tutoriel_intermediaire_1

June 29, 2019

1 Tutoriel Intermédiaire 1: boucles, contrôle de flux et numpy

1.1 1.1 Numpy

L'objet principal du module *numpy* est le tableau multidimensionnel(array multidimensionnelle), qui s'appelle un ndarray. Les ndarrays peuvent également être considérées des arrays. En fait. numpy.array est une fonction intuitive pour créer une ndarray.
Donc, une ndarray devrait principalement être créée avec les fonctions *array*, *zeros* ou *empty* selon la documentation de numpy.

```
In [1]: import numpy as np

        # instancier une ndarray en utilisant la fonction array (et des listes)
        arr = np.array([[1,2,3],[10,11,12]])
        arr, type(arr)
```

```
Out[1]: (array([[ 1,  2,  3],
                [10, 11, 12]]), numpy.ndarray)
```

```
In [2]: # instancier une ndarray en utilisant la fonction zeros
        zeros_arr = np.zeros((3,2)) # l'entrant est un tuple représentant le nombre de rangées
          # et de colonnes dans notre cas
        zeros_arr
```

```
Out[2]: array([[0., 0.],
               [0., 0.],
               [0., 0.]])
```

```
In [3]: # instancie une ndarray en utilisant la fonction empty
        empty_arr = np.empty((3,2))
        empty_arr
```

```
Out[3]: array([[0., 0.],
               [0., 0.],
               [0., 0.]])
```

Les attributes principaux d'une ndarray sont
* ndim: le nombre d'axes (dimensions) de l'array * shape: la forme de l'array (toutes

ses dimensions) (pour une array de forme n*m, la dimension est le tuple (n,m))* size: le nombre d'éléments dans l'array (n fois m) * dtype: tous les éléments dans une array numpy ont le même type, alors cet attribut retourne le type de ces éléments

```
In [4]: {'ndim': arr.ndim, 'shape': arr.shape, 'size': arr.shape, 'dtype': arr.dtype}
```

```
Out[4]: {'ndim': 2, 'shape': (2, 3), 'size': (2, 3), 'dtype': dtype('int32')}
```

Nous pouvons utiliser des opérations arithmétiques sur les arrays numpy, mais ces opérations sont appliquées *par éléments*.

```
In [5]: arr2 = np.arange(6).reshape(2,3) # la fonction 'arange' créee n éléments de 0 à n
          # ensuite on change la structure cette array avec la fonction reshape
        print(arr)
        print(arr2)
        arr + arr2
```

```
[[ 1  2  3]
 [10 11 12]]
[[0 1 2]
 [3 4 5]]
```

```
Out[5]: array([[ 1,  3,  5],
               [13, 15, 17]])
```

```
In [6]: arr * arr2
```

```
Out[6]: array([[ 0,  2,  6],
               [30, 44, 60]])
```

Il existe plusieurs fonctions unaires fonctionnant sur les arrays numpy. Par exemple, la fonction racine carrée, la fonction exponentielle et ainsi de suite.

```
In [7]: print(np.sqrt(arr))
        np.sqrt(arr)
```

```
[[1.         1.41421356 1.73205081]
 [3.16227766 3.31662479 3.46410162]]
```

```
Out[7]: array([[1.        , 1.41421356, 1.73205081],
               [3.16227766, 3.31662479, 3.46410162]])
```

Dans l'atelier précédent, nous avons vu comment l'*indexing* et le *slicing* peuvent être utilisés sur les listes et les strings. Maintenant, nous pouvons utiliser ces concepts sur les arrays de numpy.

```
In [8]: print(arr)
        print(arr[1:, 1:])
        print(arr[0, 0])
        print(arr[-1, -1])
```

```
[[ 1  2  3]
 [10 11 12]]
[[11 12]]
1
12
```

En passant, nous avons un nouveau caractère ... qui peut être particulièrement intéressant lorsque nous voulons indexer des arrays ayant plusieurs dimensions. $x[1,2...] = x[1,2,:,:,:]$

```
In [9]: arr[..., 0] # selectionne seulement la première "colonne"
```

```
Out[9]: array([ 1, 10])
```

