Université de Lille — Faculté des Sciences et Technologies, Département Informatique

L2 Informatique - 2018–2019

Algorithmes et Programmation 2

TD : Modules

Exercice 1 *Le module* `complex3`

Dans les sessions des premières questions de cet exercice, on suppose que le module `complex3` (vu en cours) a été importé avec la commande **`import complex3 as cplx`**.

Voici la documentation de ce module qui permet de travailler avec des nombres complexes :

```
>>> help(cplx.Complex)
Help on class Complex in module Complex.src.complex3:

class Complex(builtins.object)
 |  class for complex numbers
 |
 |  >>> z1 = Complex(1, 2)
 |  >>> z1.get_real_part()
 |  1.0
 |  >>> z1.get_imag_part()
 |  2.0
 |  >>> z1.modulus() == math.sqrt(5)
 |  True
 |  >>> z2 = Complex.from_real_number(3)
 |  >>> z1.equals(z2)
 |  False
 |  >>> z3 = z1.add(z2)
 |  >>> z3.equals(Complex(4, 2))
 |  True
 |  >>> z4 = z1.mul(z2)
 |  >>> z4.equals(Complex(3, 6))
 |  True
 |  >>> z4
 |  Complex(3.0, 6.0)
 |  >>> print(z1 + z2)
 |  4.000000 + 2.000000i
 |  >>> print(z1 * z2)
 |  3.000000 + 6.000000i
 |  >>> print(-z1)
 |  -1.000000 + -2.000000i
 |  >>> print(z2 - z1)
 |  2.000000 + -2.000000i
 |  >>> z1 * z2 == z4
 |  True
 |  >>> z1 != z2
 |  True
 |
 |  Methods defined here:
 |
 |  __abs__(self)
 |      :return: modulus of self
 |      :rtype: float
 |      :UC: none
 |      :Example:
 |
```

```
|      >>> z = Complex(3, 2)
|      >>> abs(z) == math.sqrt(13)
|      True
|
|  __add__(self, z)
|      :return: the sum of complex numbers self and z
|      :rtype: Complex
|      :UC: none
|      :Example:
|
|      >>> Complex(1, 2) + Complex(3, 4)
|      Complex(4.0, 6.0)
|
|  __eq__(self, z)
|      :return:
|         - True if complex numbers self and z are equals
|         - False otherwise
|      :rtype: bool
|      :UC: none
|      :Example:
|
|      >>> Complex(3, 2) == Complex(3, 2)
|      True
|      >>> Complex(3, 2) == Complex(2, 2)
|      False
|
|  __init__(self, real_part, imag_part)
|      create a complex number with real part real_part and imaginary part imag_part.
|
|      This method is implicitely called at object creation.
|
|      :param real_part:
|      :type real_part: int or float
|      :param imag_part:
|      :type imag_part: int or float
|      :raises: ``AssertionError`` if params are not int or float numbers
|      :Example:
|
|      >>> z = Complex(3, 2)
|      >>> z.get_real_part()
|      3.0
|      >>> z.get_imag_part()
|      2.0
|
|  __mul__(self, z)
|      :return: the product of complex numbers self and z
|      :rtype: Complex
|      :UC: none
|      :Example:
|
|      >>> Complex(1, 2) * Complex(3, 4)
|      Complex(-5.0, 10.0)
|
|  __neg__(self)
|      :return: the opposite complex number -self
|      :rtype: Complex
|      :UC: none
|      :Example:
|
|      >>> -Complex(1, 2)
|      Complex(-1.0, -2.0)
```

```
|
|  __neq__(self, z)
|      :return:
|         - True if complex numbers self and z are not equals
|         - False otherwise
|      :rtype: bool
|      :UC: none
|      :Example:
|
|      >>> Complex(3, 2) != Complex(2, 2)
|      True
|      >>> Complex(3, 2) != Complex(3, 2)
|      False
|
|  __repr__(self)
|      :return: an external representation of complex number self
|         (for visualizing complex numbers value in interactive mode for example)
|      :rtype: str
|      :UC: none
|      :Example:
|
|      >>> Complex(3, 2)
|      Complex(3.0, 2.0)
|
|  __str__(self)
|      :return: an external representation of complex number self
|         (for printing complex numbers for example)
|      :rtype: str
|      :UC: none
|      :Example:
|
|      >>> z = Complex(3, 2)
|      >>> z.__str__()
|      '3.000000 + 2.000000i'
|      >>> print(z)
|      3.000000 + 2.000000i
|
|  __sub__(self, z)
|      :return: the complex number self - z
|      :rtype: Complex
|      :UC: none
|      :Example:
|
|      >>> Complex(1, 2) - Complex(3, 4)
|      Complex(-2.0, -2.0)
|
|  add(self, z)
|      :param z:
|      :type z: Complex
|      :return: the sum of complex numbers self and z.
|      :rtype: Complex
|      :UC: none
|      :Example:
|
|      >>> z = Complex(1, 2).add(Complex(3, 4))
|      >>> z.get_real_part()
|      4.0
|      >>> z.get_imag_part()
|      6.0
|
|  equals(self, z)
```

```
|      :return:
|         - True if complex number self equals complex number z2
|         - False otherwise
|      :rtype: bool
|      :UC: none
|      :Example:
|
|      >>> z = Complex(1, 2)
|      >>> z.equals(Complex(1, 2))
|      True
|      >>> z.equals(Complex(-1, 2))
|      False
|
|  from_real_number(x)
|      create a complex number with real part x and zero imaginary part.
|
|      :param x: (int or float)
|      :return: a new complex number x + 0.0i
|      :rtype: Complex
|      :UC: none
|      :Example:
|
|      >>> z = Complex.from_real_number(3)
|      >>> z.get_real_part()
|      3.0
|      >>> z.get_imag_part()
|      0.0
|
|  get_imag_part(self)
|      return the imaginary part of complex number self
|
|      :return: the imanigary part of self
|      :rtype: float
|      :UC: None
|      :Example:
|
|      >>> z = Complex(3, 2)
|      >>> z.get_imag_part()
|      2.0
|
|  get_real_part(self)
|      return the real part of complex number self
|
|      :return: the real part of self
|      :rtype: float
|      :UC: None
|      :Example:
|
|      >>> z = Complex(3, 2)
|      >>> z.get_real_part()
|      3.0
|
|  modulus(self)
|      :return: modulus of complex number self, ie :math:`\sqrt{x^2 + y^2}`
|          if :math:`z = x + yi`.
|      :rtype: float
|      :UC: none
|      :Example:
|
|      >>> z = Complex(3, 2)
|      >>> z.modulus() == math.sqrt(13)
```

```
|      True
|
|  mul(self, z)
|      :param z:
|      :type z: Complex
|      :return: the product of complex numbers self and z.
|      :rtype: Complex
|      :UC: none
|      :Example:
|
|      >>> z = Complex(1, 2).mul(Complex(3, 4))
|      >>> z.get_real_part()
|      -5.0
|      >>> z.get_imag_part()
|      10.0
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __hash__ = None
```

