Mathématiques et programmation fonctionnelle

Pascal CHAUVIN
pascal.chauvin@ac-nantes.fr

Groupe « Environnements Interactifs et Enseignement des Mathématiques »
IREM des Pays de la Loire

Journées académiques de l'IREM des Pays de la Loire

17 & 18 avril 2013

Table des matières

1 L'ordinateur : un super-calculateur !

```
$ python3
Python 3.2.3 (default, Feb 20 2013, 14:44:27)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print(2**200)
1606938044258990275541962092341162602522202993782792835301376
>>>
$
```

2 Un jeu d'enfants... pour voir

2.1 La distributivité avec Python

Programme 1 – distributivite.py

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  a = 1.1
5  b = 5.3
6  k = 2.2
7
8  print( k*(a + b) )
9  print( k*a + k*b )
10 print( (k*(a + b)) - (k*a + k*b) )
```

```
$ ./distributivite.py
14.080000000000002
14.08
1.7763568394002505e-15
$
```

2.2 La distributivité avec C

Programme 2 – distributivite.c

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  double a = 1.1;
  double b = 5.3;
  double k = 2.2;

  printf("%f\n", k*(a + b) );
  printf("%f\n", k*a + k*b );
  printf("%f\n", (k*(a + b)) - (k*a + k*b) );
  return 0;
}
```

Le langage C exige une étape de compilation :

```
$ gcc distributivite.c
$
```

```
$ ./a.out
14.080000
14.080000
0.000000
$
```

2.3 La distributivité avec Objective Caml

Programme 3 – distributivite.ml

```
#!/usr/bin/env ocaml

let calcul =
  let a = 1.1 in
  let b = 5.3 in
  let k = 2.2 in
    print_float ( k *. (a +. b) );
    print_string "\n";

    print_float ( k *. a +. k *. b );
    print_string "\n";

    print_float ( (k *. (a +. b)) -. (k *. a +. k *. b) );
    print_string "\n"
;;

calcul;;
```

```
$ ./distributivite.ml
14.08
14.08
1.7763568394e-15
$
```

2.4 La distributivité avec Python : le retour !

Programme 4 – distributivite2.py

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

from decimal import *

a = Decimal("1.1")
b = Decimal("5.3")
k = Decimal("2.2")

print( k*(a + b) )

print( k*a + k*b )

print( (k*(a + b)) - (k*a + k*b) )
```

```
$ ./distributivite2.py
14.08
14.08
0.00
$
```

3 Nombres décimaux et représentation binaire

Un nombre décimal x en système décimal :

$$x = \overline{a_p\, a_{p-1}\, \ldots\, a_2\, a_1\, a_0\, ,a_{-1}\, a_{-2}\, a_{-3}\, \ldots\, a_{-q}}_{\ 10}$$

C'est-à-dire :

$$x = \left(\sum_{i=0}^{p} a_i \times 10^i\right) + \left(\sum_{i=0}^{q} a_{-i} \times 10^{-i}\right)$$

En système binaire :

$$x = \left(\sum_{i=0}^{p'} b_i \times 2^i\right) + \left(\sum_{i=0}^{q'} b_{-i} \times 2^{-i}\right)$$

Le nombre « 1,1 » ne s'écrit pas bien en système binaire [1] !

$$\overline{1{,}1}_{\ 10} = \overline{1{,}000110011\ldots}_{\ 2}$$

Le module « `outils.py` » présenté plus loin permet quelques conversions :

