PROGRAMACIÓN ORIENTADA A OBJETOS

La programación Orientada a objetos se define como un paradigma de la programación, una manera de programar específica, donde se organiza el código en unidades denominadas clases, de las cuales se crean objetos que se relacionan entre sí para conseguir los objetivos de las aplicaciones.

La programación Orientada a objetos (POO) es una forma especial de programar, más cercana a como expresaríamos las cosas en la vida real que otros tipos de programación.

Al programar orientado a objetos tenemos que aprender a pensar cómo resolver los problemas de una manera distinta a como se realizaba anteriormente, en la programación estructurada. Ahora tendremos que escribir nuestros programas en términos de clases, objetos, propiedades, métodos y otras cosas que veremos rápidamente para aclarar conceptos y dar una pequeña base que permita soltarnos un poco con los conceptos de este tipo de programación.

Motivación

Durante años, los programadores se han dedicado a construir aplicaciones muy parecidas que resolvían una y otra vez los mismos problemas. Para conseguir que los esfuerzos de los programadores puedan ser reutilizados se creó la posibilidad de utilizar módulos. El primer módulo existente fue la función, que somos capaces de escribir una vez e invocar cualquier número de veces.

Sin embargo, la función se centra mucho en aportar una funcionalidad dada, pero no tiene tanto interés con los datos. Es cierto que la función puede recibir datos como parámetros, pero los trata de una forma muy volátil. Simplemente hace su trabajo, procesando los parámetros recibidos y devuelve una respuesta.

En las aplicaciones en realidad los datos están muy ligados a la funcionalidad. Por ejemplo podemos imaginar un punto que se mueve por la pantalla. El punto tiene unas coordenadas y podemos trasladarlo de una posición a otra, sumando o restando valores a sus coordenadas. Antes de la programación orientada a objetos ocurría que cada coordenada del punto tenía que guardarse en una variable diferente (dos variables para ser exacto: x, y) y las funciones de traslación estaban almacenadas por otra parte. Esta situación no facilitaba la organización del código ni tampoco su reutilización.

Con la Programación Orientada a Objetos se buscaba resolver estas situaciones, creando unas mejores condiciones para poder desarrollar aplicaciones cada vez más complejas, sin que el código se volviera un caos. Además, se pretendía dar una de pautas para realizar las cosas de manera que otras personas puedan utilizarlas y adelantar su trabajo, de manera que consigamos que el código se pueda reutilizar.

La POO no es difícil, pero es una manera especial de pensar, a veces subjetiva de quien la programa, de manera que la forma de hacer las cosas puede ser diferente según el programador. Aunque podamos hacer los programas de formas distintas, no todas ellas son correctas, lo difícil no es programar orientado a objetos sino programar bien. Programar bien es importante porque así nos podemos aprovechar de todas las ventajas de la POO.

Cómo se piensa en objetos

Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real. Por ejemplo vamos a pensar en un coche para tratar de modelizarlo en un esquema de POO. Diríamos que el coche es el elemento principal que tiene una serie de características, como podrían ser el color, el modelo o la marca. Además tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o aparcar.

Pues en un esquema POO "el coche" sería lo que se conoce como "Clase". Sus características, como el color o el modelo, serían propiedades y las funcionalidades asociadas, como ponerse en marcha o parar, serían métodos.

La clase es como un libro, que describe como son todos los objetos de un mismo tipo. La clase coche describe cómo son todos sus coches, qué propiedades tienen y qué funcionalidades deben poder realizar. A partir de una clase podemos crear cualquier número de objetos de esa clase. Un coche rojo que es de la marca Ford y modelo Fiesta, otro verde que es de la marca Seat y modelo Ibiza.

Por poner otro ejemplo vamos a ver cómo modelizaríamos en un esquema POO una fracción, es decir, esa estructura matemática que tiene un numerador y un denominador que divide al numerador, por ejemplo 3/2.

La fracción será la clase y tendrá dos propiedades, el numerador y el denominador. Luego podría tener varios métodos como simplificarse, sumarse con otra fracción o número, restarse con otra fracción, etc.

