# Un langage pour enseigner la programmation, Scheme ou Python ?

Laurent Bloch [1]

*Le texte qui suit est issu d'une séance du séminaire* Codes Sources [5]. *Comme de juste il présentera des codes sources de différents programmes informatiques, mais aussi des codes sources de protéines et d'acides nucléiques, puisqu'aussi bien la cellule est un ordinateur (distribué) dont l'unité centrale (le ribosome) exécute les programmes de la vie, écrits selon le code génétique, dont nous donnerons aussi le source.*

## Au cœur de la question

« *La programmation fonctionnelle est un assaut radical et élégant contre toute la pratique de l'écriture de programmes. Elle s'écarte totalement de la mentalité du programmeur qui* fait ci et puis après fait ça ». Vous devez recâbler votre cerveau d'une façon assez différente [2] ».

## Faire ci et puis après faire cela

C'est la tendance spontanée du programmeur débutant, qui ne sait pas très bien par quel bout aborder le problème qu'il doit programmer. Elle est encouragée par le style de beaucoup de langages, surtout les langages de script en mode interprété. En général, cela finit par des séquences d'instructions pas forcément très cohérentes

1. https://laurentbloch.net.

2. Citation traduite de Simon Peyton-Jones, https://www.cs.cmu.edu/~popl-interviews/peytonjones.html

DOI:10.48556/SIF.1024.20.85 

mais qui donnent des résutats parce qu'une heure avant, dans la boucle d'interaction de l'interpréteur, on aura donné aux variables impliquées dans le calcul des valeurs, on ne sait plus très bien lesquelles, comme le programme de gauche ci-après et, après une (courte) réflexion, comme celui de droite :

```
#!/usr/bin/env python3
x = 5
y = 0

if x != 0:
    y = x + x**2
    y = (y - 3) // x
    if y != 0:
        y = y % x
        print (x, y)
```

```
#!/usr/bin/env python3
# coding: utf-8

x = int(input("Entrez la valeur de x : "))
y = int(input("Entrez la valeur de y : "))
if x != 0:
    y = x + x**2
    y = (y - 3) // x
    if y != 0:
        y = y % x
        print (x, y)
```

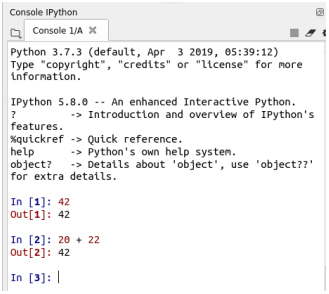
Les enseignants de l'option NSI des lycées n'en peuvent plus de ces programmes monolithiques à base d'`input` et de `print`. Pour un exposé raisonnable de l'enseignement de Python à des débutants, on pourra consulter le livre de Benjamin Wack et de ses coauteurs [6].

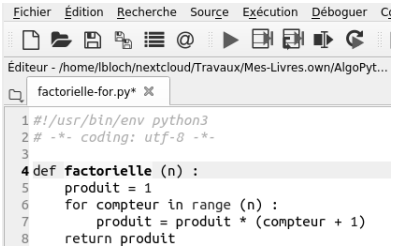
## Pour construire un programme il faut...

Corrado Böhm et Giuseppe Jacopini [3] ont montré en 1966 que tout algorithme pouvait être programmé avec trois constructions : séquence d'instructions, alternative ou répétitive. Leur article est une des deux références de celui de Dijkstra [4], *Go to Statement Considered Harmful* (1968), l'autre est un article de Niklaus Wirth et d'Anthony Hoare sur le développement du langage Algol. L'art de la programmation réside pour une bonne part dans l'agencement harmonieux, agréable à lire et compréhensible de ces trois constructions, organisées pour constituer des fonctions.

## Environnement de programmation

Ci-dessous quelques images d'un environnement de développement Python assez répandu, mais vous pouvez utiliser Emacs !



