M1 – Python
Examen

Stéphane Vialette

17 Janvier 2013

Des arbres binaires non étiquetés

Un **arbre binaire non étiqueté** est une structure qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud, le nœud initial étant appelé **racine**. Dans un arbre binaire, chaque nœud possède au plus deux fils, appelés **fils gauche** et **fils droit**. Du point de vue de ces nœuds fils, l'élément dont ils sont issus (au niveau supérieur) est appelé **père**.

À noter que si un nœud possède un unique fils, celui-ci peut être soit le fils gauche, soit le fils droit (il existe donc deux arbres binaires distincts à 2 nœuds). Un nœud de l'arbre est appelé un **nœud interne** s'il posséde au moins un fils, et une **feuille** sinon. Enfin, nos arbres binaires sont non étiquetés, les nœuds ne portent donc aucune autre information que le fait d'avoir ou non un fils à gauche et/ou à droite.

Nous prendrons comme point de départ le squelette de la classe `BinaryTree` (résidant dans un module `binarytree`) donné ci-dessous :

```
class BinaryTree(object):

    def __init__(self, left = None, right = None):
        self._left  = left   # a BinaryTree object or None
        self._right = right  # a BinaryTree object or None

    @property
    def left(self):
        return self._left

    @property
    def right(self):
        return self._right

    def __str__(self):
        return "(%s,%s)" % \
          (str(self._left)  if self._left  is not None else "()",
           str(self._right) if self._right is not None else "()")
```

```
    def __len__(self):
        left_size  = len(self._left)  if self._left  is not None else 0
        right_size = len(self._right) if self._right is not None else 0
        return 1 + left_size + right_size
```

`BinaryTree` est donc une structure récursive : les fils gauche et droit sont eux-mêmes des `BinaryTrees` (ou `None` pour un arbre vide). Un court exemple depuis ipython pour illustrer l'utilisation de la classe `BinaryTree` :

```
In [1]: from binarytree import BinaryTree

In [2]: BT = BinaryTree # pour ecrire BT au lieu de BinaryTree

In [3]: t = BT(BT(), BT())
In [4]: t
Out[4]: <binarytree.BinaryTree at 0x101d94110>

In [5]: print t
((),()),((),()))

In [6]: len(t)
Out[6]: 3

In [7]: t = BT(BT(BT(BT())))

In [8]: print t
(((((),()),()),()),())

In [9]: len(t)
Out[9]: 4

In [10]: print t.left
((((),()),()),())

In [11]: print t.right
None

In [12]: t.left = None
-----------------------------------------------
AttributeError Traceback (most recent call last)
AttributeError: can't set attribute

In [13]: t.right = None
-----------------------------------------------
AttributeError Traceback (most recent call last)
AttributeError: can't set attribute

In [14]:
```

Toutes les questions sont suivies d'un court exemple python illustrant l'utilisation de la méthode ou fonction demandée.

Diverses mesures

Nous commençons par deux méthodes bien pratiques.

Question 1 Écrire la méthode `is_leaf(self)` qui retourne `True` si le nœud d'appel est une feuille, et `False` sinon.

```
>>> from binarytree import *
>>> BT = BinaryTree
>>> t = BT(BT())
>>> t.is_leaf()
False
>>> t.left.is_leaf()
True
>>>
```

Correction :

```
class BinaryTree(object):
    ...
    def is_leaf(self):
        return self._left is None and self._right is None
```

Question 2 Écrire la méthode `all_nodes(self)` qui retourne (sous forme d'un générateur) un à un tous les nœuds de l'arbre enraciné en le nœud d'appel. Aucun ordre de parcours des nœuds de l'arbre n'est imposé.

```
>>> from binarytree import *
>>> BT = BinaryTree
>>> t = BT(BT(), BT())
>>> t
<binarytree.BinaryTree object at 0x1004a6410>
>>> t.left
<binarytree.BinaryTree object at 0x1004a6490>
>>> t.right
<binarytree.BinaryTree object at 0x1004a6510>
>>>
>>> list([node for node in t.all_nodes()])
[<binarytree.BinaryTree object at 0x1004a6490>,
 <binarytree.BinaryTree object at 0x1004a6410>,
 <binarytree.BinaryTree object at 0x1004a6510>]
>>>
```

