Tecnicatura Universitaria en Programación
Programación II

UNIDAD N° 3 y 4: Programación Orientada a Objetos - Apunte 2

Índice

Clases y Objetos - Conceptos básicos

Python es un lenguaje de programación orientado a objetos. Casi todo en Python es un objeto, con sus propiedades y métodos.

Comencemos escribiendo una clase simple, Dog que represente un perro, no un perro en particular, sino cualquier perro. ¿Qué sabemos sobre la mayoría de los perros domésticos? Pues todos tienen un nombre y una edad. También sabemos que la mayoría de los perros se sientan y se dan vuelta. Esos dos datos (nombre y edad) y esos dos comportamientos (sentarse y darse vuelta) se incluirán en nuestra clase Dog porque son comunes a la mayoría de los perros. A partir de esta clase se puede instanciar un objeto que represente a un perro.

dog.py > ...

```python
class Dog:
    def __init__(self, name, age):
        """Initialize attributes."""
        self.name = name
        self.age = age

    def sit(self):
        """Simulate a dog sitting in response to a command."""
        print(f"{self.name} is now sitting.")

```

Por convención, los nombres en mayúscula se refieren a clases en Python.

El método __init__()

Una función que es parte de una clase es un método . Todo lo que aprendiste sobre funciones se aplica también a los métodos; La única diferencia práctica por ahora es la forma en que llamaremos a los métodos. El método __init__() es un método especial que Python ejecuta automáticamente cada vez que creamos una nueva instancia basada en la clase Dog. Es un método de clase.

Este método tiene dos guiones bajos iniciales y dos guiones bajos finales, una convención que ayuda a evitar que los nombres de los métodos predeterminados de Python entren en conflicto con los nombres de sus métodos. Asegúrese de utilizar dos guiones bajos a cada lado de __init__(). Si usa solo uno en cada lado, el método no se llamará automáticamente cuando use su clase, lo que puede resultar en errores difíciles de identificar.

Definimos el método __init__() en este ejemplo para que tenga tres parámetros: self, name y age. El parámetro self es obligatorio en la definición del método y debe aparecer primero, antes que los demás parámetros que son opcionales.

Debe incluirse el parámetro self en la definición porque cuando Python llame a este método más adelante (para crear una instancia de Dog), la llamada al método pasará automáticamente el argumento self. Cada llamada a un método asociado con una instancia pasa automáticamente self, que es una referencia a la instancia misma; le da a la instancia individual acceso a los miembros de la clase. Cuando creamos una instancia de Dog, Python llamará al método __init__() de la clase Dog. Pasaremos a Dog() un nombre y una edad como argumentos;

self se pasa automáticamente, por lo que no es necesario que lo pasemos. Siempre que queramos crear una instancia de la clase Dog, proporcionaremos valores solo para los dos últimos parámetros name y age.

app.py > ...

```
1   from dog import *
2
3   my_dog = Dog('Willie', 6)
4
5   print(f"My dog's name is {my_dog.name}.")
6   print(f"My dog is {my_dog.age} years old.")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\valon\Desktop\PythonEjercicios-master\oop> python app.py
My dog's name is Willie.
My dog is 6 years old.
```

La clase Dog que estamos usando aquí es la que acabamos de escribir en el ejemplo anterior. Aquí, le decimos a Python que cree un perro cuyo nombre es 'Willie' y edad sea 6 . Cuando Python lee esta línea, llama al método __init__() con los argumentos 'Willie' y 6. El método __init__() crea una instancia que representa a este perro en particular y establece los atributos name y age utilizando los valores que proporcionamos. Luego, Python devuelve una instancia que representa a este perro. Asignamos esa instancia a la variable my_dog.

La convención de nomenclatura resulta útil en este caso; Por lo general, podemos suponer que un nombre en mayúscula como Dog se refiere a una clase, y un nombre en minúscula como my_dog se refiere a una única instancia creada a partir de una clase.

