0 Algorithmique

I – C'est quoi un ordinateur ? Et un algorithme ?

1 – Un ordinateur, qu'est-ce que c'est ?

2 – Un algorithme, qu'est-ce que c'est ?

Un **algorithme** est une suite finie et non ambiguë d'instructions qui permettent de résoudre un problème, *i.e.* d'obtenir un résultat en sortie à partir de données fournies en entrée. Par exemple, une recette de cuisine est un algorithme permettant d'obtenir un plat à partir de ses ingrédients !

Un **programme informatique** est la rédaction d'un algorithme dans un **langage de programmation** de sorte que celui-ci soit compréhensible et exécutable par l'ordinateur. Il existe une foultitude de langages différents, de plus ou moins *haut-niveau*, *compilés* ou *interprétés*.

Python est un langage dit *haut-niveau*, c'est-à-dire qu'il permet d'écrire des programmes facilement, à l'aide de mots et de symboles mathématiques usuels, et interprété, c'est-à-dire qu'il ne demande pas une étape de *compilation* supplémentaire avant exécution. Il est inventé en 1991 par Guido van Rossum et ainsi nommé en hommage aux *Monty Python*.

II – Découverte de Python

Avant de rentrer dans les détails, voici un opérateur et une fonction qui vont permettre d'illustrer les exemples à venir :

- l'opérateur *sharp* # (*a.k.a.* "*dièse*" mais à aucun moment "*hashtag*") qui permet de commenter du code : tout ce qui se situe après le #, jusqu'à la fin de la ligne, ne sera pas interprété par l'ordinateur.
- la fonction `print()` qui permet d'afficher à l'écran l'expression souhaitée.

1 – Types

Une expression est une suite de caractères qui définit une valeur. Pour connaître cette valeur, la machine doit évaluer l'expression. Voici quelques exemples numériques :

```
>>> 2+3
5
>>> 3.2+4
7.2
>>> 7/2
3.5
>>> 9//4*2.5
5.0
```

Chaque valeur possède un certain type : par exemple *entier, flottant, booléen, chaîne de caractères, liste.* Ce type permet à Python de savoir quoi lire dans la mémoire et quoi faire avec cette valeur lors de l'exécution. Pour connaître le type d'une expression après évaluation, il suffit de le demander à Python à l'aide de la commande `type()`.

```
>>> type(2+3)
<class 'int'>
>>> type(3.2+4)
<class 'float'>
>>> type(4<3)
<class 'bool'>
>>> type("un deux trois")
<class 'str'>
>>> type([1,2,3])
<class 'list'>
```

Les entiers

Les entiers sont les mêmes en informatique qu'en mathématiques. Ils sont de type `int`, pour *integer*. À la différence d'autres langages, Python travaille avec des entiers de tailles arbitraires, ce qui permet de faire du calcul exact avec des entiers pourtant gigantesques.

```
>>> 47**47
3877924263464448622666648186154330754898344901344205917642325627886496385062863
>>> ((((((2**2)**2)**2)**2)**2)**2)**2
115792089237316195423570985008687907853269984665640564039457584007913129639936
>>> 2**256
115792089237316195423570985008687907853269984665640564039457584007913129639936
```

Le tableau suivant détaille le fonctionnement des opérateurs avec des nombres entiers :

opérateur	opération
+	addition
-	soustraction
*	multiplication
/	division
//	division entière
%	modulo (reste de la division)
**	exponentiation (puissance)

Comme en mathématiques, les parenthèses sont de mises afin de bien prioriser les calculs.

```
>>> 7+31%3*5**2
32
>>> (7+31%(3*5))**2
64
```

Les flottants

Les flottants sont les représentations informatiques des réels. Ils sont de type `float`, pour *floating point number*. En raison d'un espace mémoire limité (1 bit de signe, 11 bits d'exposant et 52 bits de mantisse), seul un nombre fini de réels sont représentables en mémoire, ce qui ne permet pas de faire des calculs exacts.