## Biopython

Biopython est une bibliothèque de courts programmes Python (des *scripts*) qui permettent d'enchaîner de façon cohérente les programmes classiques de la bioinformatique (la plupart écrits en C) en les appliquant aux données du biologiste et aux banques de données du domaine. Biopython assure les conversions de format entre banques et fichiers et leur adaptation aux divers logiciels. C'est pour ce genre d'opérations que Python a été conçu ; il les effectue très bien et on se trompe en essayant de lui faire faire autre chose. Le programme qui suit, provenant du portail ExPASy de l'institut suisse de bioinformatique, correspond à la recherche de la séquence de la protéine identifiée par le numéro `O23729` (*Accession Number*). En général, les séquences biologiques sont identifiées sans ambiguïté par cet identifiant. Le lecteur trouvera le texte complet des programmes Biopython sur le site de l'auteur [1].

```
#!/usr/bin/env python3

from Bio import ExPASy
from Bio import SeqIO
with ExPASy.get_sprot_raw("O23729") as handle:
    seq_record = SeqIO.read(handle, "swiss")
    print(seq_record.id)
    print(seq_record.name)
    print(seq_record.description)
    print(repr(seq_record.seq))
    print("Length %i" % len(seq_record))
    print(seq_record.annotations["keywords"])
```

avec comme résultat d'exécution :

```
O23729
CHS3_BROFI
RecName: Full=Chalcone synthase 3; EC=2.3.1.74; AltName:
                        Full=Naringenin-chalcone synthase 3;
Seq('MAPAMEEIRQAQRAEGPAAVLAIGT...GAE', ProteinAlphabet())
Length 394
['Acyltransferase', 'Flavonoid biosynthesis', 'Transferase']
```

Biopython est un outil formidable, qui permet au biologiste d'enchaîner rapidement et facilement les analyses qu'il a l'habitude d'effectuer en routine. Mais aligner des lignes de Biopython n'est pas faire de la bioinformatique, c'est utiliser un outil sans vraiment comprendre ce qu'il fait, au risque d'être abusé par des résultats fallacieux. Disons que c'est un travail de technicien, et qu'un chercheur, à tout moment, peut être (doit être) son propre technicien.

### *Recherche de séquences*

On peut essayer d'organiser un peu mieux ce travail en créant des fonctions qui pourront être réutilisées :

```
#!/usr/bin/env python3
def FetchSeqMulti(mydb, myrettype, myretmode, myid):
    from Bio import Entrez
    from Bio import SeqIO
    Entrez.email = "lb@laurentbloch.org"
    with Entrez.efetch(db=mydb, rettype=myrettype,\
                       retmode=myretmode, id=myid) as handle:
        for seq_record in SeqIO.parse(handle, "gb"):
            print("%s %s..." % (seq_record.id,\
                                seq_record.description[:50]))
            print("Sequence length %i, %i features, from: %s"
                  % (len(seq_record), len(seq_record.features),\
                     seq_record.annotations["source"]))

            FetchSeqMulti("nucleotide", "gb", "text",\
                          "6273291,6273290,6273289")
AF191665.1 Opuntia marenae rpl16 gene; chloroplast gene for c...
Sequence length 902, 3 features, from: chloroplast Grusonia marenae
AF191664.1 Opuntia clavata rpl16 gene; chloroplast gene for c...
Sequence length 899, 3 features, from: chloroplast Grusonia clavata
AF191663.1 Opuntia bradtiana rpl16 gene; chloroplast gene for...
Sequence length 899, 3 features, from: chloroplast Grusonia bradtiana
```

### *Code génétique*

Dans la colonne de gauche du tableau ci-dessous, on trouve le nom des acides aminés du monde vivant et dans la colonne de droite, les motifs de trois acides ribonucléiques (ARN, un brin d'ARN est transcrit depuis un brin d'ADN, nucléotide pour nucléotide, en substituant un U (pour uracile) à un T (thymine)[3]) qui codent pour chacun (on note que ce code est redondant, ce qui procure des distractions aux bioinformaticiens). Dans la seconde colonne, le caractère qui code chaque acide aminé dans les banques de données (et dans les programmes qui les exploitent).

