TP1

September 4, 2017

1 Table of Contents

2 TP 1 - Programmation pour la préparation à l'agrégation maths option info

- En Python 3, avec des annotations de types.
- Notez que ce ne sont que des annotations.
- Pour plus, il faudrait utiliser quelque chose comme `beartype`.

3 Fonctions

3.1 Exercice 4

```
In [17]: def successeur(i : int) -> int:
             assert isinstance(i, int)
             assert i >= 0
             return i + 1
```

```
In [4]: successeur(3)
        successeur(2 * 2)
        successeur(2.5)
```

```
Out[4]: 4
```

```
Out[4]: 5
```

```
        ---------------------------------------------------------------------------
        AssertionError                            Traceback (most recent call last)

        <ipython-input-4-a2f196c8f8ad> in <module>()
          1 successeur(3)
          2 successeur(2 * 2)
    ----> 3 successeur(2.5)


        <ipython-input-2-7320441638ec> in successeur(i)
          1 def successeur(i:int):
    ----> 2     assert isinstance(i, int)
          3     return i + 1


        AssertionError:
```

3.2 Exercice 5

```
In [9]: def produit3(x, y, z):
            return x * y * z
```

```
produit3_2 = lambda x, y, z: x * y * z

produit3_3 = lambda x: lambda y: lambda z: x * y * z
```

```
In [12]: produit3(1, 2, 3)
         produit3_2(1, 2, 3)
         produit3_3(1)(2)  # lambda z : 1 * 2 * z
         produit3_3(1)(2)(3)
```

```
Out[12]: 6
```

```
Out[12]: 6
```

```
Out[12]: <function __main__.<lambda>.<locals>.<lambda>.<locals>.<lambda>>
```

```
Out[12]: 6
```

3.3 Exercice 6

```
In [18]: def exercice6(n : int) -> None:
             for i in range(1, n + 1):
                 print(i)
             for i in range(n, 0, -1):
                 print(i)
```

```
In [19]: exercice6(4)
```

```
1
2
3
4
4
3
2
1
```

4 Récursivité

4.1 Exercice 7

```
In [20]: def factorielle(n : int) -> int:
             if n <= 0:
                 return 0
             elif n == 1:
                 return 1
             else:
                 return n * factorielle(n - 1)
```

```
In [25]: for i in range(8):
             print("{}! = {:>5}".format(i, factorielle(i)))
```

```
0! =     0
1! =     1
2! =     2
3! =     6
4! =    24
5! =   120
6! =   720
7! =  5040
```

4.2 Exercice 8

```
In [66]: def pgcd(x : int, y : int, log: bool=False) -> int:
             if log: print("x = {:>7}, y = {:>7}".format(x, y))
             assert x >= 0 and y >= 0
             if y == 1:
                 return 1
             elif y == x or y == 0:
                 return x
             if x < y:
                 return pgcd(x, y % x, log=log)
             else:
                 return pgcd(y, x % y, log=log)
```

```
In [67]: pgcd(10, 5)
```

```
Out[67]: 5
```

```
In [71]: from sympy import gcd
         from random import randint
         for _ in range(10):
             x = randint(1, 100)
             y = randint(1, 100)
             d = pgcd(x, y)
             print("{:>2} ^ {:>2} = {:>2}".format(x, y, d))
             assert d == gcd(x, y)
```

```
23 ^ 42 =  1
48 ^ 57 =  3
47 ^ 48 =  1
12 ^ 12 = 12
86 ^ 37 =  1
79 ^ 87 =  1
72 ^ 41 =  1
65 ^ 30 =  5
96 ^ 38 =  2
90 ^  5 =  5
```

4.3 Exercice 9

```
In [72]: def fibonacci(n : int) -> int:
             assert n >= 0
             if n == 0 or n == 1:
                 return 1
             else:
                 return fibonacci(n - 1) + fibonacci(n - 2)

In [76]: %timeit fibonacci(35)

4.15 s ś 23.4 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)


In [77]: def fibonacci_lin(n : int) -> int:
             un, uns = 1, 1
             for _ in range(n):
                 un, uns = uns, un + uns
             return un

In [79]: for i in range(10):
             assert fibonacci(i) == fibonacci_lin(i)

In [80]: %timeit fibonacci_lin(35)

1.79 ţs ś 27.3 ns per loop (mean ś std. dev. of 7 runs, 1000000 loops each)
```

Voilà la différence.