```
$ python3
>>> from outils import *
>>> approx(11, 10, 16, 10)
'1,1000000000000000'
>>> approx(11, 10, 16, 2)
'1,0001100110011001'
>>> approx(11, 10, 16, 3)
'1,0022002200220022'
>>>
$
```

1. Le module « `rationnel.py` » présenté plus loin offre des outils pour la conversion entre bases de numération.

4 Un type pour représenter les nombres rationnels : « rationnel.py »

Le programme ci-dessous exploite deux modules « `rationnel.py` » et « `outils.py` », dont le code source complet est donné en annexe.

Programme 5 – calculsQ.py

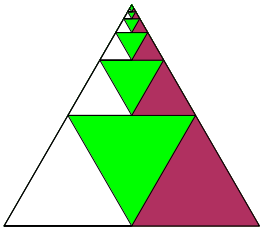
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from rationnel import *

a = rationnel(1, 2)
print(a)

b = rationnel(-2, -6)
print(b)

c = a / rationnel(3)
print(c)

print(a + b + c)

t = a * rationnel(-1, 7)
print(t)

u = (t / b) ** 5
print(u)
print(u.approximation())
print(u.approximation(30))
print(u.fraction_continue())
```

```
$ ./calculsQ.py
(1)/(2)
(1)/(3)
(1)/(6)
(1)
(-243)/(537824)
-0,0004518206699589
-0,000451820669958945677396322960
[-1, 1, 2212, 3, 1, 2, 1, 4, 1, 2]
$
```

5 Un type pour les rationnels : quelques exemples

5.1 Convergence d'une série

Python permet de réaliser aisément la figure suivante, un peu à la manière du langage **LOGO**. Elle illustre graphiquement la convergence de la série $\sum_{i=1}^{+\infty} \left(\frac{1}{4}\right)^i$, à l'aide du module natif « turtle » de **Python** :



Programme 6 – dessin.py

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

from turtle import *

def triangle(cote, code_couleur):
    if code_couleur == 0: color("green")
    if code_couleur == 1: color("white")
    if code_couleur == 2: color("maroon")
    begin_fill()
    for i in range(3):
        forward(cote)
        left(120)
    end_fill()
    color("black")
    for i in range(3):
        forward(cote)
        left(120)

def figure(n, cote):
    if n > 0:
        triangle(cote, 0)
        triangle(cote/2, 1)
        forward(cote/2)
        triangle(cote/2, 2)
        backward(cote/2)
        left(60)
        forward(cote/2)
        right(60)
        figure(n - 1, cote/2)

figure(6, 512)
```

En Python, l'indentation du code-source est fondamentale : la profondeur détermine le début et la fin d'un bloc d'instructions.

On peut aussi mener la recherche naïve de la limite de la série, avec le programme suivant :

Programme 7 – serie.py

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

from rationnel import *

#
# quelques "constantes"
#
ZERO = rationnel()
UN_QUART = rationnel(1, 4)

def somme_part(n):
  if n > 0:
    return UN_QUART ** n + somme_part(n-1)
  else:
    return ZERO

sigma = somme_part(5)
print(sigma)
print(sigma.approximation())

sigma = somme_part(6)
print(sigma)
print(sigma.approximation())
```

```
$ ./serie.py
(341)/(1024)
0,3330078125000000
(1365)/(4096)
0,3332519531250000
$
```

5.2 Algorithme de Babylone (méthode de Héron d'Alexandrie)

Programme 8 – babylone.py

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

from rationnel import *

#
# quelques "constantes"
#
UN = rationnel(1)
DEUX = rationnel(2)

def Babylone(a, x, n):
  if n > 0:
    t = Babylone(a, x, n-1)
    return (t + (a/t))/DEUX
  else:
    return UN

a = rationnel(5)
r = Babylone(a, UN, 8)

print("Une approximation de la racine carree de", a, "est:")
print(r)
print(r.approximation(40))
print(r.fraction_continue())
```

