Curso OOP: Laboratorio 1

Profesor: Eric Tanter
Ayudante: Fabián Mosso

Recomendaciones para este taller:

- Defina las clases en un mismo archivo.
- Realice la siguiente importación:

```
from __future__ import annotations
```

En algunos puntos de este taller vamos a utilizar un asistente de código. Puede utilizar cualquiera. Se recomienda el uso de cursor, que es un IDE que tiene integrado el asistente de código.

Parte 1

Para esta parte no utilice el asistente de código, al menos que se indique lo contrario, el objetivo de estos ejercicios es 'soltar la mano'

Para el siguiente ejercicio vamos a modelar distintos tipos de cuentas bancarias. Un objeto que corresponde a un cuenta bancaria soporta tres métodos:

- `deposit(amount: int) -> bool:` Deposita dinero en la cuenta, retorna si fue exitosa o no la operación.
- `withdraw(amount: int) -> bool:` Retirar dinero en la cuenta, retorna si fue exitosa o no la operación.
- `transfer(amount: int, target) -> bool:` Envía dinero a la cuenta objetivo. Por simplicidad enviar dinero es depositar dinero en la cuenta objetivo. Retorna si fue exitosa o no la operación.

1) Cree un archivo accounts.py, en él defina la clase `CurrentAccount`, la clase `CurrentAccount` representa a una cuenta corriente. Las cuentas corrientes permiten retirar dinero y hacer transferencias aunque el saldo sea negativo. Los métodos de `CurrentAccount` se definen de la siguiente forma:
   - `deposit(amount: int) -> bool:` ingresa dinero a la cuenta bancaria, siempre retorna `True`.
   - `withdraw(amount: int) -> bool:` retira dinero de la cuenta, siempre retorna `True`,
   - `transfer(amount: int, target) -> bool:` Deposita la cantidad en la cuenta objetivo. Cuando la cuenta objetivo recibe el deposito de forma exitosa, este método retorna `True`, caso contrario retorna `False`.

2) Defina la clase `CheckingAccount`, la clase `CheckingAccount` representa a una cuenta vista. Las cuentas vistas son similares a la cuenta corriente, pero su saldo nunca puede ser negativo. Los métodos de `CheckingAccount` se definen de la siguiente forma :
   - `deposit(amount: int) -> bool:` ingresa dinero a la cuenta bancaria, siempre retorna `True`.
   - `withdraw(amount: int) -> bool:` Valida que al retirar dinero, la cuenta quede con dinero, en ese caso retorna `True`, caso contrario retorna `False`.
   - `transfer(amount: int, target) -> bool:` Primero valida que la cuenta quede con saldo a favor si se realiza la transferencia. Deposita la cantidad en la cuenta objetivo. Cuando la cuenta objetivo recibe el deposito de forma exitosa, este método retorna True, caso contrario retorna `False`.

3) En un archivo `main` instancie un objeto de la clase `CurrentAccount` y otro de la clase `CheckingAccount`, luego ejecute en distintos orden depósitos, retiros, y transferencias entre las cuentas.

4) Defina la clase `StudentAccount`, la clase `StudentAccount` representa a una cuenta estudiante. La cuenta estudiante es similar a la cuenta vista, su saldo nunca puede ser negativo, ademas esta cuenta tiene montos máximos para la cantidad de dinero de la cuenta, de depósitos, y transferencias.
   Esta clase recibe en el constructor los argumentos: `max_money, max_deposit,y max_transfer.` Donde los valores por defecto son:
   - `max_money: 100000`
   - `max_deposit: 100000`
   - `max_transfer: 10000`

   Los métodos de `StudentAccount` se definen de la siguiente forma :
   - `deposit(amount: int) -> bool:` Valida que el dinero a depositar no supere ni la cantidad máxima de deposito, ni que al después de depositar supere la cantidad máxima del saldo de la cuenta. En caso que supere las validaciones ingresa dinero a la cuenta bancaria y retorna `True`. Caso contrario retorna `False`
   - `withdraw(amount: int) -> bool:` Valida que al retirar dinero, la cuenta quede con dinero, en ese caso retorna `True`, caso contrario retorna `False`.
   - `transfer(amount: int, target) -> bool:` Primero valida que la cuente quede con saldo a favor si se realiza la transferencia. Luego que la cantidad a transferir no supere la cantidad máxima del deposito. Deposita la cantidad en la cuenta objetivo. Cuando la cuenta objetivo recibe el deposito de forma exitosa, este método retorna `True`, caso contrario retorna `False`.

6) En el archivo `main`, instancie un par de cuentas de estudiante, variando los topes de estas, y realice distintas operaciones con ellas.

Para las siguientes actividades, utilice algún asistente IA.