Correction :

Choisissons, par exemple, un parours infixe. Il faut donc retourner (i) les nœuds du sous-arbre gauche (s'il y en a), (ii) le nœud courant, et (iii) les nœuds du sous-arbre gauche (s'il y en a).

```
class BinaryTree(object):
    ...
    def all_nodes(self):
        if self.left is not None:
            # yield all nodes in the left subtree
            for node in self.left.all_nodes():
                yield node

        # yield the root
        yield self

        if self.right is not None:
            # yield all nodes in the right subtree
            for node in self.right.all_nodes():
                yield node
```

Diverses méthodes pour compter des nœuds dans un arbre binaire.

Question 3 Écrire la méthode `number_of_nodes(self)` qui retourne le nombre de nœuds dans le sous-arbre enraciné en le nœud d'appel.

```
>>> from binarytree import *
>>> BT = BinaryTree
>>> t = BT(BT(), BT(BT()))
>>> t.number_of_nodes()
4
>>>
```

Correction :

Il existe, bien sûr, plusieurs façon d'écrire cette méthode. Vous êtes nombreux à avoir parcouru l'arbre en maintenant un compteur à jour :

```
class BinaryTree(object):
    ...
    def number_of_nodes(self):
        n = 1
        if self.left is not None:
            n = n + self.left.number_of_nodes()
        elif self.right is not None:
```

```
            n = n + self.right.number_of_nodes()
        return n
```

L'utilisation de la méthode `all_nodes` peut grandement simplifier l'écriture :

```
class BinaryTree(object):
    ...
    def number_of_nodes(self):
      return len([node for node in self.all_nodes()])
```

On peut aussi remarquer que la méthode `number_of_nodes` est en fait la méthode `__len__` donnée dans le squelette de la classe `BinaryTree` :

```
class BinaryTree(object):
  ...
  def number_of_nodes(self):
    return len(self)
```

ou plus simplement :

```
class BinaryTree(object):
    ...
    number_of_nodes = __len__
```

Question 4 Écrire la méthode `number_of_internal_nodes(self)` qui retourne le nombre de nœuds internes dans le sous-arbre enraciné en le nœud d'appel.

```
>>> from binarytree import *
>>> BT = BinaryTree
>>> t = BT(BT(), BT(BT()))
>>> t.number_of_internal_nodes()
2
>>>
```

Correction :

Ici, vous êtes très peu nombreux à avoir réutiliser les méthodes des questions précédentes. Effectivement, en parcourant une nouvelle fois l'arbre "à la main", il vient :

```
class BinaryTree(object):
    ...
    def number_of_internal_nodes(self):
      if self.left is not None or self.right is not None:
        n_left = 0
        if self.left is not None:
            n_left = self.left.number_of_internal_nodes()
        n_right = 0
```

```
        if self.right is not None:
            n_right = self.right.number_internal_nodes()
        return 1 + n_left + n_right
      else:
        return 0
```

Le premier test se simplifie en utilisant la méthode `is_leaf` :

```
class BinaryTree(object):
    ...
    def number_of_internal_nodes(self):
      if not self.is_leaf():
        n_left = 0
        if self.left is not None:
            n_left = self.left.number_of_internal_nodes()
        n_right = 0
        if self.right is not None:
            n_right = self.right.number_internal_nodes()
        return 1 + n_left + n_right
      else:
        return 0
```

