions>
    else :
        <instructions>
```

```
<fin du motif>
```

Un exemple d'utilisation de ce motif est le jeu “plus petit ou plus grand” :

```
import random
r=random.randint(0,100)
fin=Fralse
while not fin :
    v=int(input('Entrez une valeur entre 0 et 100'))
    if v==r or v<0 or v>100 :
        fin=True
    elif v<r :
        print('Valeur trop faible...')
    else :
        print('Valeur trop élevée')
if v==r :
    print('Bravo, vous avez trouvé')
else :
    print('Je ne joue pas avec les tricheurs...')
```

Générateurs, Listes en intension

Rechercher à *generators* et *comprehension lists* dans les documentations en anglais.

Attention, certains comportements fins ont été modifiés entre des versions 2.X et 3.X de Python.

Une écriture particulièrement efficace et concise permet de traduire en Python des idées comme : la liste des carrés des nombres entiers inférieurs à 100 divisibles par 3.

Cette idée en contient en fait 3 : **l'ensemble de départ** (entiers inférieurs à 100), qu'on **filtre** (*filter*) en ne retenant que ceux divisibles par 3, et auquels on **applique une fonction** (*map*), l'élévation au carré.

Voici les 3 étapes de construction en Python :

```
# l génère les entiers de 0 à 99
l=(x for x in range(100))
# l génère les entiers de 0 à 99 divisibles par 3
l=(x for x in range(100) if x%3==0)
# l genère la liste des carrés des entiers de 0 à 99
# divisibles par 3
l=(x**2 for x in range(100) if x%3==0)
```

Selon qu'on entoure l'expression de parenthèses ou de crochets, on a respectivement affaire à un générateur ou à une liste. La différence est qu'un générateur **retarde au maximum** l'évaluation effective des valeurs. Le choix entre l'utilisation d'un générateur ou d'une liste en intension est affaire d'efficacité en temps et en consommation mémoire.

Erreurs courantes

Tuples de listes

Un tuple n'est pas modifiable. Cependant, il peut contenir comme élément une liste. Si l'élément du tuple désignera toujours la même liste (puisque le tuple n'est pas modifiable), le contenu de cette liste peut en revanche changer :

```
l=(3,4,[1,2,3])
l[2].append(4)
print(l)
# Mais ceci ne marche pas :
l[2]=[1,2,3,4]
```

Épuiser un générateur

Épuiser un générateur (fourni par `zip` par exemple) dans une boucle, et réutiliser le même générateur pour une autre boucle :

```
l1=['tomates','citrouilles','lauriers']
l2=['2be3','M. Pokora','A. Green']

assoc=zip(l1,l2)
for i in assoc :
  print('Des',i[0],'pour',i[1])

print("J'ai exprimé un avis très personnel sur :")
for i in assoc :
  print(i[1])
```

Dans l'exemple qui précède, la seconde boucle ne produira rien puisque le générateur est épuisé. Pour corriger, il suffit de refaire : `assoc=zip(l1,l2)` avant la seconde boucle

Évaluation des paramètres par défaut une seule fois

Le mieux est de donner un exemple :

```
class MonTest :
  def __init__(self,l=[]) :
    self.liste=l
  def ajoute(self,v) :
    self.liste.append(v)
  def affiche(self) :
    print(self.liste)
```

Le constructeur crée l'attribut `liste` à partir du paramètre `l`. Ce dernier a par défaut la valeur d'une

liste vide, ce qui fait qu'on peut appeler le constructeur ainsi :

```
a=MonTest()
```

La méthode `ajoute` rajoute un élément dans l'attribut `liste` et `affiche` imprime le contenu de cet attribut.

Voici un exemple d'utilisation :

```
a=MonTest()
b=MonTest()
b.affiche()
=> []
a.ajoute(4)
a.affiche()
=> [4]
b.affiche(4)
=> [4]
```

Que s'est-il passé ? Les paramètres par défaut étant évalués à la création de la classe, cette ligne :

```
def __init__(self,l=[])
```

revient à créer une liste vide, ayant pour `id`, par exemple 32456 et à dire que cet objet est le paramètre par défaut.

Donc, si on ne passe pas de paramètre au constructeur, l'attribut `liste` sera une référence ver **cette liste** précisément (qui est vide au début). Tous les objets créés de cette façon auront donc pour attribut `liste` la **même** liste vide. Si on modifie cette liste pour un des objets, elle est modifiée pour tous.

Discussions techniques

Les tranches et les vues

Utiliser une tranche de liste ne crée par une vue sur la liste, mais une nouvelle liste. Savoir ceci évite d'utiliser des opérations qui peuvent être coûteuses en mémoire.

Supposons qu'on souhaite, comme dans le tri par sélection, rechercher le plus petit élément d'un tableau privé de ses 3 premiers éléments (ici on prend un tableau rempli des entiers dans l'ordre pour simplifier) :

