Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2024-2

Publicación: 23 de Agosto de 2024
Actividad evaluada

Actividad 2
Programación Orientada a Objetos Avanzado

Entrega

- **Lugar:** Repositorio personal de GitHub — Carpeta: `Actividades/AC2`
- **Fecha máxima de entrega:** 29 de Agosto 23:59
- **Ejecución de actividad**: La Actividad será ejecutada **únicamente** desde la terminal del computador. **Los *paths* relativos utilizados en la Actividad deben ser coherentes con esta instrucción, y no pueden modificarse.**

Introducción

Tras implementar tu propio programa para consultar animes, te estás volviendo muy popular así que desde el departamento de tránsito, liderado por el famoso Cristiano Ruznaldo, te han pedido una ayuda con su trabajo. En particular, empezar a programar algunas clases que se utilizarán en una simulación que analiza los gastos energéticos asociada a distintos tipos de vehículos.

Flujo del programa

Esta actividad consta en completar diferentes clases programadas bajo el paradigma de Programación Orientada a Objetos (POO) para cumplir con diferentes objetivos. En particular, primero se te presentará la modelación del problema y luego se presentará el detalle de cada clase con sus atributos y métodos. En base a esas 2 elementos, deberás completar el archivo `classes.py` con las clases esperadas. Este archivo será corregido exclusivamente mediante el uso de *tests*.

Finalmente, debes asegurarte de entregar, como mínimo, el archivo que tenga el *tag* de `Entregar` en la siguiente sección. Los demás archivos no es necesario subir, pero tampoco se penalizará si se suben al repositorio personal.

Archivos

En el directorio de la actividad encontrarás los siguientes archivos con código:

- Entregar Modificar `clases.py`: Aquí encontrarás la definición básica de las clases que debes implementar y completar. Al finalizar la actividad, debes asegurar que este archivo esté subido en el lugar de entrega correspondiente.
- No modificar `tests_publicos`: Carpeta que contiene diferentes `.py` para ir probando si lo desarrollado hasta el momento cumple con lo esperado. **En la última hoja del enunciado se encuentra un anexo de cómo ejecutar los *tests* por parte o todos**.

Modelo de datos

Para reflejar y hacer la simulación, te han entregado una serie de especificaciones técnicas que deberás implementar en tu modelo de datos.

Habrán distintos tipos de vehículos de distintas marcas, y todos estos tendrán un rendimiento de kilómetros por cantidad de energía consumida. Además de esto, todos poseen un identificador único que debe ser calculado en tiempo de ejecución de forma automática, y una energía disponible que por razones físicas no puede ser menor a cero.

La energía de los autos de divide en dos tipos de consumos, estos pueden ser a bencina (L) o eléctricos (W). Por un lado, los autos a bencina poseen una bencina favorita, y tienen una forma particular de recorrer kilómetros. Por otro lado, los autos eléctricos poseen una vida útil de la batería, y también poseen su propia forma de recorrer.

Actualmente solo se evaluarán 3 tipos de autos distintos: una camioneta genérica, un Telsa y un FaitHibrido. Las camionetas además de ser autos a bencina, poseen una capacidad de maleta. Por otro lado, el Telsa es un auto eléctrico, pero que al momento de recorrer lo hace de forma inteligente, pues tiene conducción automática. Finalmente, el FaitHibrido es un auto que tiene un comportamiento especial, pues es un auto a bencina y eléctrico a la vez, por lo que posee las características de ambos autos, y tiene una forma de recorrer muy especial.

Llevando modelo al código

- Modificar `class Vehiculo`:

  Clase que representa un vehículo. Posee la variable de clase `identificador`, y además incluye los siguientes métodos:

  - Modificar `def __init__(self, rendimiento, marca, energia, *args, **kwargs) -> None`:

    Este es el inicializador de la clase, y debe asignar los siguientes atributos:

`self.rendimiento`	Este atributo guarda la información del argumento `rendimiento` recibido en el inicializador. Es un `int` que representa los kilómetros que puede andar un vehículo por unidad de energía utilizada (ya sea bencina, electricidad o híbrido).
`self.marca`	Este atributo guarda la información del argumento `marca` recibido en el inicializador. Es un `str` que representa la marca del vehículo.
`self._energia`	Este atributo privado guarda la información del argumento `energia` recibido en el inicializador. Es un `float` de máximo 1 decimal que representa la cantidad de Litros/Watts que contiene el vehículo. Además, es posible que el argumento `energía` no sea recibido en el inicializador. En dicho caso, energía debe tomar un valor por defecto de `111.5`. Se recomienda investigar sobre valores por defecto en Python para lograr este objetivo. Finalmente, este atributo debe estar restringido para que su valor no puede bajar de 0.
`self.identificador`	Un `int` que sirve para identificar al vehículo. Para setearlo, debe fijarlo como el valor actual de la variable de clase `Vehiculo.identificador`, y luego se le debe sumar 1 a la variable indicada anteriormente. De esta forma, se logra que cada vehículos tengan un identificador diferente.