Nota: *Al método __init__ se lo conoce cómo métodos mágicos en Python. Los métodos mágicos son métodos especiales que tienen doble guión bajo al principio y al final de sus nombres. Son también conocidos en inglés como dunders (de doble underscores). Son utilizadas para crear funcionalidades que no pueden ser representadas en un método regular. (Los veremos más adelante)*

Accediendo a los atributos

Para acceder a los atributos de una instancia, se utiliza la notación de puntos. Accedemos al valor del atributo name de my_dog escribiendo:

```
5   print(f"My dog's name is {my_dog.name}.")
```

La notación de puntos se usa con frecuencia en Python. Esta sintaxis demuestra cómo Python encuentra el valor de un atributo. Aquí, Python mira la instancia my_dog y luego encuentra el atributo name asociado con my_dog. Este es el mismo atributo al que se hace referencia self.name en la clase Dog.

Ejecutando comportamiento

Después de crear una instancia de la clase Dog, podemos usar la notación de puntos para llamar a cualquier método definido en Dog. Hagamos que nuestro perro se siente.

Para llamar a un método, proporcione el nombre de la instancia (en este caso, my_dog) y el método que desea llamar, separados por un punto. Cuando Python lee my_dog.sit(), busca el método sit()en la clase Dogy ejecuta ese código.

Nota: *Esta forma de acceder a los atributos y llamar a los métodos sigue el principio de acceso uniforme. Este principio establece que no debería haber diferencia sintáctica entre trabajar con un atributo , propiedad calculada, o método de un objeto.*

Visibilidad de atributos y métodos en Python

Python no distingue entre métodos o atributos públicos y privados, sino que todos los miembros dentro de una clase o módulo son públicos y pueden ser accedidos por fuera de ellos.

No obstante, como convención se prefiere un guión bajo antes del nombre de un miembro para interpretar el mismo como protected y dos guiones bajo para interpretarlo como privado.

Los modificadores de acceso cumplen con el propósito de Encapsulamiento. Su misión es hacer inaccesible los detalles internos de la clase, con el fin de evitar que otras entidades sepan de su existencia y accedan a ellos directamente produciendo comportamiento inesperado. Esto, con el fin de minimizar el coupling entre entidades.

mi_clase.py > MiClase

```python
class MiClase:

    def __init__(self):
        self._atributo_protejido = 1
        self.__atributo_privado = ''

    def _metodo_protejido(self):
        return self._atributo_protejido

    def __metodo_privado(self):
        return self.__atributo_privado
```

Trabajar con atributos y métodos privados y públicos

Puede utilizar clases para representar muchas situaciones del mundo real. Una vez que escriba una clase, pasará la mayor parte de su tiempo trabajando con instancias creadas a partir de esa clase. Una de las primeras tareas que querrás realizar es modificar los atributos asociados con una instancia en particular. Puede modificar los atributos de una instancia directamente o escribir métodos que actualicen los atributos de formas específicas.

Escribamos una nueva clase que represente un automóvil. Nuestra clase almacenará información sobre el tipo de automóvil con el que estamos trabajando y tendrá un método que resuma esta información:

car.py > Car > _get_short_descriptive

```python
class Car:

    def __init__(self, model: str, year: int):
        """Initialize attributes"""
        self.model = model
        self.year = year
        self._cost = 0

    def get_description(self):
        if self._cost > 0:
            return self._get_full_descriptive()
        else:
            return self._get_short_descriptive()

    def _get_full_descriptive(self):
        full = f"El auto es modelo {self.model} del año {self.year} y la valuación es de {self._cost}"
        return full.title()

    def _get_short_descriptive(self):
        short = f"El auto es modelo {self.model} del año {self.year}, se desconoce su valuación"
        return short.title()
```

```
app.py > ...
1   from car import *
2
3   my_car = Car('Siena', 1998)
4   print(my_car.get_description())

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\valon\Desktop\PythonEjercicios-master\oop> python app.py
El Auto Es Modelo Siena Del Año 1998, Se Desconoce Su Valuación
```

La clase auto tiene atributos y métodos públicos y protegidos.

Cuando se crea una instancia, los atributos se pueden definir sin pasarlos como parámetros. Estos atributos se pueden definir en el método __init__(), donde se les asigna un valor predeterminado. En este caso el atributo privado _cost le asignamos 0 por default.