Nous pouvons empiler des arrays ensemble afin de les agrandir. Nous pouvons utiliser la fonction *vstack* afin d'empiler nos données verticalement ou bien utiliser la fonction *hstack* afin de les empiler horizontalement.

```
In [10]: nouvelle_rangée = np.array([1000, 2000, 3000])
         np.vstack((arr, nouvelle_rangée)) # les entrants ont la forme d'un tuple
```

```
Out[10]: array([[   1,    2,    3],
                [  10,   11,   12],
                [1000, 2000, 3000]])
```

```
In [11]: nouvelle_colonne = np.array([999, 888])
         nouvelle_colonne = nouvelle_colonne.reshape(2,1)
         print(nouvelle_colonne)
         np.hstack((nouvelle_colonne, arr))
```

```
[[999]
 [888]]
```

```
Out[11]: array([[999,   1,   2,   3],
                [888,  10,  11,  12]])
```

Souvenez-vous que mis à part les *numbers*, la plupart des types en Python sont accédéss par référence et non par valeur. En d'autres mots, lorsque je réassigne une array, c'est la même array qui est passé une deuxième fois.

```
In [12]: nouvelle_array = arr
         print(arr is nouvelle_array)
         arr[0,0] = 999999
         nouvelle_array
```

```
True
```

```
Out[12]: array([[999999,      2,      3],
                [    10,     11,     12]])
```

Si vous voulez copier une array afin de modifier cette variable de façon indépendante de l'originale, vous pouvez utiliser la fonction *copy*.

```
In [13]: nouvelle_arr = arr.copy()
         print(arr is nouvelle_arr)
         arr[0,0] = 22222
         nouvelle_arr
```

```
False
```

```
Out[13]: array([[999999,      2,      3],
                [    10,     11,     12]])
```

Désormais, créons une array plus significative avec la fonction *arange*.

```
In [14]: grosse_arr = np.arange(100).reshape(10, 10)
         grosse_arr
```

```
Out[14]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
                [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
                [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
                [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
                [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
                [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
                [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
                [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
                [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

1.1.1 Exercice 1

Retrouvez la dernière rangée de la *grosse_arr*

```
In [15]: def exercice1(arr):
             x = None
             return x

         assert (exercice1(grosse_arr) == np.array([90, 91, 92, 93, 94, 95, 96, 97, 98, 99])).a
```

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-15-b9f75975153f> in <module>
```

```
      3   return x
      4
----> 5 assert (exercice1(grosse_arr) == np.array([90, 91, 92, 93, 94, 95, 96, 97, 98, 99])


AssertionError:
```

1.2 1.2 Les *if statements*

Puisque cet atelier a comme objectif de vous rendre très confortable avec les boucles en Python, le deuxième bloc nécessaire afin d'atteindre cet objectif est la compréhension des contrôles de flux et des ordres de contrôle. Nous allons commencer avec le *if statement*.

```
In [16]: condition = True # une variable booléenne déclarée comme True

         if condition == True:
             print("cette expression s'évalue")

         condition = False

         if condition == True:
           print("cette expression ne s'évalue pas")
```

```
cette expression s'évalue
```

Dans la cellule précédente, nous avons vu beaucoup de nouveaux mots clefs et des concepts. Nous allons maintenant les décrire

```
if condition == True
```

L'opérateur de comparaison == compare la valeur à sa droite de celle à sa gauche. Si les deux valeurs sont égales, alors l'expression complète s'évalue comme *True* et peut-être vu comme:

```
if True == True
if True: # True == True se réduit à True
```

Maintenant, le *if statement*, étant *True* exécute le code indenté qui le suit. Dans notre cas, le suivant

```
print('the condition is true')
```

exécute.
Dans le deuxième cas,

```
if condition == True
```

devient

```
if False == True
```

et puisque $False \neq True$, alors l'expression se réduit à *False*

```
if False
```

donc le bloc de code délimité par le *if statement* ne s'exécute pas. Pour résumer, les *if statements* s'exécute seulement si leur condition est *True*.

```
if condition:
  leur code
```

Où leur code est indenté une coche de plus que le *if statement*.

