Tipos de métodos e interacciones entre objetos

Abstracción y Encapsulamiento

{desafío} latam_

Codificar un programa con clases, atributos y métodos utilizando colaboración y composición para resolver un problema de baja complejidad acorde al lenguaje Python.

- Unidad 1:
  Introducción a la programación orientada a objetos con Python
- Unidad 2:
  Tipos de métodos e interacciones entre objetos
- Unidad 3:
  Herencia y polimorfismo
- Unidad 4:
  Manejo de errores y archivos

Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Describe los conceptos de colaboración y composición distinguiendo sus diferencias de acuerdo al paradigma de orientación a objetos.*

{desafío} latam_

¿Has escuchado sobre los cuatro pilares de la programación orientada a objetos?

{desafío} latam_

Principios básicos

En programación orientada a objetos, se busca abstraer la lógica de los métodos de una clase, dejando expuesto solamente los nombres de los métodos y sus parámetros y encapsular el estado de los objetos, ocultando sus atributos.

```
from abc import ABC, abstractmethod
class PelotaAbstracta(ABC):
    # No se tiene la lógica del método en la clase abstracta
    @abstractmethod
    def rebotar(self, altura: int):
        pass
class PelotasDeJuguete(PelotaAbstracta):
    def rebotar(self, altura: float):
        self.rebotes = []
        while altura > 0:
            self.rebotes.append(altura)
            self.rebotes.append(0)
            altura //= 2
pelota_andy = PelotasDeJuguete()
pelota_andy.rebotar(10)
```

Principios básicos

Pelota: Abstracción

¿Qué hace?

¡Puedo rebotar!

¿Cómo rebota?

...

Pelota de Andy: Encapsulamiento

¿Qué información entrega la pelota?

Soy amarilla, de plástico, y mido 16 cm.

Tengo una obsolescencia programada de 1 año, pero no entregaré esa información

¿Puedo hacerla transparente?

¡No! Debo tener un color

/* Abstracción */

Abstracción

En programación orientada a objetos, el objetivo de la abstracción es **disponibilizar solamente la información esencial que permite definir un objeto**.

En Python, pueden existir tanto clases abstractas como métodos abstractos, la cual posee al menos 1 método abstracto, y este es aquel que solo posee firma (sin implementación).

Para poder definir una clase abstracta, es necesario importar la clase ABC del módulo abc. Puedes revisar más al respecto en los siguientes enlaces: abc y Introducing Abstract Base Classes.

Abstracción

Clase ABC

Se debe entregar como argumento de la clase abstracta.

Los métodos abstractos (definidos dentro de la clase abstracta), deben hacer uso del decorador **`@abstractmethod`**, el cual también se debe importar desde el módulo **abc**.

Como el método abstracto no tiene implementación, en su bloque solo debe tener la palabra reservada **`pass`**.

Al intentar instanciar la clase, se produce un error.

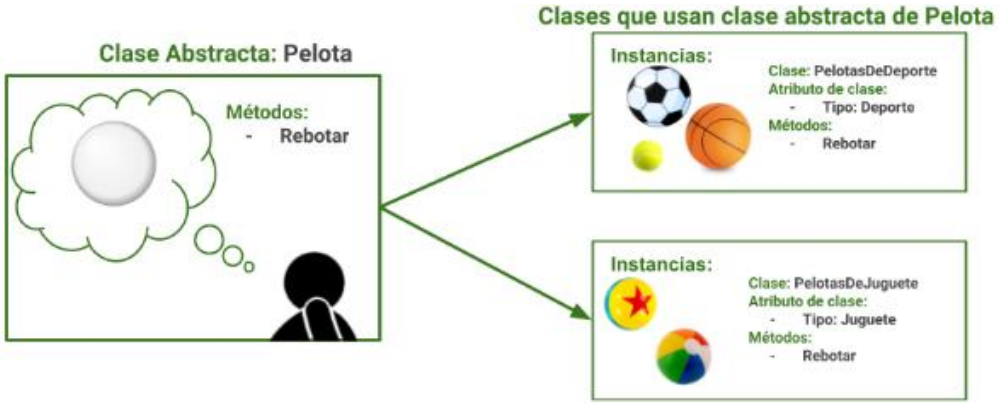
```
from abc import ABC, abstractmethod

class Pelota(ABC):
    @abstractmethod
    def rebotar(self, altura: int):
        pass
```