4.4 Exercice 10

Aucune hypothèse n'est faite sur les arguments de la fonction, on supposera seulement qu'elle est itérable sur sa sortie.

```
In [44]: from types import FunctionType

         # première version, f : type x -> type x (simple)
         def itere(f : FunctionType, n : int) -> FunctionType:
             def fn(x):
                 for _ in range(n):
                     x = f(x)
                 return x
             return fn

In [45]: itere(lambda x: x + 1, 10)(1)

Out[45]: 11
```

```
In [119]: # deuxième version, f : tuple -> tuple (simple)
          def itere2(f : FunctionType, n : int) -> FunctionType:
              def fn(*args):
                  for _ in range(n):
                      if isinstance(args, tuple):
                          args = f(*args)
                      else:
                          args = f(args)
                  if isinstance(args, tuple) and len(args) == 1:
                      return args[0]
                  else:
                      return args
              return fn

In [132]: def plusun(x):
              return x + 1

          itere2(plusun, 10)(1)

          def foisdeux(x, y):
              return x * 2, y * 2

          itere2(foisdeux, 10)(1, 2)

Out[132]: 11

Out[132]: (1024, 2048)
```

4.5 Exercice 11

```
In [144]: def hanoi(x : str, y : str, z : str, n : int) -> None:
              def hanoiaux(orig : str, dest : str, inter : str, n : int):
                  if n == 0:
                      return 0
                  c1 = hanoiaux(orig, inter, dest, n - 1)
                  print("{} -> {}".format(orig, dest))
                  c2 = hanoiaux(inter, dest, orig, n - 1)
                  return c1 + 1 + c2
              return hanoiaux(x, z, y, n)

In [145]: hanoi("T1", "T2", "T3", 1)

T1 -> T3


Out[145]: 1

In [146]: hanoi("T1", "T2", "T3", 2)
```

```
T1 -> T2
T1 -> T3
T2 -> T3
```

```
Out[146]: 3
```

```
In [148]: hanoi("T1", "T2", "T3", 5)  # 31 = 2^5 - 1
```

```
T1 -> T3
T1 -> T2
T3 -> T2
T1 -> T3
T2 -> T1
T2 -> T3
T1 -> T3
T1 -> T2
T3 -> T2
T3 -> T1
T2 -> T1
T3 -> T2
T1 -> T3
T1 -> T2
T3 -> T2
T1 -> T3
T2 -> T1
T2 -> T3
T1 -> T3
T2 -> T1
T3 -> T2
T3 -> T1
T2 -> T1
T2 -> T3
T1 -> T3
T1 -> T2
T3 -> T2
T1 -> T3
T2 -> T1
T2 -> T3
T1 -> T3
```

```
Out[148]: 31
```

---

5 Listes

5.1 Exercice 12

Les listes en Python sont des `list`. Elles ne fonctionnent **pas** comme des listes simplement chaînées comme en Caml.

```
In [152]: def concatene(liste1 : list, liste2 : list) -> list:
              # return liste1 + liste2  # solution facile
              n1, n2 = len(liste1), len(liste2)
              res = []
              for i in range(n1 + n2):
                  if i < n1:
                      res.append(liste1[i])
                  else:
                      res.append(liste2[i - n1])
              return res
```

```
In [153]: concatene([1, 2, 3], [4, 5])
```

```
Out[153]: [1, 2, 3, 4, 5]
```

5.2 Exercice 13

```
In [163]: from types import FunctionType

          def applique(f : FunctionType, liste : list) -> list:
              res = [f(x) for x in liste]  # solution facile
              res = map(f, liste)  # encore plus facile
              # en itérant la liste :
              res = []
              for x in liste:
                  res.append(f(x))
              # en itérant ses valeurs :
              res = []
              for i in range(len(liste)):
                  res.append(f(liste[i]))
              return res
```

```
In [164]: applique(lambda x : x + 1, [1, 2, 3])
```

```
Out[164]: [2, 3, 4]
```