```
$ ./babylone.py
Une approximation de la racine carree de (5) est:
(31683700840009422215077673848376823600642097148698060 7)
   /(14169381771405651323470996587541191965770779495819 9867)
2,2360679774997896964091736687312762354406
[2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    4, 4, 4, 4, 4, 4, 3]
$
```

On peut démontrer rapidement à la main la périodicité du développement en fraction continue de $\sqrt{5}$:

$$\sqrt{5} = 2 + \cfrac{1}{4 + \cfrac{1}{4 + \cfrac{1}{\ldots}}}$$

Soit en notation abrégée de la période :

$$\sqrt{5} = \left[2,\ \overline{4}\right]$$

6 Les listes en Python

6.1 Le type « `list` » de Python3

```
$ python3
Python 3.2.3 (default, Feb 20 2013, 14:44:27)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = ["chevre", "loup", "chou"]
>>> a
['chevre', 'loup', 'chou']
>>> del a[1]
>>> a
['chevre', 'chou']
>>> len(a)
2
>>> a.append("loup")
>>> a
['chevre', 'chou', 'loup']
>>> x = a.pop()
>>> x
'loup'
>>> a
['chevre', 'chou']
>>> a.insert(0, x)
>>> a
['loup', 'chevre', 'chou']
>>> a.reverse()
>>> a
['chou', 'chevre', 'loup']
>>>
$
```

```
$ python3
Python 3.2.3 (default, Feb 20 2013, 14:44:27)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = [ n*(n+1)/2 for n in range(10) ]
>>> a
[0.0, 1.0, 3.0, 6.0, 10.0, 15.0, 21.0, 28.0, 36.0, 45.0]
>>> a = [ n*(n+1)//2 for n in range(10) ]
>>> a
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
>>> b = range(3, 20, 4)
>>> print(list(b))
[3, 7, 11, 15, 19]
>>>
$
```

6.2 Fractions continues : le retour !

Programme 9 – reduite.py

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

from rationnel import *

def reduite(quotients, r):
  if len(quotients) > 0:
    return rationnel(quotients[0]) + rationnel(1) / reduite(quotients[1:], r)
  else:
    return r

def calcul_reduite(quotients):
  return reduite(quotients, rationnel())

def tests_unitaires():
  r = rationnel(4291, 1329)
  print(r)
  a = r.fraction_continue()
  print(a)
  print(calcul_reduite(a))

if __name__ == "__main__":
  tests_unitaires()
```

```
$ ./reduite.py
(4291)/(1329)
[3, 4, 2, 1, 2, 4, 2, 1, 2]
(4291)/(1329)
$
```

On consultera en annexe une méthode de calcul faisant appel aux suites définies par récurrence. La méthode valable pour les nombres rationnels s'étend aux irrationnels quadratiques.

Le module `rationnel.py` offre déjà dans forme minimale actuelle un bon champ d'investigation en classe, avec une construction déjà accessible aux élèves.

7 La programmation fonctionnelle

Programmation fonctionnelle

On exprime la solution d'un problème en termes d'appels de fonctions. La notion de récursivité (et partant, de fonction) est prépondérante dans ce modèle de programmation.

Programmation impérative

On procède par modifications successives des états de la machine. Dans ce style de programmation, l'assignation de valeur (affectation) joue un rôle central. L'aspect séquentiel est très important à considérer.

Une précision terminologique, importante pour la classe

On distingue deux sens au mot « fonction », en Mathématiques et en programmation :

- en Mathématiques : (...)
- en programmation : une *fonction* est un sous-programme ou *routine* (c'est-à-dire un programme dans le programme) que le programme (ou toute routine du programme) pourra invoquer à loisir.

  Une fonction peut comporter des paramètres, et modifier (selon les règles du langage) son environnement externe.

  Un appel de *fonction* rend un résultat ; une *procédure* est une fonction ne délivrant aucun résultat, plus exactement le résultat « vide ».

En programmation fonctionnelle, les fonctions sont des types de données, au même titre que les entiers, flottants, chaînes de caractères, types composés ou objets... Une fonction peut par exemple servir de paramètre à une autre fonction.