Para un atributo público se puede cambiar el valor directamente a través de una instancia, cómo lo vimos anteriormente accediendo con punto: my_object.atribute.

Para un método público se lo llama de la misma manera a través de una instancia cómo sigue: my_object.method( ).

Si bien Python permite acceder más o menos directamente con punto a los tributos y métodos que definimos cómo privados o protegidos (ya que el que sean privados o protegidos es sólo una convención de nombres) los programadores respetan el hecho de que esto no es conveniente.

Getters y Setters

Digamos que la clase House es parte de su programa y que por el momento, la clase solo tiene definido un atributo de instancia de precio .

```
house.py > House > __init__
1   class House:
2
3       def __init__(self, price):
4           self.price = price
```

Este atributo de instancia es público porque su nombre no tiene un guión bajo inicial. Dado que el atributo es público actualmente, es muy probable que usted y otros desarrolladores de su equipo hayan accedido y modificado el atributo directamente en otras partes del programa usando notación de puntos, como esta:

```
9   # Access value
10  obj.price
11  # Modify value
12  obj.price = 40000
```

Pero digamos que se le pide que haga que este atributo esté protegido (no público) y valide el nuevo valor antes de asignarlo . Específicamente, debe verificar si el valor es un flotante positivo.

En este punto, si decide agregar getters y setters, usted y su equipo probablemente entrarán en pánico. Esto se debe a que cada línea de código que acceda o modifique el valor del atributo deberá modificarse para llamar al getter o al setter, respectivamente. De lo contrario, el código se romperá:

house.py > ...

```python
class House:

    def __init__(self, price):
        self._price = price

    # getter method
    def get_price(self):
        return self._price

    # setter method
    def set_price(self, price):
        self._price = price

# Changed from obj.price
obj.get_price()
# Changed from obj.price = 40000
obj.set_price(40000)
```

Estos getters y setters no son más que métodos públicos de la clase House que permiten acceder o modificar el valor almacenado en el campo _price.

El decorador @property en Python

Una función decoradora es básicamente una función que agrega nueva funcionalidad a una función que se pasa como argumento. ¿Usar una función decorativa es como agregar chispas de chocolate a un helado? Nos permite agregar nueva funcionalidad a una función existente sin modificarla.

Con @property, usted y su equipo no necesitarán modificar ninguna de esas líneas porque podrán agregar getters y setters "entre bastidores" sin afectar la sintaxis que utilizó para acceder o modificar el atributo cuando era público.

house.py > ...

```python
class House:

    def __init__(self, price):
        self._price = price

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, new_price):
        if new_price > 0 and isinstance(new_price, float):
            self._price = new_price
        else:
            print("Please enter a valid price")

    @price.deleter
    def price(self):
        del self._price
```

Específicamente, puede definir tres métodos para una propiedad:

Un getter: para acceder al valor del atributo.
Un setter: para establecer el valor del atributo.
Un deleter: para eliminar el atributo de instancia.

El precio ahora está "protegido". Tenga en cuenta que el atributo de precio ahora se considera "protegido" porque agregamos un guión bajo a su nombre en self._price.

```python
@property
def price(self):
    return self._price
```

@property
Se utiliza para indicar que vamos a definir una propiedad. Observe cómo esto mejora inmediatamente la legibilidad porque podemos ver claramente el propósito de este método.
def price(self):
Observe cómo el getter tiene el mismo nombre que la propiedad que estamos definiendo: price. Este es el nombre que usaremos para acceder y modificar el atributo fuera de la clase. El método sólo toma un parámetro formal, self, que es una referencia a la instancia.
return self._price
Esta línea es exactamente lo que se esperaría de un getter normal. Se devuelve el valor del atributo protegido.