5.3 Exercice 14

```
In [165]: def liste_carree(liste : list) -> list:
              return applique(lambda x: x**2, liste)
```

```
In [167]: liste_carree([1, 2, 3])
```

```
Out[167]: [1, 4, 9]
```

5.4 Exercice 15

```
In [180]: def miroir_quad(liste : list) -> list:
              # aucune idée pour le faire innefficacement en Python,
              # ce ne sont pas des listes chaînées
              return liste[::-1]

          def miroir_lin(liste : list) -> list:
              return [liste[-i] for i in range(1, len(liste) + 1)]

In [181]: miroir_lin([1, 2, 3])

Out[181]: [3, 2, 1]
```

5.5 Exercice 16

```
In [186]: def insertion_dans_liste_triee(entiers : list, x : int) -> list:
              i = 0
              while i < len(entiers) and entiers[i] < x:
                  i += 1
              return entiers[:i] + [x] + entiers[i:]

In [187]: insertion_dans_liste_triee([1, 2, 5, 6], 4)

Out[187]: [1, 2, 4, 5, 6]

In [190]: def tri_insertion(entiers : list) -> list:
              assert all([isinstance(i, int) for _ in entiers])
              def trix(e : list, acc : list) -> list:
                  if len(e) == 0: return acc
                  else: return trix(e[1:], insertion_dans_liste_triee(acc, e[0]))
              return trix(entiers, [])

In [191]: tri_insertion([5, 2, 6, 1, 4])

Out[191]: [1, 2, 4, 5, 6]
```

5.6 Exercice 17

```
In [193]: def ordre_decroissant(x, y):
              return x > y

In [195]: from types import FunctionType

          def insertion_dans_liste_triee2(entiers : list, x : int, ordre : FunctionType) -> lis
              i = 0
              while i < len(entiers) and ordre(entiers[i], x):
                  i += 1
              return entiers[:i] + [x] + entiers[i:]
```

```
In [196]: insertion_dans_liste_triee2([6, 5, 2, 1], 4, ordre_decroissant)

Out[196]: [6, 5, 4, 2, 1]

In [197]: def tri_insertion2(entiers : list, ordre : FunctionType) -> list:
              assert all([isinstance(i, int) for _ in entiers])
              def trix(e : list, acc : list) -> list:
                  if len(e) == 0: return acc
                  else: return trix(e[1:], insertion_dans_liste_triee2(acc, e[0], ordre))
              return trix(entiers, [])

In [200]: tri_insertion([5, 2, 6, 1, 4])
          tri_insertion2([5, 2, 6, 1, 4], ordre_decroissant)

Out[200]: [1, 2, 4, 5, 6]

Out[200]: [6, 5, 4, 2, 1]
```

---

6 Exponentiation rapide

6.1 Exercice 18

```
In [3]: def exp(x : int, n : int) -> int:
            res = 1
            for _ in range(n):
                res *= x
            return res

In [4]: x = 3
        for n in range(8):
            print("  {} ** {} = {:>5}".format(x, n, exp(x, n)))

  3 ** 0 =     1
  3 ** 1 =     3
  3 ** 2 =     9
  3 ** 3 =    27
  3 ** 4 =    81
  3 ** 5 =   243
  3 ** 6 =   729
  3 ** 7 =  2187
```

6.2 Exercice 19

```
In [201]: def exp2(x : int, n : int) -> int:
              assert n >= 0
              if n == 0:
```

```
            return 1
        elif n == 1:
            return x
        elif n % 2 == 0:
            return exp2(x ** 2, n // 2)
        elif n % 2 == 1:
            return exp2(x ** 2, (n - 1) // 2) * x
```

```
In [205]: x = 3
          for n in range(8):
              print("  {} ** {} = {:>5}".format(x, n, exp2(x, n)))
```

```
  3 ** 0 =     1
  3 ** 1 =     3
  3 ** 2 =     9
  3 ** 3 =    27
  3 ** 4 =    81
  3 ** 5 =   243
  3 ** 6 =   729
  3 ** 7 =  2187
```

