Python OOP

Link al curso: https://www.linkedin.com/learning/python-object-oriented-programming

Introducción

- Python es un lenguaje orientado a objetos y todo lo que incluye son en efecto objetos, pero se puede utilizar sin definir clases tranquilamente como lenguaje de scripting o definir estructuras, variables de manera independiente, mientras la sintaxis sea correcta, funcionará.
- El enfoque OOP ayuda a organizar y estructurar programas complejos:
  - Agrupar datos y su comportamiento asociado en un mismo lugar
  - Modularización (caja negra)
- Conceptos principales de OOP:

Class	A blueprint for creating objects of a particular type
Methods	Regular functions that are part of a class
Attributes	Variables that hold data that are part of a class
Object	A specific instance of a class
Inheritance	Means by which a class can inherit capabilities from another
Composition	Means of building complex objects out of other objects

- def **__init__**(): Esta función es la primera que se ejecuta (antes que cualquier otra función de la clase) al instanciar una clase, pero **no es un constructor** como en otros lenguajes, ya que se ejecuta **luego de haber creado el objeto**. Se dice que es el "inicializador".
- def __init__(**self**, otro_param): Cada vez que llamamos a un método, python pasa implícitamente como primer argumento al objeto, por eso no hace falta explicitar "self" cada vez que se invoque el método.
- Atributos privados (_name) : Dentro de un método de instancia si escribimos **_**nombre estamos indicando que ese atributo es **privado**.
  Python no tiene una forma de forzar atributos privados, por defecto todos son públicos. Pero cualquier atributo con _ al inicio **no deberá ser llamado desde fuera de la clase.**
- Atributos secretos (doble _ _): Dentro de un método de instancia si escribimos **__**nombre será un atributo **secreto** que queda asociado a la clase y no puede accederse desde otras.
  Para poder acceder habrá que hacer referencia a la misma, no se puede acceder directamente. Ej: print(instancia._Clase__nombre) lo mostrará (aunque no se supone que hagamos esto) pero print(instancia.__nombre) dará error. Servirá cuando se trabaje con subclases, para que no tengan atributos con el mismo nombre que la clase padre, **son atributos que no queremos que tengan overriding**.
- En python todo es una subclase de la clase <object>. Ej: print(isinstance(book, object)) → True
- Python Magic Functions: Son funciones entre doble "_" y están asociadas a la definición de clases.
- "Data Class" está disponible desde Python 3.7 para automatizar: __init__, __repr__ y __eq__
- Python no soporta **overloading** (2 métodos con igual firma) pero sí soporta **herencia múltiple** (Java no lo soporta) y **polimorfismo.**

Ej completo:

```
class Book:
    # Properties defined at the class level are shared by all instances
    BOOK_TYPES = ("HARDCOVER", "PAPERBACK", "EBOOK")
    # double-underscore properties are hidden from other classes
    __booklist = None

    # static methods do not receive class or instance arguments
    # and usually operate on data that is not instance- or class-specific
    @staticmethod # Se usa para el patrón SINGLETON!
    def getbooklist():
        if Book.__booklist == None:
            Book.__booklist = []
        return Book.__booklist

    # class methods receive a class as their argument and can only
    # operate on class-level data
    @classmethod # Para llamarlo habrá que ejecutarlo en la CLASE, no en la INSTANCIA
    def getbooktypes(cls):
        return cls.BOOK_TYPES

    # instance methods receive a specific object instance as an argument
    # and operate on data specific to that object instance
    def setTitle(self, newtitle):
        self.title = newtitle

    def __init__(self, title, booktype):
        self.title = title
        self.author = author
        if (not booktype in Book.BOOK_TYPES):
            raise ValueError(f"{booktype} is not a valid book type")
        else:
            self.booktype = booktype

# access the class attribute
print("Book types: ", Book.getbooktypes())

# Create some book instances
b1 = Book("Title 1", "Autor 1", "HARDCOVER")
b2 = Book("Title 2", "Autor 2", "PAPERBACK")

# Use the static method to access a singleton object
thebooks = Book.getbooklist()
thebooks.append(b1)
thebooks.append(b2)
print(thebooks)
```

Herencia

- La herencia habilita a una clase a heredar métodos y atributos de una o más clases base.
- La definición se hace una sola vez en lugar de tenerla repetida en varios lados (Code organization).
- super().__init__(parametros) → Se hace dentro del def __init__ de la clase hijo para llamar al initializer de la clase base.

