Objetos. Dos ejercicios resueltos

Complejos

Define un módulo `Complejo` basado en la representación binómica y con operaciones de creación (en forma binómica o polar), para obtener la parte real, imaginaria, módulo y argumento de un complejo, así como operaciones de escritura (en forma binómica o polar). Operaciones con complejos.

In [1]:

```
from math import sin, cos, sqrt, atan

class Complejo:
    """Clase Complejo, basada en su representación binómica"""

    componente_real = None
    componente_imag = None

    def __init__(self, modo, real_o_radio, imag_o_argumento):
        if modo == "binom":
            self.componente_real = real_o_radio
            self.componente_imag = imag_o_argumento
        if modo == "polar":
           self.componente_real = real_o_radio * cos(imag_o_argumento)
           self.componente_imag = real_o_radio * sin(imag_o_argumento)

    def parte_real(self):
        return self.componente_real

    def parte_imag(self):
        return self.componente_imag

    def modulo(self):
        return sqrt(self.componente_real ** 2 + self.componente_imag ** 2)

    def argumento(self):
        return atan(self.componente_real / self.componente_imag)

    def escribir_complejo_binom(self):
        print("[" + str(self.componente_real)
              + ", " + str(self.componente_imag) + "]")

    def escribir_complejo_polar(self):
        print("<" + str(self.modulo())
              + ", " + str(self.argumento()) + ">")

    def __add__(self, other):
        parte_real_nueva = self.componente_real + other.parte_real()
        parte_imag_nueva = self.componente_imag + other.parte_imag()
        return Complejo("binom", parte_real_nueva, parte_imag_nueva)
```

Además de la suma (+), las demás operaciones también pueden sobrecargarse:

```
https://www.programiz.com/python-programming/operator-overloading
```

Un método muy útil es el que permite ver una clase como un string (digamos que en forma bonómica), mejor que un método para escribirlo.

```
def __str__(self):
    return "[" + str(self.componente_real) \
        + ", " + str(self.componente_imag) + "]"
```

Añádelo y comprueba el funcionamiento de la función print. Lo veremos en el siguiente ejemplo.

Veamos un ejemplo de uso del complejo.

In [2]:

```
from math import pi

z1 = Complejo("binom", 4, 3)
print(z1)
z1.escribir_complejo_binom()
z1.escribir_complejo_polar()
print(z1.parte_real(), z1.parte_imag(), z1.modulo(), z1.argumento())

print()

z2 = Complejo("polar", 5, pi/6)
z2.escribir_complejo_binom()
z3 = z1 + z2
z3.escribir_complejo_binom()
```

```
<__main__.Complejo object at 0x00000288085FE080>
[4, 3]
<5.0, 0.9272952180016122>
4 3 5.0 0.9272952180016122

[4.330127018922194, 2.4999999999999996]
[8.330127018922195, 5.5]
```

Lo corriente es definir un módulo (un archivo) separado en el que se incluye la clase, y luego otros programas de aplicación en las que se usa dicho módulo.

Si el módulo de definición del complejo se define en un archivo, digamos, bajo el nombre `complejoBin.py` , un programa que usa dicha clase debe imnportar las definiciones que necesite:

In [3]:

```
from complejoBin import *
from math import pi

z1 = Complejo("binom", 4, 3)
z1.escribir_complejo_binom()
(z1 + Complejo("polar", 5, pi/6)).escribir_complejo_polar()
```

```
[4, 3]
<9.982034669914624, 0.9872463845991051>
```

En Spyder, para que la clase sea accesible desde el programa, es necesario que dicho programa esté en la

misma carpeta.

Un automóvil

Define el objeto “automóvil”, que únicamente puede estar orientado según las direcciones de los puntos cardinales, N, S, E, O, y cuyas velocidades son números naturales, con el estado inicial siguiente…

- motor_encendido: False
- velocidad: 0
- dirección: “N”

… y los métodos siguientes:

- encender_motor()
- apagar_motor(), que únicamente funciona si la velocidad es nula.
- acelerar(), decelerar(), frenar()
- giro_dcha(), giro_izda()
- Operación de escritura, para operar con print.

In [4]:

```
class Automovil():
    motor_encendido = False
    velocidad = 0
    direccion = "N"

    def encender_motor(self):
        self.motor_encendido = True
        self.velocidad = 0

    def apagar_motor(self):
        self.motor_encendido = False
        self.velocidad = 0

    def acelerar(self, cuanto=1):
        self.velocidad = self.velocidad + cuanto

    def decelerar(self, cuanto=1):
        if self.velocidad >= cuanto:
            self.velocidad = self.velocidad - cuanto

    def frenar(self):
        self.velocidad = 0

    def giro_dcha(self):
        if self.direccion == "N":
            self.direccion = "E"
        elif self.direccion == "E":
            self.direccion = "S"
        elif self.direccion == "S":
            self.direccion = "O"
        else:
            self.direccion = "N"

    def giro_izda(self):
        if self.direccion == "N":
            self.direccion = "O"
        elif self.direccion == "O":
            self.direccion = "S"
        elif self.direccion == "S":
            self.direccion = "E"
        else:
            self.direccion = "N"

    def __str__(self):
        if self.motor_encendido:
            return "[On, " \
                + "v=" + str(self.velocidad) + ", " \
                + self.direccion + "]"
        else:
            return "[Off]"
```

In [5]:

```
a = Automovil()
print(a)
a.encender_motor()
print(a)
a.acelerar()
print(a)
a.acelerar(30)
print(a)
a.decelerar(50) # imposible
print(a)
a.giro_dcha()
print(a)
```

```
[Off]
[On, v=0, N]
[On, v=1, N]
[On, v=31, N]
[On, v=31, N]
[On, v=31, E]
```