| | | | |
|---|---|---|---|
| Alanine | A | Ala | GCU, GCC, GCA, GCG. |
| Arginine | R | Arg | CGU, CGC, CGA, CGG ; AGA, AGG. |
| Asparagine | N | Asn | AAU, AAC. |
| Acide aspartique | D | Asp | GAU, GAC. |
| Cystéine | C | Cys | UGU, UGC. |
| Glutamine | Q | Gln | CAA, CAG. |
| Acide glutamique | E | Glu | GAA, GAG. |
| Glycine | G | Gly | GGU, GGC, GGA, GGG. |
| Histidine | H | His | CAU, CAC. |
| Isoleucine | I | Ile | AUU, AUC, AUA. |
| Leucine | L | Leu | UUA, UUG ; CUU, CUC, CUA, CUG. |
| Lysine | K | Lys | AAA, AAG. |
| Méthionine | M | Met | AUG. |
| Phénylalanine | F | Phe | UUU, UUC. |
| Proline | P | Pro | CCU, CCC, CCA, CCG. |
| Pyrrolysine | O | Pyl | UAG, avant élément PYLIS. |
| Sélénocystéine | U | Sec | UGA, avec séquence SECIS. |
| Sérine | S | Ser | UCU, UCC, UCA, UCG ; AGU, AGC. |
| Thréonine | T | Thr | ACU, ACC, ACA, ACG. |
| Tryptophane | W | Trp | UGG. (UGA) |
| Tyrosine | Y | Tyr | UAU, UAC. |
| Valine | V | Val | GUU, GUC, GUA, GUG. |
| Initiation | | | AUG. (UUG, CUG) |
| Terminaison | * | | UAG, UAA ; UGA. |

3. https://fr.wikipedia.org/wiki/Transcription_(biologie)

### *Une protéine*

Voici une protéine, une vraie, pas les choses un peu visqueuses que vous pouvez acheter dans les salles de gymnastique. Elle est issue d'une grenouille griffue d'Afrique de l'Ouest. Chaque ligne du code source ci-dessous est identifiée par un code de deux caractères en début de ligne. Le texte proprement dit de la séquence est à la fin, avec un code ligne vide. Le code ligne `//` marque la fin de la séquence. Le code ligne `ID` est un identifiant, peu utile parce qu'il peut changer au fil du temps. Le code ligne `AC` indique une ligne qui contient un ou plusieurs *Accession Numbers*, immuables, et de ce fait le premier est généralement utilisé pour identifier la séquence.

```
ID   1433B_XENTR             Reviewed;         244 AA.
AC   Q5XGC8; Q28HK2;
DT   22-NOV-2005, integrated into UniProtKB/Swiss-Prot.
DT   23-NOV-2004, sequence version 1.
DT   28-NOV-2006, entry version 18.
DE   14-3-3 protein beta/alpha.
GN   Name=ywhab;
OS   Xenopus tropicalis (Western clawed frog) (Silurana tropicalis).
OC   Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
OC   Amphibia; Batrachia; Anura; Mesobatrachia; Pipoidea; Pipidae;
OC   Xenopodinae; Xenopus; Silurana.
OX   NCBI_TaxID=8364;
RN   [1]
RP   NUCLEOTIDE SEQUENCE [LARGE SCALE MRNA].
RG   Sanger Xenopus tropicalis EST/cDNA project;
RL   Submitted (MAR-2006) to the EMBL/GenBank/DDBJ databases.
RN   [2]
RP   NUCLEOTIDE SEQUENCE [LARGE SCALE MRNA].
RC   TISSUE=Embryo;
RG   NIH - Xenopus Gene Collection (XGC) project;
RL   Submitted (OCT-2004) to the EMBL/GenBank/DDBJ databases.
CC   -!- FUNCTION: Adapter protein implicated in the regulation of a large
CC       spectrum of both general and specialized signaling pathway. Binds
CC       to a large number of partners, usually by recognition of a
CC       phosphoserine or phosphothreonine motif. Binding generally results
CC       in the modulation of the activity of the binding partner (By
CC       similarity).
CC   -!- SUBUNIT: Homodimer (By similarity).
CC   -!- SUBCELLULAR LOCATION: Cytoplasm (By similarity).
CC   -!- SIMILARITY: Belongs to the 14-3-3 family.
CC   ---------------------------------------------------------------------
CC   Copyrighted by the UniProt Consortium, see http://www.uniprot.org/terms
CC   Distributed under the Creative Commons Attribution-NoDerivs License
CC   ---------------------------------------------------------------------
DR   EMBL; CR760847; CAJ82973.1; -; mRNA.
DR   EMBL; BC084514; AAH84514.1; -; mRNA.
DR   UniGene; Str.8742; -.
DR   SMR; Q5XGC8; 1-231.
DR   Ensembl; ENSXETG00000022830; Xenopus tropicalis.
DR   InterPro; IPR000308; 14-3-3.
DR   Gene3D; G3DSA:1.20.190.20; 14-3-3; 1.
DR   PANTHER; PTHR18860; 14-3-3; 1.
DR   Pfam; PF00244; 14-3-3; 1.
DR   PRINTS; PR00305; 1433ZETA.
DR   ProDom; PD000600; 14-3-3; 1.
DR   SMART; SM00101; 14_3_3; 1.
DR   PROSITE; PS00796; 1433_1; 1.
DR   PROSITE; PS00797; 1433_2; 1.
KW   Acetylation.
FT   CHAIN         1    244       14-3-3 protein beta/alpha.
FT                                /FTId=PRO_0000058600.
FT   MOD_RES       1      1       N-acetylmethionine (By similarity).
SQ   SEQUENCE   244 AA;  27721 MW;  FF766793EA1CA9E5 CRC64;
     MDKSELVQKA KLSEQAERYD DMAASMKAVT ELGAELSNEE RNLLSVAYKN VVGARRSSWR
     VISSIEQKTE GNDKRQQMAR EYREKVETEL QDICKDVLGL LDKYLVPNAT PPESKVFYLK
     MKGDYYRYLS EVASGDSKQE TVTCSQQAYQ EAFEISKSEM QPTHPIRLGL ALNFSVFYYE
     ILNSPEKACS LAKSAFDEAI AELDTLNEES YKDSTLIMQL LRDNLTLWTS ENQGEEADNA
     EADN
```

