objectifs
- rappeler les avantages de la programmation modulaire ;
- apprendre à créer une classe ;
- apprendre à créer des objets du type de la classe et à les manipuler ;
- apprendre certaines spécificités du langage python concernant les classes.

1 Un module pour représenter les dates

Dans une application (par exemple un agenda), on souhaite pouvoir manipuler des dates.

Pour cela nous allons créer un module qui contiendra les différentes fonctions manipulant les dates. Rassembler les fonctions dans un seul module a plusieurs avantages :

- favoriser la réutilisabilité ;
- rendre le code robuste face aux changements de représentations des dates (les fonctions utilisant le module date ne se soucient pas de la représentation interne des dates) ;
- rendre le code plus lisible.

En résumé … c'est une bonne pratique.

1.1 Spécification du module date

Quels sont les opérations que l'on souhaite réaliser sur les dates ?

- **creer une date** (j, m, a)
- **représenter une date sous forme de chaîne**
- **tester l'égalité de deux dates**
- **savoir si une date est avant/après une autre**
- **obtenir la date du lendemain**
- **calculer la date après** `$n$` **jours**
- ..

1.2 Une première version du module date

Pour représenter une date, on peut utiliser un type construit, à savoir :

- un tuple de longueur trois ou,
- une liste de longueur trois

Voici une première implantation du module date utilisant des tuples.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

"""
:mod:`date1` module :  a  module for date

:author: `FIL - Faculté des Sciences et Technologies -
          Univ. Lille <http://portail.fil.univ-lille1.fr>`_

:date: 2024, january. Last revision: 2026, january
"""

NOM_MOIS = ['janvier', 'février', 'mars', 'avril', 'mai', 'juin', 'juillet',
            'août', 'septembre', 'octobre', 'novembre', 'décembre']
DUREE_MOIS = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]


def est_bissextile(annee: int) -> bool:
    """Renvoie True si et seulement si annee est bissextile.

    précondition: annee >= 1582.

    $$$ est_bissextile(2024)
```

```
    True
    $$$ est_bissextile(2000)
    True
    $$$ est_bissextile(2100)
    False
    """
    return annee % 4 == 0 and (annee % 100 != 0 or annee % 400 == 0)


def nombre_de_jour_dans_mois(mois: int, annee: int) -> int:
    """Renvoie le nombre de jours dans le mois `mois` de l'année `année`.

    précondition : 0 <= mois < 12 et annee >= 1582.
    """
    duree_mois = DUREE_MOIS[mois - 1]
    if est_bissextile(annee) and mois == 2:
        return duree_mois + 1
    return duree_mois


def nom_mois(mois: int) -> str:
    """Renvoie le nom du mois en français."""
    return NOM_MOIS[mois - 1]


def cree_date(jour: int, mois: int, annee: int) -> tuple[int, int, int]:
    """Initialise une nouvelle date.

    Précondition : jour/mois/annee est une date valide

    Exemples :
    $$$ adate = cree_date(4, 6, 2024)
    $$$ isinstance(adate, tuple)
    True
    """
    return (jour, mois, annee)


def en_chaine(adate: tuple[int, int, int]) -> str:
    """Renvoie une chaîne représentant la date.

    Précondition : \

    Exemples :
    $$$ adate = cree_date(23, 1, 2024)
    $$$ en_chaine(adate)
    '23 janvier 2024'
    """
    jour, mois, annee = adate
    return f"{jour} {nom_mois(mois)} {annee}"


def sont_egales(adate: tuple[int, int, int],
                other: tuple[int, int, int]) -> bool:
    """Renvoie True si, et seulement si, deux dates sont égales.

    Exemples :
    $$$ adate1 = cree_date(23, 1, 2024)
    $$$ adate2 = cree_date(23, 1, 2024)
    $$$ id(adate1) == id(adate2)
    False
```

```
    $$$ sont_egales(adate1, adate2)
    True
    """
    jour1, mois1, annee1 = adate
    jour2, mois2, annee2 = other
    return jour1 == jour2 and \
        mois1 == mois2 and \
        annee1 == annee2


def est_avant(adate: tuple[int, int, int],
              other: tuple[int, int, int]) -> bool:
    """Renvoie True si, et seulement si, la date représentée par adate est avant
    celle représentée par other.