Le tableau suivant détaille le fonctionnement des opérateurs avec des nombres flottants :

opérateur	opération
+	addition
-	soustraction
*	multiplication
/	division
**	exponentiation (puissance)

On peut remarquer que les opérateurs sont les mêmes que pour les entiers. Lorsque l'on utilise l'un de ces opérateurs avec des entiers et des flottants, les entiers sont automatiquement convertis en flottants (on pourrait forcer la conversion de l'entier `n` en flottant avec `float(n)`). C'est le cas également pour la division flottante utilisée avec des entiers.

```
>>> 5*2.3
11.5
>>> 3/5
0.6
```

Les booléens

Les booléens sont essentiels en informatique. Ils sont de type `bool`, pour *boolean*, et sont appelés ainsi en référence à George Boole (1854). Ce type ne comprend que deux constantes : `True` et `False` (Vrai et Faux).

Le tableau suivant détaille le fonctionnement des opérateurs avec des booléens. Il n'y en a que trois, qui correspondent au *non* logique (unaire) ainsi qu'au *et* et *ou* logiques (binaires) :

`not a` :

a	not a
True	False
False	True

`a and b` :

a \ b	True	False
True	True	False
False	False	False

`a or b` :

a \ b	True	False
True	True	True
False	True	False

En informatique, le *ou* est toujours inclusif : si `a` et `b` sont vrais, alors `a` *ou* `b` est vrai aussi. L'utilité des booléens est qu'ils sont les résultats des opérateurs de comparaisons, éléments primordiaux des structures de contrôles.

Le tableau suivant détaille le fonctionnement des opérateurs de comparaisons :

opérateur	comparaison
==	égal
!=	non égal
<	inférieur strict
<=	inférieur ou égal
>	supérieur strict
>=	supérieur ou égal

```
>>> 3<1
False
>>> 4==2*2
True
>>> not (3==2)
True
>>> (4==2*2) and (not 3==2)
True
>>> (3<1) or (4==2*2)
True
>>> (3<1) or (4==5)
False
```

Dans une expression de la forme `a and b`, où `a` et `b` sont des expressions, si `a` s'évalue en `False`, on n'a pas besoin d'évaluer `b` pour s'apercevoir que `a and b` s'évalue en `False`. De même avec `a or b` si `a` s'évalue en `True`. Python respecte cette logique : si la partie gauche suffit à déterminer si l'expression s'évalue en `True` ou `False`, il n'évalue pas la partie droite (on parle du caractère paresseux des opérateurs).

C'est particulièrement utile lors de l'évaluation d'une expression dont la seconde partie pourrait produire une erreur, mais dont la première sert de garde-fou : `x>0 and sqrt(x)>2` ne produit pas d'erreur, même si `x` est un nombre négatif. En effet, dans ce cas `x>0` s'évalue en `False` et on n'a pas besoin d'évaluer `sqrt(x)>2` qui produirait une erreur, la fonction `sqrt` n'étant pas définie pour des nombres négatifs.

Les listes

Les listes sont des séquences finies d'éléments. Elles sont de type `list`. Elles se construisent par la donnée, entre crochets, d'éléments séparés par des virgules. On verra un exemple très particulier de listes au moment d'aborder les boucles.

Les chaînes de caractères

Les chaînes de caractères sont des suites de caractères quelconques. Elles sont de type `str`, pour *string*. Elles sont données soit 'entre apostrophes' ou entre "guillemets". L'ordinateur n'interprète alors plus la valeur de cette expression mais ne la retranscrit que comme une écriture sans sens algorithmique.

Il peut être utile d'insérer des chaînes de caractères au moment d'afficher des résultats à l'écran :

```
>>> "Hello " + 'World!'
'Hello World!'
>>> x=3+2
>>> print('la variable x vaut',x)
la variable x vaut 5
```

2 – Variables et affectation

En mathématiques, une **variable** est le nom que l'on donne à un élément d'un ensemble, qui peut prendre n'importe quelle valeur dans cet ensemble, mais qui n'en prend en réalité aucune. En informatique, la notion de variable est à l'opposé de cela :
Une **variable** est l'association d'un *identifiant* avec une *valeur*. Cette association s'appelle l'**affectation**.