### *Alignement de séquences*

Les biologistes sont intéressés par les similitudes entre séquences, qui peuvent les mettre sur la piste de similitudes fonctionnelles, ou de parentés génétiques entre organismes. Le calcul consiste, pour le dire de façon très simplifiée, à affecter des coûts aux différences (mutations) et aux décalages *(gaps)* et à chercher l'alignement qui minimise la somme de ces coûts.

Voici un exemple jouet d'alignement de deux séquences nucléiques (ADN), avec le code source à gauche et l'éxécution à droite :

```
from Bio import Align

aligner = Align.PairwiseAligner()
seq1 = "GAACT"
seq2 = "GAT"
score = aligner.score(seq1, seq2)
alignments = aligner.align(seq1, seq2)
for alignment in alignments:
    print(alignment)
```

```
GAACT
||--|
GA--T
<BLANKLINE>
GAACT
|-|-|
G-A-T
<BLANKLINE>
```

Les deux solutions proposées comportent chacune deux vides (*gaps*) et trois concordances (*matches*), soit avec les paramètres par défaut de la méthode `PairwiseAligner()` un score égal à 3 (moins on additionne de coûts plus le score augmente).

### *BLAST bien sûr...*

Le programme Blast, dont la première version a été publiée en 1990, est toujours le logiciel phare de la biologie moléculaire. Il peut être vu comme une extension et une généralisation des méthodes évoquées ci-dessus. Il est bien sûr accessible dans l'environnement Biopython.

```
#!/usr/bin/env python3
import sys
from Bio.Blast import NCBIWWW
from Bio.Blast import NCBIXML
def Blast_sonde(blastversion, collection, gi):
    result_handle = NCBIWWW.qblast(blastversion, \
                                   collection, gi)
    blast_record = NCBIXML.read(result_handle)
    E_VALUE_THRESH = 0.04
    for alignment in blast_record.alignments:
        for hsp in alignment.hsps:
            if hsp.expect < E_VALUE_THRESH:
                print("****Alignment****")
                print("sequence:", alignment.title)
                print("length:", alignment.length)
                print("e value:", hsp.expect)
                print(hsp.query[0:75] + "...")
                print(hsp.match[0:75] + "...")
                print(hsp.sbjct[0:75] + "...")

blastversion = sys.argv[1]
collection = sys.argv[2]
gi = sys.argv[3]
Blast_sonde(blastversion, collection, gi)
```

Si on connaît l'identifiant GenBank, avec la ligne de commande :

```
./BLAST-biopython.py blastn nt 8332116
```