A partir de la definición de una fracción (la clase) podremos construir un número indeterminado de objetos de tipo fracción. Por ejemplo podemos tener el objeto fracción 2/5 o 3/9, 4/3, etc. Todos esos son objetos de la clase fracción de números enteros.

Estas clases se podrán utilizar en los programas, por ejemplo en un programa de matemáticas harás uso de la clase fracción y construirás muchos objetos de tipo fracción para hacer cuentas diversas. En un programa que gestione un taller de coches utilizarás la clase coche y se instanciarán diversos objetos de tipo coche para hacer las operativas.

En los lenguajes puramente orientados a objetos, tendremos únicamente clases y objetos. Las clases permitirán definir un número indeterminado de objetos, que colaboran entre ellos para resolver los problemas. Con muchos objetos de diferentes clases conseguiremos realizar las acciones que se

desean implementar en la funcionalidad de la aplicación. Además, las propias aplicaciones como un todo, también serán definidas por medio de clases. Es decir, el taller de coches será una clase, de la que podremos crear el objeto taller de coches, que utilizará objetos coche, objetos de clase herramienta, objetos de clase mecánico, objetos de clase recambio, etc.

Clases en POO

Como habrás podido entender, las clases son declaraciones de objetos, también se podrían definir como abstracciones de objetos. Esto quiere decir que la definición de un objeto es la clase. Cuando programamos un objeto y definimos sus características y funcionalidades en realidad lo que estamos haciendo es programar una clase. En los ejemplos anteriores en realidad hablábamos de las clases coche o fracción porque sólo estuvimos definiendo, aunque por encima, sus formas.

Propiedades en clases

Las propiedades o atributos son las características de los objetos. Cuando definimos una propiedad normalmente especificamos su nombre y su tipo. Nos podemos hacer a la idea de que las propiedades son algo así como variables donde almacenamos datos relacionados con los objetos.

Métodos en las clases

Son las funcionalidades asociadas a los objetos. Cuando estamos programando las clases las llamamos métodos. Los métodos son como funciones que están asociadas a un objeto.

Objetos en POO

Los objetos son ejemplares de una clase cualquiera. Cuando creamos un ejemplar tenemos que especificar la clase a partir de la cual se creará. Esta acción de crear un objeto a partir de una clase se llama instanciar (que viene de una mala traducción de la palabra instace que en inglés significa ejemplar). Por ejemplo, un objeto de la clase fracción es por ejemplo 3/5. El concepto o definición de fracción sería la clase, pero cuando ya estamos hablando de una fracción en concreto 4/7, 8/1000 o cualquier otra, la llamamos objeto.

Para crear un objeto se tiene que escribir una instrucción especial que puede ser distinta dependiendo el lenguaje de programación que se emplee.

Definición de clase, objeto, atributos y métodos

- Llamamos clase a la representación abstracta de un concepto. Por ejemplo, "perro", "número entero" o "servidor web".
- Las clases se componen de atributos y métodos.
- Un objeto es cada una de las instancias de una clase.
- Los atributos definen las características propias del objeto y modifican su estado. Son datos asociados a las clases y a los objetos creados a partir de ellas.
- Un atributo de clase es compartida por todas las instancias de una clase. Se definen dentro de la clase (después del encabezado de la clase) pero nunca dentro de un método. Este tipo de variables no se utilizan con tanta frecuencia como las variables de instancia.

- Un atributo de objeto se define dentro de un método y pertenece a un objeto determinado de la clase instanciada.
- Los métodos son bloques de código (o funciones) de una clase que se utilizan para definir el comportamiento de los objetos.

Definimos nuestra primera clase:

```
>>> class clase():
...     at_clase=1
...     def metodo(self):
...       self.at_objeto=1
...
>>> type(clase)
<class 'type'>
>>> clase.at_clase
1
>>> objeto=clase()
>>> objeto.at_clase
1
>>> objeto.at_objeto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'clase' object has no attribute 'at_objeto'
>>> objeto.metodo()
>>> objeto.at_objeto
1
```

Atributos de objetos

Para definir atributos de objetos, basta con definir una variable dentro de los métodos, es una buena idea definir todos los atributos de nuestras instancias en el constructor, de modo que se creen con algún valor válido.