7.1 Fonctions pures

En programmation, une fonction est dite « fonction pure » lorsque :

- elle a pour seul effet de rendre un résultat, *i.-e.* elle ne modifie pas son environnement (son contexte d'exécution) ;
- entre différentes invocations, la fonction rend le même résultat pour les mêmes arguments.

7.2 Curryfication

Programme 10 – curry.py

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 def ajoute(x):
5   return lambda y: x + y
6
7 print( ajoute(3)(4) )
```

Programme 11 – curry.ml

```
1 #!/usr/bin/env ocaml
2
3 let ajoute x =
4   function y -> x + y;;
5
6 print_int ((ajoute 3) 4);;
7 print_string "\n";;
```

7.3 Fonction comme argument d'une autre fonction

Programme 12 – trinome.py

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

from rationnel import *

#
# p(x) = 3*x^2 + 3*x + 2/3
#
def p(x):
  a = rationnel(3)
  b = rationnel(2, 3)
  return a * x ** 2 + a * x + b

PI = rationnel(22, 7)

u = p(PI)
print(u)
print(u.approximation(20))

def image(f, x):
  return f(x)

v = image(p, PI)
print(v)
print(v.approximation())
```

Deux modèles de calcul de $\sum_{i=1}^{30} i$:

Programme 13 – somme1.py

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

somme = 0

for i in range(30 + 1):
  somme = somme + i

print(somme)
```

Programme 14 – somme2.py

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

def somme(n):
  if n > 0:
    return n + somme(n-1)
  else:
    return 0

print(somme(30))
```

Note : le programme `somme2.py` peut être amélioré par une version « récursive terminale ».

8 La programmation récursive

8.1 Retour sur les boucles

Programme 15 – boucle1.py

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

def boucle(n):
  if n > 0:
    print(n)
    boucle(n - 1)

boucle(10)
```

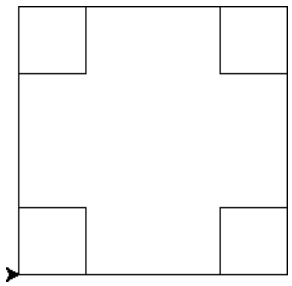
Programme 16 – boucle2.py

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

def boucle(n):
  if n > 0:
    boucle(n - 1)
    print(n)

boucle(10)
```

Programme 17 – boucle3.py

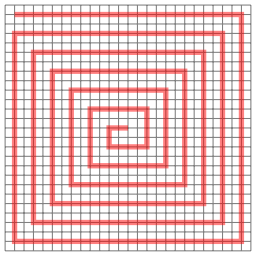
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def boucle(a, n):
  if n >= a:
    boucle(a, n - 1)
    print(n)

boucle(3, 15)
```

À titre d'exercice, on pourra s'entraîner à concevoir récursivement le dessin suivant :



8.2 Spirale « carrée »

On fixe un entier naturel p strictement positif et on recherche la plus longue spirale à côtés entiers, de longueur inférieure à p, que l'on peut construire sur le modèle ci-dessous :



(on conviendra d'un nombre donné de pixels pour une unité de longueur)

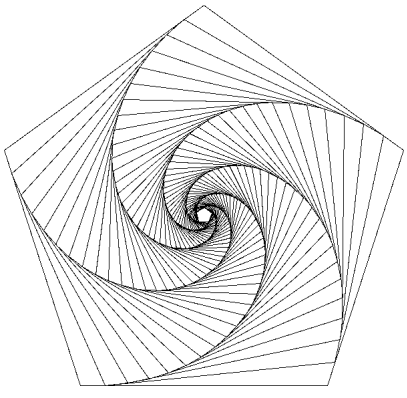
Le programme de dessin de la spirale et la détermination de la longueur de son plus côté donnent lieu à deux programmes qui illustrent parfaitement la récursivité.

8.3 Pentagones

Le programme suivant est un exemple de programmation impérative. La traduction en programmation récursive constitue un excellent exercice.

Programme 18 – penta1.py

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

from math import *
from turtle import *

def pentagone(cote):
  for n in range(5):
    forward(cote)
    left(72)

def spirale(nombre_triangles, cote):
  coeff = 0.1
  a = cote
  d = coeff * a
  for i in range(nombre_triangles):
    pentagone(a)
    d = coeff * a
    forward(d)
    a_prime = sqrt((a-d)**2 + d**2 -2*(a-d)*d*(1-sqrt(5))/4)
    angle = (180/pi) * acos( (a_prime**2 + a**2 - 2*a*d)/(2*a_prime*(a-d)) )
    left(angle)
    a = a_prime

def dessin():
  up()
  goto(-190,-210)
  down()
  spirale(50, 370)
  up()

if __name__ == "__main__":
  speed("fastest")
  dessin()

  print "Appuyez sur la touche de VALIDATION pour terminer."
  touche = str(raw_input())
```



