Programmation en Python pour les mathématiques

- N°29 - Mars 2012 -

Date de mise en ligne : mardi 6 mars 2012

Les nouvelles technologies pour l'enseignement des mathématiques

Pour mémoire, voici d'autres articles de Guillaume Connan dans MathémaTICE.

Nous [1] venons de publier un **ouvrage consacré à l'utilisation du langage Python** en cours de mathématiques, du collège jusqu'aux premières années universitaires.

Ce langage, qui est un des plus utilisés actuellement, permet de faire des mathématiques rapidement et simplement à tous les niveaux tout en étant soutenu et sans cesse amélioré par une immense communauté à travers le monde : libre et puissant, Python permet en effet une utilisation en toute sérénité. Il incite à programmer de manière concise et claire.

Le langage universel n'existe pas et l'utilisation de Python pourra être mise en parallèle avec d'autres langages, notamment fonctionnels, qui éclairent d'autres pans des mathématiques que nous aimons enseigner.

Ces dernières pourront être travaillées avec profit grâce à Python qui reste, tout en étant à la fois simple et clair, robuste et professionnel. Ce langage est largement répandu et illustre naturellement de nombreux concepts de notre matière.

Pour poursuivre la réflexion sur les rapports entre le langage mathématique et les langages de programmation, on pourra lire cet article de Gilles Dowek

L'introduction de l'algorithmique dans l'enseignement secondaire en est à ses balbutiements mais de nombreux professeurs de mathématiques, notamment ceux travaillant dans ou avec les IREM, réfléchissent depuis longtemps aux rapports et enrichissements de son enseignement pour les mathématiques.

Au moins deux pistes s'offrent à nous : illustrer de manière efficace et justifiée une notion mathématique à l'aide de l'outil informatique ou inversement, faire des mathématiques en explorant une notion informatique.

Nous avons cherché à explorer ces deux options tout en proposant une présentation des fonctionnalités de Python, les plus simples, comme certaines un peu plus techniques en fin d'ouvrage.

Voici quelques exemples permettant de s'en faire une idée.

Au sujet du PGCD de deux nombres entiers naturels

Nous pouvons établir, dès la classe de troisième de collège, l'égalité :

$$\mathrm{PGCD}(295400101920462517154, 10720242531918724) = 74$$
 En fait, la calculatrice comme le tableur sont impuissants devant un tel calcul, car les entiers en jeu sont trop grands pour une représentation binaire ordinaire sur quatre octets : le recours aux logiciels spécialisés s'impose, sauf à calculer « à la main », ce qui restera fastidieux dans l'exemple choisi.

Le langage de programmation Python n'impose pas de limite de taille pour la représentation des entiers (hormis les limites fixées par son environnement d'exécution) et vient à notre secours, de plusieurs façons.

L'algorithme d'Euclide repose sur la propriété suivante du PGCD :

$$\mathrm{PGCD}(a, b) = \mathrm{PGCD}(b, r)$$
où r désigne le reste dans la division euclidienne de a par b, b étant non nul.

C'est d'ailleurs cette propriété que nos élèves mettent en oeuvre avec la calculatrice ou un logiciel tableur. avec toutefois un inconvénient majeur pour l'apprentissage : en effet, une fois la feuille de calcul correctement programmée (dans le langage spécialisé du tableur) et exécutée, il ne reste au mieux sous les yeux de l'élève qu'une table de nombres, dont on a perdu la trace de l'obtention, sinon la signification. En particulier, les égalités des divisions euclidiennes successives sont visuellement absentes, ce qui est pour le moins dommage dans un contexte pédagogique.

Passons alors à la programmation en langage Python de cet algorithme :

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

 import sys
 sys.setrecursionlimit(100000)

 def reste(a, b):
 if a < b:
 return a
 else:
 return reste(a-b, b)

 def pgcd(a, b):
 if b == 0:
 return a
 else:
 return pgcd(b, reste(a, b))