Observe cómo accedemos al atributo precio como si fuera un atributo público. No estamos cambiando la sintaxis en absoluto, pero en realidad estamos utilizando el getter como intermediario para evitar acceder a los datos directamente. Respetando el principio de encapsulamiento.

```
@price.setter
def price(self, new_price):
    if new_price > 0 and isinstance(new_price, float):
        self._price = new_price
    else:
        print("Please enter a valid price")
```

@price.setter
Se utiliza para indicar que este es el método setter del precio de la propiedad. Observe que no estamos usando @property.setter, estamos usando @price.setter. El nombre de la propiedad se incluye antes de .setter.
def price(self, new_price):
Observe cómo el nombre de la propiedad se utiliza como nombre del setter. También tenemos un segundo parámetro new_price, que es el nuevo valor que se asignará al atributo de precio si es válido.
Finalmente, tenemos el cuerpo del setter donde validamos el argumento para comprobar si es un float positivo y luego, si el argumento es válido, actualizamos el valor del atributo. Si el valor no es válido, se imprime un mensaje descriptivo. Puede elegir cómo manejar los valores no válidos según las necesidades de su programa.

Observe cómo no estamos cambiando la sintaxis, pero ahora estamos usando un intermediario (el setter) para validar el argumento antes de asignarlo.

```
@price.deleter
def price(self):
    del self._price
```

@price.deleter
Se utiliza para indicar que este es el método de eliminación de la propiedad de precio . Observe que esta línea es muy similar a @price.setter, pero ahora estamos definiendo el método de eliminación, por lo que escribimos @price.deleter.
def price(self)
Este método sólo tiene definido un parámetro formal, self.
del self._price
El cuerpo, donde eliminamos el atributo de instancia.

¿El atributo de instancia se eliminó correctamente? Cuando intentamos acceder a él nuevamente, se genera un error porque el atributo ya no existe.

No es necesario que defina los tres métodos para cada propiedad. Puede definir propiedades de solo lectura incluyendo sólo un método getter. También puede optar por definir un setter y un getter sin deleter.

Métodos en Python: instancia, clase y estáticos

Hemos visto cómo se pueden crear métodos con def dentro de una clase, pudiendo recibir parámetros como entrada y modificar el estado (como los atributos) de la instancia. Pues bien, haciendo uso de los decoradores, es posible crear diferentes tipos de métodos:

- Lo métodos de instancia “normales” que ya hemos visto
- Métodos de clase usando el decorador @classmethod
- Y métodos estáticos usando el decorador @staticmethod

```python
mi_clase.py > ...
class Clase:
    def metodo(self):
        return 'Método normal'

    @classmethod
    def metododeclase(cls):
        return 'Método de clase'

    @staticmethod
    def metodoestatico():
        return "Método estático"
```

Los métodos de instancia son los métodos normales, que hemos visto anteriormente. Reciben como parámetro de entrada self que hace referencia a la instancia que llama al método. También pueden recibir otros argumentos como entrada. Y como ya sabemos, requieren instanciar un objeto para que puedan ser llamados.

```python
mi_clase = Clase()
mi_clase.metodo()
```

A diferencia de los métodos de instancia, los métodos de clase **@classmethod** reciben como argumento cls, que hace referencia a la clase. Por lo tanto, pueden acceder a la clase pero no a la instancia. No hace falta instanciar un objeto para llamarlos. Pero también se pueden llamar sobre el objeto si se requiere.

```python
Clase.metododeclase()
```

Por último, los métodos estáticos se pueden definir con el decorador **@staticmethod** y no aceptan como parámetro ni la instancia ni la clase. Es por ello por lo que no pueden modificar el estado ni de la clase ni de la instancia. Pero por supuesto pueden aceptar parámetros de entrada si se requiere.

```python
Clase.metodoestatico()
```

Atributos en Python: instancia y clase

Variables de instancia

También llamadas atributos o variables de objeto, estas variables representan la información particular de cada una de las instancias de una clase, como por ejemplo, el nombre de cada ser humano, el color de una silla o el importe total de una factura. Una variable de instancia es exclusiva y particular de cada instancia.

Cuando creamos una clase en Python, lo más común es inicializar los atributos o variables de instancia en el método de inicialización __init__, aunque también podrían crearse variables de instancia en otro métodos de instancia.

