PRO - UT2-A1- Introducción POO

Paradigmas de programación

En programación, un **paradigma** es un enfoque concreto de desarrollar y estructurar el desarrollo de programas

En la unidad anterior hemos aprendido a programar utilizando el paradigma **imperativo**

No es el único paradigma que existe:

- Funcional
- Orientado a objetos

Paradigma de la programación imperativa o estructurada

Consiste en una secuencia de instrucciones que el ordenador debe ejecutar.

Los elementos más importantes en esta forma de programar son:

1. **Variables**: zonas de memoria donde guardamos información.
2. **Tipos de datos**: son los valores que se pueden almacenar.
3. **Expresiones**: corresponde a operaciones entre variables (del mismo o distinto tipo)
4. **Estructuras de control**:

- **Secuenciales**: ejecución de instrucciones de forma consecutiva
- **Bucles**: permiten ejecutar un conjunto de instrucciones varias veces
- **Condicionales**: ejecutar una parte del código u otra en función de que se cumpla una condición o abortar la ejecución del programa.

Paradigma de la programación orientada a objetos

Se basa en la idea de agrupación de datos y funciones relacionados en "unidades" de información.

El tipo de datos nuevo que permite agrupar datos y funciones se llama **clase**

A cada variable de tipo clase se le denomina **objeto**

Ventajas de la POO

La OOP no es mejor ni peor que otros paradigmas. Cada paradigma es más o menos útil en función del tipo de problema que queremos solucionar.

Algunas ventajas de la OOP son:

- **Encapsulación de datos**: los datos y las operaciones para modificarlos pertenecen al objeto y solo son accesibles desde el mismo.
- **Simplicidad**: la creación de grandes sistemas es una tarea compleja, con muchos problemas que resolver. La capacidad de dividir la complejidad en problemas más pequeños, en **objetos** permite simplificar la tarea global.
- **Facilidad de modificación**: cuando se basa en objetos y modela el sistema con ellos, es más fácil realizar el seguimiento de qué partes del sistema se deben modificar. Todo esto facilita la corrección de errorers o agregar una característica nueva.

- **Capacidad de mantenimiento**: en general, el mantenimiento del código es difícil y con el tiempo se complica más. Requiere disciplina en forma de una nomenclatura correcta y una arquitectura clara y coherente, entre otros aspectos. El uso de **objetos** facilita la **búsqueda de un área concreta** del código que necesita mantenimiento.
- **Reusabilidad**: la definición de un objeto se puede usar muchas veces en muchas partes del sistema o, potencialmente, también en otros sistemas.

Clases

Una **clase** es una entidad que define una serie de elementos que determinan un estado (datos) y un comportamiento (operaciones sobre los datos que modifican su estado).

Hemos visto que python define una serie de tipos de datos **primitivos** incluidos con el lenguaje. Los hemos utilizado al definir variables (int, float, str, list, tuple, ...)

Cuando creamos una clase estamos definiendo un nuevo tipo de dato.

Para crear una clase en Python usamos la palabra reservada `class`

Una clase básica la podemos crear en Python de la siguiente forma:

```
class Coche:     # Declaración de la clase
    pass         # Código de la clase
```

> Por convención se usa notación **CamelCase** a la hora de asignar nombres a las clases. Esta notación consiste en poner en mayúscula el primer caracter del nombre de la clase. Si el nombre tiene varias palabras se ponen juntas sin espacios y con el primer caracter de cada palabra en mayúscula (Ejemplo `CocheCarrerasFormula1`)

Creación de objetos

Una clase es un molde, una plantilla, un tipo de dato definido por el programador.

Un objeto es una variable del tipo de la clase que hemos definido.

En Python creamos un objeto asignando a una variable la clase que hemos definido:

```
coche = Coche()
```

Con la asignación anterior hemos creado una **objeto** de la clase **Coche**. A este proceso se ledenomina **instanciar** o crear una **instancia**

Podemos crear todos los objetos de una clase que queramos:

```
c1 = Coche()
c2 = Coche()
```

Añadir atributos a una clase

Los atributos o datos encapsulados en un objeto se crean cuando se crea una instancia de un objeto.

Hay una función especial a la que se llama en el momento de la creación, denominada **constructor**.

Constructor

Un **constructor** es una función especial que solo se invoca al crear por primera vez el objeto. Por tanto, el constructor solo se llamará una vez.

En este método, se crean los **atributos** que debe tener el objeto. Además, se asignan **valores iniciales** a los atributos creados.

Por tanto, este método es el encargado de crear el **estado inicial** de un objeto,

En Python, el constructor tiene el nombre `__init__()`.

A `__init__()` le podemos dar el número de parámetros que queramos, pero el primer parámetro de `__init__()` debe ser siempre una variable especial de nombre `self` que hace referencia al **propio** objeto y permite que le añadamos atributos al objeto:

```
class Coche:
    def __init__(self, color, velocidad):
        self.color = color
        self.velocidad = velocidad
```

> La indentación de `def` es la que permite saber a Python que el método `__init__()` pertenece a la clase `Coche`.