Dans la mémoire physique, la création d'une variable assigne un identifiant à un emplacement de la mémoire. La valeur de la variable est alors stockée dans cette *case* de la mémoire.

L'identifiant est laissé au choix de l'utilisateur :

- Il est composé d'une suite de lettres et de chiffres mais doit débuter par une lettre.
- Il ne doit pas être un mot *réservé* du langage (par exemple : `if`, `for`, `print`, etc.).
- Il est sensible à la casse : `x` et `X` ne symbolisent pas la même lettre.

Une bonne idée est de choisir un identifiant concis mais explicite, en utilisant l'*underscore* _ au besoin (*a.k.a.* le "*tiret du 8*").

Contrairement à plusieurs autres langages de programmation, les variables n'ont pas besoin d'être déclarées en amont : la première affectation suffit à la déclaration. On remarque aussi qu'il n'est pas nécessaire de spécifier le type de celle-ci (Python s'en occupe seul) et qu'il est même possible que le type de la variable change.

Pour **affecter** une valeur à un identifiant, on utilise la syntaxe `variable=expression` :

```
>>> x=2+1   #on évalue 2+1, on obtient 3, qu'on affecte à x
>>> y=x**x+2   #ici x représente la valeur 3, 3**3+2 vaut 29, qu'on affecte à y
>>> print(y-1)   #y-1 s'évalue en 28, qu'on affiche
28
>>> x=y/2   #nouvelle affectation de x
>>> print(x)
14.5
```

III – Les structures de contrôle

Jusqu'ici toutes les séquences étudiées étaient *séquentielles* : elles se présentaient sous la forme d'expressions simples qui s'enchaînaient les unes après les autres. Il existe aussi des structures *conditionnelles*, pouvant mener à différentes exécutions en fonction de l'état des données, et des structures *itératives* permettant de répéter la même exécution.

Contrairement à beaucoup de langages, Python ne demandent pas d'accolades ou de balises `begin/end` autour des blocs d'instructions. En revanche, l'**indentation** est primordiale. Il s'agit de laisser une marge blanche devant un bloc d'instructions :

- Une séquence d'instructions est faite d'instructions écrites avec la même indentation.
- Dans une structure de contrôle (que l'on va voir tout de suite), une séquence d'instructions subordonnées doit avoir une indentation supérieure à celle de la séquence englobante. Toute ligne écrite avec la même indentation que cette séquence englobante marque la fin de la séquence subordonnée.

1 – Instructions conditionnelles

Le `if` (*si* en français) permet de mettre une condition à l'exécution d'instructions. La syntaxe Python est la suivante. On notera la présence obligatoire de l'indentation et le *deux-points* situé à la fin de la ligne d'en-tête qui contient le test à vérifier.

```
>>> if condition:
...     instruction(s)
```

Dans ce cas, si la condition est vérifiée, les instructions seront exécutées. Sinon, rien ne se passera.

Si l'on souhaite que quelque chose (de différent) se passe lorsque la condition n'est pas vérifiée, il suffit de rajouter une autre ligne d'en-tête, avec un `else` (*sinon* en français). Ici, pas de condition derrière puisqu'il s'agit du cas où la condition du `if` n'est pas vérifiée.

```
>>> if condition :
...     instruction(s)
... else:
...     autre(s)_instruction(s)
```

Dans le cas où il y a plus de trois issues possibles, on utilise `elif`, contraction de `else if` (*sinon si* en français). Finalement, la structure générale est donnée par :

```
>>> if condition:
...     [instructions effectuées si condition s'évalue en True]
... elif autre_condition:
...     [instructions eff. si condition est False et autre_condition est True]
... else:
...     [instructions effectuées si les deux conditions s'évaluent en False]
```

Chaque test (ce qui suit le `if` ou le `elif`) est une expression booléenne : l'évaluation doit fournir un *booléen*, `True` ou `False`.