En dehors de l'opérateur de comparaison $(==)$ vu dans la cellule précédente, il existe beaucoup plus d'opérateurs logiques en Python. L'opposé de l'opérateur de comparaison est l'opérateur *not equal* $(!=)$. Il existe également l'opérateur *plus grand ou égal*, l'opérateur *plus grand que*, l'opérateur *plus petit ou égal*, l'opérateur *plus petit* et l'opérateur de négation (not).

```
In [17]: condition = True
         petit_nombre = 5
         grand_nombre = 100000

         if not condition:
           print('Faux')

         if condition != False:
           print('Vrai')

         if petit_nombre > grand_nombre:
           print('Faux')

         if petit_nombre < grand_nombre:
           print('Vrai')

         if grand_nombre >= petit_nombre:
           print('Vrai')
```

```
Vrai
Vrai
Vrai
```

L'opérateur de comparaison d'égalité avec une valeur booléenne est optionnel

```
if condition == True:
  print('bonjour')
```

est exactement la même expression que la suivante

```
if condition:
  print('bonjour')
```

et

```
if condition == False:
  print('bonjour')
```

est exactement la même expression que la suivante

```
if not condition:
  print('bonjour')
```

```
In [18]: condition = False
         if not condition:
           print('Vrai')

         if condition:
           print('Faux')
```

```
Vrai
```

Nous pouvons mélanger des conditions ensemble pour avoir plus de flexibilité dans les expressions logiques comme les *if statements*. Nous avons les opérateurs *and* et *or*.

Condition 1	Condition 2	or	and
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True

Nous pouvons interpréter la table précédente avec le code suivant

```
if False and True:
  print('Faux')

if False or True:
  print('Vrai')
```

```
In [19]: if False or False:
           print('Faux')

         if True and True:
           print('Vrai')
```

```
Vrai
```

Les exemples vus dans les cellules précédentes utilisent tous des valeurs booléennes ou bien des nombres. Par contre, nous pouvons utiliser des conditions plus complexes avec des strings et des fonctions par exemple.

```
In [20]: nom = 'Jack O'

         if nom == 'Jack O':
           print('Vrai')

         if len(nom) == 6:
           print('Vrai')

         if nom == 'Jack O' and len(nom) == 6:
           print('Vrai')
```

```
Vrai
Vrai
Vrai
```

1.2.1 Exercice 2

Validez que au moins un des deux nombres soient entre ses deux seuils

```
In [21]: def exercice2(nombre1, seuil_inférieur1, seul_supérieur1,
                       nombre2, seuil_inférieur2, seul_supérieur2):
           pass

         assert exercice2(10, 15, 20, 12, 10, 11) == False

         assert exercice2(8, 6, 9, 10, 11, 22) == True
```

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

<ipython-input-21-478c1c57526f> in <module>
  3   pass
  4
----> 5 assert exercice2(10, 15, 20, 12, 10, 11) == False
  6
  7 assert exercice2(8, 6, 9, 10, 11, 22) == True


AssertionError:
```

1.3 1.3 Boucles

Les boucles sont très importantes dans quelconque language de programmation de qualité et Python ne fait pas exception. Elles sont souvent utilisées afin d'itérer sur des listes, des arrays ou quelconques structures de données itérables, ou bien afin de répéter une tâche répétitive.