    Exemples :
    $$$ adate1 = cree_date(23, 1, 2024)
    $$$ adate2 = cree_date(25, 1, 2024)
    $$$ est_avant(adate1, adate2)
    True
    $$$ est_avant(adate2, adate1)
    False
    $$$ est_avant(adate1, cree_date(23, 1, 2024))
    False
    $$$ est_avant(cree_date(25, 12, 2023), cree_date(1, 2, 2024))
    True
    """
    jour1, mois1, annee1 = adate
    jour2, mois2, annee2 = other
    if annee1 == annee2:
        if mois1 == mois2:
            res = jour1 < jour2
        else:
            res = mois1 < mois2
    else:
        res = annee1 < annee2
    return res


def est_avant_ou_egale(adate: tuple[int, int, int],
                       other: tuple[int, int, int]) -> bool:
    """Renvoie True si, et seulement si, la date représentée par
    self est avant ou egale à celle représentée par other.

    Exemples :
    $$$ adate1 = cree_date(23, 1, 2024)
    $$$ adate2 = cree_date(25, 1, 2024)
    $$$ est_avant_ou_egale(adate1, adate2)
    True
    $$$ est_avant_ou_egale(adate2, adate1)
    False
    $$$ est_avant_ou_egale(adate1, cree_date(23, 1, 2024))
    True
    """
    return est_avant(adate, other) or sont_egales(adate, other)


def lendemain(adate: tuple[int, int, int]) -> tuple[int, int, int]:
    """Renvoie la date du lendemain.

    Exemples :
```

```
    $$$ lendemain(cree_date(31, 12, 2023))
    (1, 1, 2024)
    $$$ lendemain(cree_date(31, 1, 2024))
    (1, 2, 2024)
    $$$ lendemain(cree_date(24, 1, 2024))
    (25, 1, 2024)
    """
    jour, mois, annee = adate
    if jour == nombre_de_jour_dans_mois(mois, annee):
        jour = 1
        if mois == 12:
            annee = annee + 1
            mois = 1
        else:
            mois = mois + 1
    else:
        jour = jour + 1
    return cree_date(jour, mois, annee)


def ajoute_jours(adate: tuple[int, int, int],
                 njour: int) -> tuple[int, int, int]:
    """Ajoute un nombre de jours à une date.

    Exemple :
    $$$ ajoute_jour(cree_date(31, 1, 24), 7)
    (7, 2, 24)
    """
    res = adate
    for _ in range(njour):
        res = lendemain(res)
    return res


def difference(adate: tuple[int, int, int],
               other: tuple[int, int, int]) -> int:
    """Renvoie le nombre de jours entre deux dates.

    Exemple :
    $$$ difference(cree_date(7, 2, 24), cree_date(31, 1, 24))
    7
    """
    if est_avant_ou_egale(adate, other):
        start, ending = adate, other
    else:
        start, ending = other, adate
    res = 0
    while start != ending:
        start = lendemain(start)
        res = res + 1
    return res


if (__name__ == '__main__'):
    import apl1test
    apl1test.testmod('date1.py')
```

Ce module est parfaitement fonctionnel et peut-être utilisé dans un programme de la manière suivante :

```
>>> import date1 as date
>>> d = date.cree_date(3, 2, 26)
>>> print(date.en_chaine(d))
```

```
3 février 26
>>> d2 = date.ajoute_jours(d, 100)
>>> print(date.en_chaine(d2))
14 mai 26
```

1.3 Problèmes introduits par cette représentation

La représentation des dates par des tuples pose quand même un certain nombre de problèmes :

- problème de **typage** : les dates sont du type tuple ;
- problème de **cohérence** : les opérations sur les tuples ne vont pas donner des résultats satisfaisants sur les dates.

```
>>> date1 = date.cree_date(3, 2, 2025)
>>> type(date1)
<class 'tuple'>
>>> date2 = date.cree_date(4, 2, 2024)
>>> date1 < date2
True
```

- incompatibilité avec les opérateurs python : $+$ et $<$ ne fonctionnent pas comme on pourrait s'y attendre
- problème de **responsabilité** : on peut créer des dates, c'est-à-dire des tuples sans avoir recours aux opérations primitives déclarées dans le module.

2 Une nouvelle version utilisant les classes

Pour palier les problèmes évoqués, nous allons définir une *classe*.

2.1 Qu'est-ce qu'une classe ?

- une **classe** permet de définir un nouveaux type : celui des **objets** que l'on peut ensuite créer à partir de la classe ;
- elle définit des **attributs** : chaque objet peut avoir des valeurs différentes pour les attributs.
- elle définit des **méthodes** : ce sont les opérations applicables sur les objets de la classe.

2.2 Création d'une classe

Pour définir une nouvelle classe, on utilise le mot-clef `class` :