7) Cree un archivo llamado `mainIA.py,` indique a su asistente que tomando las definiciones del archivo accounts, genere 5 pruebas unitarias. Pruebe utilizar el siguiente prompt:

“usando las definiciones del archivo 'accounts.py' genera 5 pruebas unitarias dentro del archivo mainIA.py”

Ejecute las pruebas unitarias dentro del archivo `mainIA.py`, y compare el código generado por el asistente con el que usted escribió en el archivo `main.py`.

8) También puede usar el asistente para labores ‘tediosas’. Un ejemplo de estas labores es generar documentación del código escrito.

Pruebe escribiendo el siguiente prompt:

“Generar documentación a cada clase y método del archivo 'accounts.py'”

El asistente va a generar alrededor 200 lineas de comentarios.
¿Cómo usted puede validar con los comentarios generados correspondan con la especificación?
¿De quién es la responsabilidad que los comentarios o el código generado sea correcto?

Parte 2

El objetivo de la siguiente parte es extender un código que fue generado por un asistente. Para esto utilice el siguiente enunciado como prompt del asistente.

Crea la carpeta car, y dentro de esta:

En los siguientes problemas vamos a modelar el funcionamiento de un automóvil. El automóvil puede realizar diferentes acciones: prender, apagar, acelerar y frenar. Y dos estados: prendido y apagado.

Los objetos que representan estados tienen los siguientes métodos:

- turn_on() -> None: Prender el automóvil
- turn_off() -> None: Apagar el automóvil
- accelerate() -> None: Acelerar el automóvil
- brake() -> None: Frenar el automóvil

Ademas, tienen como atributo objeto car, que es recibido como argumento en el constructor

Defina la clase TurnedOn que representa el estado 'Prendido'. La clase TurnedOn define los métodos de la siguiente forma:

- turn_on() -> None: Imprime en pantalla "Ya estoy prendido".
- turn_off() -> None: Imprime en pantalla "Apagándome" y cambia el estado del auto a apagado.
- accelerate() -> None: Imprime en pantalla "Acelerando".
- brake() -> None: Imprime en pantalla "Frenando".

Defina la clase TurnedOff que representa el estado 'Apagado'. La clase TurnedOff define los métodos de la siguiente forma:

- turn_on() -> None: Imprime en pantalla "Prendiendo" y cambia el estado del auto a prendido.
- turn_off() -> None: Imprime en pantalla "Ya estoy apagado".
- accelerate() -> None: Imprime en pantalla "No puedo acelerar, estoy apagado".
- brake() -> None: Imprime en pantalla "No puedo frenar, estoy apagado".

Defina la clase Car, que tiene:

- El campo state.
- Un constructor, que no recibe parámetros, e instancia el campo state. Un auto es inicializado como apagado.
- Los métodos turn_on(), turn_off(), accelerate() y brake().

Los métodos de la clase Car funcionan de la siguiente forma: cuando es llamado un método de Car, Car llama al método del mismo nombre de state. Por ejemplo cuando se llama a turn_on() de Car, Car llama al turn_on de state, consigo mismo como argumento.

En un archivo main instancie un objeto de la clase Car, luego ejecute en distintos orden los métodos de la clase.

A continuación no utilice el asistente e intente realizar solos las siguientes extensiones:

1) Agregue el estado de `OverHeat` (sobre-calentado). Solo se puede llegar a este estado cuando un auto que esta encendido, es acelerado 3 veces seguidas.

La clase OverHeat define los métodos de la siguiente forma:

- `turnOn() -> None`: Imprime en pantalla "Estoy prendido, pero el motor deberá ser apagado pronto".
- `turnOff() -> None`: Imprime en pantalla "Apagándome, por fin descansare" y cambia el estado del auto a apagado.
- `accelerate() -> None`: Imprime en pantalla "No puedo acelerar mas".
- `brake() -> None`: Imprime en pantalla "Frenando, pero el motor sigue caliente".

2) En un método `main` instancie un objeto de la clase `Car`, luego ejecute en distintos orden los métodos de la clase `Car` de forma que el auto pase por los 3 estados.

3) Agregue a la clase `Car` el campo used_gas, que es un int que representa la cantidad de gasolina consumida por el vehículo. La gasolina consumida es definida de la siguiente forma:

- Al encender un auto apagado consume 30.
- Al acelerar un auto encendido consume 10.
- Al sobre-calentar el auto consume 50 adicionales
- Al acelerar un auto sobre-calentado consume 20
- En auto apagado no consume gasolina
- No consume gasolina apagar o frenar un auto que este encendido o sobre-calentado

En caso de estar atascado, le puede preguntar al asistente que le explique parte del código.