Método constructor **init**

Como hemos visto anteriormente los atributos de objetos no se crean hasta que no hemos ejecutado el método. Tenemos un método especial, llamado constructor `__init__`, que nos permite inicializar los atributos de objetos. Este método se llama cada vez que se crea una nueva instancia de la clase.

Definiendo métodos. El parámetro self

El método construcutor, al igual que todos los métodos de cualquier clase, recibe como primer parámetro a la instancia sobre la que está trabajando. Por convención a ese primer parámetro se lo suele llamar `self` (que podríamos traducir como yo mismo), pero puede llamarse de cualquier forma.

Para referirse a los atributos de objetos hay que hacerlo a partir del objeto `self`.

Definición de objetos

Vamos a crear una nueva clase:

```
import math
class punto():
""" Representación de un punto en el plano, los atributos son x e y
que representan los valores de las coordenadas cartesianas."""

def __init__(self,x=0,y=0):
    self.x=x
    self.y=y

def distancia(self, otro):
    """ Devuelve la distancia entre ambos puntos. """
    dx = self.x - otro.x
    dy = self.y - otro.y
    return math.sqrt((dx*dx + dy*dy))
```

Para crear un objeto, utilizamos el nombre de la clase enviando como parámetro los valores que va a recibir el constructor.

```
>>> punto1=punto()
>>> punto2=punto(4,5)
>>> print(punto1.distancia(punto2))
6.4031242374328485
```

Podemos acceder y modificar los atributos de objeto:

```
>>> punto2.x
4
>>> punto2.x = 7
>>> punto2.x
7
```

Atributos de clase (estáticas)

En Python, las variables definidas dentro de una clase, pero no dentro de un método, son llamadas variables estáticas o de clase. Estas variables son compartidas por todos los objetos de la clase.

Para acceder a una variable de clase podemos hacerlo escribiendo el nombre de la clase o a través de self.

```
>>> class Alumno():
...     contador=0
...     def __init__(self,nombre=""):
...         self.nombre=nombre
...         Alumno.contador+=1
...
>>> a1=Alumno("jose")
>>> a1.contador
1
>>> Alumno.contador
1
```

Usamos las variables estáticas (o de clase) para los atributos que son comunes a todos los atributos de la clase. Los atributos de los objetos se definen en el constructor.

Atributos privados y ocultos

Las variables que comienzan por un guión bajo _ son consideradas privadas. Su nombre indica a otros programadores que no son públicas: son un detalle de implementación del que no se puede depender — entre otras cosas porque podrían desaparecer en cualquier momento. **Pero nada nos impide acceder a esas variables.**

```
>>> class Alumno():
...     def __init__(self,nombre=""):
...         self.nombre=nombre
...         self._secreto="asdasd"
...
>>> a1=Alumno("jose")
>>> a1.nombre
'jose'
>>> a1._secreto
'asdasd'
```

Dos guiones bajos al comienzo del nombre __ llevan el ocultamiento un paso más allá, "enmaráñando" (name-mangling) la variable de forma que sea más difícil verla desde fuera.

```
>>> class Alumno():
...     def __init__(self,nombre=""):
...         self.nombre=nombre
...         self.__secreto="asdasd"
...
>>> a1=Alumno("jose")
>>> a1.__secreto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Alumno' object has no attribute '__secreto'
```

Pero en realidad sigue siendo posible acceder a la variable.

```
>>> a1._Alumno__secreto
'asdasd'
```

Se suelen utilizar cuando una subclase define un atributo con el mismo nombre que la clase madre, para que no coincidan los nombres.

Métodos de clase (estáticos)

Los método estáticos (static methods) son aquellos que no necesitan acceso a ningún atributo de ningún objeto en concreto de la clase.

```
>>> class Calculadora():
...   def __init__(self,nombre):
...     self.nombre=nombre
...   def modelo(self):
...     return self.nombre
...   @staticmethod
...   def sumar(x,y):
...     return x+y
...
>>> a=Calculadora("basica")
>>> a.modelo()
'basica'
>>> a.sumar(3,4)
7
>>> Calculadora.sumar(3,4)
7
```

Nada nos impediría mover este método a una función fuera de la clase, ya que no hace uso de ningún atributo de ningún objeto, pero la dejamos dentro porque su lógica (hacer sumas) pertenece conceptualmente a Calculadora.