  - Modificar `def autonomia(self) -> float`:

    Property encargada de informar la autonomía. Retorna la cantidad de kilómetros que puede recorrer el vehículo. Este valor se obtienen multiplicando la energía disponible y el rendimiento del vehículo. Este resultado **siempre** debe ser del tipo `float`.

  - Modificar `def energia(self) -> float`:

    Property encargada de manejar la energía disponible. El *getter* debe retornar el valor del atributo privado de energía, y el *setter* debe encargarse de que el atributo privado no pueda tener un valor menor a 0, es decir, en caso de recibir uno menor a 0, este debe ser acotado a 0. Además, esta *property* siempre debe asegurarse que la energía sea trabajada como un `float` y redondeada a 1 decimal. Para esto último debes investigar y utilizar el comando `round` de Python.

- Modificar `class AutoBencina`:

  Clase utilizada para representar vehículos que utilizan bencina como combustible. Hereda de `Vehiculo`, y posee los siguientes métodos:

  - Modificar `def __init__(self, bencina_favorita, *args, **kwargs) -> None`:

    Este es el inicializador de la clase. Además de llamar al método de la clase padre, debe asignar los siguientes atributo:

`self.bencina_favorita`	Este atributo guarda la información del argumento `bencina_favorita` recibido en el inicializador. Es un `str` que representa el octanaje de la bencina de preferencia.

- **Modificar** `def recorrer(self, kilometros) -> str`:

  Método encargado de simular el desplazamiento del vehículo. Si el vehículo puede andar el total de los kilómetros pedidos, entonces recorre dicha cantidad de kilómetros. En caso de no poder, recorre solo lo informado por la autonomía. Además, debe calcular el gasto de este movimiento que corresponde a `kilometros\_recorridos/self.rendimiento` para luego disminuir la energía en el gasto calculado.

  Finalmente, el método retorna el siguiente texto:

```
"Anduve {N}Km y eso consume {Z}L de bencina"
```

  Donde N es la cantidad de kilómetros que logró recorrer y Z corresponde al gasto de este movimiento. Este último valor (Z), debe estar siempre redondeado 1 decimal utilizando el comando `round`.

- **Modificar** `class AutoElectrico`:

  Clase utilizada para representar autos eléctricos. Hereda de `Vehiculo`, y posee los siguientes métodos:

  - **Modificar** `def __init__(self, vida_util_bateria, *args, **kwargs) -> None`:

    Inicializador de la clase. Además de llamar al método de la clase padre, debe setear el siguiente atributo.

`self.vida_util_bateria`	Este atributo guarda la información del argumento `vida_util_bateria`" recibido en el inicializador. Es un `int` que representa los años que le quedan de vida util a la batería.

  - **Modificar** `def recorrer(self, kilometros) -> str`:

    Método encargado de simular el desplazamiento del vehículo. Si el vehículo puede andar el total de los kilómetros pedidos, entonces recorre dicha cantidad de kilómetros. En caso de no poder, recorre solo lo informado por la autonomía. Además, debe calcular el gasto de este movimiento que corresponde a `kilometros\_recorridos/self.rendimiento` para luego disminuir la energía en el gasto calculado.

    Finalmente, el método retorna el siguiente texto:

```
"Anduve {N}Km y eso consume {Z}W de energia electrica"
```

    Donde N es la cantidad de kilómetros que logró recorrer y Z corresponde al gasto de este movimiento. Este último valor (Z), debe estar siempre redondeado 1 decimal utilizando el comando `round`.

- Modificar `class Camioneta`:

  Clase que representa una camioneta. Debe heredar de la clase `AutoBencinca` de la siguiente forma:

  - Modificar `def __init__(self, capacidad_maleta, *args, **kwargs) -> None`:

    Inicializador de la clase. Además de llamar al método de la clase padre, debe setear el siguiente atributo.

`self.capacidad_maleta`	Este atributo guarda la información del argumento `capacidad_maleta` recibido en el inicializador. Es un `int` que representa la capacidad de la maleta.