Abstracción

Ejemplo

Se crea la subclase PelotaDeJuguete, que recibe como argumento la clase base Pelota, de la cual deberá implementar su método abstracto rebotar.




```python
class PelotaDeJuguete(Pelota):
    def __init__(self):
        self.rebotes = []

    def rebotar(self, altura: float):
        self.rebotes = []
        while altura > 0:
            self.rebotes.append(altura)
            self.rebotes.append(0)
            altura //= 2
        return self.rebotes
```

La clase abstracta o clase base funciona como un “molde” que contiene aspectos comunes de un conjunto de subclases. Por otro lado, las subclases implementan los métodos definidos en la clase base, y además pueden definir otros métodos que son propios de la subclase específica.

/* Encapsulamiento */

Encapsulamiento

Delimita un conjunto de datos, y los métodos que afectan esos datos, dentro de una misma unidad, restringiendo así el acceso y uso de los datos y métodos “encapsulados”. Un ejemplo clásico corresponde a una Clase, ya que dentro de su estructura encierra todos los métodos y atributos que permiten definir un objeto de dicha clase.

Normalmente, en programación orientada a objetos, al encapsular el comportamiento y el estado de un objeto dentro de la clase que lo define, estos no son posibles de acceder ni modificar desde otras clases, sino que necesariamente debe hacerse por medio del objeto.

En Python sí es posible acceder y modificar los atributos de una clase, esto porque a diferencia de lo que ocurre en otros lenguajes, todos los métodos y atributos de la clase son públicos.

Encapsulamiento

- Es posible indicar que un atributo o un método es **privado**, limitando en cierta forma su acceso desde fuera de la clase.
- Se debe anteponer al nombre del atributo o del método dos underscore “__”.
- Es posible aplicar getters y setters para los atributos de la clase, conservando el mismo nombre del atributo para las propiedades.

Si quieres profundizar más al respecto, puedes revisar los siguientes enlaces:
Variables privadas y Descriptive: Naming Styles.

Encapsulamiento

Ejemplo

La subclase **PelotaDeJuguete** tiene el atributo de instancia **`__color`** definido de forma privada. Es decir, no es posible acceder a él ni modificarlo directamente desde fuera de la clase; ya que al solicitar el valor del **`__color`** de la instancia creada, se genera un error de tipo **AttributeError**, indicando que el atributo solicitado no existe. (Se omite la implementación del método rebotar para dar enfoque al atributo **`__color`**).

```
class PelotaDeJuguete(Pelota):
   def __init__(self, color):
       self.__color = color

   def rebotar(self, altura: float):
       pass

p = PelotaDeJuguete("amarilla")
# Genera salida:
# AttributeError: 'PelotaDeJuguete' object has no attribute '__color'
print(p.__color)
```

Encapsulamiento

Ejemplo

Para poder acceder o modificar el color, se puede hacer uso de una propiedad color (sin los underscore al principio), por lo que para acceder al color de la instancia se hará uso del getter de la propiedad, y para modificarlo se hará uso del setter de la propiedad.

```
class PelotaDeJuguete(Pelota):
    def __init__(self, color: str):
        self.__color = color

    @property
    def color(self):
        # Como se está dentro de la clase, sí puede
        # acceder al atributo aunque sea privado
        return self.__color

    @color.setter
    def color(self, nuevo_color: str):
        # Puede modificar el atributo privado, porque está dentro de la clase
        self.__color = nuevo_color
    def rebotar(self, altura: float):
        pass

p = PelotaDeJuguete("amarilla")
# Salida: amarilla
print(p.color)
```

Encapsulamiento

Ejemplo

A continuación se muestra cómo a partir de una instancia de la clase anterior, se puede acceder al atributo privado **`__color`** mediante esta sintaxis.

```
p = PelotaDeJuguete("amarilla")
# Salida: amarilla
print(p._PelotaDeJuguete__color)
```

