Vocabulaire programmation objet.

Bilan

En Programmation Orientée Objet, on fabrique de nouveau types de données (int, str, bool, etc.) correspondant aux besoin du programme que l'on construit. On réfléchit alors aux caractéristiques des objets qui seront de ce type (les attributs) et aux actions possibles à partir de ces objets (les méthodes).
Ces caractéristiques et ces actions sont regroupées dans un code spécifique associé au type de données, appelé classe.

Classe, Attributs, Méthodes, Accesseur et mutateurs, encapsulation, agrégation.

— Le type de données avec ses **caractéristiques** et ses **actions** possibles s'appelle **classe**.
— Les **caractéristiques (ou variables)** de la classe s'appellent les **attributs**.
— Les **actions possibles** à effectuer avec la classe s'appellent les **méthodes**.
— La **classe** définit donc les **attributs** et les actions possibles sur ces attributs, les **méthodes**.

— Un **objet** du type de la classe s'appelle une **instance de la classe** et la création d'un objet d'une classe s'appelle une instanciation de cette classe.
— Lorsqu'on définit les **attributs** d'un objet de la classe, on parle d'**instanciation**.
— **L'encapsulation** désigne le principe de **regrouper des données brutes** avec un ensemble de **routines (méthodes)** permettant de les lire ou de les manipuler.
— On dit que les attributs et les méthodes sont **encapsulés** dans la classe.

— **Constructeur** : la manière « normale » de spécifier l'initialisation d'un objet est d'écrire un constructeur . Son nom est __init__ ().
— **Accesseur** ou « **getter** » : une fonction qui retourne la valeur d'un attribut de l'objet. Par convention son nom est généralement sous la forme : *getNom_attribut()*.
— Un **Mutateur** ou **setter** : une procédure qui permet de modifier la valeur d'un attribut d'un objet. Son nom est généralement sous la forme : *setNom_attribut()*.

— But de l'encapsulation : cacher la représentation interne des classes,
- — pour simplifier la vie du programmeur qui les utilise ;
- — pour masquer leur complexité (diviser pour régner) ;
- — pour permettre de modifier celle-ci sans changer le reste du programme.
- — La liste des méthodes devient une sorte de mode d'emploi de la classe.

— L'agrégation est le fait de construire un objet (en créant une classe) en faisant appel à plusieurs autres classes, un **objet agrégat** constitué d'**objet composant**.
Cette architecture permet de transformer une classe sans toucher aux autres, et donc de se partager le travail dans une équipe.

Exemple de mise à disposition de classes dans un module.
- Créer un fichier `figures_geometriques.py` (contenant des classes des différentes figures, Triangle, Carre, Rectangle) dont le nom, `figures_geometriques` est le nom du module.
- Dans un autre fichier où l'on veut utiliser les figures déjà crées, il faut importer le fichier précédent soit :
- `import figures_geometriques` puis lorsqu'on en a besoin, écrire
`tri = figures_géometriques.Triangle`
- ou bien `from figures_geometriques import Triangle` puis lorsqu'on en a besoin, écrire
`tri = Triangle`

Exemples de classe.

Exemple 1.


```
class Time :
   "une classe temporelle"
```

Documentation qui s'affichera en utilisant la fonction help()

Création d'une classe Time :
Par convention, Les noms de classe commencent par une majuscule.
Pas de parenthèse si la classe n'a pas d'héritage.
Les méthodes (fonctions) de la classe Time seront indentés sous celle-ci

```
    def __init__(self, hh =12, mm =0, ss =0):
```

méthode constructeur qui sera appelée lors de la l'instanciation (création) d'un objet de la classe Time

```
      self.heure =hh
      self.minute =mm
      self.seconde =ss
```

Attributs (variables) qui seront initialisés lors de la création de l'objet

méthode utilisable avec toutes les instances de la classe Time. Cette méthode sera appelée sans arguments

```
def affiche_heure(self):
    print("{0}:{1}:{2}".format(self.heure, self.minute, self.seconde))
```

Le paramètre **self est nécessaire**, il désigne l'instance à laquelle la méthode est associée

```
  def set_heure(self,h) :
     if h>=0 and h<=24 :
       self.heure=h
```