BLAST évalue la pertinence statistique du score par une analyse de la distribution des scores d'alignement entre la séquence-test et l'ensemble des séquences cibles, et calcule la probabilité et l'espérance mathématique de trouver un alignement donnant un score donné parmi les cibles, uniquement du fait du hasard. L'espérance mathématique est notée *e-value*.

```
****Alignment****
sequence: gi|1219041180|ref|XM_021875076.1| PREDICTED: Chenopodium quinoa
cold-regulated 413 plasma membrane protein 2-like (LOC110697660), mRNA
length: 1173
e value: 1.63199e-117
ACAGAAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAATTGGC ...
|| ||||||||| |||| | |||| ||  |||| |||| | |||| ||| | || ...
ACCGAAAATGGGCAGAGGAGTGAATTATATGGCAATGACACCTGAGCAACTAGC ...
****Alignment****
sequence: gi|1226796956|ref|XM_021992092.1| PREDICTED: Spinacia oleracea
cold-regulated 413 plasma membrane protein 2-like (LOC110787470), mRNA
length: 672
e value: 1.0299e-113
AAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAATTGGCCGT ...
|||||||| |||  |||| | || ||||| |||||||| || ||||| ||||  ...
AAAATGGGTAGACGAATGGATTATTTGGCGATGAAAACCGAGCAATTAGCCGC ...
****Alignment****
sequence: gi|731339628|ref|XM_010682658.1| PREDICTED: Beta vulgaris subsp.
vulgaris cold-regulated 413 plasma membrane protein 2 (LOC104895996), mRNA
length: 847
e value: 2.76359e-108
TTGGCCATGAAAACTGATCAATTGGCCGTGGCTAATATGATCGATTCCGATAT ...
|||||||||||||||| ||| ||||  |||||||| |||| ||||  ||||| ...
TTGGCCATGAAAACTGAGCAAATGGCGTTGGCTAATTTGATAGATTATGATAT ...
****Alignment****
sequence: gi|1389679838|ref|XM_016034586.2| PREDICTED: Ziziphus jujuba
cold-regulated 413 plasma membrane protein 2-like (LOC107424728), mRNA
length: 946
e value: 1.43158e-105
AAAATGGGGAGAGAAATGAAGTACTTGGCCATGAAAACTGATCAATTGGCCGT ...
||||||||||    ||| |||  ||||| |||| |||||||| |   |||   ...
AAAATGGGGAGG---ATGGAGTTTTTGGCTATGAGAACTGATCCA---GCCAC ...
```

## Apprend-on ainsi à programmer ?

L'expérience accumulée par les chercheurs dans le domaine des interfaces entre l'être humain et la machine (IHM) peut nous aider à comprendre les démarches d'apprentissage de la programmation. Ainsi, ils ont remarqué que la difficulté ne résidait en général pas dans l'écriture d'une instruction individuelle. Ce résultat est confirmé par l'expérience des tableurs, qui ont permis à des milliers de comptables de programmer des systèmes de gestion assez complexes en écrivant des formules de calcul dans des cases, ce qui en soi est un résultat remarquable. Ce constat peut être étendu à toutes sortes de *Frameworks*, *Content Management Systems* ou *Cascading Style Sheets*, qui permettent à des professionnels peu pourvus en compétences informatiques de réaliser les applications qu'ils souhaitent en insérant ou en modifiant une instruction ponctuelle dans un cadre plus vaste qui leur est fourni tout préparé.

Ce qui est vraiment difficile, malgré l'enseignement du *Discours de la méthode*, c'est de décomposer un problème en sous-problèmes, ce qui explique la propension du débutant à « faire ci et puis après faire ça », parce que la démarche itérative semble

contourner cette difficulté. Comme on le sait, ce contournement ne tarde jamais à se retourner contre son auteur, sous la forme du célèbre « plat de spaghettis » auquel va bientôt ressembler son programme. Ceci nous semble une bonne raison d'encourager le style de programmation fonctionnel, qui encourage et facilite cette décomposition des problèmes en sous-problèmes, résolus par des sous-programmes. Le concept de sous-programme nous semble ici un pivot de l'art de la programmation.

Le style fonctionnel est possible avec la plupart des langages, mais certains l'encouragent plus que d'autres.