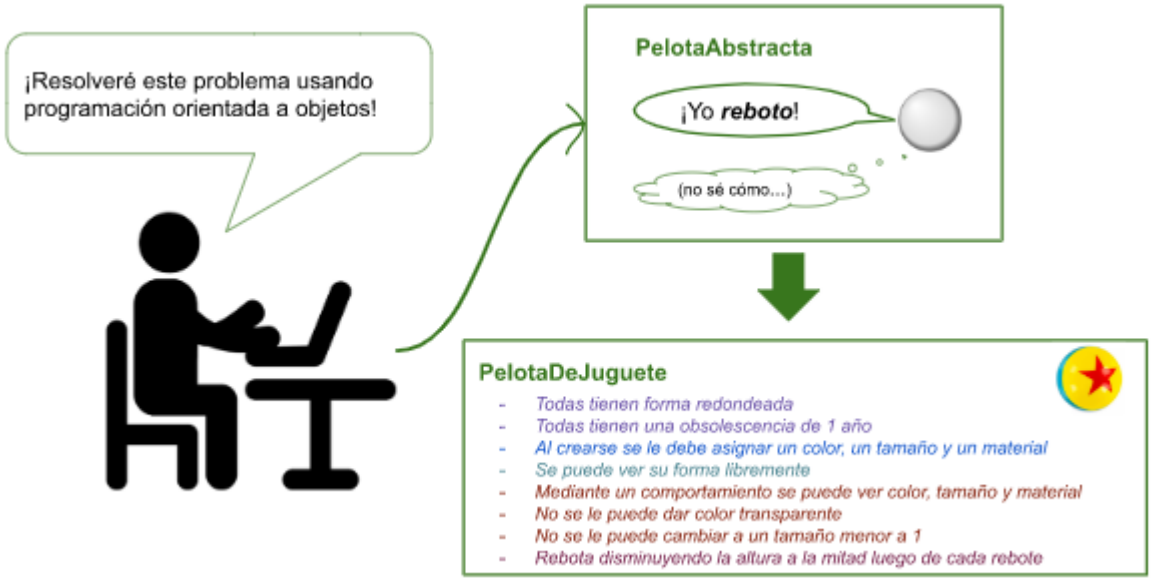
Abstraer y encapsular un algoritmo dentro de una clase

Mediante la abstracción, es posible establecer la pauta a seguir para todas las subclases que comparten comportamientos en común, dados por la clase base, por otro lado, el encapsulamiento permite delimitar dentro de una clase todo el comportamiento y estado de cada objeto instanciado a partir de dicha clase.

Por último, haciendo uso de distintos tipos de métodos (constructor, getters, setters, métodos de instancia, métodos estáticos) y atributos (de clase, de instancia), en conjunto con el nivel de acceso (privado o público) y las propiedades, es posible establecer suficientes reglas en cuanto al comportamiento y posibles estados de un objeto.

Abstraer y encapsular un algoritmo dentro de una clase

Ejemplo



Ejercicio guiado

"Solicitudes en Créditos"

{desafío} latam_

Solicitudes en Créditos

Desde una institución bancaria, te han solicitado realizar un prototipo de programa que permita a un usuario ingresar una solicitud de crédito. En esta primera etapa, solo se quiere crear las solicitudes de los distintos tipos de crédito, sin considerar las características del usuario solicitante.

El prototipo debe consistir en una aplicación de consola en Python, la cual al ser ejecutada debe solicitar al usuario ingresar la siguiente información mediante línea de comandos:

- El tipo de crédito: las opciones son crédito de consumo, crédito comercial y crédito hipotecario
- Monto del crédito
- Correo de contacto del usuario solicitante

Solicitudes en Créditos

Con los datos ingresados, se debe crear una instancia de solicitud de crédito adecuada. Para ello, se debe considerar la siguiente información entregada por la entidad bancaria:

- Los créditos de consumo y los créditos comerciales solo pueden tener montos entre $1.000.000 y $5.000.000 inclusive. Los créditos hipotecarios solo pueden tener montos entre $20.000.000 y $100.000.000.
- Si se solicita un crédito fuera del rango permitido, se debe forzar el monto solicitado al valor posible más cercano. Ejemplo: Si se solicita un crédito de consumo $400.000, se forzará esta cantidad a ser $1.000.000.
- Los correos de contacto para las solicitudes de créditos de consumo e hipotecarios solo pueden tener terminaciones “.cl” o “.com”. Para el caso de las solicitudes de créditos comerciales, también pueden terminar en “.org”, pero no pueden contener los textos “gmail”, “outlook” ni “hotmail”. En todos los casos, los correos deben tener 1 símbolo arroba (@).
- Si se ingresa un correo no válido para el tipo de crédito solicitado, se debe forzar a ser un texto vacío.

Solicitudes en Créditos

Solución

Paso 1

En un archivo **solicitud_credito.py**, importar ABC y **abstractmethod** desde abc, y definir la clase **SolicitudCredito**.

```
# archivo credito.py
from abc import ABC, abstractmethod
class SolicitudCredito(ABC):
```

Paso 2

Dentro de la clase **SolicitudCredito**, definir el método abstracto **validar_monto**, el cual recibe el parámetro “monto” además de la instancia.

```
    @abstractmethod
    def validar_monto(self, monto: str):
        pass
```

Solicitudes en Créditos

Solución

Paso 3

También dentro de la clase **SolicitudCredito**, definir el método abstracto **validar_correo**, el cual recibe el parámetro “correo” además de la instancia.

```
@abstractmethod
    def validar_correo(self, correo: str):
        pass
```

Paso 4

En el mismo archivo **solicitud_credito.py**, a continuación de la clase **SolicitudCredito**, definir la clase **SolicitudCreditoDeConsumo**, como subclase de la clase **SolicitudCredito**, y dentro de ella, definir el atributo de clase privado **`__terminaciones`**.

```
class SolicitudCreditoDeConsumo(SolicitudCredito):
    __terminaciones = (".cl", ".com")
```

{desafío} latam_

Solicitudes en Créditos

Solución

Paso 5

Definir e implementar el método **validar_monto**, siguiendo las reglas entregadas en la descripción.

```
def validar_monto(self, monto: int):
      if monto < 1000000:
          monto = 1000000
      elif monto > 5000000:
          monto = 5000000
      return monto
```

Solicitudes en Créditos

Solución

Paso 6

Definir e implementar el método **validar_correo**, siguiendo las reglas entregadas en la descripción.

```
def validar_correo(self, correo: str):
        return (correo correo.count("@") == 1
        and correo.endswith(SolicitudCreditoDeConsumo.__terminaciones)
        else "")
```

Solicitudes en Créditos

Solución

Paso 7

Al comienzo de la clase **SolicitudCreditoDeConsumo**, definir el constructor de la clase, el cual recibe por parámetro el monto y el correo, los que se deben asignar a los atributos de la instancia, haciendo uso de los métodos de validación. Los atributos de la instancia se definen como privados.

```
def __init__(self, monto: int, correo: str):
        self.__monto = self.validar_monto(monto)
        self.__correo = self.validar_correo(correo)
```

Solicitudes en Créditos

Solución

Paso 8

Agregar las propiedades con getter y setter para **monto** y **correo**. En los setter, se debe llamar a los métodos de validación para asignar el valor al atributo.

```
@property
    def monto(self):
        return self.__monto

    @monto.setter
    def monto(self, monto: int):
        self.__monto = self.validar_monto(monto)

    @property
    def correo(self):
        return self.__correo

    @correo.setter
    def correo(self, correo: str):
        self.__correo = self.validar_correo(correo)
```

{desafío} latam_

Solicitudes en Créditos

Solución

Paso 9

A continuación de la clase **SolicitudCreditoDeConsumo**, definir la clase **SolicitudCreditoComercial**, siguiendo los mismos pasos que para la clase **SolicitudCreditoDeConsumo**, pero aplicando en las implementaciones de los métodos la lógica correspondiente a este caso, según la descripción.