```
In [22]: for element in grosse_arr:
             print("élément présent: {}".format(element))
```

```
élément présent: [0 1 2 3 4 5 6 7 8 9]
élément présent: [10 11 12 13 14 15 16 17 18 19]
élément présent: [20 21 22 23 24 25 26 27 28 29]
élément présent: [30 31 32 33 34 35 36 37 38 39]
élément présent: [40 41 42 43 44 45 46 47 48 49]
élément présent: [50 51 52 53 54 55 56 57 58 59]
élément présent: [60 61 62 63 64 65 66 67 68 69]
élément présent: [70 71 72 73 74 75 76 77 78 79]
élément présent: [80 81 82 83 84 85 86 87 88 89]
élément présent: [90 91 92 93 94 95 96 97 98 99]
```

Analysons la cellule précédente.

```
for element in grosse_arr:
```

puisque la *grosse_arr* est une array multidimensionnelle de 2 dimensions, nous itérons sur les rangées. En Python, tous le travail des for loops est caché ce qui permet de rendre Python beaucoup plus simple. Si on compare la syntaxe de Python avec un autre language comme C, on s'aperçoit rapidement de la force de Python.

```
int[][] grosse_arr = new int[10][10];
for (int i = 0; i < 10; i++){
  printf("élément présent:");
  for (int j = 0; j < 10; j++)}
    printf("%f", grosse_arr[i][j]);
  }
  printf("\n");
}
```

Nous avons parfois besoin de l'index lorsque nous itérons sur une liste ou une array afin de savoir nous sommes présentement à quelle itération. Nous utilisons la fonction *enumerate* afin de parvenir à nos besoins.

```
In [23]: for index, élément in enumerate(grosse_arr):
             print('index présent: {}, élément présent: {}'.format(index, élément))
```

```
index présent: 0, élément présent: [0 1 2 3 4 5 6 7 8 9]
index présent: 1, élément présent: [10 11 12 13 14 15 16 17 18 19]
index présent: 2, élément présent: [20 21 22 23 24 25 26 27 28 29]
```

```
index présent: 3, élément présent: [30 31 32 33 34 35 36 37 38 39]
index présent: 4, élément présent: [40 41 42 43 44 45 46 47 48 49]
index présent: 5, élément présent: [50 51 52 53 54 55 56 57 58 59]
index présent: 6, élément présent: [60 61 62 63 64 65 66 67 68 69]
index présent: 7, élément présent: [70 71 72 73 74 75 76 77 78 79]
index présent: 8, élément présent: [80 81 82 83 84 85 86 87 88 89]
index présent: 9, élément présent: [90 91 92 93 94 95 96 97 98 99]
```

Remarquez que la boucle commence à l'index 0 et non 1.
Nous pouvons imbriquer des boucles dans d'autres. Par exemple, pour une boucle embriquée dans une autre, nous appelons cette dernière la boucle intérieure et la première la boucle extérieure.

```
In [24]: for i, rangée in enumerate(grosse_arr): # boucle extérieure
           for j, élément in enumerate(rangée): # boucle intérieure
             if i * grosse_arr.shape[0] + j < 20: # imprime seulement les 20 premiers éléments
               print(élément)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

1.3.1 Exercice 3

Calculez la moyenne de l'array x.

```
In [25]: x = np.random.randn(100)

         def exercice3(arr):
           somme = 0
```

```
    return somme


assert '{:3f}'.format(exercice3(x)) == '{:3f}'.format(np.average(x))


---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-25-1e3afaa2a9f4> in <module>
  6
  7
----> 8 assert '{:3f}'.format(exercice3(x)) == '{:3f}'.format(np.average(x))


AssertionError:
```

Il existe une autre façon d'interagir avec les boucles *for* avec la compréhension de liste. La syntaxe de base est la suivante

```
[ expression for item in liste if condition ]
```

Remarquez les deux crochets autour de la ligne complète, et les mots-clefs suivants: * for: la variable suivant le for est un élément individuel provenant de la liste, nous pouvons accéder à cet élément dans l'expression. * in: la variable suivant le mot-clef in est la liste sur laquelle nous sommes en train d'itérer. * if: la condition suivant le if spécifie si l'expression s'exécute, cette partie est optionnelle.

```
In [26]: valeurs_aléatoires = np.random.randn(12)
         print(np.average(valeurs_aléatoires))
         top_valeurs = [el for el in valeurs_aléatoires if el > np.average(valeurs_aléatoires)]
         top_valeurs

-0.13799857116047187


Out[26]: [1.3711684476870805,
          -0.1084060082115768,
          0.9965783910861999,
          1.327095426767479,
          0.1660907120758052,
          0.9648090960785973]

In [28]: # créer une simple liste vide ayant 2 dimensions
         deuxdim_liste = [[] for peuimporte in range(5)]
         deuxdim_liste

Out[28]: [[], [], [], [], []]
```