```
class A:
    """Une nouvelle classe A.
    Exemples:
    $$$ a=A()
    $$$ isinstance(a, A)
    True
    """
    val = 42
```

Ces lignes définissent une nouvelle classe A. Une *bonne pratique* est de documenter la classe à l'aide d'une docstring.

À l'instar des fonctions , nous devons ensuite créer un ou plusieurs objets de la classe A :

```
obj1 = A()
obj2 = A()
```

Les variables `obj1` et `obj2` sont associées à deux nouvelles instances de la classe `A`. Ces deux instances sont différentes : elle n'ont pas la même identité.

```
>>> id(obj1)
140117805493152
>>> id(obj2)
140117804139536
>>> obj1 == obj2
False
```

Les instances sont bien du type de la classe :

```
>>> type(obj1)
<class '__console__.A'>
>>> isinstance(obj1, A)
True
```

Les instances possèdent un attribut que l'on peut manipuler en utilisant la notation pointée :

```
>>> obj1.val
42
>>> obj2.val
42
```

2.3 Créer une classe avec une méthode

À titre d'exemple, on souhaite ajouter à notre classe une méthode permettant d'incrémenter la valeur de l'attribut `val`. Pour cela nous allons définir une fonction à l'intérieur de la classe.

définition
Une *méthode* est une fonction définie à l'intérieur d'une classe.

```
class A:
    """Une nouvelle classe A.
    Exemples:
    $$$ a=A()
    $$$ isinstance(a, A)
    True
    """
    val = 42

    # une méthode
    def incrementer(self):
        """Incrémente la valeur de l'attribut.

        précondition: \
        """
        self.val = self.val + 1
```

Les objets de la classe `A` disposent maintenant d'une méthode `incrementer`.

L'appel à la méthode `incrementer` s'effectue sur les objets de la classe `A` de la manière suivante :

```
>>> obj = A()
>>> obj.val
42
>>> obj.incrementer()
>>> obj.val
43
```

On remarque que dans la définition de la méthode, nous avons utilisé un paramètre `self`. Ceci est nécessaire (en python) car :

1. lorsque l'on appelle la méthode, elle s'utilise sur une instance de la classe (c'est-à-dire un objet) ;
2. lors de la définition de la classe, ces instances ne sont pas encore créées et il n'y a aucun moyen d'y accéder. `self` désigne donc l'instance sur laquelle portera la méthode.

La classe définie précédemment nous permet d'envisager la programmation d'une nouvelle classe `Date`. Toutefois, la manière de définir les attributs est relativement limitée.

2.4 Construction paramétrée

Comment initialiser de nouveaux objets de manière à ce qu'ils possèdent tous les mêmes attributs (nom et type) et que les valeurs associées à ces attributs puissent être définies à la création de l'objet ?

Dans les langages orientés objets, on utilise pour cela une méthode spéciale permettant d'initialiser un objet. Cette méthode est appelée **constructeur** , même si son rôle n'est pas de construire mais d'initialiser.

En python, le nom de cette méthode spéciale est `__init__`

> En python, tous les noms des méthodes spéciales commençent et finissent par deux blancs soulignés (double underscore), on les appelle également :
> - magic methods, ou
> - dunder methods.

Voici la nouvelle classe `A` avec un constructeur :

```
class A:
    """Une classe d'exemple v2"""

    def __init__(self, init_value: int = 42):
        """Initialise un nouvel objet de la classe.
        précondition: \

        Exemples :
        $$$ a = A()
        $$$ a.val
        42
        $$$ b = A(101)
        $$$ b.val
        101
        """
        self.val = init_value

    def incrementer(self):
        """Incrémente la valeur de l'attribut.
        Précondition : \

        Exemples:
        $$$ a = A()
        $$$ a.incrementer()
        $$$ a.val
        43
        """
        self.val += 1
```

On constate que la valeur associée aux attributs peut être fournie à la construction. La valeur par défaut est 42 et l'initialisation des attributs se fait dans le constructeur (*bonne pratique*).