Bien que cela ne soit pas obligatoire, il existe une convention de passer par des **getter** (ou accesseur en français) et des **setter** (mutateurs) pour lire ou changer la valeur d'un attribut

```
tstart = Time()
```

Instantiation de la classe Time (création d'un objet) sans passage de paramètre car les arguments ont des valeurs par défaut.

```
tstart.affiche_heure()
```

On obtiendra l'affichage 12:0:0

```
tstart.set_heure(13)
```

Modifie l'attribut tstart.heure à 13

```
recreation = Time(10, 15, 18)
```

Instantiation de la classe Time (création d'un objet) avec passage de paramètre. La méthode __init__ va créer les attributs
recreation.heure = 10,
recreation.minute = 15
et recreation.seconde=18

```
recreation.affiche_heure()
```

On obtiendra l'affichage 10:15:18

Exemple 2.

A - Classes et Attributs

Une *classe* définit et nomme une structure de données qui peut regrouper plusieurs attributs (aussi appelés *champs* ou *propriétés*). Une fois qu'une classe est définie, on peut facilement obtenir plusieurs *instances* différentes de cette classe (on dit parfois *objet* au lieu d'instance). La classe peut être vue comme un modèle, les instances comme des exemplaires différents du modèle.

-- Définition d'une classe et création des instances ou objets

En Python la définition d'une classe :
- est introduite par le mot-clef `class` suivi du nom choisi pour la classe commençant par une majuscule,
- comporte une méthode *constructeur* `__init__` qui permet d'initialiser les attributs des différentes instances,
- utilise par convention le nom de variable `self` qui désigne l'instance de la classe,
- `self` *possédant l'extraordinaire particularité d'être obligatoire dans les en-têtes des méthodes mais ne devant pas être mentionné lors des appels (il est passé en argument de façon transparente !).*

```
class Date:
    '''Une classe pour représenter une date'''

    def __init__(self, j, m, a):
        self.jour = j
        self.mois = m
        self.annee = a

anniv_alice = Date(7, 12, 2003)
anniv_bob = Date(29, 2, 2004)
```

L'exemple ci-contre est archétypique. Lorsqu'on crée les deux instances (ou objets) `anniv_alice` et `anniv_bob`, la méthode constructeur `__init__` est appelée deux fois avec :
- l'argument `self` (égal à `anniv_alice` puis `anniv_bob`) qui est transmis de façon invisible,
- les arguments `j, m, a` (égaux à `7, 12, 2003` puis `29, 2, 2004`) qui sont transmis de façon explicite pour être affectés aux trois attributs `jour, mois, annee`.

-- Lecture et modification des attributs des instances

```
>>> date_mensualite = Date(7, 10, 2020)
>>> date_paie = Date(30, 9, 2020)
>>> date_paie.mois
9
>>> date_mensualite.jour
7
>>> date_paie.mois = date_paie.mois + 1
>>> date_paie.mois
10
```

On accède en lecture ou en écriture aux attributs des instances grâce à une *notation pointée*.
Les attributs sont en effet *mutables*.
Ainsi pour accéder à l'attribut att d'un objet (ou instance) obj on utilise la notation obj.att qui permet aussi bien de lire la valeur de l'attribut que de la modifier.
Ainsi pour accéder à l'attribut `att` d'un objet (ou instance) `obj` on utilise la notation `obj.att` qui permet aussi bien de lire la valeur de l'attribut que de la modifier.

B - Méthodes d'instances

Une fois les attributs des objets d'une classe définis, on a souvent besoin de fonctions pour manipuler les attributs des objets. Ces fonctions, définies à l'intérieur des classes, sont appelées des *méthodes* : elles peuvent par exemple modifier certains attributs, renvoyer des valeurs calculées à partir des attributs, créer de nouveaux objets, modifier d'autres objets etc.

-- Définition des méthodes et appels aux méthodes

Les méthodes sont définies à l'intérieur de la définition des classes comme les fonctions habituelles à l'aide du mot-clef `def`. La seule différence, déjà évoquée dans le A, est que leur en-tête comporte obligatoirement comme premier paramètre `self` qui ne sera pas mentionné lors des appels puisqu'il sera remplacé automatiquement de façon invisible lors de l'appel par l'instance elle-même.