Il y a également un autre type de boucle qui se nomme la boucle *while* et qui a une structure différente. La boucle *while* utilise la même structure que le *if statement* vu précédemment. Cette boucle s'exécute jusqu'à sa condition s'évalue à *False*.

```
while condition:
  code
```

Donc, le code s'exécute seulement lorsque la condition s'évalue à *True*.

```
In [29]: i = 0
         while i < 5:
           print('i value: {}'.format(i))
           i = i + 1 # alternatively i += 1
```

```
i value: 0
i value: 1
i value: 2
i value: 3
i value: 4
```

Finissons notre conversation sur les boucles avec la fonction *range*.

```
range(start, stop, step)
```

Les trois paramètres de la fonction *range* sont optionnels. Par défaut, le paramètre *start* à la valeur 0, le paramètre *stop* n'a pas de valeur et le paramètre *step* à la valeur 1. En d'autres mots, c'est une *range* ∞. Si l'on donne seulement un paramètre à la fonction *range*, par défaut c'est le paramètre *stop*.

```
range(10)
```

Donc en utilisant l'expression précédente dans notre code, nous pourrions boucler 10 fois.

```
In [30]: for x in range(5):
           print(x)
```

```
0
1
2
3
4
```

Utilisons les notions vues aujourd'hui afin de traiter des données au préalable. Nous allons utiliser le *World Happiness Report* de 2019. Nous allons utiliser des DataFrame afin de manipuler des données.

```
In [31]: import io
         import pandas as pd
         whr = pd.read_csv('world-happiness-report-2019.csv')
         whr.loc[30:60]
```

Out[31]:

	Country (region)	Ladder	SD of Ladder	Positive affect	\
30	Panama	31	121	7.0	
31	Brazil	32	116	69.0	
32	Uruguay	33	88	10.0	
33	Singapore	34	5	38.0	
34	El Salvador	35	112	23.0	
35	Italy	36	31	99.0	
36	Bahrain	37	83	39.0	
37	Slovakia	38	39	53.0	
38	Trinidad and Tobago	39	89	14.0	
39	Poland	40	28	76.0	
40	Uzbekistan	41	99	19.0	
41	Lithuania	42	55	138.0	
42	Colombia	43	120	30.0	
43	Slovenia	44	54	114.0	
44	Nicaragua	45	133	31.0	
45	Kosovo	46	107	71.0	
46	Argentina	47	97	28.0	
47	Romania	48	75	80.0	
48	Cyprus	49	95	60.0	
49	Ecuador	50	113	11.0	
50	Kuwait	51	98	89.0	
51	Thailand	52	81	20.0	
52	Latvia	53	30	119.0	
53	South Korea	54	57	101.0	
54	Estonia	55	32	50.0	
55	Jamaica	56	102	51.0	
56	Mauritius	57	94	55.0	
57	Japan	58	43	73.0	
58	Honduras	59	151	13.0	
59	Kazakhstan	60	40	81.0	
60	Bolivia	61	71	70.0	

	Negative affect	Social support	Freedom	Corruption	Generosity	\
30	48.0	41.0	32.0	104.0	88.0	
31	105.0	43.0	84.0	71.0	108.0	
32	76.0	35.0	30.0	33.0	80.0	
33	2.0	36.0	20.0	1.0	21.0	
34	84.0	83.0	74.0	85.0	134.0	
35	123.0	23.0	132.0	128.0	48.0	
36	83.0	59.0	24.0	NaN	23.0	
37	47.0	21.0	108.0	142.0	70.0	
38	52.0	29.0	51.0	141.0	41.0	
39	33.0	44.0	52.0	108.0	77.0	
40	15.0	11.0	1.0	18.0	29.0	
41	41.0	17.0	122.0	113.0	124.0	
42	88.0	52.0	56.0	124.0	111.0	
43	71.0	14.0	13.0	97.0	54.0	