Lo podemos llamar desde el objeto o desde la clase.

Funciones getattr,setattr,delattr,hasattr

```
>>> a1=Alumno("jose")
>>> getattr(a1,"nombre")
'jose'
>>> getattr(a1,"edad","no tiene")
'no tiene'

>>> setattr(a1,"nombre","pepe")
>>> a1.nombre
'pepe'

>>> hasattr(a1,"nombre")
True

>>> delattr(a1,"nombre")
```

Propiedades: getters, setters, deleter

Para implementar la encapsulación y no permitir el acceso directo a los atributos, podemos poner los atributos ocultos y declarar métodos especificos para acceder y modificar los atributos (mutadores). Estos métodos se denominan getters y setters.

```
class circulo():
    def __init__(self,radio):
        self.set_radio(radio)
    def set_radio(self,radio):
        if radio>=0:
            self._radio = radio
        else:
            raise ValueError("Radio positivo")
            self._radio=0
    def get_radio(self):
        print("Estoy dando el radio")
        return self._radio
```

```
>>> c1=circulo(3)
>>> c1.get_radio()
Estoy dando el radio
3
>>> c1.set_radio(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jose/github/curso_python3/curso/u51/circulo.py", line 8, in
set_radio
    raise ValueError("Radio positivo")
ValueError: Radio positivo
```

En Python, las propiedades nos permiten implementar la funcionalidad exponiendo estos métodos como atributos.

```
class circulo():
    def __init__(self,radio):
        self.radio=radio

    @property
    def radio(self):
        print("Estoy dando el radio")
        return self._radio

    @radio.setter
    def radio(self,radio):
        if radio>=0:
            self._radio = radio
        else:
            raise ValueError("Radio positivo")
            self._radio=0
```

```
>>> c1=circulo(3)
>>> c1.radio
Estoy dando el radio
3
>>> c1.radio=4
>>> c1.radio=-1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jose/github/curso_python3/curso/u52/circulo2.py", line 15,
in radio
    raise ValueError("Radio positivo")
ValueError: Radio positivo
```

Hay un tercera property que podemos crear: el deleter

```
...
@radio.deleter
def radio(self):
    del self._radio


>>> c1=circulo(3)
>>> c1.radio
Estoy dando el radio
3
>>> del c1.radio
>>> c1.radio
Estoy dando el radio
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jose/github/curso_python3/curso/u52/circulo2.py", line 8, in
radio
    return self._radio
AttributeError: 'circulo' object has no attribute '_radio'
>>> c1.radio=3
```

Representación de objetos __str__ y __repr__

La documentación de Python hace referencia a que el método `__str()__` ha de devolver la representación "informal" del objeto, mientras que `__repr()__` la "formal".

- La función `__str()__` debe devolver la cadena de texto que se muestra por pantalla si llamamos a la función `str()`. Esto es lo que hace Python cuando usamos `print`. Suele devolver el nombre de la clase.
- De `__repr()__`, por el otro lado, se espera que nos devuelva una cadena de texto con una representación única del objeto. Idealmente, la cadena devuelta por `__repr()__` debería ser aquella que, pasada a `eval()`, devuelve el mismo objeto.

Continuamos con la clase `circulo`:

```
...
def __str__(self):
    clase = type(self).__name__
    msg = "{0} de radio {1}"
    return msg.format(clase, self.radio)

def __repr__(self):
    clase = type(self).__name__
    msg = "{0}({1})"
    return msg.format(clase, self.radio)
```

Suponemos que estamos utilizando la clase `circulo` sin la instrucción `print` en el getter.

```
>>> c1=circulo(3)
>>> print(c1)
circulo de radio 3
>>> repr(c1)
'circulo(3)'
>>> type(eval(repr(c1)))
<class 'circulo2.circulo'>
```

Comparación de objetos __eq__

Tampoco podemos comparar dos `circulos` sin definir `__eq()__`, ya que sin este método Python comparará posiciones en memoria.

