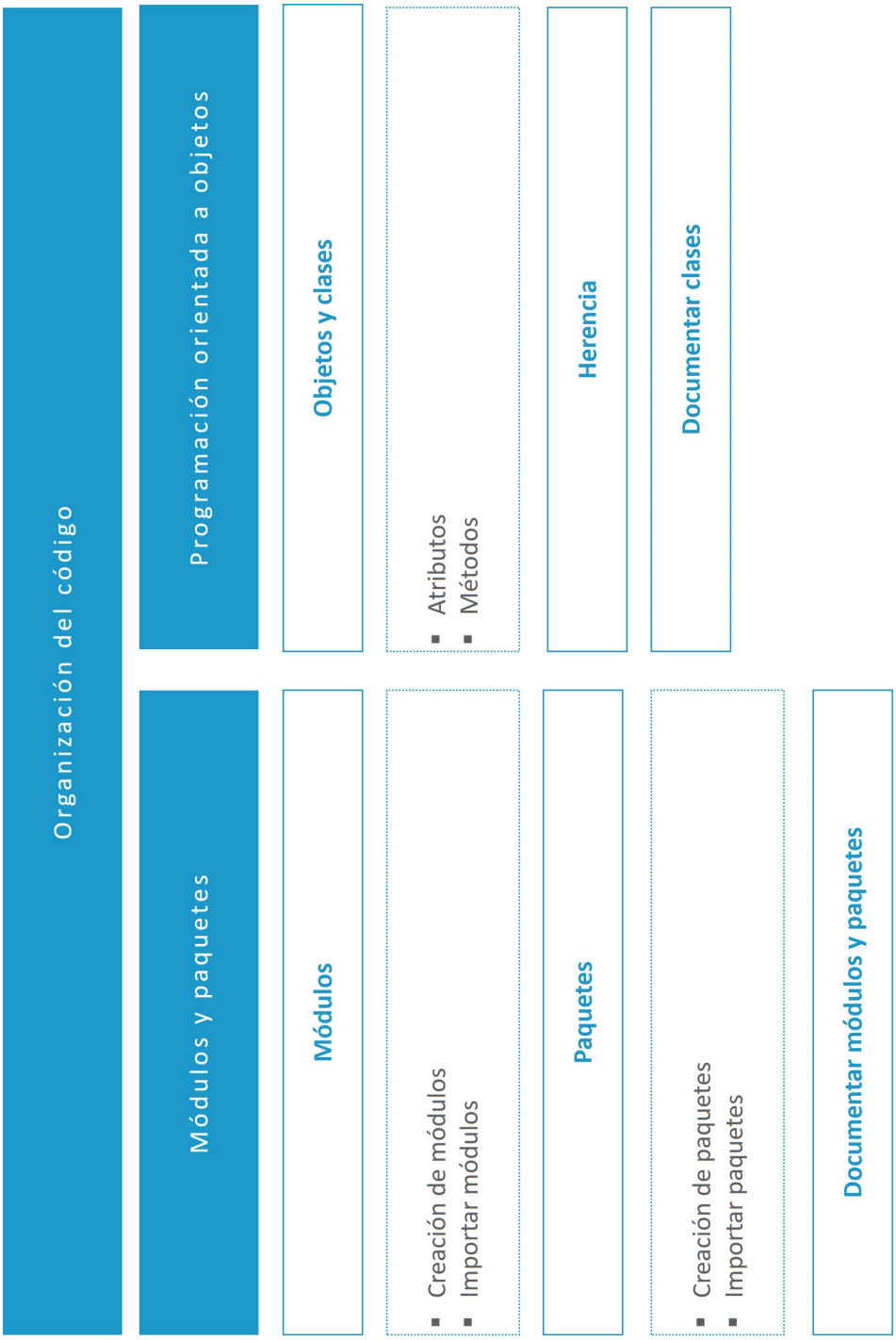
Curso de Programación en Python

Tema 5. Organización del código

Índice

Esquema



5.1. Introducción y objetivos

A la hora de desarrollar proyectos más grandes en Python es necesario organizar nuestro código. En este tema veremos cómo hacer esto con dos aproximaciones. La primera de ellas es el uso de módulos y paquetes que nos permitirán organizar nuestro código en diferentes *scripts* y, además, crear una organización de carpetas para almacenar cada uno de estos *scripts.* La segunda aproximación es utilizar la programación orientada a objetos para encapsular la información en objetos y crear diferentes operaciones que nos permitan trabajar con ellos.