6.3 Exercice 20

On ne dispose pas de typage pour faire ça aussi joliment qu'en Caml...

```
In [6]: def mult_float(x : float, y : float) -> float:
            return x * y

        monoide_flottants = {
            'mult': mult_float,   # (*.) en Caml
            'neutre': 1.
        }
```

```
In [15]: import numpy as np

         def mult_ndarray(A : np.array, B : np.array) -> np.array:
             return np.dot(A, B)

         # Pas possible d'avoir un neutre pour une taille quelconque !
         monoide_nparray = lambda n, m: {
             'mult': mult_ndarray,
             'neutre': np.eye(n, m)
         }
```

```
In [16]: mult_ndarray([[1, 1], [1, 1]], [[1, 2], [3, 4]])
```

```
Out[16]: array([[4, 6],
                [4, 6]])
```

Manuellement ce n'est pas trop dur :

```
In [31]: def mult_mat(A : list, B : list) -> list:
             n, m = len(A), len(A[0])  # A est (n, m)
             m2, p = len(B), len(B[0]) # B est (m2, p)
             assert m == m2
             C = [[0 for _ in range(p)] for _ in range(n)]  # C est (n, p)
             for i in range(n):
                 for j in range(p):
                     for k in range(m):
                         C[i][j] += A[i][k] * B[k][j]
             return C

         # Pas possible d'avoir un neutre pour une taille quelconque !
         monoide_mat = lambda n, m: {
             'mult': mult_mat,
             'neutre': [[int(i==j) for j in range(m)] for i in range(n)]  # I est (n, m)
         }

In [32]: mult_mat([[1, 1], [1, 1]], [[1, 2], [3, 4]])

Out[32]: [[4, 6], [4, 6]]
```

6.4 Exercice 21

```
In [33]: def exp_rapide(monoide : dict, x, n : int):
             mult = monoide['mult']
             assert n >= 0
             if n == 0:
                 return monoide['neutre']
             elif n == 1:
                 return x
             elif n % 2 == 0:
                 return exp_rapide(monoide, mult(x, x), n // 2)
             elif n % 2 == 1:
                 return mult(exp_rapide(monoide, mult(x, x), (n - 1) // 2), x)
```

6.5 Exercice 22

```
In [34]: def exp_rapide_float(x : float, n : int) -> float:
             return exp_rapide(monoide_flottants, x, n)

In [35]: exp_rapide_float(2.0, 8)

Out[35]: 256.0

In [36]: exp_rapide_float(0.2, 8)

Out[36]: 2.560000000000002e-06
```

Et pour les matrices, un petit piège à cause des tailles :

```
In [37]: def exp_rapide_mat(A : list, k : int) -> float:
             n, m = len(A), len(A[0])
             mono = monoide_mat(n, m)
             return exp_rapide(mono, A, k)

In [38]: for k in range(5):
             exp_rapide_mat([[1, 1], [1, 1]], k)

Out[38]: [[1, 0], [0, 1]]

Out[38]: [[1, 1], [1, 1]]

Out[38]: [[2, 2], [2, 2]]

Out[38]: [[4, 4], [4, 4]]

Out[38]: [[8, 8], [8, 8]]

In [42]: for k in range(5):  # nilpotente
             exp_rapide_mat([[0, 1, 2], [0, 0, 1], [0, 0, 0]], k)

Out[42]: [[1, 0, 0], [0, 1, 0], [0, 0, 1]]

Out[42]: [[0, 1, 2], [0, 0, 1], [0, 0, 0]]

Out[42]: [[0, 0, 1], [0, 0, 0], [0, 0, 0]]

Out[42]: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

Out[42]: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

6.6 Exercice 23

```
In [48]: from operator import mul

         def exp_rapide_2(x, n : int):
             def mul_par_x(y):
                 return x * y
             return itere(mul_par_x, n)(x)

In [49]: exp_rapide_2(2.0, 8)

Out[49]: 512.0
```

---

7 Formule du calcul propositionnel

Spoiler alert : en Python, c'est chaud.

On va faire des choses non typées, avec des dictionnaires, pour bidouiller.

7.1 Exercice 24

`(not p ^ ((q ^ not p) v (r v q)))` s'écrit en Caml `(Conj(Not(V("p")),Disj(Conj(V("q"),Not(V("p"))),`

On va imbriquer des dictionnaires, les types, `V`, `Not`, `Conj` et `Disj` seront des clés, et les valeurs seront des formules ou couples de formules. Mais on va cacher ça et l'utilisateur verra exactement la même chose qu'en Caml !