```
>>> obj1 = A()
>>> obj2 = A(2026)
>>> obj1.incrementer()
>>> obj1.val
43
>>> obj2.val
2026
```

Nous commençons à envisager l'écriture de la classe `Date`. Avant d'en donner une version plus aboutie, nous allons maintenant aborder certaines possibilités offertes par Python.

2.5 Lien avec les opérateurs python

2.5.1 Affichages d'un objet

En python, le programmeur peut solliciter l'affichage d'un objet de deux manières :

1. en utilisant l'instruction `print` ;
2. en utilisant l'instruction `repr`.

Cette dernière est celle appelée par l'interpréteur lorsque l'on évalue un objet.

Pour l'instant, les deux affichages de la classe `A` sont ceux utilisés par la classe `Object` de python :

```
>>> obj1
<__console__.A object at 0x7f6fb805ef90>
```

```
>>> print(obj1)
<__console__.A object at 0x7f6fb805ef90>
```

… pas très satisfaisant.

Pour personnaliser la classe `A`, nous allons définir deux nouvelles méthodes magiques :

- `__str__(self) -> str` qui sera appelée par l'instruction `print` ;'
- `__repr__(self) -> str` qui sera appelée par la fonction `repr`.

La différence sémantique entre les deux méthodes est subtile. En théorie, la méthode `__repr__` devrait fournir une chaîne permettant de créer un nouvel objet égal à celui manipulé (c'est une *bonne pratique*), tandis que la méthode `__str__` est libre.

Voici la nouvelle version de la classe `A` :

```
class A:
    """Une classe d'exemple v3."""

    def __init__(self, init_value: int = 42):
        """Initialise un nouvel objet de la classe.

        Précondition: \

        Exemples :
        $$$ a = A()
        $$$ a.val
        42
        $$$ b = A(101)
        $$$ b.val
        101
        """
        self.val = init_value

    def incrementer(self):
        """Incrémente la valeur de l'attribut.

        Précondition : \

        Exemples:
        $$$ a = A()
        $$$ a.incrementer()
        $$$ a.val
        43
        """
        self.val += 1

    def __repr__(self) -> str:
        """Renvoie une représentation textuelle de self.

        précondition: \

        Exemple :
        $$$ a = A()
        $$$ repr(a)
        'A(42)'
        $$$ a = A(51)
        $$$ repr(a)
        'A(51)'
        """
        return f"A({repr(self.val)})"

    def __str__(self) -> str:
        """Renvoie une représentation textuelle de self.
```

```
        précondition: \

        Exemple :
        $$$ a = A()
        $$$ str(a)
        'Je suis un objet de classe A, associé à la valeur 42'
        $$$ a = A(51)
        $$$ str(a)
        'Je suis un objet de classe A, associé à la valeur 51'
        """
        return f"Je suis un objet de la classe A, associé à la valeur {repr(self.val)}"
```

On constate maintenant que les objets peuvent être affichés dans l'interpréteur et sur la sortie standard.

```
>>> obj1 = A(2026)
>>> obj1
A(2026)
>>> print(obj1)
Je suis un objet de la classe A, associé à la valeur 2026
```

Dans les tests, nous avons utilisé les fonctions `str` et `repr`, qui appellent respectivement les méthodes `__str__` et `__repr__` de la classe. Ce sont ces fonctions qui sont respectivement utilisées à leur tour par `print` et l'interpréteur.

Fournir une représentation d'une classe est une *bonne pratique* : elle facilite la lisibilité et permet un déboggage plus facile.

2.5.2 Égalité de deux objets

Nous savons maintenant programmer une classe disposant d'affichage. Dans cette partie nous allons nous intéresser à la comparaison d'objet. Commençons par comparer l'objet précédent à 2026 :

```
>>> obj1 == 2026
False
```

le résultat est celui attendu : même si la variable `obj1` est associée à un objet portant la valeur 2026, cet objet n'est pas du type `int`.

Créons maintenant un nouvel objet du type `A`, associé à la valeur 2026 et comparons les deux variables.

