Vers la programation vraiment objet

Nous allons commencer à créer nos propres objets, appelés classes. Cette façon de programmer est moderne et efficace dans la mesure où le langage est complètement prévu pour.

Il est possible de créer son propre module. Un module est simplement une boîte à outils qui définit des objets réutilisables. Cela permet de rendre le code plus modulaire, permet de le découper en briques élémentaires et ainsi d'améliorer la lisibilité.

Définition d'une classe

Nous commençons par définir la notion de classe. Voici une première définition de classe

In [1]:
```
class compteur:
    """une première classe pour faire un compteur"""
    def __init__(self):  # fonction appelée à la créaction de l'obj
et
        print("La fonction __init__ est utilisée !")
        self.n = 5       # l'objet ne contient qu'un champ : n
    def avance(self):    # une fonction membre
        self.n += 1      # qui incrémente le champ n
```

In [2]:
```
compt = compteur()

while compt.n < 10:
    print(compt.n)
    compt.avance()

compt.__init__()
print(compt.n)
```

```
La fonction __init__ est utilisée !
5
6
7
8
9
La fonction __init__ est utilisée !
5
```

On peut améliorer cette classe en choisissant à la création d'un compteur le pas (ou l'incrément). Voici une proposition :

In [3]:
```
class compteur:
    """une première classe pour faire un compteur"""
    def __init__(self, begin=0, incr=1):  # fonction appelée à la c
réaction de l'objet
        self.n = begin                    # l'objet contient deux c
hamps : n et i
        self._i = incr
    def avance(self):                     # une fonction membre
        self.n += self._i                 # qui incrémente le champ
n de l'incrément i
```

In [4]:
```
compt = compteur(begin=-17, incr=2)
while compt.n < 10:
    print(compt.n)
    compt.avance()
```

```
-17
-15
-13
-11
-9
-7
-5
-3
-1
1
3
5
7
9
```

Pour rendre les choses plus jolies, on peut avoir envie d'écrire ce code :

```
compt = compteur(incr=2)
while compt < 10:
    print(compt)
    compt.avance()
```

Pour pouvoir le faire, il faut ajouter quelques fonctions à notre classe. Une fonction de comparaison `__lt__` et une fonction d'affichage `__str__` . Voici une solution

In [44]:
```
class compteur:
    """une première classe pour faire un compteur"""
    def __init__(self, incr=1):
        self.n = 0
        self.i = incr

    def avance(self):
        self.n += self.i

    def __eq__(self, val):        # fonction utilisée pour faire co
```

```
mpt == val
        print(f"On passe dans __eq__({val})")
        if isinstance(val, self.__class__):
            return self.n == val.n
        return self.n == val

    def __ne__(self, val):         # fonction utilisée pour faire co
mpt != val
        print(f"On passe dans __ne__({val})")
        if isinstance(val, self.__class__):
            return self.n != val.n
        return self.n != val

    def __lt__(self, val):         # fonction utilisée pour faire com
pt < val
        print(f"On passe dans __lt__({val})")
        if isinstance(val, self.__class__):
            return self.n < val.n
        return self.n < val

    def __gt__(self, val):         # fonction utilisée pour faire com
pt > val
        print(f"On passe dans __gt__({val})")
        if isinstance(val, self.__class__):
            return self.n > val.n
        return self.n > val

    def __le__(self, val):         # fonction utilisée pour faire com
pt <= val
        print(f"On passe dans __le__({val})")
        if isinstance(val, self.__class__):
            return self.n <= val.n
        return self.n <= val

    def __ge__(self, val):         # fonction utilisée pour faire com
pt >= val
        print(f"On passe dans __ge__({val})")
        if isinstance(val, self.__class__):
            return self.n >= val.n
        return self.n >= val

    def __rlt__(self, val):
        print(f"On passe dans __rlt__({val})")
        return val < self.n

    def __rgt__(self, val):
        print(f"On passe dans __rgt__({val})")
        return val > self.n

    def __format__(self, fmt):
        """pour formater le compteur"""
        return f"{self.n:{fmt}}"
```

```
    def __str__(self):            # fonction utilisée par print
        print("On passe dans __str__()")
        return f"{self.n}"
```

