td2_eco_rappels_1a

June 24, 2022

1 2A.eco - Rappel de ce que vous savez déjà mais avez peut-être oublié

pandas et numpy sont essentiels pour manipuler les données. C'est ce que rappelle ce notebook. Voir aussi Essential Cheat Sheets for Machine Learning and Deep Learning Engineers.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

1.1 Les quelques règles de Python

Python est un peu susceptible et protocolaire, il y a quelques règles à respecter :

1) L'indentation est primordiale : un code mal indenté ne fonctionne pas. L'indentation indique à l'interpréteur où se trouvent les séparations entre des blocs d'instructions. Un peu comme des points dans un texte. Si les lignes ne sont pas bien alignées, l'interpréteur ne sait plus à quel bloc associer la ligne.

2) On commence à compter à 0. Ca peut paraitre bizarre mais c'est comme ça. Le premier élément d'une liste est le 0-ème.

3) Les marques de ponctuation sont importantes :

- Pour une liste : []
- Pour un dictionnaire : {}
- Pour un tuple : ()
- Pour séparer des éléments : ,
- Pour commenter un bout de code : #
- Pour aller à la ligne dans un bloc d'instructions :

- Les majuscules et minuscules sont importantes
- Par contre l'usage des ' ou des " est indifférente. Il faut juste avoir les mêmes début et fin.
- Pour documenter une fonction ou une classe """ documentation """

1.2 Les outputs de Python : l'opération, le print et le return

Quand Python réalise des opérations, il faut lui préciser ce qu'il doit en faire : - est-ce qu'il doit juste faire l'opération, - afficher le résultat de l'opération, - créer un objet avec le résultat de l'opération ?
Remarque : dans l'environnement Notebook, le dernier élement d'une cellule est automatiquement affiché (print), qu'on lui demande ou non de le faire. Ce n'est pas le cas dans un éditeur classique comme Spyder.

[2]:
```
# on calcule : dans le cas d'une opération par exemple une somme
2+3 # Python calcule le résultat mais n'affiche rien dans la sortie

# le print : on affiche

print(2+3) # Python calcule et on lui demande juste de l'afficher
# le résultat est en dessous du code
```

```
5
```

[3]:
```
# le print dans une fonction

def addition_v1(a,b) :
    print(a+b)

resultat_print = addition_v1(2,0)
print(type(resultat_print))

# dans la sortie on a l'affichage du résultat, car la sortie de la fonction est un␣
 ↪print
# en plus on lui demande quel est le type du résultat. Un print ne renvoie aucun type,␣
 ↪ce n'est ni un numérique,
# ni une chaine de charactères, le résultat d'un print n'est pas un format utilisable
```

```
2
<class 'NoneType'>
```

Le résultat de l'addition est affiché
La fonction addition_v1 effectue un print
Par contre, l'objet crée n'a pas de type, il n'est pas un chiffre, ce n'est qu'un affichage.
Pour créer un objet avec le résultat de la fonction, il faut utiliser **return**

[4]:
```
# le return dans une fonction
def addition_v2(a,b) :
    return a+b

resultat_return = addition_v2(2,5) #
print(type(resultat_return))
## là on a bien un résultat qui est du type "entier"
```

```
<class 'int'>
```

Le résultat de addition_v2 n'est pas affiché comme dans addition_v1
Par contre, la fonction addition_v2 permet d'avoir un objet de type int, un entier donc.

1.3 Type de base : variables, listes, dictionnaires ...

Pyhon permet de manipuler différents types de base
On distingue deux types de variables : les immuables qui ne peuvent être modifiés et les modifiables

1.3.1 Les variables - types immuables

Les variables immuables ne peuvent être modifiées

- None : ce type est une convention de programmation pour dire que la valeur n'est pas calculée
- bool : un booléen
- int : un entier
- float : un réel
- str : une chaine de caractères
- tuple : un vecteur

[5]:
```
i = 3          # entier = type numérique (type int)
r = 3.3        # réel   = type numérique (type float)
s = "exemple" # chaîne de caractères = type str
n = None       # None signifie que la variable existe mais qu'elle ne contient rien
               # elle est souvent utilisée pour signifier qu'il n'y a pas de résultat
a = (1,2)      # tuple

print(i,r,s,n,a)
```

```
3 3.3 exemple None (1, 2)
```

Si on essaie de changer le premier élément de la chaine de caractères **s** on va avoir un peu de mal.
Par exemple si on voulait mettre une majuscule à "exemple", on aurait envie d'écrire que le premier élément de la chaine s est "E" majuscule
Mais Python ne va pas nous laisser faire, il nous dit que les objets "chaine de caractère" ne peuvent être modifiés

[6]:
```
s[0] = "E"  # déclenche une exception
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-0edb952b04f7> in <module>()
----> 1 s[0] = "E"

TypeError: 'str' object does not support item assignment
```

Tout ce qu'on peut faire avec une variable immutable, c'est le réaffecter à une autre valeur : il ne peut pas être modifié
Pour s'en convaincre, utilisons la fonction id() qui donne un identifiant à chaque objet.