 print(pgcd(295400101920462517154, 10720242531918724))
```

L'exécution du programme ci-dessus donnera immédiatement 74, nonobstant la « grosseur » des entiers. Mentionnons que Python dispose d'un opérateur binaire (noté malheureusement « % » pour les professeurs qui auront à introduire cette notation avec la notion déjà délicate en elle-même de pourcentage, suffisant à rechercher d'éviter des expressions telles que « 20%3 », dont la syntaxe est pourtant licite dans la plupart des langages de programmation), parfaitement mobilisable ; la programmation de la fonction (au sens informatique) « reste » est au demeurant un exercice fondamental d'algorithmique.

Faisons quelques commentaires sur ce programme :

- le texte du programme (le code source) est limpide et très proche de la propriété mathématique utilisée : elle y est en littéralement inscrite, et de plus complétée par la propriété :

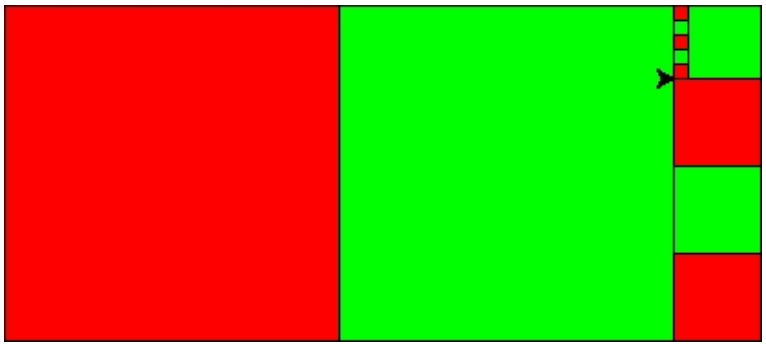
  $$\mathrm{PGCD}(a, 0) = a$$
  Notre programme Python n'est finalement rien d'autre qu'une démonstration (au sens visuel) des mathématiques en action, et cela de façon évidente, contrairement à ce que permettraient la calculatrice et le tableur !

- En aucun cas, nous n'avons procédé à des affectations ou allocations de mémoire, ce qui nous est offert par

l'emploi d'une programmation récursive, possible en Python, qui évoque bien sûr la notion mathématique de récurrence. À l'inverse, un style itératif nous aurait conduit à introduire et gérer nous-mêmes des variables ;

- Le programme pourra même, en Python, être aisément modifié, pour produire sous la forme d'un affichage ou d'un fichier texte, les égalités des divisions euclidiennes successives.

Nous allons maintenant voir une autre utilisation de Python, graphique cette fois, pour mettre en évidence toujours sur le thème du PGCD l'algorithme des différences, moins raffiné que le précédent, néanmoins porteur de sens quant à la question de la commensurabilité. Nous produisons ci-dessous en Python le texte d'un programme qui réalise l'anthyphérèse d'un rectangle de côtés entiers, en pavant progressivement et visuellement celui-ci par des carrés, en commençant toujours par le plus grand carré possible. L'algorithme est alors opéré, carré après carré, sous nos yeux, ce qui n'est pas le moins pour forger une image mentale !



Le programme « anthypherese.py » est reproduit ci-après :

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

 from turtle import * # pour utiliser une tortue à la mode "LOGO"

 bascule = 0 # alterner entre 0 (rouge) et 1 (vert)

 def carré(a):
 """ tracer un carré de côté a """
 global bascule
 if bascule == 0:
 color("red")
 else:
 color("green")
 bascule = 1 - bascule

 begin_fill()
 for k in range(4):
 forward(a)
 left(90)
 end_fill()

 color("black")
 for k in range(4):
 forward(a)
```



```
left(90)

def rectangle(a, b):
""" tracer le rectangle de côtés a et b """
for k in range(2):
forward(a)
left(90)
forward(b)
left(90)

def anthyphérèse(a, b):
""" a et b sont les dimensions du rectangle """
if a == b:
carré(a)
else:
while a > b:
carré(b)
forward(b)
a -= b # signifie: a = a-b
forward(a)
left(90)
anthyphérèse(b, a)

longueur, largeur = 416 , 184 # affectations multiples
rectangle(longueur, largeur)
anthyphérèse(longueur, largeur)
```

Listes Python

Dans cette section, nous allons nous intéresser à quelques « confiseries syntaxiques » sur les listes. La liste est certainement la structure de donnée la plus utilisée en **Python**. Une liste **Python** s'apparente quelque peu à un tableau en Java, mais en mieux ! C'est un objet qui croît dynamiquement au fur et à mesure que de nouveaux éléments y sont ajoutés.

1. Prise en main rapide des listes

Donnons toute de suite un exemple de liste :

```
>>> liste = ['a', 12, 12.23, 0, 'exemple', 15]
 >>> liste
 ['a', 12, 12.23, 0, 'exemple', 15]
```

Comme on le voit dans l'exemple précédent, une liste est une séquence d'éléments, rangés dans un certain ordre ; de plus, en **Python**, une liste n'est pas nécessairement homogène : elle peut contenir des objets de types différents les uns des autres.