- Modificar `class Telsa`:

  Clase que representa un auto eléctrico. Debe heredar de la clase `AutoElectrico`, con un cambio en el siguiente método:

  - Modificar `def recorrer(self, kilometros) -> str`:

    Debe llamar al método de la clase padre para recorrer los kilómetros indicados. Luego, modificar el texto retornado por la clase padre para agregar `"de forma muy inteligente"` al final, y considerando un espacio entre ambos textos. Finalmente, se retorna este nuevo texto modificado.

- Modificar `class FaitHibrido`:

  Clase que representa un auto híbrido. Debe heredar de las clases `AutoBencina` y `AutoElectrico`:

  - Modificar `def __init__(self, *args, **kwargs) -> None`:

    Debe llamar al método de la clase padre. Además de esto, todos los `FaitHibrido` poseen una vida útil de la batería de 5 años.

  - Modificar `def recorrer(self, kilometros) -> str`:

    Método que simula el recorrido del auto. De los kilómetros recibidos, si la cantidad a recorrer es mayor a 10.0 kilómetros, este auto va a recorrer los primeros 10.0 kilómetros como si fuera un auto a bencina. Luego, los kilómetros restantes los hará como un auto eléctrico. En otro caso, que la cantidad a recorrer sea menor o igual a 10.0 kilómetros, solo los recorrerá como un auto eléctrico.

    Finalmente, el retorno de este método varía según si logra o no recorrer más de 10 kilómetros.

    - Si recorre 10 o menos kilómetros, solamente retornará el texto de la clase padre correspondiente al auto eléctrico.
    - En otro caso, debe retornar la concatenación de ambos textos del método recorrer de las clases padre, considerando un espacio entre ellos, y respetando el orden de primer andar como un auto de bencina y luego como un auto eléctrico.

Notas

- **No puedes hacer *import* de otras librerías externas a las entregadas en el archivo.**
- Recuerda que la ubicación de tu entrega es en **tu repositorio de Git**. En la rama (*branch*) por defecto del repositorio: `main`.
- Se recomienda completar la actividad en el orden del enunciado.

- Recuerda que esta evaluación presenta corrección **automatizada**. Si entregas un código que se cae al momento de correr los *tests*, será evaluado con 0 puntos.
- Si aparece un error inesperado, ¡léelo! Intenta interpretarlo.

Objetivos de la actividad

- Implementar clases siguiendo especificaciones técnicas.
- Manejar concepto de herencia y multiherencia correctamente.
- Hacer correcto uso de *properties*.
- Utilizar métodos de clases padres correctamente.
- Resolver bien el problema del diamante.
- Probar código mediante la ejecución de *test*.

Ejecución de *tests*

En esta actividad se provee de varios archivos `.py` los cuáles contiene diferentes *tests* que ayudan a validar el desarrollo de la actividad. Para ejecutar estos *tests*, **primero debes posicionar tu terminal/consola en la carpeta de la actividad (`Actividades/AC2`)**. Luego, desde esta misma, debes escribir el siguiente comando para ejecutar todos los *tests* de la actividad:

- `python3 -m unittest discover tests_publicos -v -b`

En cambio, si deseas ejecutar un subconjunto de *tests*, puedes hacerlo si escribes lo siguiente en la terminal/consola:

- `python3 -m unittest -v -b tests_publicos.test_vehiculo`
  Para ejecutar solo el subconjunto de *tests* relacionado a la clase `Vehiculo`.
- `python3 -m unittest -v -b tests_publicos.test_autobencina`
  Para ejecutar solo el subconjunto de *tests* relacionado a la clase `AutoBencina`.
- `python3 -m unittest -v -b tests_publicos.test_autoelectrico`
  Para ejecutar solo el subconjunto de *tests* relacionado a la clase `AutoElectrico`.
- `python3 -m unittest -v -b tests_publicos.test_camioneta`
  Para ejecutar solo el subconjunto de *tests* relacionado a la clase `Camioneta`.
- `python3 -m unittest -v -b tests_publicos.test_telsa`
  Para ejecutar solo el subconjunto de *tests* relacionado a la clase `Teslsa`.
- `python3 -m unittest -v -b tests_publicos.test_faithibrido`
  Para ejecutar solo el subconjunto de *tests* relacionado a la clase `FaitHibrido`.

Importante: recuerda que si `python3` no funciona, probar con el comando específico de tu computador. Este puede ser `py`, `python`, `py3` o `python3.11`.