A Le module « outils.py »

Programme 19 – outils.py

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

import sys
sys.setrecursionlimit(10000)

#
# (a, b) dans N x N*
#
def reste(a, b):
  if a < b:
    return a
  else:
    return reste(a - b, b)

#
# (a, b) dans N x N*
#
def quotient_entier(a, b):
  if a < b:
    return 0
  else:
    return 1 + quotient_entier(a - b, b)

#
# (a, b) dans N x N
#
def pgcd(a, b):
  if b == 0:
    return a
  else:
    return pgcd(b, reste(a, b))

#
# (a, b) dans N x N*
#
def fraction_cont_pos(q, a, b):
  q.append(quotient_entier(a, b))
  if reste(a, b) > 0:
    return fraction_cont_pos(q, b, reste(a, b))
  else:
    return q

#
# (a, b) dans Z x N*
#
def fraction_cont_b_pos(a, b):
  q = []
  if a < 0:
    q.append(-quotient_entier(-a + b, b))
  else:
    q.append(quotient_entier(a, b))
  return q + fraction_cont_pos([], a - b * q[0], b)[1:]

#
```

```
# (a, b) dans Z x Z*
#
def fraction_cont(a, b):
  if b < 0:
    return fraction_cont_b_pos(-a, -b)
  else:
    return fraction_cont_b_pos(a, b)

def partie_ent(chiffres, x, base):
  if x > 0:
    partie_ent(chiffres, quotient_entier(x, base), base)
    chiffres.append(str(reste(x, base)))

def partie_frac(chiffres, n, a, b, base):
  if n > 0:
    chiffres.append(str(quotient_entier(a * base, b)))
    partie_frac(chiffres, n - 1, reste(a * base, b), b, base)

def approx_pos(chiffres, p, q, nombre_chiffres, base):
  t = quotient_entier(p, q)
  if t == 0:
    chiffres.append("0")
  else:
    partie_ent(chiffres, t, base)
  if reste(p, q) > 0:
    chiffres.append(",")
    partie_frac(chiffres, nombre_chiffres, reste(p, q), q, base)
  return chiffres

def approx_q_pos(p, q, nombre_chiffres, base):
  chiffres = []
  if p < 0:
    chiffres.append("-")
    approx_pos(chiffres, -p, q, nombre_chiffres, base)
  else:
    approx_pos(chiffres, p, q, nombre_chiffres, base)
  return "".join(chiffres)

def approx(p, q, nombre_chiffres, base):
  if q < 0:
    return approx_q_pos(-p, -q, nombre_chiffres, base)
  else:
    return approx_q_pos(p, q, nombre_chiffres, base)

def approximation(p, q, nombre_chiffres =16, base =10):
  #
  # on impose la base <= 10 (pour les digits)
  #
  if base > 10:
    base = 10
  return approx(p, q, nombre_chiffres, base)