Variables de clase

También se llaman atributos de clase o, a veces, variables estáticas (que no tiene nada que ver con constantes, pues su valor se puede modificar). Las variables de clase representan información que es común para todas las instancias de una clase.

De esta manera podemos tener diversas instancias de una clase y todas ellas compartirán los valores de las variables de clase. Si una instancia modifica el valor de una variable de clase, dicho valor queda modificado para todas las instancias.

Así, cuando necesitamos compartir un valor común para todas las instancias de una clase definiremos una variable de clase. La manera más sencilla de definir una variable de clase es hacerlo dentro de la clase pero fuera de las funciones.

dog.py > ...

```python
class Dog:

    animal_type = "Mammal" #class variable

    def __init__(self, name, age, color):
        """Initialize attributes."""
        #instance variable
        self.name = name
        self.age = age
        self._color = color
```

Herencia

No siempre es necesario empezar desde cero al escribir una clase. Si la clase que estás escribiendo es una versión especializada de otra clase que escribiste, puedes usar herencia . Cuando una clase hereda de otra, adquiere los atributos y métodos de la primera clase. La clase original se llama clase padre y la nueva clase es clase hija. La clase hija hereda todos los atributos y métodos de su clase padre, pero también es libre de definir nuevos atributos y métodos propios.

El método __init__() para una sub clase

Cuando escribes una nueva clase y extiendes otra clase existente, a menudo querrás llamar al método __init__() de la clase padre. Esto inicializará todos los atributos que se definieron en el método __init__() y los hará disponibles en la clase hija.

Para ello en el método __init__() de la clase hija escribimos la siguiente línea: super().__init__()

```python
class MiSubClase(MiClase):
    def __init__(self):
        super().__init__()
```

En el ejemplo anterior tenemos una clase MiSubClase que extiende la MiClase (hereda todos los miembros). Luego al crear un objeto de MiSubClase, se ejecuta ambos métodos __init__( ).

La función super() es una función especial que le permite llamar a un método de otra clase superior en la jerarquía. El nombre super proviene de una convención de llamar a la clase principal superclase y a la clase secundaria subclase.

car.py > ...

```python
class Car:

    def __init__(self, model: str, year: int):
        self._model = model
        self._year = year

    @property
    def model(self):
        return self._model
    @model.setter
    def model(self, new_model):
        self._model = new_model

    @property
    def year(self):
        return self._year
    @year.setter
    def year(self, new_year):
        self._year = new_year

class ElectricCar(Car):

    def __init__(self, autonomy_km: float, year: int, model: str):
        self.autonomy_km = autonomy_km
        super().__init__(model, year)

my_car = ElectricCar(250.6,2022,'Tito')
print(my_car.model)
```

```
PROBLEMS 6  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\valon\Desktop\PythonEjercicios-master\oop> python car.py
Tito
```

Polimorfismo

El polimorfismo es uno de los pilares básicos en la programación orientada a objetos, por lo que para entenderlo es importante tener las bases de la POO y la herencia bien asentadas.

En programación orientada a objetos, el polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito

que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.

Definimos las clases Gato y Perro que extienden la clase Animal. Todas implementan el método hablar( )

```
1   class Animal:
2
3       def __init__(self) -> None:
4           pass
5
6       def hablar(self) -> str:
7           pass
8
9   class Perro(Animal):
10
11      def __init__(self) -> None:
12          pass
13      def hablar(self) -> str:
14          return "Guau!"
15
16  class Gato(Animal):
17
18      def __init__(self) -> None:
19          pass
20
21      def hablar(self) -> str:
22          return "Miau!"
```

A continuación creamos un objeto de cada clase y llamamos al método hablar(). Podemos observar que cada animal se comporta de manera distinta al usar hablar().

```
25  pet = Perro()
26  print(pet.hablar())
27  pet = Gato()
28  print(pet.hablar())
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS C:\Users\valon\Desktop\PythonEjercicic
Guau!
Miau!
```

En el caso anterior, la variable pet ha ido “tomando las formas” de Perro y Gato, pero en cualquier caso al llamar al método hablar se ejecuta el comportamiento correspondiente.