Ej:

```
class Publication:
    def __init__(self, title, price):
        self.title = title
        self.price = price

class Periodical(Publication): # Hereda de Publication
    def __init__(self, title, price, publisher, period):
        super().__init__(title, price) # Llama al initializer de la clase base
        self.period = period
        self.publisher = publisher

class Book(Publication): # Hereda de Publication
    def __init__(self, title, author, pages, price):
        super().__init__(title, price) # Llama al initializer de la clase base
        self.author = author
        self.pages = pages

class Magazine(Periodical): # Hereda de Periodical, que a la vez hereda de Publication
    def __init__(self, title, publisher, price, period):
        super().__init__(title, price, publisher, period)

class Newspaper(Periodical): # Hereda de Periodical, que a la vez hereda de Publication
    def __init__(self, title, publisher, price, period):
        super().__init__(title, price, publisher, period)


b1 = Book("Brave New World", "Aldous Huxley", 311, 29.0)
n1 = Newspaper("NY Times", "New York Times Company", 6.0, "Daily")
m1 = Magazine("Scientific American", "Springer Nature", 5.99, "Monthly")

print(b1.author)
print(n1.publisher)
print(b1.price, m1.price, n1.price)
```

Abstract base class

- Se usan cuando no se desea que se instancien objetos de la clase base, sino que cada una de las subclases tenga una implementación propia.
- Hay ciertos métodos de la clase base (abstract method) que las subclases deberán implementar si o si.
- Un **@abstractmethod no implementado** en las clases que heredan, dará ERROR.
- Hay que importarlo de la librería abc: **from abc import ABC, abstractmethod**

Ej: Se tiene una clase padre llamada GraphicShape que tiene un método para calcular el área pero que **no deberá estar implementado a este nivel**, porque se tienen dos clases heredadas: Circle y Square. Ambas clases deben poder calcular el área, pero cada una tendrá su propia implementación.

Ej:

```
from abc import ABC, abstractmethod

class GraphicShape(ABC): # Inheriting from ABC indicates this is an abstract base class
    def __init__(self):
        super().__init__()

    @abstractmethod #declaring a method as abstract requires a subclass to implement it
    def calcArea(self):
        pass

class Circle(GraphicShape):
    def __init__(self, radius):
        self.radius = radius

    def calcArea(self):
        return 3.14 * self.radius ** 2

class Square(GraphicShape):
    def __init__(self, side):
        self.side = side

    def calcArea(self):
        return self.side * self.side

# g = GraphicShape() # Esto dará error porque la clase abstracta NO puede instanciarse

c = Circle(10)
print(c.calcArea())
s = Square(12)
print(s.calcArea())
```

Herencia múltiple

- Una clase podrá heredar de más de una clase base.
- print(Clase.__mro__) muestra el orden de la herencia → *Method Resolution Order*
- Por la complejidad extra de esto, no se usa mucho al desarrollar pero es útil al desarrollar **interfaces** (próxima sección).

```
class A:
    def __init__(self):
        super().__init__()
        self.foo = "foo"
        self.name = "Class A"

class B:
    def __init__(self):
        super().__init__()
        self.bar = "bar"
        self.name = "Class B"


class C(B, A): # Clase C hereda primero de B y luego de A
    def __init__(self):
        super().__init__() #En este caso ambas clases padre setean el name

    def showprops(self):
        print(self.foo)
        print(self.bar)
        print(self.name)


# create the class and call showname()
c = C()
print(C.__mro__) # El método de clase __mro__ permite ver el orden de la herencia
c.showprops() # El resultado es “foo” “bar” “Class B” porque B se heredó primero
```