[7]:
```
print(s)
id(s)
```

```
exemple
```

[7]:
```
2287124957368
```

[8]:
```
s = "autre_mot"
id(s)
```

[8]:
```
2287124971312
```

On voit bien que s a changé d'identifiant : il peut avoir le même nom, ce n'est plus le même objet

1.3.2 Les variables - types modifiable : listes et dictionnaires

Heureusement, il existe des variables modifiables comme les listes et les dictionnaires.

Les listes - elles s"écrivent entre [ ] Les listes sont des élements très utiles, notamment quand vous souhaitez faire des boucles
Pour faire appel aux élements d'une liste, on donne leur position dans la liste : le 1er est le 0, le 2ème est le 1 ...

[9]:
```
ma_liste = [1,2,3,4]


print("La longueur de ma liste est de", len(ma_liste))

print("Le premier élément de ma liste est :", ma_liste[0])

print("Le dernier élément de ma liste est :", ma_liste[3])
print("Le dernier élément de ma liste est :", ma_liste[-1])
```

```
La longueur de ma liste est de 4
Le premier élément de ma liste est : 1
Le dernier élément de ma liste est : 4
Le dernier élément de ma liste est : 4
```

Les dictionnaires - ils s'écrivent entre Un dictionnaire associe à une clé un autre élément, appelé une valeur : un chiffre, un nom, une liste, un autre dictionnaire etc.

- **Format d'un dictionnaire : {Clé : valeur}**

Dictionnaire avec des valeurs int On peut par exemple associer à un nom, un nombre

[10]:
```
mon_dictionnaire_notes = { 'Nicolas' : 18 , 'Pimprenelle' : 15}
# un dictionnaire qui à chaque nom associe un nombre
# à Nicolas, on associe 18

print(mon_dictionnaire_notes)
```

```
{'Nicolas': 18, 'Pimprenelle': 15}
```

Dictionnaire avec des valeurs qui sont des listes Pour chaque clé d'un dictionnaire, il ne faut pas forcément garder la même forme de valeur
Dans l'exemple, la valeur de la clé "Nicolas" est une liste, alors que celle de "Philou" est une liste de liste

[11]:
```
mon_dictionnaire_loisirs =  \
{ 'Nicolas' : ['Rugby','Pastis','Belote'] ,
  'Pimprenelle' : ['Gin Rami','Tisane','Tara Jarmon','Barcelone','Mickey Mouse'],
  'Philou' : [['Maths','Jeux'],['Guillaume','Jeanne','Thimothée','Adrien']]}
```

Pour accéder à un élément du dictionnaire, on fait appel à la clé et non plus à la position, comme c'était le cas dans les listes

[12]:
```
print(mon_dictionnaire_loisirs['Nicolas']) # on affiche une liste
```

```
['Rugby', 'Pastis', 'Belote']
```

[13]:
```
print(mon_dictionnaire_loisirs['Philou']) # on affiche une liste de listes
```

```
[['Maths', 'Jeux'], ['Guillaume', 'Jeanne', 'Thimothée', 'Adrien']]
```

Si on ne veut avoir que la première liste des loisirs de Philou, on demande le premier élément de la liste

[14]:
```
print(mon_dictionnaire_loisirs['Philou'][0]) # on affiche alors juste la première liste
```

```
['Maths', 'Jeux']
```

On peut aussi avoir des valeurs qui sont des int et des listes

[15]:
```
mon_dictionnaire_patchwork_good = \
{ 'Nicolas' : ['Rugby','Pastis','Belote'] ,
  'Pimprenelle' : 18 }
```

1.4 A retenir

- L'indentation du code est importante (4 espaces et pas une tabulation)
- Une **liste** est entre [] et on peut appeler les positions par leur place
- Un **dictionnaire**, clé x valeur, s'écrit entre {} et on appelle un élément en fonction de la clé

1.5 Questions pratiques :

- Quelle est la position de 7 dans la liste suivante

[16]:
```
liste_nombres = [1,2,7,5,3]
```

- Combien de clés a ce dictionnaire ?

