
<https://pa.dilla.fr/19>



ACTIVITÉ 1

Définir une classe `Fraction` pour représenter un nombre rationnel. Cette classe possède deux attributs, `num` et `denom`, qui sont des entiers et désignent respectivement le numérateur et le dénominateur. On demande que le dénominateur soit plus particulièrement un entier strictement positif.

1. Écrire le constructeur de cette classe. Le constructeur doit lever une `ValueError` si le dénominateur fourni n'est pas strictement positif.
2. Ajouter une méthode `__str__` qui renvoie une chaîne de caractères de la forme `"12 / 35"`, ou simplement de la forme `"12"` lorsque le dénominateur vaut 1.
3. Ajouter des méthodes `__eq__` et `__lt__` qui reçoivent une deuxième fraction en argument et renvoient `True` si la première fraction représente respectivement un nombre égal ou un nombre strictement inférieur à la deuxième fraction.
4. Ajouter des méthodes `__add__` et `__mul__` qui reçoivent une deuxième fraction en argument et renvoient une nouvelle fraction représentant respectivement la somme et le produit des deux fractions.
5. Tester ces opérations.
6. (bonus) S'assurer que les fractions sont toujours sous forme réduite.

```
[ ]: from doctest import testmod

class Fraction:

    def __init__(self, num, denom):
        """Nombre rationnel défini par la fraction
        num/denom avec denom > 0

        Args:
            num (int): numérateur
            denom (int): dénominateur > 0

        Exemples:
        >>> nb = Fraction(1,2)
        >>> print(nb.num)
        1
        >>> print(nb.denom)
        2
        """
        if denom <= 0:
            raise ValueError("le dénominateur doit être strictement positif")
        self.num = num
        self.denom = denom

    def __str__(self):
        """Affichage de la classe Fraction

        Exemple :
        >>> print(Fraction(1,2))
        1 / 2
        >>> print(Fraction(5,1))
        5
        """
        if self.denom == 1:
            return str(self.num)

        texte = str(self.num) + " / " + str(self.denom)
        return texte

    def __eq__(self, frac):
        """Est ce que le nombre rationnel est égal à frac?

        Args:
            frac (Fraction): nombre rationnel à comparer

        Exemples:
        >>> Fraction(1,2) == Fraction(1,2)
        True
        >>> Fraction(1,2) == Fraction(2,1)
        False
        >>> Fraction(1,2) == Fraction(5,10)
        True
        """
        if self.denom == frac.denom:
            return self.num == frac.num

        # méthode du produit en croix pour tester égalité
        # l'avantage est de ne comparer que des nombres entiers
        # et pas des float
        return self.num * frac.denom == self.denom * frac.num
```

```
def __lt__(self, frac):
    """Est ce que le nombre rationnel est inférieur à frac?

    Args:
        frac (Fraction): nombre rationnel à comparer

    Returns:
        bool: True si inférieur à frac

    Exemple:
    >>> Fraction(5,7) < Fraction(6,7)
    True
    >>> Fraction(5,7) < Fraction(4,7)
    False
    >>> Fraction(5,7) < Fraction(10,14)
    False
    >>> Fraction(3,4) < Fraction(1,2)
    False
    """
    if self.denom == frac.denom:
        return self.num < frac.num

    # méthode de comparaison par le produit en croix
    # manipulation d'entiers (moins de pb que les float)
    return self.num * frac.denom < self.denom * frac.num


def __mul__(self, frac):
    """résultat du produit par frac

    Args:
        frac (Fraction): nb rationnel avec lequel on multiplie

    Returns:
        Fraction: nb rationnel produit des deux fractions

    Exemple:
    >>> Fraction(1,3) * Fraction(3,5) == Fraction(1,5)
    True
    """
    num = self.num * frac.num
    denom = self.denom * frac.denom
    return Fraction(num, denom)


def __add__(self, frac):
    """
    >>> Fraction(1,2) + Fraction(3,5) == Fraction(11,10)
    True
    >>> Fraction(1,2) + Fraction(50,50) == Fraction(3,2)
    True
    """
    denom = self.denom * frac.denom
    num = self.num * frac.denom + frac.num * self.denom
    return Fraction(num, denom)
```

```
testmod()
```



ACTIVITÉ 2

Définir une classe `Intervalle` représentant des intervalles de nombres. Cette classe possède deux attributs `a` et `b` représentant respectivement l'extrémité inférieure et l'extrémité supérieure de l'intervalle. Les deux extrémités sont considérées comme incluses dans l'intervalle. Tout intervalle avec $b < a$ représente l'intervalle vide.

1. Écrire le constructeur de la classe `Intervalle` et une méthode `est_vide` renvoyant `True` si l'objet représente l'intervalle vide et `False` sinon.
2. Ajouter des méthodes `__len__` renvoyant la longueur de l'intervalle (l'intervalle vide a une longueur 0) et `__contains__` testant l'appartenance d'un élément `x` à l'intervalle.
3. Ajouter une méthode `__eq__` permettant de tester l'égalité de deux intervalles avec `==` et une méthode `__le__` permettant de tester l'inclusion d'un intervalle dans un autre avec `<=`. Attention : toutes les représentations de l'intervalle vide doivent être considérées égales, et incluses dans tout intervalle.
4. Ajouter des méthodes `intersection` et `union` calculant respectivement l'intersection de deux intervalles et le plus petit intervalle contenant l'union de deux intervalles (l'intersection est bien toujours un intervalle, alors que l'union ne l'est pas forcément). Ces deux fonctions doivent renvoyer un nouvel intervalle sans modifier leurs paramètres.
5. Tester ces méthodes.

```
[ ]: class Intervalle:

    def __init__(self, debut, fin):
        self.a = debut
        self.b = fin

    def est_vide(self):
```

```
    return self.b < self.a

def __len__(self):
    if self.est_vide() :
        return 0
    return self.b - self.a

def __contains__(self, x):
    if self.est_vide():
        return False
    if x > self.b or x < self.a:
        return False
    return True

def __eq__(self, inter2):
    if self.est_vide() and inter2.est_vide():
        return True

    if self.a == inter2.a and self.b == inter2.b:
        return True

    return False

def __le__(self, inter2):
    if self.est_vide():
        return True

    return self.a >= inter2.a and self.b <= inter2.b

def intersection(self, inter2):
    if self.est_vide():
        return self

    if inter2.est_vide():
        return inter2

    tmp_a = max(self.a, inter2.a)
    tmp_b = min(self.b, inter2.b)

    return Intervalle( tmp_a, tmp_b )

def reunion(self, inter2):
    if self.est_vide():
        return inter2
    if inter2.est_vide():
        return self

    intersec = self.intersection(inter2)
    if intersec.est_vide():
        if len(self) <= len(inter2):
            return self
        else:
            return inter2

    tmp_a = min(self.a, inter2.a)
    tmp_b = max(self.b, inter2.b)
```

```

        return Intervalle(tmp_a, tmp_b)

    def __str__(self):
        return "[" + str(self.a) + " ; " + str(self.b) + "]"

```

```

[ ]: # Tests des méthodes

mon_inter = Intervalle(5,12)
print("mon_inter.a:", mon_inter.a)
print("mon_inter.b:", mon_inter.b)

mon_inter2 = Intervalle(5, 3)
print("mon_inter2.a:", mon_inter2.a)
print("mon_inter2.b:", mon_inter2.b)

print("mon_inter.est_vide():" , mon_inter.est_vide() )
print("mon_inter2.est_vide():", mon_inter2.est_vide() )

print("len(mon_inter):" , len(mon_inter) )
print("len(mon_inter2):", len(mon_inter2) )

print("mon_inter.__contains__(5):", mon_inter.__contains__(5) )
print("5 in mon_inter:", 5 in mon_inter )

print( "3 in mon_inter:", 3 in mon_inter )
print( "12.001 in mon_inter:", 12.001 in mon_inter )

print( "mon_inter == mon_inter2:", mon_inter == mon_inter2)

print("Intervalle(4,10) <= Intervalle(5,12):", Intervalle(4,10) <= Intervalle(5,12) )

print("[4,6] inters. [5,10] == [5, 6]:", Intervalle(4,6).intersection(Intervalle(5,10)) ==
↪Intervalle(5,6) )

print("[4,6] reunion [9,10] == [9,10]:", Intervalle(4,6).reunion(Intervalle(9,10)) ==
↪Intervalle(9,10) )
print("[4,6] reunion [5, 8] == [4, 8]:", Intervalle(4,6).reunion(Intervalle(5, 8)) == Intervalle(4,
↪8) )

```

Voici ci-dessous une version du code précédent :

- avec une documentation correcte
- avec des tests unitaires utilisant la bibliothèque `doctest`.

```

[ ]: from doctest import testmod

class Intervalle:
    def __init__(self, debut, fin):
        """Intervalle d'extrémité [a ; b]

        Args:
            debut (int): borne inf de l'intervalle
            fin (int): borne sup de l'intervalle

        Exemples:
        >>> inter = Intervalle(5,12)
        >>> inter.a

```

```
5
>>> inter.b
12
"""
self.a = debut
self.b = fin

def est_vide(self):
    """Est ce que l'intervalle est vide?
    Est considéré vide un intervalle
    [a,b] dont b <= a.

    Returns:
        bool: True si intervalle vide

    Exemple:
    >>> inter1 = Intervalle(5,12)
    >>> inter1.est_vide()
    False
    >>> inter2 = Intervalle(5, 3)
    >>> inter2.est_vide()
    True
    """
    return self.b < self.a

def __len__(self):
    """Longueur d'un intervalle.

    Returns:
        int: longueur de l'intervalle [a,b]

    Exemple:
    >>> len(Intervalle(5,12))
    7
    >>> len(Intervalle(5,3))
    0
    """
    if self.est_vide() :
        return 0
    return self.b - self.a

def __contains__(self, x):
    """Est ce que la valeur x appartient
    à l'intervalle [a,b]?

    Args:
        x (float/int): nombre

    Returns:
        bool: True ssi x in [a,b]

    Exemples:
    >>> 6 in Intervalle(5,12)
    True
    >>> 5 in Intervalle(5,12)
    True
    >>> 4.9 in Intervalle(5,12)
    False
    """
    if self.est_vide():
```

```
        return False
    if x > self.b or x < self.a:
        return False
    return True

def __eq__(self, inter):
    """Est ce que l'intervalle est
    égale à l'intervalle inter

    Args:
        inter2 (Intervalle): Intervalle à comparer

    Returns:
        bool: True ssi les deux intervalles ont les
        mêmes bornes

    Exemple:
    >>> Intervalle(5,12) == Intervalle(5,7)
    False
    >>> Intervalle(5,12) == Intervalle(5,12)
    True
    >>> Intervalle(5,3) == Intervalle(7,2)
    True
    """
    if self.est_vide() and inter.est_vide():
        return True

    if self.a == inter.a and self.b == inter.b:
        return True

    return False

def __le__(self, inter):
    """Est ce que l'intervalle est inclu
    dans l'intervalle inter?

    Args:
        inter (Intervalle): Intervalle [a,b]

    Returns:
        bool: True ssi l'intervalle est inclus dans inter

    Exemples:
    >>> Intervalle(4,10) <= Intervalle(5,12)
    False
    >>> Intervalle(5,12) <= Intervalle(4,10)
    False
    >>> Intervalle(6,10) <= Intervalle(5,12)
    True
    """
    if self.est_vide():
        return True

    return self.a >= inter.a and self.b <= inter.b

def intersection(self, inter):
    """Intersection avec l'intervalle inter.

    Args:
        inter (Intervalle)
```



```
Returns:
    Intervalle: égal à l'intersection de self
    avec inter.

Exemples:
>>> Intervalle(4,6).intersection(Intervalle(5,10)) == Intervalle(5,6)
True
"""
if self.est_vide():
    return self

if inter.est_vide():
    return inter

tmp_a = max(self.a, inter.a)
tmp_b = min(self.b, inter.b)

return Intervalle( tmp_a, tmp_b )

def reunion(self, inter):
    """Réunion avec l'intervalle inter. Si la réunion
    n'est pas un intervalle, renvoie le plus petit
    des deux intervalles.

    Args:
        inter (Intervalle)

    Returns:
        Intervalle: réunion avec inter

    Exemples :
    >>> Intervalle(4,6).reunion(Intervalle(9,10)) == Intervalle(9,10)
    True
    >>> Intervalle(4,6).reunion(Intervalle(5,8)) == Intervalle(4,8)
    True
    """
    if self.est_vide():
        return inter
    if inter.est_vide():
        return self

    intersec = self.intersection(inter)
    if intersec.est_vide():
        if len(self) <= len(inter):
            return self
        else:
            return inter

    tmp_a = min(self.a, inter.a)
    tmp_b = max(self.b, inter.b)

    return Intervalle(tmp_a, tmp_b)
```

```
testmod()
```

**ACTIVITÉ 3**

Définir une classe `Angle` pour représenter un angle en degrés. Cette classe contient un unique attribut, `angle`, qui est un entier. On demande que, quoiqu'il arrive, l'égalité $0 \leq \text{angle} < 360$ reste vérifiée.

1. Écrire le constructeur de cette classe.
2. Ajouter une méthode `__str__` qui renvoie une chaîne de caractères de la forme "60 degrés". Observer son effet en construisant un objet de la classe `Angle` puis en l'affichant avec `print`.
3. Ajouter une méthode `ajoute` qui reçoit un autre angle en argument (un objet de la classe `Angle`) et l'ajoute au champ `angle` de l'objet. Attention à ce que la valeur d'`angle` reste bien dans le bon intervalle.
4. Ajouter deux méthodes `cos` et `sin` pour calculer respectivement le cosinus et le sinus de l'angle. On utilisera pour cela les fonctions `cos` et `sin` de la bibliothèque `math`. Attention : il faut convertir l'angle en radians (en le multipliant par $\pi/180$) avant d'appeler les fonctions `cos` et `sin`.
5. Tester les méthodes `ajoute`, `cos` et `sin`.

```
[ ]: # version avec tests unitaires
# de toutes les méthodes
from doctest import testmod

class Angle:
    """Angles compris entre 0 et 360°
    """

    def __init__(self, mesure):
        """Angle de mesure entière comprise entre 0 et 360.

        Args:
            mesure (int): mesure de l'angle

        Exemple:
        >>> a = Angle(90)
        >>> a.angle
        90
        >>> a = Angle(365)
```

```

>>> a.angle
5
"""
mesure = Angle._ajuste(mesure)
self.angle = mesure

def _ajuste(angle):
    """Méthode de classe auxiliaire (hors interface)
    utilisée dans le constructeur __init__ pour
    vérifier, quoi qu'il arrive la contrainte d'un
    angle compris entre 0 et 360.

    Renvoie la valeur angle restreinte à
    l'intervalle [0 ; 360[.

    Args:
        angle (int): mesure d'un angle

    Returns:
        int: valeur ajustée de l'angle

    >>> Angle._ajuste(360)
    0
    >>> Angle._ajuste(365)
    5
    >>> Angle._ajuste(90)
    90
    """
    return angle % 360

def __str__(self):
    """Affiche un angle sous la forme
    '90 degrés'

    Returns:
        str:

    Exemples:
    >>> print(Angle(90))
    90 degrés
    >>> print(Angle(390))
    30 degrés
    """
    return str(self.angle) + " degrés"

def ajoute(self, obj):
    """Ajoute l'angle actuel avec un autre angle obj

    Args:
        obj (Angle): angle à ajouter

    Returns:
        Angle: Angle de mesure égale à la somme des deux angles

    Exemples:
    >>> a1 = Angle(90)
    >>> a2 = Angle(350)
    >>> print( a1.ajoute(a2) )
    80 degrés
    """

```

```

mesure = self.angle + obj.angle
return Angle(mesure)

def _radian(mesure):
    """Methode de classe auxiliaire utilisée
    dans cos et sin.
    Convertit la mesure en degré en radian

    Args:
        mesure (int): valeur en degré

    Returns:
        float: valeur en radian

    Exemple:
    >>> round( Angle._radian(90) ,5)
    1.5708
    """
    from math import pi
    return mesure * pi / 180

def cos(self):
    """Cosinus de l'angle.

    Returns:
        float: cosinus de la mesure de l'angle

    Exemple :
    >>> round( Angle(45).cos() ,5)
    0.70711
    """
    from math import cos
    return cos( Angle._radian(self.angle) )

def sin(self):
    """Sinus de l'angle

    Returns:
        float: sinus de la mesure de l'angle

    Exemple:
    >>> round ( Angle(45).sin() ,5 )
    0.70711
    """
    from math import sin
    return sin( Angle._radian (self.angle) )

```

testmod()



ACTIVITÉ 4

Définir une classe Date pour représenter une date, avec trois attributs jour, mois et année.

1. Écrire son constructeur.
2. Ajouter une méthode `__str__` qui renvoie une chaîne de caractères de la forme "8 mai 1945". On pourra se servir d'un attribut de classe qui est un tableau donnant les noms des douze mois de l'année. Tester en construisant des objets de la classe `Date` puis en les affichant avec `print`.
3. Ajouter une méthode `__lt__` qui permet de déterminer si une date `d1` est antérieure à une date `d2` en écrivant `d1 < d2`. La tester.

```
[ ]: # version avec des test unitaire

from doctest import testmod

class Date:
    mois_fr = [ None,
                'janvier', 'février', 'mars',
                'avril', 'mai', 'juin',
                'juillet', 'août', 'septembre',
                'octobre', 'novembre', 'décembre']

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

    def __str__(self):
        """Affiche une date en français.
        Par exemple '8 mai 1945'.

        Returns:
            str

        Exemple:
        >>> print(Date(8,5,1945))
        8 mai 1945
        >>> print(Date(5,6,1977))
        5 juin 1977
        """
        date = str(self.jour) + " "
        date += Date.mois_fr[self.mois] + " "
        date += str(self.annee)
        return date

    def __lt__(self, date):
        """Est ce que la date est antérieure à date?

        Args:
            date (Date)

        Returns:
```

bool: True ssi est strictement antérieure à date

Exemples:

```
>>> Date(5,6,1978) < Date(4,6,1978)
False
>>> Date(5,6,1978) < Date(7,6,1978)
True
>>> Date(1,1,2006) < Date(1,1,2006)
False
>>> Date(1,1,2004) < Date(1,1,2006)
True
>>> Date(1,4,2003) < Date(1,5,2003)
True
"""
```

```
if self.annee < date.annee:
    return True
if self.annee > date.annee:
    return False
if self.mois < date.mois:
    return True
if self.mois > date.mois:
    return False
if self.jour < date.jour:
    return True
return False
```

testmod()



ACTIVITÉ 5

Dans certains langages de programmation les tableaux ne sont pas nécessairement indexés à partir de 0. Par exemple, on peut déclarer un tableau dont les indices vont de -10 à 9 si on le souhaite. Dans cet exercice, on se propose de construire une classe `Tableau` pour réaliser de tels tableaux. Un objet de cette classe aura deux attributs, un attribut `premier` qui est la valeur de premier indice et un attribut `contenu` qui est un tableau Python contenant les éléments. Ce dernier est un vrai tableau Python, indexé à partir de 0.

Écrire un constructeur `__init__(self, imin, imax, v)` où `imin` est le premier indice, `imax` le dernier indice et `v` la valeur utilisée pour initialiser toutes les cases du tableau. Ainsi, on peut écrire `t = Tableau(-10, 9, 42)` pour construire un tableau de vingt cases, indexées de -10 à 9 et toutes initialisées avec la valeur 42.

Écrire une méthode `__len__(self)` qui renvoie la taille du tableau.

Écrire une méthode `__getitem__(self, i)` qui renvoie l'élément du tableau `self` d'indice `i`. De même, écrire une méthode `__setitem__(self, i, v)` qui modifie l'élément du tableau `self` d'indice `i` pour lui donner la valeur `v`. Ces deux méthodes doivent vérifier que l'indice `i` est bien valide et, dans le cas contraire, lever l'exception `IndexError` avec la valeur de `i` en argument (c'est-à-dire `raise IndexError (i)`).

Enfin, écrire une méthode `__str__(self)` qui renvoie une chaîne de caractères décrivant le contenu du tableau.

```
[ ]: from doctest import testmod

class TableauIndex():
    def __init__(self, imin, imax, v):
        """Tableau d'indices quelconque.

        Args:
            imin (int): indice de début
            imax (int): indice de fin de tableau
            v (object): valeur initiale de toutes les cellules du tableau

        Exemples:
        >>> t = TableauIndex(-10, 9, 42)
        >>> t.contenu
        [42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
        """
        self.premier = imin
        longueur = imax - imin + 1
        self.contenu = [v] * longueur

    def __len__(self):
        """
        Exemples:
        >>> t = TableauIndex(-10, 9, 42)
        >>> len(t)
        20
        """
        return len(self.contenu)

    def __getitem__(self, i):
        """Exemples:
        >>> t = TableauIndex(-10, 9, 42)
        >>> t[-10]
        42
        """
        if i < self.premier or i >= self.premier + len(self):
            raise IndexError (i)
```

```
    indice = i + self.premier
    return self.contenu[indice]

def __setitem__(self, i, v):
    """Exemples:
    >>> t = TableauIndex(-10, 9, 42)
    >>> t[-10] = 12
    >>> t[-10]
    12
    >>> t[-9]
    42
    """
    if i < self.premier or i >= self.premier + len(self):
        raise IndexError (i)

    indice = i - self.premier
    self.contenu[indice] = v

def __str__(self):
    """Exemples:
    >>> t = TableauIndex(5,9,10)
    >>> print(t)
    [10, 10, 10, 10, 10]
    """
    return str(self.contenu)
```

```
testmod()
```