La première manipulation que l'on a besoin d'effectuer sur une liste, c'est d'en extraire un élément : la syntaxe est alors `liste[indice]`. Par exemple, cherchons à extraire un élément de notre liste :

```
>>> liste[2]
 12.23
```

Le résultat peut surprendre : on aurait peut-être attendu comme réponse 12 au lieu de 12.23. En fait, les éléments d'une liste sont indexés à partir de 0 et non de 1.

Pour l'extraction de parties d'une liste, on dispose d'outils de « saucissonnage » particulièrement conviviaux [2] :

```
>>> liste = [12, 11, 18, 7, 15, 3]
 >>> liste[2:]
 [18, 7, 15, 3]
 >>> liste[:2]
 [12, 11]
 >>> liste[0:len(liste)]    # len(liste) fournit la longueur de la liste
 [12, 11, 18, 7, 15, 3]
 >>> liste[:]               # le même en mieux
 [12, 11, 18, 7, 15, 3]
 >>> liste[2:5]
 [18, 7, 15]
 >>> liste[-1]                # équivaut à : liste[len(liste)-1]
 3
```

2. Quelques méthodes.

Parmi les nombreuses méthodes que possède un objet (au sens de la programmation orientée objet) du type `list`, il y en a une qui permet d'ajouter un élément en fin de liste, c'est la méthode `append()`. Voici comment l'utiliser :

```
>>> liste = [12, 11, 18, 7, 15, 3]
 >>> liste
 [12, 11, 18, 7, 15, 3]
 >>> liste.append(5)
 >>> liste
 [12, 11, 18, 7, 15, 3, 5]
```

Remarquons cette syntaxe qui peut surprendre la première fois qu'on la rencontre ; il s'agit de la notation point : l'instruction `liste.append(5)` signifie que l'on modifie la valeur de l'objet `liste` en lui appliquant la méthode `append()` avec comme paramètre effectif `5`.

Parmi les méthodes associées aux listes, voici les plus utiles :

la méthode...	**son effet...**
`list.append(x)`	**ajoute l'élément** `x` en fin de liste
`list.extend(L)`	**ajoute en fin de liste les éléments de L**
`list.insert(i, x)`	**insère un élément** `x` en position `i`
`list.remove(x)`	**supprime la première occurrence de** `x`
`list.pop([i])`	**supprime l'élément d'indice** `i` et le renvoie
`list.index(x)`	**renvoie l'indice de la première occurrence de** `x`

`list.count(x)`	**renvoie le nombre d'occurrences de** `x`
`list.sort()`	**modifie la liste en la triant**
`list.reverse()`	**modifie la liste en inversant l'ordre des éléments**

3. Définir des listes par compréhension.

Pour créer des listes, **Python** fournit une facilité syntaxique particulièrement agréable, à savoir les listes (définies) par compréhension (en anglais, *list-comprehensions*). Elles permettent de générer des listes d'une manière très concise, sans avoir à utiliser de boucles.

Les listes définies par compréhension remplacent simultanément un « mappage » et un « filtrage » comme il y en a dans tous les langages fonctionnels.

La syntaxe pour définir une liste par compréhension est proche de celle utilisée en mathématiques pour définir un ensemble par compréhension. En voici quelques exemples :

```
>>> liste = [2, 4, 6, 8, 10]
 >>> [3*x for x in liste]
 [6, 12, 18, 24, 30]
 >>> [[x, x**3] for x in liste]
 [[2, 8], [4, 64], [6, 216], [8, 512], [10, 1000]]
 >>> [3*x for x in liste if x > 5]  # on filtre avec une condition
 [18, 24, 30]
 >>> [3*x for x in liste if x**2 < 50] # idem
 [6, 12, 18]
 >>> liste2 = list(range(3))
 >>> [x*y for x in liste for y in liste2]
 [0, 2, 4, 0, 4, 8, 0, 6, 12, 0, 8, 16, 0, 10, 20]
```

Dans la suite de cette section, on présente pêle-mêle plusieurs utilisations des listes définies par compréhension.

- Voici par exemple, un moyen efficace d'obtenir la liste des années bissextiles dans un intervalle donné :

```
>>> bissextile = [b for b in range(2000, 2100)
 ...               if (b % 4 ==0 and b % 100 != 0) or (b % 400 == 0)]
 >>> bissextile
 [2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028, 2032, 2036, 2040, 2044, 2048, 2052, 2056, 2060, 2064, 2068,
2072, 2076, 2080, 2084, 2088, 2092, 2096]
```

- On appelle *triplet pythagoricien* un triplet d'entiers naturels (x, y, z) non nuls tels que $x^2+y^2=z^2$. Voici un moyen de trouver les triplets pythagoriciens dont les composantes sont inférieures à 10.

```
>>>  n = 11
 >>> [(x, y, z) for x in range(1, n) for y in range(1, n)
 ...            for z in range(1, n) if x**2 + y**2 == z**2]
 [(3, 4, 5), (4, 3, 5), (6, 8, 10), (8, 6, 10)]
```