Interfaces

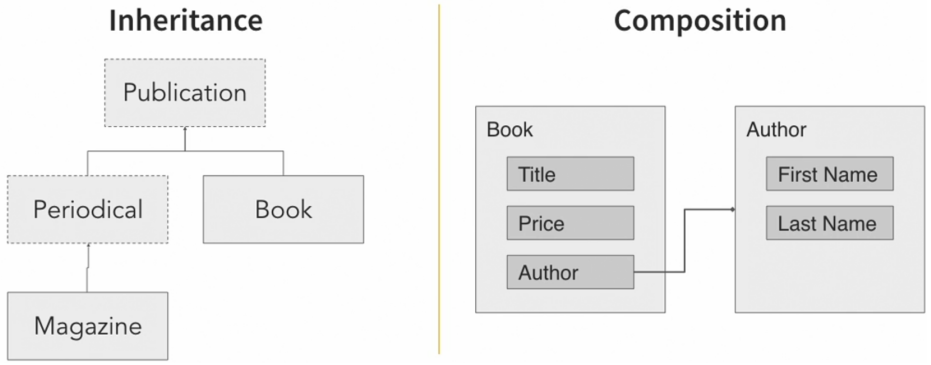
- Una interfaz es una promesa/contrato que hace una clase, que asegurará cierto comportamiento o capacidad en cada clase que herede de ella.
- Python no trae built-in interfaces, hay que desarrollarlas utilizando *una combinación de abstract base classes y herencia múltiple*.
- Las interfaces podrán utilizarse como clase heredada cada vez que se necesite.

Ej: A partir del ejemplo de herencia entre GraphicShape, Circle y Square, se desea contar con una representación en formato JSON de cada objeto instanciado. La implementación será distinta para cada figura, por eso se desarrolla una **interfaz "JSONify" como clase base que tendrá un método abstracto "toJSON" no implementado**.
Luego, Circle y Square, heredarán no solo de GraphicShape, sino también de JSONify.

```
from abc import ABC, abstractmethod

class GraphicShape(ABC):
    def __init__(self):
        super().__init__()

    @abstractmethod
    def calcArea(self):
        pass

class JSONify(ABC):
    @abstractmethod
    def toJSON(self):
        pass

class Circle(GraphicShape, JSONify):
    def __init__(self, radius):
        self.radius = radius

    def calcArea(self):
        return 3.14 * (self.radius ** 2)

    def toJSON(self):
        return f"{{ \"Circle\": {str(self.calcArea())} }}"


c = Circle(10)
print(c.calcArea())
print(c.toJSON())
```

Composición

- Consiste en crear objetos complejos a partir de objetos más simples.



Ej:

```python
class Book:
    def __init__(self, title, price, author=None):
        self.title = title
        self.price = price
        # Use references to other objects, like author and chapters
        self.author = author
        self.chapters = []

    def addchapter(self, chapter):
        self.chapters.append(chapter)

    def getbookpagecount(self):
        result = 0
        for ch in self.chapters:
            result += ch.pagecount
        return result

class Author:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def __str__(self): #Magic function → Se ve en la sección siguiente
        return f"{self.fname} {self.lname}"

class Chapter:
    def __init__(self, name, pagecount):
        self.name = name
        self.pagecount = pagecount

auth = Author("Leo", "Tolstoy")
b1 = Book("War and Peace", 39.95, auth)
b1.addchapter(Chapter("Chapter 1", 104))
b1.addchapter(Chapter("Chapter 2", 89))
```

Magic functions

- Son métodos asociados a cada definición de clase.
- Se reconocen porque están entre doble _: __init__, __str__, etc.
- Se puede hacer overriding de estos, especificando:
    - Cómo se representan los objetos como string.
    - Controlar acceso a atributos (getters y setters)
    - Cómo se hará la comparación entre objetos definidos por mí
    - Invocar objetos como si fueran funciones

A continuación un review de los más utilizados, para ver más ir a la docu de python, sección “Data Models”.

Magic strings:

1. __str__: Para devolver un string cuando se invoque el objeto → Para mostrar al **usuario**
   Por defecto sobreescribe a print(objeto). Una vez definido si se hace: print(objeto) o print(str(objeto)) el resultado será el mismo: En lugar de mostrar Clase.object at 0x00… mostrará algo más bonito.