```
l=list(range(2500000))
min(l)
min(l[3:])
```

La dernière ligne est particulièrement inefficace, car elle provoque la création d'une seconde liste de 24999997 éléments, puis recherche le min, puis libère la liste (garbage collector). On peut constater

cette activité en surveillant l'espace mémoire utilisé par l'interpréteur.

Affectation, mode de passage des paramètres

Le modèle d'affectation en Python (toutes les variables sont des références) est un peu particulier. Si le comportement observé correspond généralement à celui des autres langages, une bonne compréhension des mécanismes ci-dessous permet d'éviter les surprises.

Modèle d'affectation

En Python, on peut se représenter une affectation du type :

```
a = <expression >
```

en imaginant que l'expression est évaluée, donnant naissance éventuellement à un nouvel objet. Puis, la variable a devient une nouvelle référence à cet objet.

Réfléchissez à ce que fait l'interpréteur lorsque vous entrez~:

```
a=6
b=a
b=b+1

t=[666,667,668]
m=t
m[1]=m[1]-1
```

Vous pouvez aussi consulter le document suivant : Le symbole d'affectation en Python

id et is

La fonction `id(obj)` indique l'identifiant de l'objet `obj`. Deux objets différents ont des identifiants différents. L'opérateur `is` compare ces identifiants : `a is b` vaut `True` si a et b ont le même identifiant c'est à dire si les variables a et b référencent **le même objet en mémoire**.

Testez le code suivant :

```
a=(5,6,7)
b=(5,6,7)
c=a
id(a),id(b),id(c)
```

Dans ce qui précède, il y a deux objets en mémoire (qui se trouvent avoir la même

valeur). Le premier est référencé par les variables c et a et le second est référencé par la variable b.

Selon les implémentations de l'interpréteur, Python peut *parfois* optimiser la place mémoire en utilisant deux fois le même objet en mémoire (s'il est non modifiable).

Attention, `is` et == sont différents (le premier implique le deuxième, mais l'inverse n'est psa vrai). Voici un exemple à enter dans le shell :

```
l1=[1,1,3,5,8]
l2=[1,1,3,5,8]
l3=l1
l4=[42]
l1,l2,l3,l4
l1==l2,l1 is l2
l1==l3, l1 is l3
l1==l4, l1 is l4
```

Passage des paramètres

En Python, le passage des paramètres peut être comparé à une affectation :

```
def fonction(a,b) :
  pass

c,d=5,6
fonction(c,d)
```

Dans le code qui précède, on peut *imaginer* que l'exécution de la fonction débute par :

```
a,b=c,d
```

On comprend ainsi que modifier a ou b dans la fonction ne modifiera pas le contenu des variables c et d dans le programme principal (on parle de passage par valeur)

Ce comportement sera donc différent, si c'est une liste qui est passée en paramètre (on parle de passage par référence)

La remarque suivante vaut pour tous les langages :
Si dans une fonction, on change le contenu d'une variable passée par valeur, l'original ne change pas. Si on revanche, on modifie l'objet désigné par une variable passée par référence, alors l'original sera modifié.

En Python, on ne parle pas de passage par valeur ou par référence. Tous les objets sont

passés par *affectation*. Mais certains étant non modifiable, tout se passe comme s'ils étaient passés par valeur. Dans le cas des objets modifiables (listes), c'est l'usage qu'on en fait dans la fonction qui déterminera si l'objet utilisé lors de l'appel est modifié ou non.

Pour faire un parallèle avec le C, lors du passage en paramètre :

- d'un argument de type *non modifiable*, le comportement observable est le même que lors d'un passage par valeur en C.
- d'un argument de type *modifiable*, le comportement observable est celui d'un passage par référence (attention à ne pas réaffecter la variable utilisée en paramètre....)

Vous pouvez tester en Python le code qui suit :

```
def suivant(a) :
    a=a+1

def tabsuivant(t) :
    for i in range(len(t)) :
        t[i]=t[i]+1

b=5
suivant(b)
print(b)
m=[1,2,3,4]
tabsuivant(m)
print(m)
```

- Mini diaporama illustrant le déroulement du programme précédent

Voici un programe tiré de l'ouvrage *Python Programming Fundamentals* :

```
def reverseInPlace(lst) :
    for i in range(len(lst)//2) :
        tmp=lst[i]
        lst[i]=lst[len(lst)-1-i]
        lst[len(lst)-1-i]=tmp

s=input("Please enter a sentence : ")
lst=s.split()
reverseInPlace(lst)
print("The sentence backward is : ",end="")
for word in lst :
    print(word,end=" ")
print()
```

Peut-on utiliser la procedure `reverseInPlace` ainsi :

```
print(reverseInPlace([1,2,3,4]))
```

Expliquez.

Peut-on utiliser cette fonction pour retourner une chaîne de caractère plutôt qu'une liste ? Expliquez et proposez une solution.

Quelles est la différence entre les deux programmes suivants ? Expliquez.