In [45]:
```
compt = compteur(incr=2)
while 10 < compt:
    print(f"{compt:04d}")
    compt.avance()
```

```
On passe dans __gt__(10)
```

In [46]:
```
compt1, compt2 = compteur(incr=2), compteur(incr=1)
compt2.n = 3
while compt1 != compt2:
    print(f"{compt1} < {compt2}")
    compt1.avance()
    compt2.avance()
```

```
On passe dans __ne__(3)
0 < 3
On passe dans __ne__(4)
2 < 4
On passe dans __ne__(5)
4 < 5
On passe dans __ne__(6)
```

Fabrication du premier module

Maintenant que nous avons créer notre premier objet, nous allons le ranger dans un module qui sera réutilisable directement dans n'importe quel code. Il est même possible d'ajouter ce module à notre installation de python afin que `python` connaisse le chemin, sinon il faut nécessairement que le fichier contenant le module soit rangé dans le répertoire de travail.

L'écriture dans un fichier à partir d'un notebook peut se faire facilement à l'aide de la commande magique `%%writefile` .

In [ ]:
```
%%writefile 'module_compteur.py'
"""
Ici se trouve la documentation du module.
C'est très important pour que l'on puisse l'utiliser

Module qui définit une classe compteur

@BG
"""
class compteur:
    """
    Documentation de la classe compteur

    Parameters
    ----------

    n: int
        la valeur du compteur
    i: int (optional)
        l'incrément (default=1)

    Attributes
    ----------

    n: int
        la valeur du compteur
    i: int
        l'incrément
    nmax: int
        une borne à ne pas dépasser (default 1000)
    """
    def __init__(self, incr=1):
        self._n = 0
        self._i = incr

    def avance(self):
        """Fonction qui incrémente le compteur"""
        self._n += self._i

    def __lt__(self, val):       # fonction utilisée pour faire com
pt < val
        return self._n < val     # pas besoin de la commenter car t
out le monde sait ce qu'elle fait !!!

    def __str__(self):           # fonction utilisée par print
        return f"{self._n}"      # pas besoin de la commenter car t
out le monde sait ce qu'elle fait !!!
```

On peut afficher le contenu d'un fichier à l'aide de la commande `cat` :

In [ ]:
```
!cat module_compteur.py
```

In [ ]:
```
import module_compteur as mc  # on importe le module avec un raccourci
```

In [ ]:
```
help(mc)                      # on affiche l'aide du module
```

In [ ]:
```
help(mc.compteur)    # on affiche seulement l'aide de la classe `compteur`
```

Et on peut l'utiliser facilement.

In [ ]:
```
compt = mc.compteur(incr=2)
while compt < 21:
    print(compt)
    compt.avance()
```

Question

- Ajoutez les opérateurs `__le__` (pour $\leq$), `__gt__` (pour $>$) et `__ge__` (pour $\geq$) sur le même modèle que l'opérateur `__lt__` .
- Testez vos nouveaux opérateurs

In [ ]:
```
compt = mc.compteur(incr=2)
while 20 >= compt:
    print(compt)
    compt.avance()
```

Question

- Afin d'éviter les boucles infinies, nous pouvons imposer que le compteur ne dépasse pas une certaine valeur `nmax` qui vaut 1000 par exemple. Ajoutez cette fonctionnalité.
- Afin d'empêcher l'utilisateur de proposer un incrément non entier (même si cette fonctionnalité pourrait être utile), modifiez la fonction `__init__` afin qu'elle prenne la partie entière de l'incrément dans le cas où il n'est pas entier. Vous pourrez aussi ajouter un message qui prévient l'utilisateur.
- Testez vos nouvelles fonctionnalités en exécutant la cellule suivante.

Indication : vous pourrez générer une erreur dans le cas où le compteur dépasse la valeur `nmax` en utilisant la commande `raise ValueError("Le compteur va trop loin !")`

In [ ]:
```
compt = mc.compteur(incr=2.)
while True:
    compt.avance()
```

In [ ]:
```

```