```
In [63]: V = lambda x: {'V': x}
         Not = lambda x: {'Not': x}
         Disj = lambda x, y: {'Disj': (x, y)}
         Conj = lambda x, y: {'Conj': (x, y)}

         p = V('p')
         r = V('r')
         q = V('q')
         not_p = Not(p)

         f = Conj(Not(p), Disj(Conj(q, not_p), Disj(r, q)))
         f

Out[63]: {'Conj': ({'Not': {'V': 'p'}},
           {'Disj': ({'Conj': ({'V': 'q'}, {'Not': {'V': 'p'}})},
             {'Disj': ({'V': 'r'}, {'V': 'q'})})})}

In [59]: (Conj(Not(V("p")),Disj(Conj(V("q"),Not(V("p"))),Disj(V("r"),V("q")))))

Out[59]: {'Conj': ({'Not': {'V': 'p'}},
           {'Disj': ({'Conj': ({'V': 'q'}, {'Not': {'V': 'p'}})},
             {'Disj': ({'V': 'r'}, {'V': 'q'})})})}
```

7.2 Exercice 25

Ensuite on fait une filtration "à la main" sur les clés du dictionnaire (de niveau 1, on ne récurse pas quand on teste `'V' in formule`).

```
In [79]: def taille(formule : dict) -> str:
             if 'V' in formule:
                 return 0
             elif 'Not' in formule:
                 return 1 + taille(formule['Not'])
             elif 'Conj' in formule:
                 x, y = formule['Conj']
                 return 1 + taille(x) + taille(y)
             elif 'Disj' in formule:
                 x, y = formule['Disj']
                 return 1 + taille(x) + taille(y)

In [69]: taille(f)

Out[69]: 6
```

7.3 Exercice 26

```
In [109]: def formule_to_string(formule : dict) -> str:
              if 'V' in formule:
                  return formule['V']
              elif 'Not' in formule:
                  return "~" + formule_to_string(formule['Not'])
              elif 'Conj' in formule:
                  x, y = formule['Conj']
                  return "(" + formule_to_string(x) + " ^ " + formule_to_string(y) + ")"
              elif 'Disj' in formule:
                  x, y = formule['Disj']
                  return "(" + formule_to_string(x) + " V " + formule_to_string(y) + ")"

In [110]: def affiche(formule):
              print(formule_to_string(formule))

In [111]: affiche(f)

(~p ^ ((q ^ ~p) V (r V q)))
```

Et voilà. Pas tellement plus dur hein !

7.4 Exercice 27

Les valeurs des variables seront données comme un dictionnaire associant nom de variable à valeurs booléennes. Et comme on veut frimer, on prend un `defaultdict`.

```
In [115]: from collections import defaultdict

          d = defaultdict(lambda: False, {'p': True, 'q': False})
          d['p']  # -> True car présent et True
          d['q']  # -> False car présent et False
          d['x']  # -> False car absent

Out[115]: True

Out[115]: False

Out[115]: False

In [113]: valeurs_1 = defaultdict(lambda: False, {'p': True})

In [114]: valeurs_2 = defaultdict(lambda: False, {'r': True})

In [89]: def eval(valeurs : dict, formule : dict) -> bool:
             if 'V' in formule:
                 return valeurs[formule['V']]
             elif 'Not' in formule:
```

```
            return not eval(valeurs, formule['Not'])
        elif 'Conj' in formule:
            x, y = formule['Conj']
            return eval(valeurs, x) and eval(valeurs, y)
        elif 'Disj' in formule:
            x, y = formule['Disj']
            return eval(valeurs, x) or eval(valeurs, y)
```