#
# tests unitaires
#
def tests_unitaires():
  #
  # quelques "constantes" utiles pour les tests
  #
```

```
115     BASE_DEUX = 2
116     BASE_DIX = 10
117     CHIFFRES_DROITE_VIRGULE = 16
118 
119     #
120     # approximation du rationnel -4565/21
121     #
122     print(approx(-4565, 21, CHIFFRES_DROITE_VIRGULE, BASE_DIX))
123 
124     #
125     # approximation du rationnel (-4565)/(-21)
126     #
127     print(approx(4565, -21, CHIFFRES_DROITE_VIRGULE, BASE_DIX))
128 
129     #
130     # expression du rationnel 4565/21 comme fraction continue
131     #
132     print(fraction_cont(4565, 21))
133 
134     #
135     # expression du rationnel -4565/21 comme fraction continue
136     #
137     print(fraction_cont(-4565, 21))
138 
139     #
140     # expression du rationnel 2/3 comme fraction continue
141     #
142     print(fraction_cont(2, 3))
143 
144     #
145     # expression du rationnel -2/3 comme fraction continue
146     #
147     print(fraction_cont(2, -3))
148 
149     #
150     # approximation du rationnel 11/10
151     #
152     print(approx(11, 10, CHIFFRES_DROITE_VIRGULE, BASE_DIX))
153 
154     #
155     # approximation du rationnel 11/10 en base 2
156     #
157     print(approx(11, 10, CHIFFRES_DROITE_VIRGULE, BASE_DEUX))
158 
159     print(approx(100, 20, CHIFFRES_DROITE_VIRGULE, BASE_DIX))
160     print(approx(21, 4565, CHIFFRES_DROITE_VIRGULE, BASE_DIX))
161     print(approximation(179, 260, 25))
162 
163 if __name__ == "__main__":
164     tests_unitaires()
```

Note : pour des raisons purement techniques, et éviter d'alourdir le code source sans offrir le moindre intérêt mathématique, le choix à été fait de limiter les conversions aux bases inférieures ou égales à 10.

Après l'étude sur le calcul du reste, du quotient... on peut s'affranchir des fonctions `reste` et `quotient_entier` définies « à la main », en utilisant celles fournies par le langage. Par exemple :

Programme 20 – outils2.py

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

import sys
sys.setrecursionlimit(10000)

#
# (a, b) dans N x N*
#
def reste(a, b):
  return (a % b)

#
# (a, b) dans N x N*
#
def quotient_entier(a, b):
  return (a // b)

#
# (a, b) dans N x N
#
def pgcd(a, b):
  if a == 0 and b == 0:
    return 1
  else:
    while b > 0:
      r = reste(a, b); a = b; b = r
    return a

#
# ... la suite du fichier est identique.
#
```

La fonction `pgcd` a également écrite de façon impérative, avec en plus un test initial, de sorte que :

$$\text{PGCD}\,(0,0) = 1\,.$$

B Le module « rationnel.py »

Programme 21 – rationnel.py

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

from outils import *

class rationnel(object):

  def __init__(self, num =0, denom =1):
    if denom == 0:
      #
      # signaler une erreur ici
      #
      num = 0
      denom = 1

    if denom < 0:
      p = -num
      q = -denom
    else:
      p = num
      q = denom

    d = pgcd(abs(p), q)

    self.__num = p // d
    self.__denom = q // d

  def __str__(self):
    if self.__denom == 1:
      return ("("+str(self.__num)+")")

    return ("("+str(self.__num)+")/("+str(self.__denom)+")")

  def __add__(self, autre):
    num = self.__num * autre.__denom + self.__denom * autre.__num
    denom = self.__denom * autre.__denom
    return rationnel(num, denom)

  def __neg__(self):
    return rationnel(-(self.__num), self.__denom)

  def __sub__(self, autre):
    num = self.__num * autre.__denom - self.__denom * autre.__num
    denom = self.__denom * autre.__denom
    return rationnel(num, denom)

  def __mul__(self, autre):
    num = self.__num * autre.__num
    denom = self.__denom * autre.__denom
    return rationnel(num, denom)

  def __truediv__(self, autre):
    if autre.__num == 0:
      #
      # signaler une erreur ici
```