El concepto de polimorfismo, desde una perspectiva más general, se puede aplicar tanto a funciones como a tipos de datos. Así nacen los conceptos de funciones polimórficas y tipos polimórficos. Las primeras son aquellas funciones que pueden evaluarse o ser aplicadas a

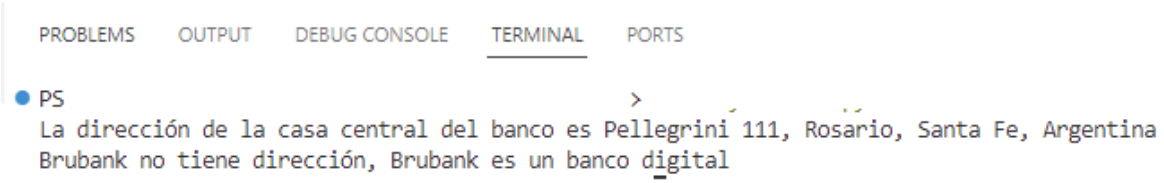
diferentes tipos de datos de forma indistinta; los tipos polimórficos, por su parte, son aquellos tipos de datos que contienen al menos un elemento cuyo tipo no está especificado.

El concepto de polimorfismo podemos aplicarlo para sobrecargar o sobrescribir métodos.

Sobreescritura de métodos

Puede anular o sobreescribir la implementación de cualquier método de la clase padre que no se ajuste a lo que intenta modelar con la clase hija. Para hacer esto, define un método en la clase hija con el mismo nombre que el método que desea anular en la clase padre. Python ignorará la implementación del método de la clase padre y solo prestará atención a la implementación redefinida en la clase hija.

```python
class Banco():
    ...

    def get_address(self) -> str:
        return "La dirección de la casa central del banco es " + self.__address

    ...

class Brubank():
    ...

    def get_address(self) -> str:
        return "Brubank no tiene dirección, Brubank es un banco digital"

    ...
```



Sobrecarga de métodos

En Python la sobrecarga de métodos (overloading) como tal, no existe. Quienes vienen de otros lenguajes como Java o C# se encuentran con algunas confusiones puesto que es algo muy común en estos lenguajes. La sobrecarga de métodos o comúnmente llamada Overloading es un práctica que consiste en tener diferentes métodos con el mismo nombre en una misma clase, y que el intérprete o compilador logre diferenciarlos por los tipos de datos que se envían como argumentos para los parámetros en la llamada al método.

Herencia Múltiple

Herencia múltiple hace referencia a la característica de los lenguajes de programación orientada a objetos en la que una clase puede heredar comportamientos y características de

más de una superclase. Esto contrasta con la herencia simple, donde una clase sólo puede heredar de una superclase. Python soporta la herencia múltiple pero no es una buena práctica en programación y hay pocos casos dónde deba ser implementada.

Clase Abstracta

Un concepto importante en programación orientada a objetos es el de las clases abstractas. Unas clases en las que se pueden definir tanto métodos como propiedades, pero que no pueden ser instancias directamente. Solamente se pueden usar para construir subclases. Permitiendo así tener una única implementación de los métodos compartidos, evitando la duplicación de código.

Otra característica de estas clases es que no es necesario que tengan una implementación de todos los métodos. Pudiendo ser estos abstractos. Los métodos abstractos son aquellos que solamente tienen una declaración, pero no una implementación.

Las clases derivadas de las clases abstractas deben implementar necesariamente todos los métodos abstractos para poder crear una clase que se ajuste a la interfaz definida. En el caso de que no se defina alguno de los métodos no se podrá crear la clase.

Resumiendo, las clases abstractas definen una interfaz común para las subclases. Proporciona atributos y métodos comunes para todas las subclases evitando así la necesidad de duplicar código. Imponiendo además los métodos que deben ser implementados para evitar inconsistencias entre las subclases.