Al finalizar este tema se habrán alcanzado los siguientes objetivos que mejorarán la forma en la que organicemos el código en Python:

1. Aprender a crear módulos y paquetes en Python.
2. Saber cómo se deben importar los módulos y paquetes en nuestro código.
3. Conocer cómo se debe documentar los módulos y los paquetes.
4. Saber cómo aplicar la programación orientada a objetos en Python.
5. Aprender cómo funciona la herencia dentro de la programación orientada a objetos en Python.
6. Conocer cómo se puede documentar las clases desarrolladas en Python.

5.2. Módulos y paquetes

Hasta ahora hemos estado desarrollando nuestro programa usando un único *script* o en un mismo *notebook*. Sin embargo, en proyectos más grandes es necesario dividir nuestro código en diferentes *scripts* para organizarlo. Lo más normal es organizar nuestro código en *scripts* que tengan funciones o variables con una misma funcionalidad.

Para ayudarnos a organizar nuestro código usaremos los módulos y paquetes. A continuación, explicaremos qué son cada uno de estos elementos, cómo podemos crearlos y cómo podemos usarlos en otros *scripts.*

Módulos

Los módulos son cada uno de los ficheros .py que creemos en nuestro proyecto. Estos ficheros contienen elementos que hemos visto anteriormente, como variables o funciones, y clases que veremos en el siguiente apartado. Dividir el código en módulos nos permite organizar un conjunto de elementos por su funcionalidad.

Creación de módulos

Imaginemos que queremos crear un módulo que contendrá dos funciones que nos permiten calcular el perímetro de una circunferencia y el área del círculo contenido en una circunferencia. Para ello, crearemos un *script* llamado «circunferencia.py» que contendrá el siguiente código:

```python
# Módulo circunferencia.py
import math

def perimetro(radio):
    return 2 * math.pi * radio

def area(radio):
    return math.pi * radio ** 2
```

Este módulo contiene las funciones `perímetro` y `area` que, a partir del valor del radio, obtiene ambos valores. Para el número pi, hemos usado el módulo `math` que vimos en temas anteriores.

Hasta aquí ya tendríamos nuestro módulo, que nos permite hacer algunos cálculos sobre las circunferencias. A continuación, veremos cómo usar nuestro módulo en otros *scripts* o en un *notebook.*

Importar módulos

Para poder utilizar las funciones que hemos incluido en un módulo es necesario importar dicho módulo. Python tiene la sentencia `import` que, seguida del nombre del módulo (sin la extensión .py), nos permite utilizar todo el código que hayamos implementado en dicho módulo. Es importante tener en cuenta que el módulo tiene que ser accesible por el *script* que lo importe, si no es accesible, nos devolverá un error indicando que no sabe dónde se encuentra.

Por ejemplo, vamos a importar el módulo circunferencia para hacer algunos cálculos en nuestro *notebook*. Para ello, comenzaremos nuestro código con la siguiente sentencia:

```
import circunferencia
```

Una vez hecho esto, podemos acceder al código del módulo `circunferencia`. Sin embargo, es necesario usar el nombre del módulo seguido de un punto (.) para acceder a las funciones. Por ejemplo, para calcular el perímetro de una circunferencia con un radio de 5 unidades, usaríamos la siguiente instrucción:

```
circunferencia.perimetro(5) # Devolverá 31.41592653589793
```