### *Tri Rapide en Python*

Le programme Python ci-dessous a l'avantage d'être lisible, ce qui en fait un bon pseudo-code. Ses performances sont pour le moins médiocres, et le rôle sémantique du caractère de tabulation pour dénoter la structure du code, s'il est acceptable pour un script bref avec un seul niveau d'indentation, est rédhibitoire pour de vrais programmes, des fonctions de plus de dix lignes, du texte que l'on voudra déplacer dans le corps du programme, etc.

```
from partition import partition

def tri_rapide (T):
    imin = 0
    imax = len(T) - 1
    tri_part(T, imin, imax)
    print (T)

def tri_part(T, imin, imax):
    if imin < imax:
        q = partition (T, imin, imax)
        tri_part(T, imin, q)
        tri_part(T, q + 1, imax)
```

```
from echange import echange

def partition (T, imin, imax):
    x = T[imin]
    while True:
        while T[imax] > x:
            imax -= 1
        while T[imin] < x:
            imin += 1
        if imin < imax:
            echange (T, imin, imax)
            imax -= 1a
            imin += 1
        else:
            return imax
```

```
def echange(V, i, j):
    tmp = V[i]
    V[i] = V[j]
    V[j] = tmp
```

Le lecteur trouvera le texte complet des programmes, y compris les fonctions auxiliaires pour mesurer les temps de calcul, sur le site de l'auteur [2].

### *Tri Rapide en Scheme*

Le même tri est programmé en langage Scheme et présenté ci-après. Après avoir constaté l'allergie des biologistes aux manuels de programmation dont les exercices s'attaquent à la résolution numérique d'équations différentielles, l'auteur a rédigé son propre manuel, avec des exemples d'algorithmes bioinformatiques [**?**].

```
(define (tri-rapide v imin imax)
  (if (<fx imin imax)
      (let ((q (partition v imin imax)))
        (tri-rapide v imin q)
        (tri-rapide v (+fx q 1) imax)))
  v)

(define (swap v i j)
  (let ((x (vector-ref v i)))
    (vector-set! v i (vector-ref v j))
    (vector-set! v j x)))
```

```
(define (partition v imin imax)
  (let ((x (vector-ref v imin)))
    (let loop ()
      (let loop1 ()
        (if (>fx (vector-ref v imax) x)
            (begin
              (set! imax (-fx imax 1))
              (loop1))))
      (let loop2 ()
        (if (<fx (vector-ref v imin) x)
            (begin
              (set! imin (+fx imin 1))
              (loop2))))
      (if (<fx imin imax)
          (begin
            (swap v imin imax)
            (set! imin (+fx imin 1))
            (set! imax (-fx imax 1))
            (loop))
          imax))))
```

### *Tri rapide en C*

Toujours le même programme mais dans le langage C ci-après.

```
void quicksort(int tableau[], int premier, int dernier){
  int i, j, pivot, temp;
  if(premier < dernier){
    pivot = premier;
    i = premier;
    j = dernier ;
    while(i < j){
      while(tableau[i] <= tableau[pivot] && i < dernier)
    i++;
      while(tableau[j] > tableau[pivot])
    j--;
      if(i < j){
    temp = tableau[i];
    tableau[i] = tableau[j];
    tableau[j] = temp;
      }
    }
    temp = tableau[pivot];
    tableau[pivot] = tableau[j];
    tableau[j] = temp;
    quicksort(tableau, premier, j-1);
    quicksort(tableau, j+1, dernier);
  }
}
```

### *Temps d'exécution du tri rapide*

En considérant $200 \times 10^6$ valeurs prises au hasard entre 1 et $10^8$, le tableau ci-après donne les temps d'exécution du tri rapide, respectivement en C compilé avec `clang` et l'option d'optimisation `-O3`, et avec le compilateur Scheme Bigloo, option d'optimisation `-O5`. Compte tenu du fait que Bigloo vient avec un environnement d'exécution *runtime* et notamment un glaneur de cellules (*garbage collector*, GC), on peut considérer ce résultat comme satisfaisant. Python et les autres compilateurs Scheme échouent à effectuer ce calcul.