```
def symetrique(lst) :
    for i in reversed(range(len(lst))) :
        lst.append(lst[i])

s=[1,2,3,4]
symetrique(s)
print(s)
```

```
def symetrique(lst) :
    for v in reversed(lst) :
        lst=lst+[v]

s=[1,2,3,4]
symetrique(s)
print(s)
```

Il y a une subtilité supplémentaire. Si plutôt que `lst=lst+[v]`, vous écrivez `lst+=[v]`, Python ne **recréera pas une liste**. Vous aurez un comportement similaire à celui d'append

Types créés par l'utilisateur

Si le programmeur crée lui même une séquence, un type mapping etc., ce nouveau type peut ou non avoir les propriétés habituelles en Python.

Une séquence *custom* est toujours itérable, même si on n'écrit pas de méthode `__iter__`. Il suffit d'écrire `__getitem__` (c'est la méthode spéciale appelée lors de l'emploi de `[…]`) et le caractère ordonné suffit :

```
class SeqPuiss() :
    def __init__(self,n) :
        self.n=n
    def __getitem__(self,i) :
        if i>100 : raise StopIteration
        return i**self.n
```

```
sq=SeqPuiss(3)
for c in sq : print(c)
```

Voici un exemple de classe itérable. Le fait que la méthode `__iter__` soit implantée signifie que la classe est *itérable*, c'est à dire peut renvoyer un itérateur (qui se trouve être elle même). Le fait que la méthode `__next__` soit implantée signifie que la classe est un itérateur. Ce double rôle (itérable, itérateur) n'est pas facile à comprendre. Prenez un peu de temps pour étudier le programme d'exemple.

```
class TestIterable() :
    def __init__(self,liste) :
        self.l=list(liste)

    def __iter__(self) :
        print("Dans iter")
        self.i=0
        return self

    def __next__(self) :
        if self.i>=len(self.l) : raise StopIteration
        print("Dans next")
        v=self.l[self.i]
        self.i+=1
        return v


ti=TestIterable([1,3,5,7])
for v in ti :
    print("Read : ",v)
```

Dans ce qui précède, l'exécution de la ligne `for v in ti` provoque l'appel de la méthode `__iter__` et crée un itérateur (qui n'est pas nommé dans la boucle `for`). Puis, à chaque tour de boucle, c'est la méthode `__next__` sur l'itérateur créé qui est appelée. La boucle pourrait être simulée ainsi :

```
ti=TestIterable([1,3,5,7])
it=iter(ti)
try :
    while True :
        v=next(it)
        print("Read : ",v)
except StopIteration : pass
```

Exceptions

Python possède un système d'exception. Nous montrons ici commment intercepter une exception, et

comment en lever une.

Intercepter une exception

Exemple :

```
try :
    f=open("fichier","r")
except IOError(e) :
    print("Erreur d'entrée sortie : ",e)
```

On peut avoir plusieurs blocs `except`, ainsi qu'un bloc `else` exécuté si aucun bloc `except` ne l'est, ainsi qu'un bloc `finally` exécuté quoi qu'il arrive à la fin du bloc `try`.

Lever une exception

Il est possible de lever une exception d'un type préexistant, ou de créer son propre type.

On lève un exception avec le mot clé `raise` :

```
raise ValueError('Valeur attendue plus faible')
```

La création d'un nouveau type d'Exception est immédiat :

```
class MonException(Exception) : pass

raise MonException('Mon message')
```

Mon avis est qu'il faut, si possible utiliser les types d'exception déjà prévus plutôt que systématiquement créer un nouveau type. La liste des types d'exceptions disponibles est ici : Hiérarchie des exceptions

Voici les plus courantes :

- `IndexError` indice d'une séquence non valide (en dehors des limites)
- `KeyError` équivalent de `IndexError` pour les dictionnaires
- `TypeError` type incorrect
- `ValueError` valeur incorrecte (mais le type est correct)
- `NotImplementedError` fonction non (encore) implémentée (à utiliser durant la phase de développement)
- `RuntimeError` autres types d'erreur
- `IOError` erreur d'entrée/sortie

Mot clé ''lambda''

Le mot réservé `lambda` permet decréer des fonctions *à la volée*, dans la limite d'une expression. Par exemple, ceci :