Continuamos con la clase `circulo`:

```
...
def __eq__(self,otro):
    return self.radio==otro.radio

>>> c1=circulo(5)
>>> c2=circulo(3)
>>> c1 == c2
False
```

Si queremos utilizar `<`, `<=`, `>` y `>=` tendremos que rescribir los métodos: `__lt()__`, `__le()__`, `__gt()__` y `__ge()__`

Operar con objetos __add__ y __sub__

Si queremos operar con los operadores + y -:

```
def __add__(self,otro):
    self.radio+=otro.radio

def __sub__(self,otro):
    if self.radio-otro.radio>=0:
        self.radio-=otro.radio
    else:
        raise ValueError("No se pueden restar")

>>> c1=circulo(5)
>>> c2=circulo(3)
>>> c1 + c2
>>> c1.radio
8


>>> c1=circulo(5)
>>> c2=circulo(3)
>>> c1 - c2
>>> c1.radio
2
>>> c1 - c2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jose/github/curso_python3/curso/u52/circulo2.py", line
42, in __sub__
    raise ValueError("No se pueden restar")
ValueError: No se pueden restar
```

Más métodos especiales

Existen muchos más métodos especiales que podemos sobreescribir en nuestras clases para añadir funcionalidad a las mismas. Puedes ver la documentación oficial para aprender más sobre ellas.

Polimorfismo

El polimorfismo es la técnica que nos posibilita que al invocar un determinado método de un objeto, podrán obtenerse distintos resultados según la clase del objeto. Esto se debe a que distintos objetos pueden tener un método con un mismo nombre, pero que realice distintas operaciones.

Lo llevamos usando desde principio del curso, por ejemplo podemos recorrer con una estructura `for` distintas clases de objeto, debido a que el método especial `__iter__` está definida en cada una de las clases. Otro ejemplo sería que con la función `print` podemos imprimir distintas clases de objeto, en este caso, el método especial `__str__` está definido en todas las clases.

Además esto es posible a que python es dinámico, es decir en tiempo de ejecución es cuando se determina el tipo de un objeto. Veamos un ejemplo:

```
class gato():
    def hablar(self):
        print("MIAU")

class perro():
    def hablar(self):
        print("GUAU")

def escucharMascota(animal):
    animal.hablar()

if __name__ == '__main__':
    g = gato()
    p = perro()
    escucharMascota(g)
    escucharMascota(p)
```

Herencia

La herencia es un mecanismo de la programación orientada a objetos que sirve para crear clases nuevas a partir de clases preexistentes. Se toman (heredan) atributos y métodos de las clases viejas y se los modifica para modelar una nueva situación.

La clase desde la que se hereda se llama clase base y la que se construye a partir de ella es una clase derivada.

Si nuestra clase base es la clase `punto` estudiadas en unidades anteriores, puedo crear una nueva clase de la siguiente manera:

```
class punto3d(punto):
    def __init__(self,x=0,y=0,z=0):
        super().__init__(x,y)
        self.z=z
    @property
    def z(self):
        return self._z

    @z.setter
    def z(self,z):
        self._z=z

    def __str__(self):
        return super().__str__()+":"+str(self.z)

    def distancia(self,otro):
        dx = self.x - otro.x
        dy = self.y - otro.y
        dz = self.z - otro.z
        return (dx*dx + dy*dy + dz*dz)**0.5
```

Creemos dos objetos de cada clase y veamos los atributos y métodos que tienen definido:

```
>>> p=punto(1,2)
>>> dir(p)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_x', '_y',
'distancia', 'x', 'y']
>>> p3d=punto3d(1,2,3)
>>> dir(p3d)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_x', '_y',
'_z', 'distancia', 'x', 'y', 'z']
```

La función super()