[17]:
```
dictionnaire_evangile = {"Marc" : "Lion", "Matthieu" : ["Ange","Homme ailé"] ,
                         "Jean" : "Aigle" , "Luc" : "Taureau"}
```

- Que faut-il écrire pour obtenir "Ange" en résultat à partir du dictionnaire_evangile ?

1.6 Objets : méthodes et attributs

Mainentant qu'on a vu quels objets existaient en Python, nous allons voir comment nous en servir.

1.6.1 Un petit détour pour bien comprendre : Un objet, c'est quoi ?

Un objet a deux choses : des attributs et des méthodes

- Les attributs décrivent sa structure interne : sa taille, sa forme (dont on ne va pas parler ici)
- Les méthodes sont des "actions" qui s'appliqueront à l'objet

1.6.2 Premiers exemples de méthode

Avec les éléments définis dans la partie 1 (les listes, les dictionnaires) on peut faire appel à des méthodes qui sont directement liées à ces objets.
Les méthodes, c'est un peu les actions de Python.

Une méthode pour les listes Pour ajouter un item dans une liste : on va utiliser la méthode *.append()*

[18]:
```
ma_liste = ["Nicolas","Michel","Bernard"]

ma_liste.append("Philippe")

print(ma_liste)
```

```
['Nicolas', 'Michel', 'Bernard', 'Philippe']
```

Une méthode pour les dictionnaires Pour connaitre l'ensemble des clés d'un dictionnaire, on appelle la méthode *.keys()*

[19]:
```
mon_dictionnaire = {"Marc" : "Lion", "Matthieu" : ["Ange","Homme ailé"] ,
                            "Jean" : "Aigle" , "Luc" : "Taureau"}

print(mon_dictionnaire.keys())
```

```
dict_keys(['Matthieu', 'Jean', 'Marc', 'Luc'])
```

1.6.3 Connaitre les méthodes d'un objet

Pour savoir quelles sont les méthodes d'un objet vous pouvez : - taper help(mon_objet) ou mon_objet? dans la console iPython - taper mon_objet. + touche tabulation dans la console iPython ou dans le notebook . iPython permet la complétion, c'est-à-dire que vous pouvez faire appaître la liste

1.7 Les opérations et méthodes classiques des listes

1.7.1 Créer une liste

Pour créer un objet de la classe list, il suffit de le déclarer. Ici on affecte à **x** une liste

[20]:
```
x = [4, 5] # création d'une liste composée de deux entiers
x = ["un", 1, "deux", 2] # création d'une liste composée de 2 chaînes de caractères
# et de deux entiers, l'ordre d'écriture est important
x = [3] # création d'une liste d'un élément, sans la virgule,
x = [ ] # crée une liste vide
x = list () # crée une liste vide
```

1.7.2 Un premier test sur les listes

Si on veut tester la présence d'un élément dans une liste, on l'écrit de la manière suivante :

[21]:
```
# Exemple

x = "Marcel"

l = ["Marcel","Edith","Maurice","Jean"]

print(x in l)

#vrai si x est un des éléments de l
```

```
True
```

1.7.3 Pour concaténer deux listes :

On utilise le symbole +

[22]:
```
t = ["Antoine","David"]
print(l + t) #concaténation de l et t
```

```
['Marcel', 'Edith', 'Maurice', 'Jean', 'Antoine', 'David']
```

1.7.4 Pour trouver certains éléments d'une liste

Pour chercher des élements dans une liste, on utilise la position dans la liste.

[23]:
```
l[1] # donne l'élément qui est en 2ème position de la liste
```

[23]:
```
'Edith'
```

[24]:
```
l[1:3] # donne les éléments de la 2ème position de la liste à la 4ème exclue
```

[24]:
```
['Edith', 'Maurice']
```

1.7.5 Quelques fonctions des listes

[25]:
```
longueur = len(l) # nombre d'éléments de l

minimum = min(l) # plus petit élément de l, ici par ordre alphabétique

maximum = max(l) # plus grand élément de l, ici par ordre alphabétique

print(longueur,minimum,maximum)
```

```
4 Edith Maurice
```

[26]:
```
del l[0 : 2] # supprime les éléments entre la position 0 et 2 exclue
print(l)
```

```
['Maurice', 'Jean']
```

1.8 Les méthodes des listes

On les trouve dans l'aide de la liste. On distingue les méthodes et les méthodes spéciales : visuellement, les méthodes spéciales sont celles qui précédées et suivis de deux caractères de soulignement, les autres sont des méthodes classiques.