44	125.0	66.0	70.0	43.0	71.0
45	7.0	85.0	50.0	144.0	31.0
46	93.0	46.0	54.0	109.0	123.0
47	62.0	86.0	57.0	146.0	102.0
48	99.0	90.0	81.0	115.0	39.0
49	113.0	71.0	42.0	68.0	95.0
50	97.0	69.0	47.0	NaN	42.0
51	35.0	53.0	18.0	131.0	10.0
52	38.0	34.0	126.0	92.0	105.0
53	45.0	91.0	144.0	100.0	40.0
54	6.0	12.0	45.0	30.0	83.0
55	51.0	28.0	49.0	130.0	119.0
56	16.0	54.0	40.0	96.0	37.0
57	14.0	50.0	64.0	39.0	92.0
58	73.0	84.0	39.0	79.0	51.0
59	5.0	19.0	80.0	57.0	57.0
60	138.0	93.0	35.0	91.0	104.0

	Log of GDP\nper capita	Healthy life\nexpectancy
30	51.0	33.0
31	70.0	72.0
32	52.0	35.0
33	3.0	1.0
34	100.0	75.0
35	29.0	7.0
36	20.0	42.0
37	35.0	38.0
38	38.0	93.0
39	41.0	36.0
40	104.0	83.0
41	36.0	62.0
42	74.0	51.0
43	34.0	29.0
44	108.0	53.0
45	88.0	NaN
46	55.0	37.0
47	48.0	61.0
48	33.0	6.0
49	86.0	45.0
50	5.0	70.0
51	62.0	58.0
52	43.0	68.0
53	27.0	9.0
54	37.0	41.0
55	93.0	55.0
56	53.0	73.0
57	24.0	2.0
58	113.0	57.0

```
59                    47.0                    88.0
60                   101.0                    94.0
```

En regardant notre jeu de données de plus près, nous nous apercevons que certaines valeurs sont *NaN* (not a number). Naïvement, nous allons itérer sur notre jeu de données afin de remplacer les *NaN* par le nombre de colonnes.

```
In [32]: nombre_de_rangées = whr.shape[0]
         nombre_de_colonnes = whr.shape[1]


         for i in range(nombre_de_rangées):
           for j in range(1, nombre_de_colonnes):
             if np.isnan(whr.iloc[i, j]):
               print('i: {}, j: {}'.format(i,j))
               whr.iloc[i, j] = nombre_de_rangées
```

```
i: 20, j: 7
i: 24, j: 9
i: 24, j: 10
i: 27, j: 7
i: 28, j: 3
i: 28, j: 4
i: 28, j: 5
i: 28, j: 6
i: 28, j: 7
i: 28, j: 8
i: 36, j: 7
i: 45, j: 10
i: 50, j: 7
i: 63, j: 9
i: 63, j: 10
i: 75, j: 10
i: 86, j: 7
i: 92, j: 7
i: 100, j: 7
i: 109, j: 10
i: 111, j: 9
i: 134, j: 10
i: 148, j: 9
```

```
In [33]: whr.loc[30:60]
```

```
Out[33]:        Country (region)  Ladder  SD of Ladder  Positive affect  \
         30               Panama      31           121              7.0
         31               Brazil      32           116             69.0
         32              Uruguay      33            88             10.0
         33            Singapore      34             5             38.0
```