La función `super()` me proporciona una referencia a la clase base. Y podemos observar también que hemos reescrito el método `distancia` y `__str__`.

```
>>> p.distancia(punto(5,6))
5.656854249492381
>>> p3d.distancia(punto3d(2,3,4))
1.7320508075688772
>>> print(p)
1:2
>>> print(p3d)
1:2:3
```

Herencia múltiple

La herencia múltiple se refiere a la posibilidad de crear una clase a partir de múltiples clases superiores. Es importante nombrar adecuadamente los atributos y los métodos en cada clase para no crear conflictos.

```
class Telefono:
    "Clase teléfono"
    def __init__(self,numero):
        self.numero=numero
    def telefonear(self):
        print('llamando')
    def colgar(self):
        print('colgando')
    def __str__(self):
        return self.numero

class Camara:
    "Clase camara fotográfica"
    def __init__(self,mpx):
        self.mpx=mpx
    def fotografiar(self):
        print('fotografiando')
    def __str__(self):
        return self.mpx
```

```
class Reproductor:
    "Clase Reproductor Mp3"
    def __init__(self,capcidad):
        self.capacidad=capcidad
    def reproducirmp3(self):
        print('reproduciendo mp3')
    def reproducirvideo(self):
        print('reproduciendo video')
    def __str__(self):
        return self.capacidad

class Movil(Telefono, Camara, Reproductor):
    def __init__(self,numero,mpx,capacidad):
        Telefono.__init__(self,numero)
        Camara.__init__(self,mpx)
        Reproductor.__init__(self,capacidad)
    def __str__(self):
        return "Número: {0}, Cámara: {1},Capacidad:
{2}".format(Telefono.__str__(self),Camara.__str__(self),Reproductor.__s
tr__(self))
```

Veamos los atributos y métodos de un objeto `Movil`:

```
>>> mimovil=Movil("645234567","5Mpx","1G")
>>> dir(mimovil)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'capacidad', 'colgar', 'fotografiar', 'mpx', 'numero', 'reproducirmp3',
'reproducirvideo', 'telefonear']
>>> print(mimovil)
Número: 645234567, Cámara: 5Mpx,Capacidad: 1G
```

Funciones issubclass() y isinstance()

La función `issubclass(SubClase, ClaseSup)` se utiliza para comprobar si una clase (SubClase) es hija de otra superior (ClaseSup), devolviendo True o False según sea el caso.

```
>>> issubclass(Movil,Telefono)
True
```

La función booleana `isinstance(Objeto, Clase)` se utiliza para comprobar si un objeto pertenece a una clase o clase superior.

```
>>> isinstance(mimovil,Movil)
True
```

Delegación

Llamamos delegación a la situación en la que una clase contiene (como atributos) una o más instancias de otra clase, a las que delegará parte de sus funcionalidades.

A partir de la clase `punto`, podemos crear la clase `circulo` de esta forma:

```
class circulo():

    def __init__(self,centro,radio):
        self.centro=centro
        self.radio=radio

    def __str__(self):
        return "Centro:{0}-Radio:
{1}".format(self.centro.__str__(),self.radio)
```

Y creamos un objeto `circulo`:

```
>>> c1=circulo(punto(2,3),5)
>>> print(c1)
Centro:2:3-Radio:5
```

EJERCICIOS PROGRAMACIÓN ORIENTADA A OBJETOS

1. Queremos programar un generador de **password** que cumpla las siguientes condiciones:
   a. Una password debe tener los atributos **longitud** y **contraseña.** Por defecto la longitud será 8.
   b. Podremos generar una contraseña por defecto de longitud 8 o generar una contraseña aleatoria con una longitud dada.
   c. Deberemos poder comprobar si es fuerte dicha contraseña. Para que una contraseña sea fuerte debe tener más de 2 mayúsculas, más de 1 minúscula y más de 5 números.
   d. Deberemos poder generar una password, es decir, generar la contraseña del objeto según la longitud que tenga.
   e. Debemos poder obtener la contraseña y la longitud de una password.
   f. Debemos poder modificar la longitud de una passsword.

   Para probarlo:
   - Crea un array de password con el tamaño solicitado por teclado.
   - Genera las contraseñas y muestra si son o no fuertes.