Que cela est laborieux ! Le plus simple est de remarquer que nous disposons déjà d'un itérateur sur l'arbre binaire :

```
class BinaryTree(object):
  ...
  def number_of_internal_nodes(self):
    n = 0
    for node in self.all_nodes():
      if not node.is_leaf():
        n = n + 1
    return 0
```

qui se simplifie à l'aide d'une compréhension de liste :

```
class BinaryTree(object):
  ...
  def number_of_internal_nodes(self):
    return len([node for node in self.all_nodes()
                if not node.is_leaf()])
```

Question 5 Écrire la méthode `number_of_leaves(self)` qui retourne le nombre de feuilles dans le sous-arbre enraciné en le nœud d'appel.

```
>>> from binarytree import *
>>> BT = BinaryTree
>>> t = BT(BT(), BT(BT()))
>>> t.number_of_leaves()
```

```
2
>>>
```

Correction :

Cf. Question 4 :

```
class BinaryTree(object):
  ...
  def number_of_leaves(self):
    return len([node for node in self.all_nodes()
                if node.is_leaf()])
```

Il a été également proposé :

```
class BinaryTree(object):
  ...
  def number_of_leaves(self):
    return self.number_of_nodes() - self.number_of_internal_nodes()
```

qui est correct mais moins efficace car nécessitant 2 parcours de l'arbre.

Le **niveau** d'un nœud dans un arbre est la distance qui le sépare de la racine. Par convention, la racine est au niveau 1 (ses éventuels fils sont donc eux au niveau 2, . . .).

Question 6 Écrire la méthode `number_of_nodes_at_level(self, level)` qui retourne le nombre de nœuds au niveau **level** dans le sous-arbre enraciné en le nœud d'appel. Il pourra être utile ici de définir une méthode auxiliaire.

```
>>> from binarytree import *
>>> BT = BinaryTree
>>> t = BT(BT(), BT(BT()))
>>> t = BT(BT(), BT(BT()))
>>> t.number_of_nodes_at_level(1)
1
>>> t.number_of_nodes_at_level(2)
2
>>> t.number_of_nodes_at_level(3)
1
>>> t.number_of_nodes_at_level(4)
0
>>>
```

Correction :

Une version avec une méthode auxiliaire pour simplifier l'écriture :

```python
class BinaryTree(object):
  ...
  def _number_of_nodes_at_level(self, level, current_level):
    if current_level == level:
      # done, no need to further explore the tree rooted at self.
      return 1

    left = 0
    if self.left is not None:
      left = self.left._number_of_nodes_at_level(level, current_level + 1)

    right = 0
    if self._right is not None:
      right = self.right._number_of_nodes_at_level(level, current_level + 1)

    return left + right

  def number_of_nodes_at_level(self, level):
    return self._number_of_nodes_at_level(level, 1)
```

De la forme

Un arbre binaire est **symétrique** si le sous-arbre gauche de la racine est l'image miroir du sous-arbre droit de la racine. L'image miroir d'un arbre binaire est l'arbre binaire obtenu en permutant la gauche et la droite, c'est-à-dire en permutant récursivement les sous-arbres gauches et droits. Un arbre binaire avec un unique nœud est donc symétrique. Par exemple, `BT(BT(BT()),BT(None, BT()))` est symétrique, alors que `BT(BT(BT()),BT(BT()))` ne l'est pas.

Question 7 Écrire la méthode `is_symmetric(self)` qui retourne `True` si l'arbre enraciné en le nœud d'appel est symétrique, et `False` sinon. Il pourra être utile ici de définir une fonction auxiliaire `mirror(bt1, bt2)` qui retourne `True` si l'arbre binaire `bt1` est l'image mirroir de l'arbre binaire `bt1`, et `False` sinon.

```
>>> from binarytree import *
>>> BT = BinaryTree
>>> t = BT()
>>> t.is_symmetric()
True
>>> t = BT(BT())
>>> t.is_symmetric()
False
>>> t = BT(None, BT())
>>> t.is_symmetric()
False
```

```
>>> t = BT(BT(), BT())
>>> t.is_symmetric()
True
>>> t = BT(BT(BT()), BT(None, BT()))
>>> t.is_symmetric()
True
>>> t = BT(BT(BT(None, BT())), BT(None, BT(BT())))
>>> t.is_symmetric()
True
>>>
```