Para poder crear clases abstractas en Python es necesario importar la clase ABC y el decorador abstractmethod del módulo abc (Abstract Base Classes). Un módulo que se encuentra en la librería estándar del lenguaje, por lo que no es necesario instalar. Así para definir una clase privada solamente se tiene que crear una clase heredada de ABC con un método abstracto.

```
1   from abc import ABC, abstractmethod
2
3   class Animal(ABC):
4       @abstractmethod
5       def mover(self) -> None:
6           pass
7
8   class Gato(Animal):
9       def mover(self) -> str:
10          return 'Mover gato'
11
12  my_cat = Gato()
```

Ahora si se intenta crear una instancia de la clase animal, Python no lo permitirá indicando que no es posible. Es importante notar que que si la clase no hereda de ABC y contiene por lo menos un método abstracto, Python permitirá instancias las clases.

```
a = Animal()
    ^^^^^^^^
TypeError: Can't instantiate abstract class Animal with abstract method mover
```

Las subclases tienen que implementar todos los métodos abstractos, en el caso de que falta alguno de ellos Python no permitirá instancias tampoco la clase hija.

Características

- **Una clase abstracta puede tener un constructor**

Las clases abstractas pueden tener constructores, pero no se pueden crear instancias de ellas directamente. Los constructores se utilizan cuando se crea una subclase concreta.
Definirías un constructor en una clase abstracta si deseas realizar alguna inicialización (en los campos de la clase abstracta) antes de que realmente tenga lugar la creación de instancias de una de las subclases.

- **Una clase abstracta puede tener campos de instancia y de clase**
- **Una clase abstracta puede tener métodos concretos y métodos abstractos**

Los métodos abstractos tiene el decorador @abstractmethod, el mismo debe situarse inmediatamente arriba del la definición del método.
Los métodos concretos, con implementación, no llevan el decorador @abstactmethod.

- **Una clase abstracta puede tener propiedades abstractas o concretas**
- **Una clase abstracta no se puede instanciar**
- **Al heredar de una clase abstracta se debe escribir la implementación de los métodos abstractos de la superclase.**
- **Se dice que una subclase sobreescribe un método de su superclase cuando define un método con las mismas características (nombre y parámetros) pero con distinta implementación.**

Nota: Sobreescritura, override o anular es el mismo concepto con diferentes nombres.

Cuidado: Si se define una clase abstracta en Python heredando de ABC pero no se define dentro ningún método abstracto la misma se puede instanciar. Si bien Python permite realizar esta acción no es correcto hacerlo.

raise NotImplementedError()