Question 1 Quels sont les constructeurs et sélecteurs définis dans ce module ?

Question 2 La lecture de cette documentation laisse-t-elle entrevoir de quelle façon les nombres complexes sont implantés ? Du point de vue de l'utilisateur du module, cette connaissance est-elle importante ?

Question 3 Indiquez comment créer les trois nombres complexes $z_1 = 1+2i$ et $z_2 = 3+4i$ et $z_3 = z_1 - 3z2$, puis imprimez le nombre z_3. Quel est l'affichage produit ?

Exercice 2 *Les cartes*

Dans cet exercice on souhaite réaliser un module pour représenter des cartes à jouer. Ce module sera nommé `card` et définira une unique classe nommée `Card`.

Nous considérons ici des cartes à jouer caractérisées par

— une valeur qui est un élément de l'ensemble $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, C, K\}$ (J pour jack, C pour knight (le cavalier des jeux de tarot), Q pour queen, K pour king et A pour ace) ; vous pourrez supposer que cet ensemble de valeurs est défini dans la classe `Card` par la constante
`VALUES = ("Ace", "2", ..., "9", "10", "Jack", "Knight", "Queen", "King")` ;

— une couleur qui est un élément de l'ensemble $\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit, \}$; vous pourrez supposer que cet ensemble de couleurs est défini dans la classe par la constante
`COLORS = ("spade", "heart", "diamond", "club")`.

è

Question 1 En supposant déjà réalisés dans cette classe

— le constructeur `__init__` qui à partir d'une valeur et d'une couleur valides renvoie une carte de cette valeur et de cette couleur ;

— et les sélecteurs `get_color` et `get_value` renvoyant respectivement la couleur et la valeur d'une carte, couleur et valeur étant l'une des chaînes de caractères des constantes `Card.COLORS` et `Card.VALUES`

```
>>> c1 = Card('Ace', 'heart')
>>> c1.get_color()
'heart'
>>> c1.get_value()
'Ace'
```

réalisez pour cette classe les méthodes suivantes :

— `random` (sans paramètre) de création aléatoire d'une carte ;
— `compare` qui renvoie un entier négatif si la carte passée en paramètre est inférieur à l'objet propriétaire de la méthode, un entier positif si elle est supérieure et un entier nul si elles sont égales (même couleur et même valeur). L'ordre total utilisé sur les cartes s'appuie d'abord sur l'ordre des valeurs tel que défini par la liste `VALUES` des valeurs (ainsi Ace < 2 < 3 < ... < King) puis en cas d'égalité sur l'ordre défini par la liste `COLORS` des couleurs (ainsi on a spade < heart < diamond < club).

```
>>> c2 = Card.random()
>>> c2.get_value(), c2.get_color()
('Ace', 'spade')
>>> c2.compare(c1) > 0
False
>>> c1.compare(c2) > 0
True
>>> c1.compare(c1) == 0
True
```

Question 2 Réalisez maintenant les méthodes spéciales permettant de

1. utiliser les opérateurs de comparaison usuels du langage Python,

```
>>> c1 == c2
False
>>> c1 != c2
True
>>> c1 < c2
False
>>> c1 > c2
True
```

2. obtenir une représentation externe plus lisible des cartes.

```
>>> Card.random()
Card("4", "club")
>>> print(c1)
Card("Ace", "heart")
```

Question 3

1. En utilisant le module `card`, réalisez une fonction `deck` sans paramètre qui renvoie un jeu complet des $56 = 4 \times 14$ cartes sous forme d'une liste dans laquelle les cartes sont rangées par couleur d'abord et au sein d'une couleur par valeur.
2. Réalisez une fonction `deck` avec un paramètre indiquant si on veut un paquet de 56 cartes, un paquet de 52 cartes (poker), ou un paquet de 32 cartes (belote, manille ou piquet).

Question 4 Proposez des réalisations du constructeur `__init__` et des sélecteurs `get_color` et `get_value` de la classe `Card`.