Correction :

Commençons par écrire une fonction `mirror` pour tester si 2 arbres binaires sont en mirroir. La procédure est très simple : (i) 2 arbres binaires "vides" sont en mirroir, et (ii) 2 arbres binaires "non vides" sont en mirroir si les sous-arbres sont en mirroir :

```
def mirror(binary_tree_1, binary_tree_2):
  if binary_tree_1 is None and binary_tree_2 is None:
    return True
  elif binary_tree_1 is None and binary_tree_2 is not None:
    return False
  elif binary_tree_1 is not None and binary_tree_2 is None:
    return False
  else:
    return mirror(binary_tree_1.left, binary_tree_2.right) and \
           mirror(binary_tree_1.right, binary_tree_2.left)
```

Avec la fonction `mirror`, tester si un arbre binaire est symétrique devient trivial :

```
class BinaryTree(object):
  ...
  def is_symmetric(self):
    return mirror(self.left, self.right)
```

Générer des arbres

Un arbre binaire est **complet** si toutes les feuilles sont au même niveau.

Question 8 Écrire la fonction `make_complete_binary_tree(level)` qui retourne un arbre binaire complet de niveau `level` (toutes les feuilles sont au niveau `level`).

```
>>> from binarytree import *
>>> BT = BinaryTree
>>> print make_complete_binary_tree(1)
((),())
>>> print make_complete_binary_tree(2)
```

```
(((),()),((),()))
>>> print make_complete_binary_tree(3)
((((),()),((),())),(((),()),((),())))
>>>
```

Correction :

Le cas de base est immédiat : au niveau 1 nous avons `BinaryTree()`. Pour des niveaux supérieurs, il suffit de construire des arbres binaires complets en décrémentant le niveau :

```
def make_complete_binary_tree(level):
  if level == 1:
    return BinaryTree()
  else:
    return BinaryTree(make_complete_binary_tree(level - 1),
                      make_complete_binary_tree(level - 1))
```

Nous avons passé ici sous silence le cas d'un niveau d'appel négatif ou nul. Il serait est néanmois très facile de gérer ce cas si le cahier des charges l'imposait :

```
def make_complete_binary_tree(level):
  if level <= 0:
    return None
  elif level == 1:
    return BinaryTree()
  else:
    return BinaryTree(make_complete_binary_tree(level - 1),
                      make_complete_binary_tree(level - 1))
```

Une remarque important maintenant. Il a été souvent proposé dans vos copies quelque chose qui se ramène en fait à :

```
def make_complete_binary_tree(level):
  if level == 1:
    return BinaryTree()
  else:
    bt = make_complete_binary_tree(level - 1)
    return BinaryTree(bt, bt)
```

Attention, les sous-arbres gauche et droit sont ici les **mêmes** sous-arbres (*i.e.*, les **mêmes** objets), si vous modifiez par la suite l'un des sous-arbres vous modifiez également l'autre sous-arbre. Bref, une implémentation très dangereuse !

Question 9 Écrire la fonction `all_complete_binary_trees(start_level = 1)` qui retourne un générateur produisant tous les arbres binaires complets de niveau `start_level`, `start_level` + 1, ...

```
>>> from binarytree import *
>>> BT = BinaryTree
```

```
>>> import itertools as it
>>> def take(n, iterable):
...     "Return first n items of the iterable as a list"
...     return list(it.islice(iterable, n))
...
>>> for t in take(3, all_complete_binary_trees(1)):
...     print t
...
((),())
(((),()),((),()))
((((),()),((),())),(((),()),((),())))
>>>
```

Correction :

Avec la fonction `make_complete_binary_tree` en main, l'écriture de la fonction `all_complete_binary_trees` ne pose aucune difficulté particulière :

```
def all_complete_binary_trees(start_level = 1):
  level = start_level
  while True:
    yield make_complete_binary_tree(level)
    level = level + 1
```