{desafío} latam_

```
class SolicitudCreditoComercial(SolicitudCredito):
    __prohibidos = ("gmail", "outlook", "hotmail")
    __terminaciones = (".cl", ".com", ".org")
    def __init__(self, monto: int, correo: str):
        self.__monto = self.validar_monto(monto)
        self.__correo = self.validar_correo(correo)
    @property
    def monto(self):
        return self.__monto
    @monto.setter
    def monto(self, monto: int):
        self.__monto = self.validar_monto(monto)
    @property
    def correo(self):
        return self.__correo
    @correo.setter
    def correo(self, correo: str):
        self.__correo = self.validar_correo(correo)
    def validar_monto(self, monto: int´):
        if monto < 1000000:
            monto = 1000000
        elif monto > 5000000:
            monto = 5000000
        return monto
    def validar_correo(self, correo: str):
        return correo (if not any(p in correo.lower() for p in
        SolicitudCreditoComercial.__prohibidos)
        and correo.count("@") == 1
        and correo.endswith(SolicitudCreditoComercial.__terminaciones)
        else "")
```

Solicitudes en Créditos

Solución

Paso 10

Finalmente, a continuación de la clase **SolicitudCreditoComercial**, definir la clase **SolicitudCreditoHipotecario**, según corresponde para su caso.

```
class SolicitudCreditoHipotecario(SolicitudCredito):
    __terminaciones = (".cl", ".com")
    def __init__(self, monto, correo: str):
        self.__monto = self.validar_monto(monto)
        self.__correo = self.validar_correo(correo)
    @property
    def monto(self):
        return self.__monto
    @monto.setter
    def monto(self, monto: int):
        self.__monto = self.validar_monto(monto)
    @property
    def correo(self):
        return self.__correo
    @correo.setter
    def correo(self, correo: str):
        self.__correo = self.validar_correo(correo)
    def validar_monto(self, monto: int):
        if monto < 20000000:
            monto = 20000000
        elif monto > 100000000:
            monto = 100000000
        return monto
    def validar_correo(self, correo: str):
        return (correo if correo.count("@") == 1
        and correo.endswith(SolicitudCreditoHipotecario.__terminaciones)
        else "")
```

Solicitudes en Créditos

Solución

Paso 11

En un archivo **programa.py**, importar las clases **SolicitudCreditoDeConsumo**, **SolicitudCreditoComercial** y **SolicitudCreditoHipotecario** desde **solicitud_credito.py**.

```
# archivo programa.py
from credito import SolicitudCreditoDeConsumo,
SolicitudCreditoComercial, SolicitudCreditoHipotecario
```

Solicitudes en Créditos

Solución

Paso 12

A continuación, solicitar al usuario los datos requeridos, y almacenar en variables.

```
print("¡Gracias por solicitar un crédito con nuestro Banco!")

tipo = int(input("\nIngrese el Tipo de Crédito a solicitar:\n"
"1. Crédito de consumo\n"
"2. Crédito Comercial\n"
"3. Crédito Hipotecario\n"))

monto = int(input("\nIngrese el monto que desea solicitar:\n"))
correo = input("\nIngrese su correo de contacto:\n")
```

Solicitudes en Créditos

Solución

Paso 13

A continuación del código anterior, crear la instancia correspondiente, según el tipo de crédito ingresado.

```
credito = None
if tipo == 1:
   credito = SolicitudCreditoDeConsumo(monto, correo)
elif tipo == 2:
   credito = SolicitudCreditoComercial(monto, correo)
elif tipo == 3:
   credito = SolicitudCreditoHipotecario(monto, correo)
```

Solicitudes en Créditos

Solución

Ejemplo de ejecución

```
¡Gracias por solicitar un crédito con nuestro Banco!

Ingrese el Tipo de Crédito a solicitar:
1. Crédito de consumo
2. Crédito Comercial
3. Crédito Hipotecario
3

Ingrese el monto que desea solicitar:
50000000

Ingrese su correo de contacto:
miau@gmail.com
```

¿Como se puede definir una clase abstracta en Python?

¿Qué permite el encapsulamiento?

Próxima sesión...

- *Describe los conceptos de colaboración y composición distinguiendo sus diferencias de acuerdo al paradigma de orientación a objetos.*

(continuación)

{desafío} latam_

{desafío}
latam_

Academia de
talentos digitales