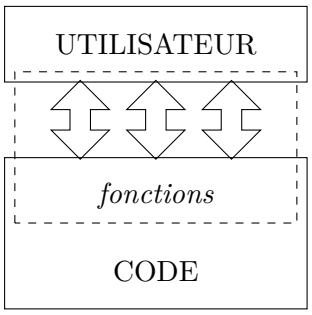
I Modules & Interfaces

Lorsque vous jouez à un jeu vidéo, vous n'avez pas accès au code mais simplement à des actions, des fonctions prévues par le créateur.

Lors du projet annuaire, vous aviez créé différentes fonctions pour interagir avec l'annuaire (`creation()`, `affichage()`, `existeTel()`, `ajout()`, etc.) et pour interagir avec l'annuaire vous n'utilisiez que ces fonctions.



Dans un cas comme dans l'autre, l'ensemble des fonctions laissées accessibles à l'utilisateur forme .....

Ci-contre, la situation schématisée :

On peut alors regrouper l'ensemble des fonctions et objets servant à appliquer une tâche dans un seul fichier qui formera alors un ..... Un ..... peut-être vu comme une .....

expl1 : considérons qu'une personne est caractérisé par son nom, son année de naissance et son téléphone.

Si l'on souhaite que l'utilisateur ne puisse que créer, afficher et avoir accès au nom et à l'âge (sans l'année de naissance), il nous faut prévoir des fonctions qui lui permettront de faire tout cela.

Si au cours de l'implémentation, on souhaite écrire une fonction nécessaire au module mais ne faisant pas partie de l'interface utilisateur, on peut écrire son nom précédé par le caractère _ (underscore).

L'interface sera composée des fonctions :

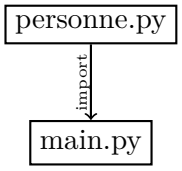
- `creer(nom : ..., annee : ..., tel : ...) -> tuple`
- `afficher(personne : ......) -> None`
- `nom(personne : ......) -> ...`
- `age(personne : ......) -> ...`

Le fichier `personne.py` se présente ainsi (sans commentaire avec la programmation défensive désactivée) :

```
DEBUG = ...

def creer(nom : ..., annee : ..., tel : ...) -> tuple :
    _verif( ..., ......, ...)
    return ...

def afficher(personne : ......) -> None :
    print( f"{personne[0]}, né(e) en {personne[1]}, à contacter au {personne[2]}" )

def nom(personne : ......) -> ... :
    return ...

def age(personne : ......) -> ... :
    import time
    sonAge = ...
    _verif2( sonAge )
    return ...

def _verif(nom : ..., annee : ..., tel : ...) -> None :
    global DEBUG
    assert not DEBUG or ( isinstance(nom, str) \
                          and isinstance(......, ...) \
                          and isinstance(..., ...) \
                          and ... > 0 )

def _verif2(sonAge : ...) -> None :
    global DEBUG
    assert ...
```

On remarque que les fonctions `_verif()` et `_verif2()` sont des fonctions cachées pour l'utilisateur.

Le fichier principal peut alors faire appel au module `personne` en l'important comme une bibliothèque. Si les fonctions importées ne sont pas précisées (*cf.* exemple suivant), nous sommes obligés d'écrire le nom du module devant chaque fonction du-dit module (*e.g.* personne.creer()).



Le fichier `main.py` se présente ainsi

```
import personne

p1 = personne.creer('Maelle', 2002, -6789)
personne.afficher(p1)   # affcihe ...
print( f"Cette année, cette personne a {personne.age(p1)} ans" ) # affiche ...
```

Quelques remarques :

◇ on peut également importer précisément les fonctions nécessaires avec `from module import fonctions`.

◇ on remarque que la variable `DEBUG` est propre au fichier `personne.py` et en définir une nouvelle dans le fichier principal n'a aucun effet (le téléphone −6789 ne lève pas d'**AssertionError**).

```
from personne import creer, afficher, age
DEBUG = True

p1 = creer('Maelle', 2002, -6789)
afficher(p1) # affiche ...
print( f"Cette année, cette personne a {age(p1)} ans" ) # affiche ...
```

◇ si le module se trouve dans un autre répertoire que votre fichier principal, on pourra ajouter temporairement au path `python` le répertoire contenant le module avec `import sys` `sys.path.insert(0, "/path/to/module/")` .

II Programmation Orientée Objets

La programmation orientée objets est l'un des nombreux paradigmes de programmation parmi lesquels la programmation impérative, fonctionnelle, par contrainte, récursive, etc.

1 Définition et création d'instances

Les modules sont pratiques pour définir un ensemble de méthodes sur les structures natives du langage de programmation.

Si l'on souhaite définir une nouvelle structure avec différents champs (on parle .....), on la définira au travers d'une classe en ajoutant dans la classe toutes les fonctions que l'on autorisera sur cette structure (on parle de .....).

expl2 : On considère toujours une personne définie par son nom, son année de naissance et son téléphone.
Le nom de la classe sera **Personne** et chaque instance (= .....) de la classe aura trois ..... : nom, annee, telephone.