2. __repr__: Para devolver el estado del objeto. → Para mostrar al desarrollador o para **debuggear**

```python
class Book:
    def __init__(self, title, author, price):
        super().__init__()
        self.title = title
        self.author = author
        self.price = price

    def __str__(self): # Returns User-friendly string
        return f"{self.title} by {self.author}, costs {self.price}"

    def __repr__(self): # Returns Dev-friendly string
        return f"title={self.title},author={self.author},price={self.price}"

b1 = Book("War and Peace", "Leo Tolstoy", 39.95)
print(b1)
print(str(b1))
print(repr(b1))
```

Magic comparators:

1. __eq__: Comprueba la igualdad entre dos objetos. Por defecto python no compara atributos. Si esto no está definido y defino dos objetos de tipo Book con iguales atributos y luego los comparo print(b1 == b2) aunque le haya seteado los mismos atributos, dará False

2. __ge__: Establece la relación “>=” con otro objeto.

3. __lt__: Establece la relación “<” con otro objeto. Al establecer esta magic function, automáticamente **habilitamos a los objetos de clase Book a ser ordenados si los agregamos a una lista**!

```
class Book:
    def __init__(self, title, author, price):
        super().__init__()
        self.title = title
        self.author = author
        self.price = price

    def __eq__(self, value): # Check de igualdad
        if not isinstance(value, Book):
            raise ValueError("Can't compare book to non-book type")
        return (self.title == value.title and
                self.author == value.author and
                self.price == value.price)

    def __ge__(self, value): # Check de >=
        if not isinstance(value, Book):
            raise ValueError("Can't compare book to non-book type")
        return self.price >= value.price

    def __lt__(self, value): # Check de <. Esto además habilita el ordenamiento
        if not isinstance(value, Book):
            raise ValueError("Can't compare book to non-book type")
        return self.price < value.price

b1 = Book("War and Peace", "Leo Tolstoy", 39.95)
b2 = Book("The Catcher in the Rye", "JD Salinger", 29.95)
b3 = Book("War and Peace", "Leo Tolstoy", 39.95)
b4 = Book("To Kill a Mockingbird", "Harper Lee", 24.95)
print(b1 == b3)
print(b2 >= b1)
print(b2 < b1)
books = [b1, b3, b2, b4]
books.sort()
print([book.title for book in books])
```

Magic attributes:

1. __getattribute__: Este método se ejecuta cada vez que un atributo es llamado. Para evitar un bucle recursivo, habrá que tomar cada atributo de la superclase (Ver ej.).
2. __setattr__: Similar pero para cuando se quiere setear el valor de un atributo
3. __getattr__: Este método se invoca cuando __getattribute__ falla, cuando lanza una excepción o cuando el atributo no existe.

Ej:

```
class Book:
    def __init__(self, title, author, price):
        super().__init__()
        self.title = title
        self.author = author
        self.price = price
        self._discount = 0.1

    def __str__(self):
        return f"{self.title} by {self.author}, costs {self.price}"

    def __getattribute__(self, name): # Called when an attribute is retrieved.
        if (name == "price"):
            p = super().__getattribute__("price") # super() para no entrar en bucle rec
            d = super().__getattribute__("_discount")
            return p - (p * d)
        return super().__getattribute__(name) # super() para no entrar en bucle recurs.

    def __setattr__(self, name, value): # Called when an attribute value is set.
        if (name == "price"):
            if type(value) is not float:
                raise ValueError("The 'price' attribute must be a float")
        return super().__setattr__(name, value)

    def __getattr__(self, name): # Called when __getattribute__ lookup fails
        return name + " is not here!"

b1 = Book("War and Peace", "Leo Tolstoy", 39.95)
b2 = Book("The Catcher in the Rye", "JD Salinger", 29.95)
b1.price = 38.95
print(b1)
b2.price = float(40)  # using an int will raise an exception
print(b2)
print(b1.atributo_falso)# If an attribute doesn't exist, __getattr__ will be called
```

Magic call::

- __call__: Permite modificar los atributos de la instancia, llamando al objeto como si fuera una función, *en lugar de utilizar los setters.* Es útil cuando se tienen objetos cuyos atributos son muy cambiantes, para tener un código más compacto y fácil de leer.