Dans une telle structure conditionnelle, les expressions booléennes sont évaluées les unes après les autres, de haut en bas, jusqu'à ce que l'une d'entre elles s'évalue en `True`. Le bloc d'instructions correspondant (et seulement celui-ci) est alors exécuté, puis on sort de la structure conditionnelle. Le bloc correspondant au `else` est exécuté seulement si **toutes** les expressions conditionnelles situées au-dessus se sont évaluées en `False`. Il est possible de mettre plusieurs `elif`.

Voici un exemple, avec `note` une variable supposée contenir un flottant entre 0 et 20 :

```
>>> if note>=16:
...     print("Mention Très Bien")
... elif note>=14:
...     print("Mention Bien")
... elif note>=12:
...     print("Mention Assez Bien")
... elif note>=10:
...     print("Mention Passable")
... else:
...     print("Essaie encore !")
```

2 – Boucle inconditionnelle

Les boucles permettent d'exécuter plusieurs fois un même bloc d'instructions. En effet, on a souvent besoin qu'une variable prenne successivement comme valeur tous les entiers entre deux bornes.

Pour cela, Python utilise une fonction, `range`, qui va créer une liste d'entiers contenant toutes les valeurs que prendra la variable souhaitée. Mais ce n'est pas une `list` au sens de Python (c'est un `range`).

- Pour `m` et `n` deux entiers, `range(m,n)` renvoie tous les entiers entre `m` inclus et `n` exclus. En fait, les deux bornes sont "la première valeur qui est dedans" et "la première valeur qui n'est pas dedans".
- Pour un entier `n`, `range(n)` renvoie tous les entiers entre 0 inclus et `n` exclus. Cela fait les n termes entre 0 et `n-1` car oui, Python commence à compter à 0.
- Pour trois entiers `m`, `n` et `p`, alors `range(m,n,p)` renvoie tous les entiers entre `m` inclus et `n` exclus, en allant de `p` en `p`. Il s'agit donc de la liste `[m,m+p,m+2p,...]` sans jamais dépasser ou égaler `n`.

La boucle `for` (*pour* en français) permet de répéter un bloc d'instructions en modifiant à chaque itération la valeur d'une variable. La syntaxe Python est donnée ci-après. On notera une nouvelle fois la présence obligatoire de l'indentation et le *deux-points* situé à la fin de la ligne d'en-tête.

```
>>> for i in range(m,n):
...     instruction(s)
```

En vérité, l'utilisation de la fonction `range` n'est pas nécessaire à l'exécution d'une boucle `for` : on pourrait la remplacer par n'importe quelle liste ou *a fortiori,* n'importe quoi qui soit itérable.

```
>>> x=1
>>> n=7
>>> for i in range(1,n+1):
...     x=x*i
>>> x
5040
```

On vérifie que la variable `x` contient bien $7! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 = 5040$ à l'issue de cette boucle.

3 – Boucle conditionnelle

La boucle `while` (*tant que* en français) permet de réaliser une suite d'instructions tant qu'une certaine condition est vraie. La syntaxe Python est la suivante :

```
>>> while condition:
...     instruction(s)
```

Le mécanisme est le suivant : on évalue `condition`. Si le résultat est `True`, on effectue toutes les instructions du bloc indenté, puis on recommence l'évaluation de `condition`. Sinon, on passe aux instructions situées après la boucle. Par exemple, la séquence

```
>>> i=0
>>> while i<10:
>>>     print(i)
>>>     i=i+1
>>> print("Terminé")
```

affiche à l'écran tous les nombres entre 0 et 9, puis `"Terminé"`. En effet, lorsque `i` atteint 9, on l'affiche à l'écran, puis on incrémente `i` (qui vaut 10 en bas de la boucle). On réévalue ensuite la condition, mais `10<10` s'évalue en `False`, donc on sort de la boucle et on affiche `"Terminé"`, qui est une expression en dehors du corps de la boucle. On note que la condition est évaluée uniquement en haut de la boucle : si elle s'évalue en `True`, on effectue **toutes** les instructions du corps de boucle avant de recommencer l'évaluation.