```
lambda x: x**2
```

est la fonction qui prend un paramètre et renvoie son carré. Ainsi, on peut écrire :

```
carre=lambda x : x**2
carre(5)
=> 25
carre(9)
=> 81
```

Les expressions peuvent être plus compliquées. Voici une autre façon de programme la suite de Syracuse, qui utilise le *pseudo* opérateur ternaire de python : `a if b>c else d` (vaut a si b>c et d sinon) :

```
suivant=lambda x: x//2 if x%2==0 else 3*x+1
tous=lambda u,n : [] if n==0 else [u]+tous(suivant(u),n-1)
```

Voici un autre exemple, tiré de [1] :

```
import math
def composition(f,g) :
    return lambda x: f(g(x))

f=composition(math.sin,math.cos)
f(3)  # Vaut sin(cos(3))
=> -0.83602
```

Modules complémentaires

NumPy

NumPy est une extension à Python. Il faut l'installer en plus du langage de base pour pouvoir l'utiliser.

```
import numpy as np

m1=np.array([[1, 2], [3, 4]])
m2=np.zeros((10,10))
m2[4][5]=3
m2[4,6]=1
```

Matplotlib

Example de code Matplotlib

```
import matplotlib.pyplot as plt
import math
```

```
t=range(1000)
y=[math.sin(x*3.14/180) for x in t]
plt.plot(y)
plt.show()
#plt.savefig('fig.png') # Au choix show ou savefig...
```

(ns)FAQ

Créer un exécutable autonome

Voici la marche à suivre pour Windows. Le même principe est applicable sous MacOS ou Linux.

Il existe plusieurs méthodes ayant pour objectif de permettre la distribution d'un programme Python sur des machines ne possédant pas l'interpréteur.

Parmi ces méthodes, nous avons retenu celle utilisant le module `cx_freeze`. Ce module est téléchargeable pour différentes plates-formes ici : http://cx-freeze.sourceforge.net/. Il est installé par défaut dans `pyzo`.

Le script de démarrage, `cxfreeze`, se trouve normalement dans le dossier `Script` de l'installation Python.

Pour utiliser le script, on se place dans le dossier contenant le programme python :

```
C:\...> Z:\
Z:\> dir
tictactoe.py
```

Puis on entre [4)]

```
Z:\> L:\Pyzo\Scripts\cxfreeze tictactoe.py --target-dir tttdir
```

Le répertoire `'tttdir'` sera créé, il contiendra un `.exe` ainsi que les DLL nécessaires à son exécution.

Utiliser l'éditeur Scite

Par défaut Scite ne sait pas où se trouve l'interpréteur Python sur les machines à disposition. Il est possible de configurer ceci localement, en mettant **dans le répertoire de travail** un fichier `SciTE.properties` contenant ceci :

SciTE.properties

```
command.go.*.py=c:\python32\pythonw -u "$(FileNameExt)"
```

Trouver le répertoire courant depuis un shell

Pour savoir où vont aller se placer les fichiers créés par exemple, il faut connaître le répertoire courant. Dans un shell, entrez :

```
import os
os.getcwd()
```

Changer de répertoire et lister les fichiers

Dans un shell, entrez :

```
import os
os.chdir('Z:/monrep')
os.listdir()
```

Modifier la limite du nombre d'appels récursifs

La limite est 1000 par défaut.

```
import sys
sys.setrecursionlimit(100000)
```

Inclure des tests dans les docstrings

Le module `doctest` permet d'inclure simplement des tests dans la documentation. Ces tests peuvent être réalisés à la demande, permettant ainsi au développeur de s'assurer que sa classe, sa fonction ou son module reste conforme au cahier des charges. Il suffit d'inclure dans la doctring les tests à faire et les réponses attendues.

```
from doctest import testmod

def parfait(n) :
    """ Détermine si un nombre est parfait
    >>> parfait(100)
    False
    >>> parfait(28)
    True
    """
    return n==sum(i for i in range(1,n) if n%i==0)

testmod()
```

Si le programme est silencieux, c'est que les tests se sont bien déroulés (on peut aussi rendre les tests plus verbeux en passant l'option `-v` à l'interpréteur.

Liens

- Documentation Python : Built-in functions
- Documentation Python : Types standards
- Doc NumPy/Scipy au LaBRI
- Le symbole d'affectation en Python

1)

`sys.float_info` contient des informations sur le codage utilisé.

2)

Efficaces... car les algorithmes sous-jacents utilisés par les concepteurs de l'interpréteur sont généralement bien choisis.

3)

en fait, l'opérateur `del` *libère* une référence

4)

vous aurez peut être des chemins différents...