2. Queremos implementar un programa que me permita calcular el perímetro y el área de distintas figuras geométricas: triángulo, cuadrilátero, circunferencia, cubo, cono y cilindro.
   Para ello debes tener en cuenta que un triángulo y un cuadrilátero son polígonos.
   También debes saber que:
   - Un cubo está compuesto de 6 caras cuadradas congruentes.
   - El cono tendrá siempre como base un círculo.
   - El cilindro estará formado por dos círculos y un rectángulo.

3. Queremos modelar un juego sobre Zombies. En este juego existe un mundo sobre el cual estas horribles criaturas se desplazan, mordiendo a todo humano que se le cruce.
   El juego, a grandes rasgos presenta las siguientes reglas:

- Todos los personajes (ya sean humanos o zombies), se mueven sobre una línea recta
- En cada turno, el jugador elige un personaje humano y lo mueve hacia la izquierda o derecha, para que los zombies, que son movidos más lentamente por la computadora, no lo atrapen. También pueden realizar otras acciones, como gritar, defenderse, pedir ayuda, etc.
- Cada vez que un personaje realiza alguna acción, este pierde una cierta cantidad de energía, y sólo puede realizarla si tiene energía suficiente.
- Los personajes puede ser perseguidos por otros personajes
- Los zombies pueden morder a una persona y convertirla en zombie.

No nos interesa desarrollar la dinámica de turnos del juego, ni las reglas por las que los zombies se mueven, nos concentramos en los siguientes requerimientos:

- Poder mover una persona una cierta distancia, en una cierta dirección.
- Poder caminar, trotar y correr: esto es que la persona se desplace una cierta distancia prefijada. Estas acciones tienen un determinado consumo energético, y no debe moverse si no cuenta con la suficiente energía:
  - caminando: se desplaza 10 metros. Consume 5 unidades de energía
  - trotando: se desplaza 20 metros. Consume 15 unidades.
  - corriendo: se desplaza 40 metros. Consume 60 unidades.
- Permitir que un personaje sea perseguido por otro, y poder gritar: cuando un personaje grita, todos los perseguidores lo escuchan. Gritar también tiene un consumo energético, de 10 puntos, como se describe en el punto anterior. Por otro lado, si a un humano le gritan, este no hace nada.
- Modelar los zombies y la mordida: son humanos que han sido mordidos por otro zombie. Cuando esto ocurre, su energía no cambia, y su comportamiento solo se ve modificado en los siguientes aspectos:
  - Se mueven a la mitad de velocidad (por ejemplo, si le digo a un zombie que corra, este solo se desplazará 40/2 = 20 metros)
  - Si son mordidos por otro zombie, siguen siendo zombies
  - Saben morder (los humanos no)
  - Si les gritan, pierden hasta 100 puntos de energía.

4.

Se quiere hacer el diseño de un robot modular. El robot estará compuesto por varios módulos entre los que se encuentran: rotación, extensión, helicoidal, cámara. Los módulos podrán ser dinámicos (capaces de moverse: rotación, extensión, helicoidal) o estáticos (no se pueden mover: cámara).

Los módulos tendrán un identificador (1-255) y unas dimensiones (largo, ancho y alto, entre 1 y 200mm). Los módulos estarán compuestos de un sistema de control y un sistema de comunicación. Los módulos dinámicos tendrán:

- motores (1 ó 2).
- un parámetro que es el tipo de movimiento que pueden realizar.
- una función que es moverse (con parámetro el tipo de movimiento).

Los módulos estáticos podrán tener sensores (de 0 a 5).

El sistema de control utiliza el sistema de mensajes para comunicarse. Los módulos pueden enviar y recibir mensajes de/hacia el usuario y otros módulos, con un parámetro que es un array de datos a mandar o recibir. También utiliza los motores para moverse y los sensores para captar información del medio.

Se pide que utilizando herencia siempre que se pueda, se realice un diseño UML de las clases necesarias para representar todas las entidades del sistema, indicando atributos y métodos, así como las relaciones existentes entre las clases.

Se pide la implementación de clases y simulación correspondiente.