Deux choses importantes sur les boucles `while` :

- Il se peut très bien qu'à la première évaluation de la condition, celle-ci soit `False` : dans ce cas on n'effectue jamais le corps de boucle.
- Enfin il faut être **très prudent** avec les boucles `while` : si on s'y prend mal, on crée un morceau de code qui boucle sans fin : c'est une boucle infinie et ça fait planter le logiciel.

IV – Fonctions

Les fonctions sont d'une importance capitale en informatique. Il s'agit d'écrire une séquence d'instructions, dépendant de paramètres donnés en entrée (les arguments) et renvoyant un résultat en sortie.

Deux points de vue, souvent complémentaires, permettent de préciser ce qu'est une fonction :

- Elle permet de réaliser un calcul précis, que l'on peut utiliser plusieurs fois.
- Elle est une brique de base d'un problème plus complexe.

La syntaxe Python est donnée ci-après. On notera une nouvelle fois la présence obligatoire de l'indentation et le *deux-points* situé à la fin de la ligne d'en-tête.

```
>>> def nom_fonction(arg_1,arg_2,...,arg_k):
...     instruction(s)
...     return resultat
```

- `def` est le *mot-clé* pour spécifier à Python que ce qui suit est la définition d'une fonction.
- `nom_fonction` est le nom que l'on utilisera par la suite pour appeler la fonction.
  Comme pour les variables, on utilise un nom concis mais clair, sans caractères spéciaux.
- Les arguments `arg_i` sont les données en entrée. Il peut ne pas y en avoir si la fonction n'a besoin de rien pour fonctionner.
- Le `return` marque la fin de la fonction et définit ce que renverra Python après l'exécution de la fonction.

Attention, le rôle d'une définition de fonction n'est pas d'exécuter les instructions qui en composent le corps, mais uniquement de mémoriser ces instructions en vue d'une exécution ultérieure. Il faudra faire appel à la fonction pour que les instructions soient exécutées.

Par exemple, définir la fonction qui suit ne provoque pas d'erreur.

```
>>> def fonction_erreur():
...     print(1/0)
```

Évidemment, l'appeler en provoquerait une ! Mais qui ferait ça ?

Voici la définition d'une fonction `disc` qui calcule le discriminant d'un polynôme de degré 2 en prenant en entrée les trois coefficients `a`, `b` et `c`.

```
>>> def disc(a,b,c):
...     delta=b**2-4*a*c
...     return delta
```

Maintenant que la fonction est stockée dans la mémoire, il suffit de faire un appel de cette fonction sur des données pour qu'elle renvoie le résultat souhaitée. Voici la syntaxe pour appeler cette fonction sur le polynôme $2x^2+7x-4$ (ici a=2, b=7 et c=-4).

```
>>> disc(2,7,-4):
81
```

Une erreur classique est de confondre `print` et `return` : `return` est une instruction de sortie de fonction, `print` est une fonction Python qui affiche l'argument passé en entrée à l'écran et qui ne renvoie rien. Lorsqu'on teste une fonction dans la console, on ne voit pas vraiment la différence mais elle est pourtant significative : une fonction sans `return` ne renvoie rien !

Trois choses à savoir concernant les variables utilisées dans les fonctions :

- les variables définies à l'intérieur d'une fonction n'existent qu'à l'intérieur de la fonction,
- les variables définies à l'extérieur d'une fonction sont utilisables à l'intérieur de la fonction si elles ont été déclarées avant la fonction,
- les variables définies à l'extérieur d'une fonction ne sont pas modifiables dans cette fonction.

On parle alors de **variable locale** pour parler des variables qui ne vont avoir de sens qu'à l'intérieur d'une fonction. C'est le cas par exemple de la variable `delta` dans la fonction `disc` décrite plus haut. À l'inverse, une **variable globale** est visible dans l'ensemble du programme.