Ej:

```
class Book:
    def __init__(self, title, author, price):
        super().__init__()
        self.title = title
        self.author = author
        self.price = price

    def __str__(self):
        return f"{self.title} by {self.author}, costs {self.price}"

    # The __call__ method can be used to call the object like a function
    def __call__(self, title, author, price):
        self.title = title
        self.author = author
        self.price = price

b1 = Book("War and Peace", "Leo Tolstoy", 39.95)
print(b1)
b1("Anna Karenina", "Leo Tolstoy", 49.95) # Acá se está usando!
print(b1)
```

Data class

Feature disponible a partir de python 3.7 que agrega tres comportamientos principales:

1. Completa automáticamente __init__, __repr__ y __eq__.
2. Permite setear “default values”.
3. Permite crear “clases inmutables”.

- Hay que importarlo de la librería dataclasses: **from dataclasses import dataclass**
- El resto del código, las llamadas al objeto, etc, quedarán igual que antes.
- Si quiero definir mi propio init, se debe usar la función **__post_init__** que también es provista por este decorator. Se ejecutará cuando la función init finalice y permite agregar otros atributos.

1. **Completa automáticamente __init__, __repr__ y __eq__.**

Ej:

```python
from dataclasses import dataclass
@dataclass # No será necesario definir explícitamente el def __init__(self,...):
class Book:
    title: str # A pesar de ser requisito de dataclass, el tipado será dinámico
    author: str
    pages: int
    price: float

    def __post_init__(self):  # Se ejecuta luego del init que es automático (x dataclass)
        self.description = f"Description: {self.title} by {self.author}"

    def bookinfo(self): # You can define methods in a dataclass like any other
        return f"{self.title}, by {self.author}"

b1 = Book("War and Peace", "Leo Tolstoy", 1225, 39.95)
b2 = Book("The Catcher in the Rye", "JD Salinger", 234, 29.95)

# access fields
print(b1.title)
print(b2.author)

print(b1) # __repr__ viene implementado por dataclass (se podrá sobreescribir)

# comparing two dataclasses
b3 = Book("War and Peace", "Leo Tolstoy", 1225, 39.95)
print(b1 == b3) # __eq__ viene implementado por dataclass (se podrá sobreescribir)

b1.title = "Anna Karenina" # change some fields, call a regular class method
print(b1.bookinfo())

print(b1.description) # Mostrará el atributo description creado en el __post_init__
```

2. **Permite setear “default values”.**

- Restricción: Los atributos SIN default siempre van primero, sino lanza error.

```
@dataclass
class Book:
    title: str # Restricción los atributos SIN default value deben ir primero
    author: str = ‘No Author’
```

- Los “default arguments” también se pueden trabajar con el módulo **field,** importandolo desde **dataclasses**
- Si hago esto, el resultado es el mismo que utilizar el “=” a secas
  - `price: float = field(default=10.0)`
- **Pero** si se quiere completar con una función y no se usa field(default_factory=...) y se hace directamente:
  - `price: float = float(random.randrange(20, 40))`

  El código se ejecutará una sola vez y seteará el mismo valor para todas las instancias.

Ej completo:

```
from dataclasses import dataclass, field # Agrego este módulo
import random

def price_func():
    return float(random.randrange(20, 40))

@dataclass
class Book:
    title: str = "No Title"
    author: str = "No Author"
    pages: int = 0
    price: float = field(default_factory=price_func) # Calculo con field utilizando la
función que crea un núm. al azar. Esto se ejecuta una vez por cada instancia que se crea.

b1 = Book() # Libro con valores por default
print(b1)
b1 = Book("War and Peace", "Leo Tolstoy", 1225) # Les paso title, author y pages,
b2 = Book("The Catcher in the Rye", "JD Salinger", 234) # price se completa solo
print(b1)
print(b2)
```

3. **Permite crear “clases inmutables”.**

Son clases donde los atributos no pueden ser modificados. Para esto hay que agregar el parámetro **frozen = True** en el decorator. **No serán modificables desde fuera ni desde dentro** (con métodos de clase).

```
from dataclasses import dataclass
@dataclass(frozen=True)  # El parámetro frozen hace a la clase inmutable
class ImmutableClass:
    value1: str = "Value 1"
    value2: int = 0
```