- Le calcul du produit scalaire de deux vecteurs représentés par des listes s'effectue en utilisant les primitives `sum` et `zip` [3].

```
>>> u = [1, 4, 7, 8, -2]
 >>> v = [0, 11, -7, 1, -3]
 >>> sum([x * y for x, y in zip(u, v)])
 9
```

- Pour obtenir les diviseurs d'un entier *n*, on peut également utiliser une liste par compréhension :

```
>>> n = 100; [d for d in range(1, n+1) if n % d == 0]  # les diviseurs de 100
 [1, 2, 4, 5, 10, 20, 25, 50, 100]
```

On en déduit une manière simple de tester si un nombre est parfait. On rappelle qu'un nombre est *parfait* lorsqu'il est la somme de ses diviseurs propres.

```
>>> def parfait(n):
 ...     return sum([d for d in range(1, n) if n % d == 0]) == n
 ...
 >>> [i for i in range(2, 1000) if parfait(i)]
 [6, 28, 496]
```

Un manière naïve de donner la liste des nombres premiers inférieurs à 100 serait :

```
>>> [p for p in range(2, 100) if [i for i in range(2, p) if p % i == 0] == []]
 [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Une façon beaucoup plus efficace de générer l'ensemble des nombres premiers inférieurs à un entier donné serait d'utiliser le crible d'Ératosthène :

```
>>> def crible(prems):
 ...     if prems == []: return []
 ...     return [prems[0]] + crible([p for p in prems[1:] if p % prems[0] != 0])
 ...
 >>> crible(range(2, 100))
 [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

On peut réutiliser le principe de cette fonction pour trouver la décomposition d'un entier en produit de facteurs premiers :

```
#!/usr/bin/python3
 #-*- coding: Utf-8 -*-
 def factorise(n, facteurs, liste):
 if n == 1: return n, facteurs, liste
 if n % liste[0] == 0:
 return factorise(n // liste[0], facteurs + [liste[0]], liste)
 return factorise(n, facteurs, [p for p in liste[1:] if p % liste[0] != 0])

 def factPrems(n):
 resultat = factorise(n, [], range(2, n + 1))[1]
 print(str(n) + ' = ' + '×'.join([str(j) for j in resultat]))
 return resultat

 import sys
 sys.setrecursionlimit(100000)
```

```
for n in crible(range(2, 18)):
factPrems(2**n - 1)
```

Après exécution de ce programme, on trouve que le premier nombre de Mersenne non premier est $2^{11} - 1 = 2047 = 23\times 89$.

Une version itérative de ces algorithmes est donnée à la section 3.5 du chapitre 3.

- Les listes définies par compréhension permettent de programmer le tri rapide (*quicksort*) de manière particulièrement concise [4] :

```
>>> def tri_rapide(l):
...     if l == []: return []
...     return (tri_rapide([x for x in l[1:] if x < l[0]])
...             + [l[0]] + tri_rapide([x for x in l[1:] if x >= l[0]]))
...
>>> import random
>>> alea = [random.randrange(100) for i in range(20)]
>>> alea
[46, 24, 38, 99, 97, 30, 60, 9, 86, 66, 69, 41, 99, 28, 59, 0, 95, 92, 82, 2]
>>> tri_rapide(alea)
[0, 2, 9, 24, 28, 30, 38, 41, 46, 59, 60, 66, 69, 82, 86, 92, 95, 97, 99, 99]
```

- Les méthodes de simulation de *Monte-Carlo* permettent de calculer une valeur numérique en utilisant des procédés aléatoires. Par exemple, le calcul approché de l'intégrale de la fonction $f(x)$ sur le segment $[a, b]$ par cette méthode repose sur l'approximation :

$$ \int_{a}^{b} f(t) dt \simeq \frac{b-a}{n} \sum_{i=1}^{n} f(x_i) $$

où $i\mapsto x_i$ désigne une variable aléatoire suivant la loi uniforme sur l'intervalle $[a, b]$ [5].