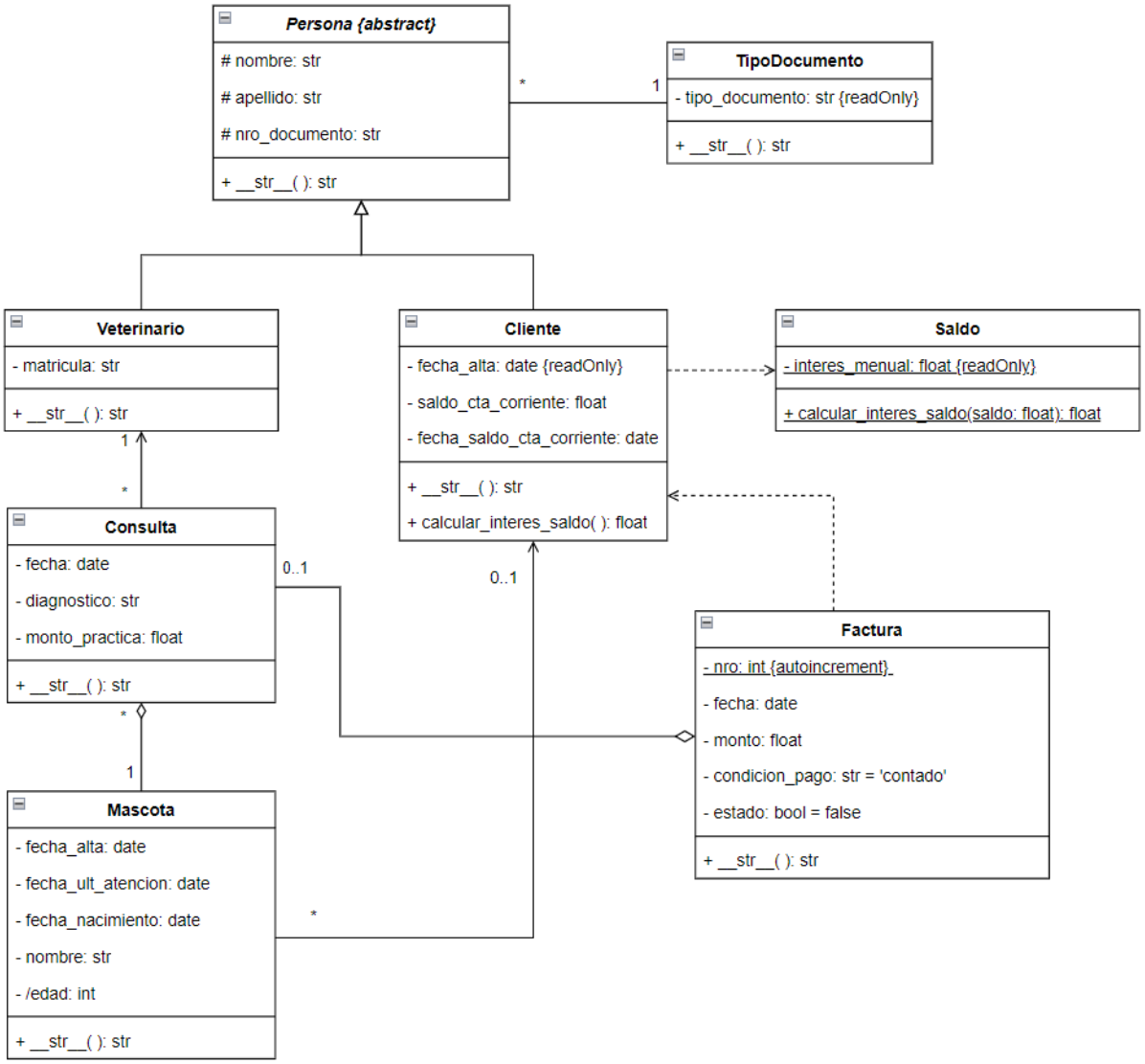
Esta excepción se deriva de RuntimeError.
En las clases base definidas por el usuario, los métodos abstractos deben generar esta excepción cuando requieren que las clases derivadas anulen el método.
También se pueden utilizar en el proceso de desarrollo mientras la clase que se está desarrollando tiene un método que aún es necesario agregar la implementación real.

```
@abstractmethod
def un_metodo(self):
    raise NotImplementedError
```

```python
from abc import ABC, abstractmethod

class Persona(ABC):

    # Constructor de la clase abstracta necesario para inciializar el campo nombre
    def __init__(self, nombre: str) -> None:
        self._nombre = nombre #campo protegido

    @property #Propiedad concreta, puede sobreescribirse en la clase hija
    def nombre(self) -> str:
        return self._nombre.title()

    @nombre.setter #Propiedad abstracta, debe sobreescribirse en la clase hija
    @abstractmethod
    def nombre(self, nuevo_nombre: str):
        self._nombre = nuevo_nombre

    @abstractmethod #Metodo abstracto, debe sobreescribirse en la clase hija
    def __str__(self) -> str:
        raise NotImplementedError

    def mensaje(self) -> str: #Metodo concreto, puede sobreescribirse en la clase hija
        return "Esto es una persona y su nombre es " + self.nombre
```

Mapeo de relaciones desde el UML a Python

Veremos cómo pasar el siguiente diagrama de clases a código Python.



Bibliografía

https://docs.python.org/3/tutorial/classes.html
https://realpython.com/python-pep8/
Matthes, E. (2023). Python crash course: A hands-on, project-based introduction to programming. no starch press.
https://www.freecodecamp.org/news/python-property-decorator

Versiones

Fecha	Versión
30/08/2023	1.0
10/10/2023	1.1 Agregados en clase abstracta y sobreescritura

Autores

María Mercedes Valoni