| C | | Bigloo | |
|---|---|---|---|
| real | 65,3 s | real | 109,0 s |

## Choix d'un langage d'initiation à la programmation

Nous identifions ici plusieurs critères :

— la facilité d'apprentissage est la première qualité d'un langage choisi pour l'enseignement à des débutants ;
— une syntaxe sobre, régulière et expressive contribue à la facilité recherchée ;
— la lisibilité : les langages qui expriment leurs constructions par des mots sont plus faciles à apprendre que ceux qui les expriment par des signes cabalistiques, ne serait-ce que parce que les mots sont plus faciles à mémoriser ;
— le niveau d'abstraction du langage doit se placer au « juste milieu » :
  — ne pas exiger d'emblée la compréhension du fonctionnement de l'OS et du processeur ;
  — mais permettre la programmation d'opérations élémentaires telles que tri ou fonctions de condensation.
— le langage choisi doit encourager la structuration en sous-programmes, grâce à des constructions faciles à comprendre ;
— le langage doit fournir une gestion automatique de la mémoire (GC) ; il y a bien assez de choses à apprendre et à comprendre avant la gestion explicite de la mémoire ;
— le langage doit pouvoir être compilé : seul un programme exécutable constitué matériellement dans un fichier peut communiquer l'intuition d'un artefact qui calcule ;
— les environnements intégrés de programmation munis d'interfaces graphiques actionnées à la souris nous semblent des obstacles à la compréhension plutôt qu'autre chose. Outre qu'ils sont intrinsèquement compliqués et difficiles à maîtriser, ils dissimulent des choses qu'il faut au contraire voir.

## Apports techniques du style fonctionnel

Outre les avantages formels, évoqués ci-dessus, que procure le style fonctionnel, il apporte des avantages techniques. Depuis quelques années les performances unitaires des processeurs progressent de moins en moins vite, et leur miniaturisation se heurtera de toute façon à des limites physiques.

L'amélioration des performances des ordinateurs et autres artefacts programmables passe désormais par la multiplication des organes de calcul, que ce soit par les processeurs multi-cœurs (en fait des multi-processeurs) ou par le recours aux processeurs graphiques (*Graphics Processing Unit*, GPU), qui sont en fait des unités de calcul vectoriel, capables d'appliquer simultanément une instruction unique à de multiples données rangées dans un vecteur.

L'informatique en nuages (*cloud computing*, infonuagique), en permettant de répartir un calcul sur un nombre indéterminé d'ordinateurs dont on ne sait pas où ils sont, suggère une évolution des architectures de calcul dans cette même direction.

Ces technologies stimulent le recours au calcul parallèle ; garantir qu'une tâche ne modifiera pas l'état du système pris comme hypothèse par une autre tâche est nécessaire au déterminisme du calcul ; le style fonctionnel limite le recours à l'affectation et par là facilite la satisfaction de cette condition.

## Le marché florissant de la lambda-expression

C'est l'aboutissement (provisoire) des évolutions mentionnées à la section précédente, la lambda-expression, qui jusqu'à très récemment appartenait au vocabulaire exclusif des informaticiens théoriciens, devient une denrée commerciale de grande consommation.

## Remerciements

Pour les méthodes bioinformatiques, je dois tout à William Saurin. Manuel Serrano m'a aidé pour les comparaisons de programmes dans des langages différents. Ils ne sont bien sûr en rien responsables des erreurs ou imprécisions de ce texte et de ces codes sources.

## Références

[1] Laurent Bloch. Python et biopython. 2019.

[2] Laurent Bloch. Python, scheme, c. 2019.

[3] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM (CACM)*, 9(5), mai 1966.

[4] Edsger Wybe Dijkstra. Letters to the editor : Go to statement considered harmful. *Commun. ACM*, 11(3) :147–148, March 1968.

[5] Raphaël Fournier-S'niehotta, Baptiste Mélès, and Lionel Tabourier. Séminaire codes sources.

[6] Benjamin Wack, Sylvain Conchon, Judicaël Courant, Marc de Falco, Gilles Dowek, Jean-Christophe Fillâtre, and Stéphane Gonnord. *Informatique pour tous en classes préparatoires aux grandes écoles - Manuel d'algorithmique et programmation structurée avec Python*. Eyrolles, Paris, 2013.