34	El Salvador	35	112	23.0
35	Italy	36	31	99.0
36	Bahrain	37	83	39.0
37	Slovakia	38	39	53.0
38	Trinidad and Tobago	39	89	14.0
39	Poland	40	28	76.0
40	Uzbekistan	41	99	19.0
41	Lithuania	42	55	138.0
42	Colombia	43	120	30.0
43	Slovenia	44	54	114.0
44	Nicaragua	45	133	31.0
45	Kosovo	46	107	71.0
46	Argentina	47	97	28.0
47	Romania	48	75	80.0
48	Cyprus	49	95	60.0
49	Ecuador	50	113	11.0
50	Kuwait	51	98	89.0
51	Thailand	52	81	20.0
52	Latvia	53	30	119.0
53	South Korea	54	57	101.0
54	Estonia	55	32	50.0
55	Jamaica	56	102	51.0
56	Mauritius	57	94	55.0
57	Japan	58	43	73.0
58	Honduras	59	151	13.0
59	Kazakhstan	60	40	81.0
60	Bolivia	61	71	70.0

	Negative affect	Social support	Freedom	Corruption	Generosity	\
30	48.0	41.0	32.0	104.0	88.0	
31	105.0	43.0	84.0	71.0	108.0	
32	76.0	35.0	30.0	33.0	80.0	
33	2.0	36.0	20.0	1.0	21.0	
34	84.0	83.0	74.0	85.0	134.0	
35	123.0	23.0	132.0	128.0	48.0	
36	83.0	59.0	24.0	156.0	23.0	
37	47.0	21.0	108.0	142.0	70.0	
38	52.0	29.0	51.0	141.0	41.0	
39	33.0	44.0	52.0	108.0	77.0	
40	15.0	11.0	1.0	18.0	29.0	
41	41.0	17.0	122.0	113.0	124.0	
42	88.0	52.0	56.0	124.0	111.0	
43	71.0	14.0	13.0	97.0	54.0	
44	125.0	66.0	70.0	43.0	71.0	
45	7.0	85.0	50.0	144.0	31.0	
46	93.0	46.0	54.0	109.0	123.0	
47	62.0	86.0	57.0	146.0	102.0	
48	99.0	90.0	81.0	115.0	39.0	

49	113.0	71.0	42.0	68.0	95.0
50	97.0	69.0	47.0	156.0	42.0
51	35.0	53.0	18.0	131.0	10.0
52	38.0	34.0	126.0	92.0	105.0
53	45.0	91.0	144.0	100.0	40.0
54	6.0	12.0	45.0	30.0	83.0
55	51.0	28.0	49.0	130.0	119.0
56	16.0	54.0	40.0	96.0	37.0
57	14.0	50.0	64.0	39.0	92.0
58	73.0	84.0	39.0	79.0	51.0
59	5.0	19.0	80.0	57.0	57.0
60	138.0	93.0	35.0	91.0	104.0

	Log of GDP\nper capita	Healthy life\nexpectancy
30	51.0	33.0
31	70.0	72.0
32	52.0	35.0
33	3.0	1.0
34	100.0	75.0
35	29.0	7.0
36	20.0	42.0
37	35.0	38.0
38	38.0	93.0
39	41.0	36.0
40	104.0	83.0
41	36.0	62.0
42	74.0	51.0
43	34.0	29.0
44	108.0	53.0
45	88.0	156.0
46	55.0	37.0
47	48.0	61.0
48	33.0	6.0
49	86.0	45.0
50	5.0	70.0
51	62.0	58.0
52	43.0	68.0
53	27.0	9.0
54	37.0	41.0
55	93.0	55.0
56	53.0	73.0
57	24.0	2.0
58	113.0	57.0
59	47.0	88.0
60	101.0	94.0

1.3.2 Exercice 4

Retournez l'élément ayant la plus petite valeur de la rangée corespondant à la Pologne

```
In [34]: def exercice4(rangée):
             plus_bas = None
             return plus_bas

         rangée = whr.iloc[39, 1:]
         assert exercice4(rangée) == 28
```

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-34-b13de929a874> in <module>
  4
  5 rangée = whr.iloc[39, 1:]
----> 6 assert exercice4(rangée) == 28


AssertionError:
```