```
>>> obj2 = A(2026)
>>> obj1 == obj2
False
```

Ici le résultat est moins prévisible : nous constatons que la comparaison des deux variables porte sur leur identité. En effet l'indentité de l'objet associé à `obj1` est différente de celle associé à `obj2`. Pour s'en convaincre nous pouvons utiliser la fonction `id` qui renvoie l'identité d'un objet :

```
>>> id(obj1)
140117805493152
>>> id(obj2)
140117804139216
```

Par défaut, la comparaison s'effectue donc par identité. Cependant nous pouvons redéfinir ce comportement en utilisant la méthode spéciale `__eq__`.

Plus précisement, la comparaison `o1 == o2` entre deux objets est égale à `o1.__eq__(o2)`.

Voici une méthode `__eq__` permettant de comparer deux objets de la classe `A` :

```
    def __eq__(self, other: Any) -> bool:
        """Renvoie True ssi les deux objet1 self et other sont égaux.

        Précondition: \

        Exemples:
```

```
        $$$ obj1 = A(2026)
        $$$ obj1 == 2026
        False
        $$$ obj2 = A()
        $$$ obj1 == obj2
        False
        $$$ obj3 = A(2026)
        $$$ id(obj1) == id(obj3)
        False
        $$$ obj1 == obj3
        True
        """
        if not isinstance(other, A):
            return False
        # ici other est bien de la classe A
        return self.val == other.val
```

La comparaison vérifie d'abord que le type de l'objet associé au paramètre `other` est `A`, puis effectue la comparaison des valeurs associées aux attributs. Il est important de respecter cet ordre car l'attribut `val` n'est défini (à priori) que pour les objets de la classe `A`.

3 Deuxième version de date

Nous sommes maintenant prêts pour écrire une deuxième version du module `date`. Cette version définit une classe `Date` disposant des fonctionnalités (sous forme de méthodes) évoquées plus haut :

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

"""
:mod:`date` module :  a  module for date

:author: `FIL - Faculté des Sciences et Technologies -
          Univ. Lille <http://portail.fil.univ-lille1.fr>`_

:date: 2024, january. Last revision: 2025, january

Date are objects
"""

from typing import Self


NOM_MOIS = ['janvier', 'février', 'mars', 'avril', 'mai', 'juin', 'juillet',
            'août', 'septembre', 'octobre', 'novembre', 'décembre']
DUREE_MOIS = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

def est_bissextile(annee: int) -> bool:
    """
    Renvoie True si et seulement si annee est bissextile.

    Précondition: annee >= 1582.

    Exemples:
    $$$ est_bissextile(2024)
    True
    $$$ est_bissextile(2000)
    True
    $$$ est_bissextile(2100)
    False
    """
```

```
    return annee % 4 == 0 and (annee % 100 != 0 or annee % 400 == 0)


def nombre_de_jour_dans_mois(mois: int, annee: int) -> int:
    """
    Renvoie le nombre de jour dans le mois `mois` de l'année `année`.

    Précondition : 0 <= mois < 12 et annee >= 1582.
    """
    duree_mois = DUREE_MOIS[mois - 1]
    if est_bissextile(annee) and mois == 2:
        return duree_mois + 1
    return duree_mois


def nom_mois(mois) -> str:
    """Renvoie le nom du mois en français."""
    return NOM_MOIS[mois - 1]


class Date:
    """Une classe permettant de représenter des dates."""

    def __init__(self, jour: int, mois: int, annee: int):
        """Initialise une nouvelle date.

        Précondition : jour/mois/annee est une date valide

        Exemples :
        $$$ adate = Date(4, 6, 2024)
        $$$ type(adate) == Date
        True
        """
        self.jour = jour
        self.mois = mois
        self.annee = annee

    def __str__(self) -> str:
        """Renvoie une chaîne représentant la date.
        Exemples:
        $$$ adate = Date(23, 1, 2024)
        $$$ str(adate)
        '23 janvier 2024'
        """
        return f"{self.jour} {nom_mois(self.mois)} {self.annee}"

    def __repr__(self) -> str:
        """Renvoie une représentation textuelle d'une date."""
        return f"Date({self.jour}, {self.mois}, {self.annee})"

    def __eq__(self, other: Self) -> bool:
        """Renvoie True si, et seulement si, deux dates sont égales.

        Exemples :
        $$$ adate1 = Date(23, 1, 2024)
        $$$ adate2 = Date(23, 1, 2024)
        $$$ id(adate1) == id(adate2)
        False
        $$$ adate1 == adate2