En el cuerpo de `__init__()` hay dos instrucciones que usan la variable `self`:

- `self.color = color` crea el atributo de nombre `color` y asigna al mismo el valor del parámetro `color`
- `self.velocidad = velocidad` crea el atributo de nombre `velocidad` y asigna al mismo el valor del parámetro `velocidad`

Si ahora queremos crear un objeto de la clase Coche de la misma forma:

```
coche = Coche()
Se produjo una excepción: TypeError       (note: full exception trace is shown
but execution is paused at: <module>)
__init__() missing 2 required positional arguments: 'color' and 'velocidad'
  File "/home/ivan/mega/clases/pro/proyectos/test/test_objetos.py", line 6, in
<module> (Current frame)
    coche = Coche()
```

Python devuelve un error indicando que debemos pasar dos parámetros al crear el objeto.

```
coche_rojo = Coche("rojo", 20)
coche_blanco = Coche("blanco", 30)
```

Acabamos de crear dos instancias de la clase `Coche` el objeto `coche_rojo` un coche rojo que circula a 20Kmh y el objeto `coche_verde` que representa a un coche de color verde que circula a 30Kmh

> Fíjate que aunque `__init__()` tiene 3 parámetros solo hemos pasado 2 al crear el objeto; Python se encarga de pasar de forma automática `self` en el momento de llamar al constructor y no debemos pasarlo como parámetro.

Después de crear un objeto podemos acceder a sus atributos usando **notación punteada**

```
>>> coche_rojo.color
'rojo'
>>> coche_blanco.velocidad
30
```

Atributos de clase

Los atributos que acababamos de definir se llaman **atributos de instancia** y son especificos de cada objeto en el momento de crearlos.

Python permite definir también **atributos de clase**, son atributos que tienen el mismo valor en todos los objetos que creemos en una determina clase.

Se crean asignando valor a una variable fuera del método `__init__()`. Dichas variables deben estar indentadas y ubicadas al principio de la definición de la clase. Ademas deben inicializarse.

Por ejemplo, todos los coches tienen en común tener 4 ruedas:

```
class Coche:
    # Atributos de clase
    ruedas = 4

    def __init__(self, color, velocidad):
        # Atributos de instancia
        self.color = color
        self.velocidad = velocidad
```

Con la notación punteada podemos acceder también a los atributos de clase

```
>>> coche_rapido = Coche("azul", 120)
>>> coche_rapido.ruedas
4
```

Métodos de instancia

Los métodos de instancia son funciones definidas dentro de una clase y qué solo pueden ser llamados desde un objeto de la clase.

El primer parámetro de un método, igual que en el constructor `__init__()` debe ser siempre `self`

```
class Coche:
    ruedas = 4

    def __init__(self, color, velocidad):
        self.color = color
        self.velocidad = velocidad

    # métodos de instancia
    def descripcion(self):
        return f"El coche tiene color {self.color} y una velocidad de
{self.velocidad} Kmh"

    def acelera(self, incremento):
        self.velocidad += incremento
```

```
        return f"Nueva velocidad: {self.velocidad} Kmh"
```

Hemos añadido a la clase dos métodos de instancia:

- `descripcion()` que devuelve una cadena mostrando el color y la velocidad del coche
- `acelera()` que incrementa la velocidad del coche con el parámetro que le pasemos y devuelve una cadena de texto que informa de la nueva velocidad.

Para ejecutar los métodos de instancia usamos notación punteada.

```
>>> coche_rojo = Coche("rojo", 80)

>>> coche_rojo.descripcion()
'El coche tiene color rojo y una velocidad de 80 Kmh'

>>> coche_rojo.acelera(20)
'Nueva velocidad: 100 Kmh'
```

El método `__str__()`

Cuando creamos una clase es buena idea disponer de un método que permita mostrar la información relevante del objeto. En el caso anterior hemos creado un método de nombre `descripcion()` que realiza dicha función, pero está no es la forma más "Pythonica" de hacerlo

Por ejemplo, si creamos una lista Python muestra una representación de la misma si usamos `print()` para mostrarla.

```
>>> names = ["Fletcher", "David", "Dan"]
>>> print(names)
['Fletcher', 'David', 'Dan']
```

Sin embargo, si llamamos con `print()` un objeto que hayamos creado de la clase Coche obtenemos algo como:

```
>>> print(coche_rojo)
<__main__.Coche object at 0x00aeff70>
```

Obtenemos un mensaje críptico que indica que `coche_rojo` es un objeto de la clase `Coche` y a continuación su dirección de memoria.

Podemos cambiar el resultado de `print()` sobre un objeto creando un método especial de nombre `__str__()`.

Si para la clase anterior cambiamos el nombre del método `descripcion()` por `__str__()` ahora obtendremos un resultado más amigable al mostrar con `print()` el objeto:

```
class Coche:
    ruedas = 4

    def __init__(self, color, velocidad):
        self.color = color
        self.velocidad = velocidad

    # métodos de instancia
```

```
    def __str__(self):
        return f"El coche tiene color {self.color} y una velocidad de
{self.velocidad} Kmh"

    def acelera(self, incremento):
        self.velocidad += incremento
        return f"Nueva velocidad: {self.velocidad} Kmh"

coche_rojo = Coche("rojo", 80)

print(coche_rojo)
```

Al ejecutar obtenemos:

```
El coche tiene color rojo y una velocidad de 80 Kmh
```

Recursos

- POO - ferestrepoca
- Tutorial POO - J2Logo
- python oop - curso microsoft.com
- OOP en Python 3 - Realpython
- Object Oriented Programming in Python - Usman Malik - stackabuse.com