Intéressons nous aux arbres binaires à n nœuds, $n \geq 1$ étant un entier fixé. Clairement, il existe un unique arbre binaire avec 1 nœud (`BT()` pour nous). Il existe 2 arbre binaires avec 2 nœuds (`BT(BT())`, et `BT(None, BT())` pour nous), et 5 arbres binaires avec 3nœuds (`BT(BT(BT()))`, `BT(BT(None, BT()))`, `BT(BT(), BT())`, `BT(None, BT(BT()))`, et `BT(None, BT(None, BT()))` pour nous). Consulter la Figure 1

Plus généralement, pour générer tous les arbres binaires à n nœuds, il suffit de générer, pour tout $0 \leq k \leq n-1$, tous les arbres binaires dont le sous-arbre gauche de la racine comporte k nœuds et le sous-arbre droit de la racine comporte $n-k-1$ nœuds.

Question 10 Écrire la fonction `all_binary_trees(n)` qui retourne un générateur produisant tous les arbres binaires à n nœuds.

```
>>> from binarytree import *
>>> BT = BinaryTree
>>> for t in all_binary_trees(3):
...     print t
...
((),(((),()),()))
((),((),((),())))
(((),()),((),()))
((((),()),()),())
```

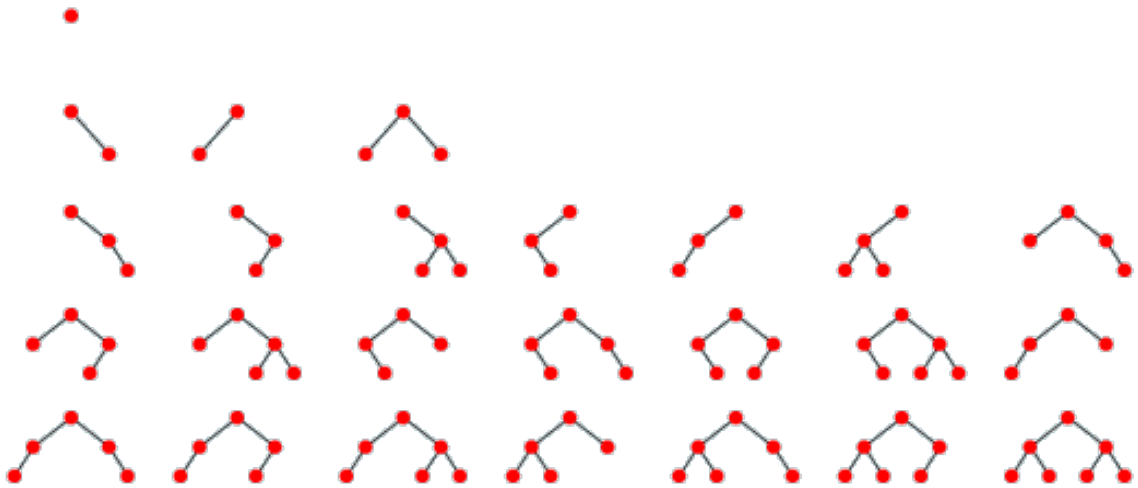

FIGURE 1 – Quelques arbres binaires.

```
(((),(),())),())
>>>
```

Correction :

Une lecture – presque mot à mot – du dernier paragraphe avant la question 10 se traduit en python par :

```
def all_binary_trees(n):
  if n == 0:
    yield None
  elif n == 1:
    yield BinaryTree()
  elif n ==2:
    yield BinaryTree(BinaryTree())
    yield BinaryTree(None, BinaryTree())
  else:
    for k in xrange(n):
      for left in all_binary_trees(k):
        for right in all_binary_trees(n-k-1):
          yield BinaryTree(left, right)
```