        True
        """
```

```
        if not isinstance(other, Date):
            return False
        return self.jour == other.jour and \
            self.mois == other.mois and \
            self.annee == other.annee

    def __lt__(self, other: Self) -> bool:
        """Renvoie True si, et seulement si, la date représentée par
        self est avant celle représentée par other.

        Exemples :
        $$$ adate1 = Date(23, 1, 2024)
        $$$ adate2 = Date(25, 1, 2024)
        $$$ adate1 < adate2
        True
        $$$ adate2 < adate1
        False
        $$$ adate1 < Date(23, 1, 2024)
        False
        """
        if self.annee == other.annee:
            if self.mois == other.mois:
                res = self.jour < other.jour
            else:
                res = self.mois < other.mois
        else:
            res = self.annee < other.annee
        return res

    def __le__(self, other: Self) -> bool:
        """Renvoie True si, et seulement si, la date représentée par
        self est avant ou egale à celle représentée par other.

        Exemples :
        $$$ adate1 = Date(23, 1, 2024)
        $$$ adate2 = Date(25, 1, 2024)
        $$$ adate1 <= adate2
        True
        $$$ adate2 <= adate1
        False
        $$$ adate1 <= Date(23, 1, 2024)
        True
        """
        return self < other or self == other

    def __gt__(self, other: Self) -> bool:
        """Renvoie True si, et seulement si, la date représentée par
        self est après celle représentée par other.

        Exemples :
        $$$ adate1 = Date(23, 1, 2024)
        $$$ adate2 = Date(25, 1, 2024)
        $$$ adate1 > adate2
        False
        $$$ adate2 > adate1
        True
        $$$ adate1 > Date(23, 1, 2024)
        False
        """
        return not (other <= self)
```

```
    def tomorrow(self) -> Self:
        """Renvoie la date du lendemain.

        Exemples:
        $$$ Date(31, 12, 2023).tomorrow() == Date(1, 1, 2024)
        True
        $$$ Date(31, 1, 2024).tomorrow() == Date(1, 2, 2024)
        True
        $$$ Date(24, 1, 2024).tomorrow() == Date(25, 1, 2024)
        True
        """
        annee = self.annee
        mois = self.mois
        jour = self.jour

        if jour == nombre_de_jour_dans_mois(mois, annee):
            jour = 1
            if mois == 12:
                annee = annee + 1
                mois = 1
            else:
                mois = mois + 1
        else:
            jour = jour + 1
        return Date(jour, mois, annee)

    def __add__(self, njour: int) -> Self:
        """Ajoute un nombre de jours à une date.

        Précondition: njour >= 0
        Exemples:
        $$$ Date(31, 1, 24) + 7
        Date(7, 2, 24)
        """
        res = self
        for _ in range(njour):
            res = res.tomorrow()
        return res

    def __sub__(self, other: Self) -> int:
        """Renvoie le nombre de jours entre deux dates.

        Exemples:
        $$$ Date(7, 2, 24) - Date(31, 1, 24)
        7
        """
        if self <= other:
            start, ending = self, other
        else:
            start, ending = other, self
        res = 0
        while start != ending:
            start = start.tomorrow()
            res = res + 1
        return res


if (__name__ == '__main__'):
    import apl1test
    apl1test.testmod('date.py')
```

Nous remarquons que certaines méthodes spéciales ont été ajoutées pour :

- comparer entre elles les dates avec les opérateurs `<, <=, >` ;
- soustraire deux dates avec l'opérateur `-` : le résultat est un nombre de jours ;
- ajouter un nombre de jours à une date avec l'opérateur `+`.

Voici un tableau résumant les méthodes spéciales à connaître :

opérateur/fonction python	méthode spéciale	sens
constructeur	`__init__`	initialise les attributs
`==`	`__eq__`	égalité de deux objets
repr	`__repr__`	représentation textuelle
str (print)	`__str__`	représentation textuelle
`<`	`__lt__`	strictement inférieur
`>`	`__gt__`	strictement supérieur
`<=`	`__le__`	inférieur ou égal
`>=`	`__ge__`	supérieur ou égal
`+`	`__add__`	ajout de deux objets
`-`	`__sub__`	différence
`*`	`__mul__`	multiplication
`/`	`__truediv__`	division
`//`	`__floordiv__`	division entière
`%`	`__mod__`	modulo
`**`	`__pow__`	exponentiation