Voici un calcul approché du nombre $ln(2)$ par la méthode de Monte Carlo et celle du point milieu [6] :

$$ \int_{a}^{b} f(t) dt \simeq \frac{b-a}{n} \sum_{i=1}^{n} f\left(a + \left(i+\frac12\right) \frac{b-a}n\right) $$

```
>>> import random, math
>>> def monteCarlo(f, a, b, n):
...     return sum([f(random.uniform(a, b)) for i in range(n)]) * (b - a) / n
...
>>> def pointMilieu(f, a, b, n):
...     return sum([f(a + (i + 0.5) * (b - a) / n) for i in range(n)]) * (b - a) / n
...
>>> print(monteCarlo(lambda x : 1 / x, 1, 2, 10**5))
0.6936520468798466
>>> print(pointMilieu(lambda x : 1 / x, 1, 2, 10**5))    # pour comparaison
0.6931471805568249
>>> print(math.log(2))                                   # pour comparaison
0.6931471805599453
```


Pour comparer l'efficacité respective de ces deux méthodes, on peut modifier le nombre de points calculés et améliorer l'affichage :

```
#!/usr/bin/python3
 #-*- coding: Utf-8 -*-
 import random, math
 def monteCarlo(f, a, b, n):
 return sum([f(random.uniform(a, b)) for i in range(n)]) * (b - a) / n

 def pointMilieu(f, a, b, n):
 return sum([f(a + (i + 0.5) * (b - a) / n) for i in range(n)]) * (b - a) / n

 f = lambda x : 1 / x

 print('-'*80)
 print('{0:>10s} | {1:^14s} | {2:^14s} | {3:^14s} | {4:^14s} |'.format('n',
 'Monte Carlo', 'erreur absolue', 'Point Milieu', 'erreur absolue'))
 print('-'*80)

 for i in range(0, 6):
 n = 10**i
 mc = monteCarlo(f, 1, 2, n)
 pm = pointMilieu(f, 1, 2, n)
 erreur_mc = math.log(2) - mc
 erreur_pm = math.log(2) - pm
 print('{0:10d} | {1: 14.10f} | {2: 14.10f} | {3: 14.10f} | {4: 14.10f} |'
 .format(n, mc, erreur_mc, pm, erreur_pm))
 print('-'*80)
```

L'exécution de ce script donne :

```
 --------------------------------------------------------------------------------
         n |  Monte Carlo   | erreur absolue |  Point Milieu  | erreur absolue |
--------------------------------------------------------------------------------
         1 |   0.6990577081 |  -0.0059105276 |   0.6666666667 |   0.0264805139 |
        10 |   0.7075502337 |  -0.0144030531 |   0.6928353604 |   0.0003118201 |
       100 |   0.6972369518 |  -0.0040897713 |   0.6931440556 |   0.0000031249 |
      1000 |   0.6915338095 |   0.0016133710 |   0.6931471493 |   0.0000000312 |
     10000 |   0.6925572119 |   0.0005899686 |   0.6931471802 |   0.0000000003 |
    100000 |   0.6923513201 |   0.0007958604 |   0.6931471806 |   0.0000000000 |
--------------------------------------------------------------------------------
```

On constate dans tous ces exemples que l'utilisation de listes définies par compréhension permet d'écrire des fonctions de manière concise et très proche de la formulation mathématique.

Annexe

Table des matières

- Résolution numérique d'équations différentielles
- Interpolation polynomiale
- Dérivation numérique
- Intégration numérique

- **Chapitre 5 : récursivité**

-
  - Quelques exemples
  - Spirale de pentagones
  - Courbe du dragon
  - Triangle de Sierpinski
  - Sommes de termes d'une suite géométrique

- **Chapitre 6 : classes**

-
  - Graphes
  - Représentation de nombres
  - Listes
  - Arbres binaires
  - Calculateur
  - Polynômes et fractions rationnelles

- **Bibliographie**

- **Index général**
- **Index des commandes**

Les auteurs :

Alexandre CASAMAYOU-BOUCAU : professeur de mathématiques en CPGE au collège Stanislas (Paris) ;

Pascal CHAUVIN : professeur de mathématiques au collège Garcie Ferrande de Saint Gilles Croix de Vie (Vendée), animateur IREM ;

Guillaume CONNAN : professeur de mathématiques à l'IUT d'informatique de Nantes, animateur IREM.

---

[1] Les trois auteurs de l'ouvrage et de l'article : voir en fin d'article

[2] Pour une explication détaillée de ces opérations, on renvoie à la section 9 du premier chapitre.

[3] présentées à la section 14 du chapitre 1

[4] Les méthodes de tri les plus classiques sont présentées à la section 13 du chapitre 1 et leur complexité est discutée à la section 1.4 du chapitre 5.

[5] Le calcul approché d'intégrales est abordé à la section 2.2 du chapitre 2, à la section 4 du chapitre 3 et à la section 6 du chapitre 4.

[6] Le calcul des logarithmes est présenté également à la section 2.3.4 du chapitre 3.