```
56        #
57        return rationnel()
58      num = self.__num * autre.__denom
59      denom = self.__denom * autre.__num
60      return rationnel(num, denom)
61
62    def __pow__(self, exposant):
63      if self.__num == 0 and exposant == 0:
64        #
65        # signaler une erreur ici
66        #
67        return rationnel()
68
69      if exposant < 0:
70        return rationnel(self.__denom ** (-exposant), self.__num ** (-exposant))
71      else:
72        return rationnel(self.__num ** exposant, self.__denom ** exposant)
73
74    def approximation(self, chiffres_apres_virgule =16):
75      return approximation(self.__num, self.__denom, chiffres_apres_virgule)
76
77    def fraction_continue(self):
78      return fraction_cont(self.__num, self.__denom)
79
80  def tests_unitaires():
81    a = rationnel(28, -561)
82    print(a)
83    print(a.approximation())
84    print(a.fraction_continue())
85
86    b = rationnel(3)
87    print(b)
88
89    t = a + b
90    print(t)
91
92    t = a - b
93    print(t)
94
95    t = a * b
96    print(t)
97
98    t = a / b
99    print(t)
100   print(t.approximation(100))
101
102   t = a ** 3
103   print(t)
104   print(t.approximation(100))
105
106 if __name__ == "__main__":
107   tests_unitaires()
```

C Fractions continues : le calcul des réduites

L'intérêt historique des fractions continues réside dans l'approximation des nombres réels par des rationnels.

On suppose établie une écriture d'un nombre réel x sous la forme des quotients partiels (entiers) $(a_i)_{i\in\mathbb{N}}$:

$$x = [a_0\,,\ a_1\,,\ a_2\,,\ a_3\,,\ \ldots\,,\ a_n\,,\ \ldots]$$

Cette écriture est finie quand x est rationnel, périodique si x est un nombre irrationnel quadratique, infinie pour les nombres transcendants.

On peut procéder au calcul des réduites successives par l'« algorithme des fractions continues » ci-dessous.

On pose :

$$\forall n \in \mathbb{N},\ \begin{cases} p_{n+1} = a_n p_n + p_{n-1} \\ q_{n+1} = a_n q_n + q_{n-1} \end{cases}$$

avec les conventions : $p_{-1} = 0\,,\ p_0 = 1$ et $q_{-1} = 1\,,\ q_0 = 0\,.$

La suite des réduites successives $\left(\dfrac{p_n}{q_n}\right)_{n\in\mathbb{N}}$ converge vers x, avec : $\lim\limits_{n\to+\infty} q_n = +\infty$.

D'un point de vue purement mathématique, on peut agrémenter les deux suites $(p_n)_{n\in\mathbb{N}}$ et $(p_n)_{n\in\mathbb{N}}$ d'une troisième suite des quotients incomplets $(a_n)_{n\in\mathbb{N}}$, définie par :

$$\forall n \in \mathbb{N}, a_n = \left\lfloor -\frac{q_{n-1}x - p_{n-1}}{q_n x - p_n} \right\rfloor$$

avec la réserve $q_n x - p_n \neq 0$ (ce qui est le cas lorsque x est irrationnel).

La suite $(a_n)_{n\in\mathbb{N}}$ nécessite dans le calcul de chaque terme l'emploi du réel x avant d'évaluer la partie entière du quotient, ce qui pose la question du choix convenable d'une valeur approchée de x pour le calcul effectif restreint au type « `rationnel` ».

Exemple de calcul des premiers quotients incomplets pour $\sqrt{11}$:

i	-1	0	1	2	3	4	$\ldots$
p_i	0	1	3	10			
q_i	1	0	1	3			
a_i	$\bullet$	3	3	6			