[27]:
```
help(l)
```

```
Help on list object:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
```

```
|
|  __contains__(self, key, /)
|      Return key in self.
|
|  __delitem__(self, key, /)
|      Delete self[key].
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(…)
|      x.__getitem__(y) <==> x[y]
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __iadd__(self, value, /)
|      Implement self+=value.
|
|  __imul__(self, value, /)
|      Implement self*=value.
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.n
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  __repr__(self, /)
|      Return repr(self).
```

```
 |
 |  __reversed__(…)
 |      L.__reversed__() -- return a reverse iterator over the list
 |
 |  __rmul__(self, value, /)
 |      Return self*value.
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(…)
 |      L.__sizeof__() -- size of L in memory, in bytes
 |
 |  append(…)
 |      L.append(object) -> None -- append object to end
 |
 |  clear(…)
 |      L.clear() -> None -- remove all items from L
 |
 |  copy(…)
 |      L.copy() -> list -- a shallow copy of L
 |
 |  count(…)
 |      L.count(value) -> integer -- return number of occurrences of value
 |
 |  extend(…)
 |      L.extend(iterable) -> None -- extend list by appending elements from the
iterable
 |
 |  index(…)
 |      L.index(value, [start, [stop]]) -> integer -- return first index of
value.
 |      Raises ValueError if the value is not present.
 |
 |  insert(…)
 |      L.insert(index, object) -- insert object before index
 |
 |  pop(…)
 |      L.pop([index]) -> item -- remove and return item at index (default
last).
 |      Raises IndexError if list is empty or index is out of range.
 |
 |  remove(…)
 |      L.remove(value) -> None -- remove first occurrence of value.
 |      Raises ValueError if the value is not present.
 |
 |  reverse(…)
 |      L.reverse() -- reverse *IN PLACE*
 |
 |  sort(…)
 |      L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
```

```
 |
 |  __hash__ = None
```

1.9 A retenir et questions

A retenir :

- Chaque objet Python a des attributs et des méthodes
- Vous pouvez créer des classes avec des attributs et des méthodes
- Les méthodes des listes et des dictionnaires sont les plus utilisées :
  - list.count()
  - list.sort()
  - list.append()
  - dict.keys()
  - dict.items()
  - dict.values()

Questions pratiques :

- Définir la liste allant de 1 à 10, puis effectuez les actions suivantes :

```
\textquestiondown  triez et affichez la liste
\textquestiondown  ajoutez l'élément 11 à la liste et affichez la liste
\textquestiondown  renversez et affichez la liste
\textquestiondown  affichez l'indice de l'élément 7
\textquestiondown  enlevez l'élément 9 et affichez la liste
\textquestiondown  affichez la sous-liste du 2e au 3e élément ;
\textquestiondown  affichez la sous-liste du début au 2e élément ;
\textquestiondown  affichez la sous-liste du 3e élément à la fin de la liste ;
```

- Construire le dictionnaire des 6 premiers mois de l'année avec comme valeurs le nombre de jours respectif.

```
- Renvoyer la liste des mois.
- Renvoyer la liste des jours.
- Ajouez la clé du mois de Juillet ?
```

1.10 Passer des listes, dictionnaires à pandas

Supposons que la variable 'data' est un liste qui contient nos données.
Une observation correspond à un dictionnaire qui contient le nom, le type, l'ambiance et la note d'un restaurant.
Il est aisé de transformer cette liste en dataframe grâce à la fonction 'DataFrame'.

[28]:
```python
import pandas

data = [{"nom": "Little Pub", "type" : "Bar", "ambiance": 9, "note": 7},
     {"nom": "Le Corse", "type" : "Sandwicherie", "ambiance": 2, "note": 8},
     {"nom": "Café Caumartin", "type" : "Bar", "ambiance": 1}]

df = pandas.DataFrame(data)

print(data)
df
```

```
[{'type': 'Bar', 'ambiance': 9, 'note': 7, 'nom': 'Little Pub'}, {'type':
'Sandwicherie', 'ambiance': 2, 'note': 8, 'nom': 'Le Corse'}, {'type': 'Bar',
'ambiance': 1, 'nom': 'Café Caumartin'}]
```

[28]:

	ambiance	nom	note	type
0	9	Little Pub	7.0	Bar
1	2	Le Corse	8.0	Sandwicherie
2	1	Café Caumartin	NaN	Bar

[29]: