```python
from math import atan2

class cplx:

    """Une classe permettant d'effectuer des manipulations de base sur les complexes.
    Version du 23 septembre 2016.
    Suggestions pour compléter la classe :
    (1) Exponentiation : z**n avec n un entier quelconque (surcharge partielle de
l'opérateur __pow__).
    (2) Racines nièmes avec n un entier naturel non nul.
    """

    # ========================================================================= #
    # CONSTRUCTION                                                              #
    # Les noms des attributs doivent permettre de lire aisément les caluls ! #
    # self.re : partie réelle du complexe.                                      #
    # self.im : partie imaginaire du complexe.                                  #
    # ========================================================================= #

    def __init__(self,x,y):
        self.re = x
        self.im = y

    # ========================================================================= #
    # Représentation classique du complexe, en tenant compte des situations #
    # particulières ...                                                         #
    # ATTENTION ! La méthode spéciale __repr__ doit renvoyer une chaîne de  #
    # caractères !                                                              #
    # ========================================================================= #

    def __repr__(self):
        s = ''
        if self.re == 0 and self.im == 0:
            return '0'
        else:
            if self.re != 0:
                s = str(self.re)
                if self.im == 0:
                    return s
                else:
                    if self.im > 0:
                        if self.im == 1:
                            s += ' + i'
                        else:
                            s += ' + ' + str(self.im) + ' i'
                    elif self.im < 0:
                        if self.im == -1:
                            s += ' - i'
                        else:
                            s += ' - ' + str(abs(self.im)) + 'i'
            else:
                if self.im > 0:
                    if self.im == 1:
                        s += 'i'
                    else:
                        s += str(self.im) + ' i'
                elif self.im < 0:
                    if self.im == -1:
                        s += ' - i'
                    else:
                        s += str(self.im) + ' i'
        return s

    # ====== #
    # MODULE #
    # ====== #
```

```
def module(self):
    """Renvoie le module du complexe
    """
    return (self.re**2 + self.im**2)**0.5

# ================= #
# COMPLEXE CONJUGUE #
# ================= #

def conj(self):
    """Renvoie le conjugué du complexe
    """
    return cplx(self.re,-self.im)

# ======= #
# INVERSE #
# ======= #

def inverse(self):
    """Renvoie, s'il est défini, l'inverse du complexe
    """
    if self.re != 0 or self.im != 0:
        m2 = self.module()**2
        return cplx(self.re/m2,-self.im/m2)
    else:
        raise ValueError('L\'inverse du complexe nul n\'est pas défini !')

# ======== #
# ARGUMENT #
# ======== #

def argument(self):
    """Renvoie, s'il est défini, l'argument, compris entre -pi et pi, du complexe
    """
    if self.re != 0 or self.im != 0:
        return atan2(self.im,self.re)
    else:
        raise ValueError('Le complexe nul n\'admet pas d\'argument !')

# ================================================================ #
# SURCHARGE DE __add__ et __radd__ POUR LE SUPPORT DE cplx par "+" #
# ================================================================ #

# Méthode spéciale __add__ : surcharge de l'opérateur + afin de pouvoir
# simplement sommer un complexe ... et un autre nombre grâce au symbole +.
# ATTENTION ! Ici, on a traite le cas où le PREMIER TERME de l'addition
# est un complexe !

def __add__(self,z):
    if type(z) != cplx and type(z) != int and type(z) != float:
        raise TypeError('L\'addition est impossible !')
    else:
        if type(z) == cplx:
            return cplx(self.re + z.re,self.im + z.im)
        else:
            return cplx(self.re + z,self.im)

# Méthode spéciale __radd__ : surcharge de l'opérateur + afin de pouvoir
# simplement sommer un complexe ... et un autre nombre grâce au symbole +.
# ATTENTION ! Ici, on a traite le cas où le DEUXIEME TERME de l'addition est
# un complexe !

def __radd__(self,z):
    if type(z) != cplx and type(z) != int and type(z) != float:
        raise TypeError('L\'addition est impossible !')
    else:
        return self + z
```

```
    # ============================================================ #
    # SURCHARGE DE __neg__ POUR LE SUPPORT DE cplx par "-" comme "-z" #
    # ============================================================ #

    def __neg__(self):
        return cplx(-self.re,-self.im)

    # ============================================================ #
    # SURCHARGE DE __sub__ et __rsub__ POUR LE SUPPORT DE cplx par "-" #
    # ============================================================ #

    # Méthode spéciale __sub__ : surcharge de l'opérateur - afin de pouvoir
    # simplement soustraire un complexe ... et un autre nombre grâce au
    # symbole -.
    # ATTENTION ! Ici, on a traite le cas où le PREMIER TERME de la soustraction
    # est un complexe !

    def __sub__(self,z):
        if type(z) != cplx and type(z) != int and type(z) != float:
            raise TypeError('La soustraction est impossible !')
        else:
            if type(z) == cplx:
                return self + (-z)
            else:
                return cplx(self.re - z,self.im)

    # Méthode spéciale __rsub__ : surcharge de l'opérateur - afin de pouvoir
    # simplement soustraire un complexe ... et un autre nombre grâce au
    # symbole -.
    # ATTENTION ! Ici, on a traite le cas où le DEUXIEME TERME de la
    # soustraction est un complexe mais PAS LE PREMIER !

    def __rsub__(self,z):
        if type(z) != cplx and type(z) != int and type(z) != float:
            raise TypeError('La soustraction est impossible !')
        else:
            return -(self - z)

    # ============================================================ #
    # SURCHARGE DE __mul__ et __rmul__ POUR LE SUPPORT DE cplx par "*" #
    # ============================================================ #

    # Méthode spéciale __mul__ : surcharge de l'opérateur * afin de pouvoir
    # simplement multiplier un complexe ... et un autre nombre grâce au
    # symbole *.
    # ATTENTION ! Ici, on a traite le cas où le PREMIER FACTEUR de la
    # multiplication est un complexe !

    def __mul__(self,z):
        if type(z) != cplx and type(z) != int and type(z) != float:
            raise TypeError('La multiplication est impossible !')
        else:
            if type(z) != cplx:
                return cplx(self.re * z,self.im * z)
            else:
                return cplx(self.re * z.re - self.im * z.im,self.re * z.im + self.im
* z.re)

    # Méthode spéciale __rmul__ : surcharge de l'opérateur * afin de pouvoir
    # simplement multiplier un complexe ... et un autre nombre grâce au
    # symbole *.
    # ATTENTION ! Ici, on a traite le cas où le SECOND FACTEUR de la
    # multiplication est un complexe mais PAS LE PREMIER !

    def __rmul__(self,z):
        if type(z) != cplx and type(z) != int and type(z) != float:
```

```
        raise TypeError('La multiplication est impossible !')
    else:
        return self * z

# ========================================================================== #
# SURCHARGE DE __truediv__ et __rtruediv__ POUR LE SUPPORT DE cplx par "/" #
# ========================================================================== #

# Méthode spéciale __truediv__ : surcharge de l'opérateur / afin de pouvoir
# simplement diviser un complexe ... et un autre nombre grâce au symbole /.
# ATTENTION ! Ici, on a traite le cas où le DIVIDENDE de la division est un
# complexe !

def __truediv__(self,z):
    if type(z) != cplx and type(z) != int and type(z) != float:
        raise TypeError('La division est impossible !')
    elif(type(z) == cplx and z.re == 0 and z.im == 0) or z == 0:
        raise ValueError('Division par 0 !')
    else:
        if type(z) != cplx:
            return cplx(self.re / z,self.im / z)
        else:
            return self * z.inverse()

# Méthode spéciale __rtruediv__ : surcharge de l'opérateur / afin de pouvoir
# simplement diviser un complexe ... et un autre nombre grâce au symbole /.
# ATTENTION ! Ici, on a traite le cas où le DIVISEUR de la division est un
# complexe mais PAS LE DIVISEUR !

def __rtruediv__(self,z):
    if (type(z) != cplx and type(z) != int and type(z) != float):
        raise TypeError('La division est impossible !')
    elif self.re == 0 and self.im == 0:
        raise ValueError('Division par 0 !')
    else:
        return self.inverse() * z
```