Personne
nom (str)
annee (int)
telephone (int)

On peut alors définir en `python` cette classe et deux objets de cette classe ainsi :

```
class Personne: # avec ...
    """
    une personne est définie par son nom, son annee de naissance et son téléphone
    """
    def __init__(self, n : ..., a : ..., t : ...) -> 'Personne' :
        self.nom = ...
        self.annee = ...
        self.telephone = ...
```

```
objet1 = Personne('Maelle', 2002, 6789)
objet2 = Personne('Mouad', 2003, 6701)
```

Lors de l'appel `Personne('Maelle', 2002, 6789)` un objet de la classe `Personne` est créé et la fonction `__init__` (.....) est appelée avec les paramètres `('Maelle', 2002, 6789)`.

2 Accès aux attributs et méthodes

On peut accéder aux attributs d'une instance de classe en écrivant le nom de l'instance suivie d'un point et du nom de l'attribut voulu (*e.g.* `objet1.nom`). De même, pour appeler les méthodes définies dans une classe, on peut les appeler en écrivant le nom de l'instance suivie d'un point et du nom de la méthode suivie de parenthèses et des paramètres requis (*e.g.* `objet1.age()`).

expl3 : pour accéder ou modifier un attribut, on peut faire :

```
print( objet1.nom )  # affiche 'Maelle'
objet1.annee = 2005  # modifie la valeur de l'attribut annee à 2005
```

la POO `python` permet d'ajouter à un objet un attribut à la volée (!). Ceci est très dangereux car deux objets d'une même classe pourraient ne pas avoir les mêmes attributs. En règle général, les autres langages acceptant le paradigme de Programmation Orientée Objets interdisent ce fonctionnement.

On peut définir des fonctions (et donc une interface) qui seront liées à la classe et à tous ses objets en définisant les objets dans la classe. Il faut alors que la fonction possède obligatoirement un 1er argument qui soit `self`. En effet, on applique la fonction à un objet et c'est cet objet qui est le 1er argument.

expl4 :

```
class Personne:
    ...

    def afficher(self) -> None :    # on préfèrera __str__() : cf les méthodes spéciales
        print( f"{self.nom}, né(e) en {self.annee}, à contacter au {self.telephone}" )

    def leNom(...) -> ... :
        return self....

    def _ageEn(..., an : ...) -> ... :    # c'est une fonction privée
        return an - self.annee

    def age(...) -> int :
        import time
        return self._ageEn( ............ )

objet1 = Personne('Maelle', 2002, 6789)
objet1.afficher()    # affiche ...
print( f"Aujourd'hui, { objet1.leNom() } a { objet1.age() } ans." )    # affiche ...
```

attribut et méthode doivent avoir ête nommés différemment. Hors d'un module, on cherchera à ne pas accéder directement à un attribut mais à toujours passer par l'interface du module.

3 Méthodes spéciales

De nombreuses méthodes sont définies par défaut. On peut les redéfinir de manière à pouvoir utiliser les structures habituelles de `python`.

fonction	but	exemple d'appel	contraire
__init__	créé l'objet de la classe (le constructeur)	`obj=MyClass(val)`	-
__repr__	(souvent) affiche comment recréer l'objet	`obj`	-
__str__	renvoie un str qui rend l'objet lisible	`str( obj )`	-
__contains__	précise l'objet contient un élément	`e in obj`	-
__getitem__	renvoie l'élément i	`obj[i]`	__setitem__
__len__	renvoie la longueur de l'objet	`len( obj )`	-
__add__	additionne deux objets	`obj + obj2`	__sub__
__eq__	vérifie l'égalité de deux objets	`obj == obj2`	__ne__
__lt__	vérifie si l'objet est le plus petit (less than)	`obj < obj2`	__ge__
__gt__	vérifie si l'objet est le plus grand (greater than)	`obj > obj2`	__le__

Ainsi plutôt que créer une fonction d'affichage, on préfèrera définir la fonction `__str__`.
Pour savoir si Maelle est strictement plus jeune que Mouad, on définira la fonction `__lt__`.

```
class Personne:
    ...

    def __str__(self) -> str :
        return f"{self.nom}, né(e) en {self.annee}, à contacter au {self.telephone}"

    def __repr__(self) -> str :
        return f"Personne('{self....}', {.........}, {.........})"

    def __lt__(self, obj2) -> bool :
        return ............................    # self est plus jeune si né(e) après obj2

objet1 = Personne('Maelle', 2002, 6789)
objet2 = Personne('Mouad', 2003, 6701)
print( objet1 )    # affiche ...
print( objet1 < objet2 )    # affiche ...
```

Dans le shell, l'appel de la fonction `__repr__` renverra
```
In [1]: objet1
Out[1]: Personne('Maelle', 2002, 6789)
```
.

Rq : par défaut si la fonction `__str__` n'est pas définie, `python` appelle la fonction `__repr__`.

Rq : la notion d'héritage est hors-programme mais essentielle en POO. On peut définir une sous-classe à toute classe. Cette sous-classe hérite alors de tous les attributs et méthodes de la classe parente, auquel on ajoute les attributs et méthodes propres à la sous-classe.

expl5 : l'exception AssertionError est une sous-classe de la classe Exception.