V – Les bibliothèques

Afin de ne pas s'encombrer de choses inutiles, Python ne lance que les fonctions de bases au départ. Cependant, il possède aussi un catalogue extrêmement riche de bibliothèques (ou librairies ou modules ou packages) externes, à importer si nécessaire.

Il existe plusieurs façons d'importer une bibliothèque :

```
>>> import numpy
>>> numpy.sqrt(16)
4
>>> import numpy as np
>>> np.sqrt(25)
5
>>> from numpy import sqrt
>>> sqrt(36)
6
```

- Dans le premier cas, on importe toute la bibliothèque `numpy`. Pour utiliser une des fonctions de la bibliothèque, on doit débuter le nom de la fonction par "`numpy.`".
- Dans le deuxième cas, on importe toute la bibliothèque `numpy` en utilisant l'*alias* `np`. Pour utiliser une des fonctions de la bibliothèque, on doit débuter le nom de la fonction par l'alias "`np.`". C'est très utile pour des bibliothèques dont les noms sont parfois longs.
- Dans le troisième cas, on n'importe de toute la bibliothèque `numpy` **QUE** la fonction `sqrt`.

On présente désormais deux bibliothèques importantes pour le programme d'ECT.

1 – La bibliothèque `numpy`

La force principale de la bibliothèque `numpy` est de travailler avec des tableaux de toutes dimensions. Ces tableaux de nombres, en dimension 2, sont la meilleure façon de représenter les *matrices*, objets clés du programme de deuxième année.

En plus de représenter les matrices (et les opérations sur les matrices), la bibliothèque `numpy` contient aussi toutes les constantes et fonctions mathématiques usuelles :

Constantes

`np.pi`	$\pi \approx 3.14159$
`np.e`	$e = \exp(1) \approx 2.718$

Fonctions

`np.sqrt()`	fonction racine carrée
`np.abs()`	fonction valeur absolue
`np.floor()`	fonction partie entière
`np.log()`	fonction logarithme népérien
`np.exp()`	fonction exponentielle

Attention, le `e` classique n'est pas le e mathématique : il s'agit d'une notation pour exprimer une puissance de 10.

```
>>> import numpy
>>> numpy.e
2.718281828459045
>>> 3e2   # 3*10**2
300.0
```

Enfin une dernière fonction, utile pour le tracé d'une courbe : `np.linspace()`.
Pour deux valeurs `a` et `b` et un entier `n`, `np.linspace(a,b,n)` fournit un tableau (au sens de `numpy`, mais ce n'est qu'une liste de nombres) qui contient un découpage régulier de l'intervalle $[a, b]$ en n valeurs.
Ce seront les abscisses des points dont Python se servira afin de tracer la courbe.

2 – La bibliothèque `matplotlib`

Pour cette bibliothèque, l'usage d'un *alias* est recommandé : `import matplotlib.pyplot as plt`.
Comme précisé précédemment, Python procède par *interpolation*, c'est-à-dire qu'il place des points puis les relie par des segments. Mais s'il y a suffisamment de points, les segments sont si petits qu'ils en deviennent imperceptibles.

Les deux fonctions principales sont les suivantes :

- `plt.plot(X,Y)`, où `X` est le tableau des abscisses et `Y` celui des ordonnées correspondantes.
  Si le `X` se construit facilement avec `np.linspace()`, pour le `Y` il suffit d'appliquer une fonction à `X` et Python comprend de lui-même qu'il faut construire un tableau de même taille, où chaque coefficient est l'image par la fonction du coefficient de `X`.
- `plt.show()` est la fonction qui affiche le graphique. En effet, la fonction précédente n'affiche pas la courbe mais crée l'objet dans la mémoire correspondant au graphique. Ici, le graphique s'affiche sur une nouvelle fenêtre.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> X=np.linspace(-5,5,500)   #les abscisses de 500 points entre -5 et 5
>>> Y=np.abs(X)   #les ordonnées des 500 points
>>> plt.plot(X,Y)   #Python crée le graphique
>>> plt.show()   #Python affiche le graphique
```