Question 11 Écrire la fonction `all_truly_binary_trees(n)` qui retourne un générateur produisant tous les arbres binaires à n nœuds dont tous les nœuds internes ont exactement 2 fils.

```
>>> from binarytree import *
>>> BT = BinaryTree
>>> for t in all_truly_binary_trees(3):
```

```
...     print t
...
(((),()),((),()))
>>> for t in all_truly_binary_trees(4):
...     print t
...
>>> for t in all_truly_binary_trees(5):
...     print t
...
(((),()),(((),()),((),())))
((((),()),((),())),((),()))
>>>
```

Correction :

C'est en première lecture une simple variation de la question 10. Il faut néanmois être attentif à la parité. (C'est probablement l'unique question de cet examen qui nécessite une petite réflexion algorithmique . . . petite réflexion algorithmique puisqu'il est explicitement mentionné en exemple que l'appel à `all_truly_binary_trees(4)` ne produit aucun arbre binaire.) Ce problème de parité régle, le code ne pose plus vraiment de difficulté :

```
def all_truly_binary_trees(n):
  if n % 2 == 0:
    raise StopIteration

  if n == 1:
    yield BinaryTree()
  elif n == 3:
    yield BinaryTree(BinaryTree(), BinaryTree())
  else:
    for k in xrange(1, n-1, 2):
      for left in all_truly_binary_trees(k):
        for right in all_truly_binary_trees(n-k-1):
          yield BinaryTree(left, right)
```

Soient t et t' deux arbres binaires. Soit u un nœud de t qui ne possède pas de fils gauche. **Greffer à gauche** t' en u consiste à ajouter t' comme nouveau fils gauche de u. Soit v un nœud de t qui ne possède pas de fils droit. **Greffer à droite** t' en v consiste à ajouter t' comme nouveau fils droit de v. Remarquons qu'une feuille peut être greffée à gauche et à droite. Un arbre binaire t' est greffé à gauche (resp. à droite) à un arbre binaire t s'il est greffé à gauche (resp. à droite) à un nœud de t. Enfin, un arbre binaire t' est greffé à un arbre binaire t s'il est greffé à gauche ou à droite à t.

Question 12 Definir au préalable l'exception `BinaryTreeException`.

– Écrire la methode `attach_left(self, t)` de la classe `BinaryTree` qui fixe l'arbre binaire `t` comme nouveau fils gauche du nœud appelant si son fils gauche est au moment de l'appel l'arbre vide, et lève l'exception `BinaryTreeException` sinon.

– Écrire la methode `attach_right(self, t)` de la classe `BinaryTree` qui fixe l'arbre binaire `t` comme nouveau fils droit du nœud appelant si son fils droit est au moment de l'appel l'arbre vide, et lève l'exception `BinaryTreeException` sinon.
– Écrire ensuite la fonction `all_grafted_trees(bt1, bt2)` qui retourne un générateur produisant tous les arbres binaires obtenus en greffant l'arbre binaire `bt1` à l'arbre binaire `bt2` ou en greffant l'arbre binaire `bt2` à l'arbre binaire `bt1`.

```
>>> from binarytree import *
>>> BT = BinaryTree
>>> t1 = BT()
>>> print t1
((),())
>>> t2 = BT(BT())
>>> print t2
(((),()),())
>>> for t in all_grafted_trees(t1, t2):
...     print t
...
((((),()),()),())
((),(((),()),()))
((((),()),()),())
(((),((),())),())
(((),()),((),()))
>>>
```

Correction :

```
def all_grafted_trees_aux(bt1, bt2):
  for node in bt1.all_nodes():
    if node.left is None:
      # node.left does not allow write access: one must use self._left
      node._left = bt2
      yield bt1
      node._left = None
    if node.right is None:
      # node.left does not allow write access: one must use self._right
      node._right = bt2
      yield bt1
      node._right = None

def all_grafter_trees(bt1, bt2):
    for bt in all_grafted_trees_aux(bt1, bt2):
        yield bt
    for bt in all_grafted_trees_aux(bt2, bt1):
        yield bt
```