```
In [90]: eval(valeurs_1, f)
```

```
Out[90]: False
```

```
In [91]: eval(valeurs_2, f)
```

```
Out[91]: True
```

7.5 Exercice 28

```
In [93]: def extrait_variables_x(formule : dict) -> list:
             if 'V' in formule:
                 return [formule['V']]
             elif 'Not' in formule:
                 return extrait_variables_x(formule['Not'])
             elif 'Conj' in formule:
                 x, y = formule['Conj']
                 return extrait_variables_x(x) + extrait_variables_x(y)
             elif 'Disj' in formule:
                 x, y = formule['Disj']
                 return extrait_variables_x(x) + extrait_variables_x(y)

         # on enlève les doublons
         def extrait_variables(formule : dict) -> list:
             return list(set(extrait_variables_x(formule)))
```

```
In [94]: extrait_variables(f)
```

```
Out[94]: ['p', 'r', 'q']
```

On trouve facilement les n variables d'une formule.
Ensuite, un `itertools.product` permet de générer les 2^n valuations.

```
In [98]: from itertools import product

         def toutes_valeurs(variables):
             for valeurs in product([False, True], repeat=len(variables)):
                 yield defaultdict(lambda: False, {k:v for k,v in zip(variables, valeurs)})
```

```
In [103]: list(toutes_valeurs(extrait_variables(f)))
```

```
Out[103]: [defaultdict(<function __main__.toutes_valeurs.<locals>.<lambda>>,
                       {'p': False, 'q': False, 'r': False}),
           defaultdict(<function __main__.toutes_valeurs.<locals>.<lambda>>,
                       {'p': False, 'q': True, 'r': False}),
           defaultdict(<function __main__.toutes_valeurs.<locals>.<lambda>>,
                       {'p': False, 'q': False, 'r': True}),
           defaultdict(<function __main__.toutes_valeurs.<locals>.<lambda>>,
                       {'p': False, 'q': True, 'r': True}),
           defaultdict(<function __main__.toutes_valeurs.<locals>.<lambda>>,
                       {'p': True, 'q': False, 'r': False}),
           defaultdict(<function __main__.toutes_valeurs.<locals>.<lambda>>,
                       {'p': True, 'q': True, 'r': False}),
           defaultdict(<function __main__.toutes_valeurs.<locals>.<lambda>>,
                       {'p': True, 'q': False, 'r': True}),
           defaultdict(<function __main__.toutes_valeurs.<locals>.<lambda>>,
                       {'p': True, 'q': True, 'r': True})]
```

```
In [106]: def str_of_bool(x : bool) -> str:
              # return str(int(x))
              return '1' if x else '0'
```

```
In [116]: def table_verite(formule : dict) -> None:
              variables = extrait_variables(formule)
              # D'abord la formule
              for k in variables:
                  print(k, end=' ')
              print('| ', end='')
              affiche(f)
              # Puis toutes ces valeurs possibles
              for valeurs in toutes_valeurs(variables):
                  for k in variables:
                      print(str_of_bool(valeurs[k]), end=' ')
                  print('| ', end='')
                  print(str_of_bool(eval(valeurs, formule)))
```

```
In [117]: table_verite(f)
```

```
p r q | (~p ^ ((q ^ ~p) V (r V q)))
0 0 0 | 0
0 0 1 | 1
0 1 0 | 1
0 1 1 | 1
1 0 0 | 0
1 0 1 | 0
1 1 0 | 0
1 1 1 | 0
```

Note : ce code est encore plus concis que celui donné dans la solution en Caml.

On peut vérifier, par exemple sur Wolfram | Alpha que l'on obtient bien le bon résultat...

---

8 Conclusion

Comme vous le voyez, on arrive à répondre aux mêmes questions dans les deux langages, et il n'y a pas de grosses différences en pratique dans la mise en oeuvre.

Là où Caml excelle pour les types définis, le filtrage et la récursion, Python gagne en simplicité sur l'affichage, sa librairie standard et les dictionnaires et ensembles...