Esta forma de llamar a la función `perimetro` se realiza mediante el *namespace.* El *namespace* es el nombre que se indica después de la instrucción `import` y que será la ruta del módulo, como veremos más adelante. En la sentencia `import` podemos incluir más de un módulo. Para ello solo tenemos que separar los módulos que vamos a importar por comas.

```
import modulo1, modulo2, modulo3, …, moduloN
```

Imaginemos que, en nuestro proyecto, tenemos otro módulo llamado «poligono», que calcula el área y el perímetro de todos los polígonos regulares. Para poder importar los dos módulos en nuestro programa (el de circunferencia y el de polígono), ejecutaríamos la siguiente sentencia:

```
import circunferencia, poligono
```

Y para llamar a las funciones de los diferentes módulos deberíamos usar el *namespace* de cada módulo, es decir, si queremos llamar a la función `area` de cada módulo, deberíamos hacerlo de la siguiente manera:

```
circunferencia.area(5)

poligono.area(10, 5)
```

Como se puede observar, el uso de los *namespace* a veces puede ser un poco tedioso, sobre todo si el nombre de los módulos es complejo o muy largo. Para solucionar esto, Python nos permite definir unos alias en los nombres de los módulos a la hora de importarlos. Así, solo deberíamos usar los alias a la hora de llamarlos dentro de nuestro código. Una restricción que existe con el uso de los alias es que no podemos hacer importaciones múltiples en una única sentencia, sino que debemos hacer las importaciones de una en una. Para definir un alias a un módulo importado, usaremos la palabra reservada as seguida del alias que queramos definir a nuestro modulo.

```
import modulo as mod
```

En el ejemplo anterior, con nuestros módulos `circunferencia` y `poligono`, vamos a importar ambos módulos con un alias. A continuación, ejecutaremos las funciones para calcular el área usando los alias que hemos definido:

```
import circunferencia as cir
import poligono as pol
cir.area(5)
pol.area(10, 5)
```

Como vemos, esto nos permite que la forma de llamar a los módulos sea más sencilla y el código más legible.

Paquetes

Para proyectos verdaderamente grandes, es necesario agrupar diferentes módulos en carpetas. Así, podemos cargar varios módulos que tienen funcionalidades similares.

Creación de paquetes

Un paquete es, básicamente, una carpeta que contiene varios módulos. Sin embargo, este paquete debe tener un fichero Python llamado __init__.py que puede estar vacío. Sin embargo, es aconsejable que el fichero __init__.py incluya los import de todos los módulos que están incluidos en el paquete. Un ejemplo de estructura de paquete sería el siguiente:

```
paquete/
├── __init__.py
├── modulo1.py
└── modulo2.py
```

En este caso, nuestro fichero __init__.py debería incluir las siguientes instrucciones para que se pudiesen importar los módulos cuando importemos únicamente el paquete:

```
import paquete.modulo1
import paquete.modulo2
```

En nuestro ejemplo anterior, podríamos incluir los módulos circunferencia y polígono en un paquete llamado figuras. Para ello, haremos una estructura de ficheros de la siguiente manera:

```
figuras/
├── __init__.py
├── circunferencia.py
└── poligono.py
```

Importando módulos en paquetes

Los módulos que están incluidos en paquetes tienen una forma un poco diferente de ser importados. La primera de las formas es importar el paquete donde se encuentran esos módulos. Para ello usaremos la instrucción `import` seguida del nombre del paquete o paquetes (separados por comas) que queremos importar:

```
import paquete1, paquete2, …, paqueteN
```

Vamos a ver un ejemplo en el que importamos el paquete `figuras` y ejecutamos las funciones del cálculo de área que hemos visto anteriormente:

```
import figuras


figuras.circunferencia.area(5)
figuras.poligono.area(10, 5)
```

Como vemos en el ejemplo, para ejecutar cada función es necesario usar el nombre del paquete, seguido del nombre del módulo y, por último, el nombre de la función que se quiere ejecutar.

Una forma de simplificar esto consiste en usar la palabra reservada `from` en la importación. Esta palabra nos indica dónde se encuentran los módulos que queremos importar. Para ello usaremos una de las siguientes estructuras de importación.

```
from paquete import modulo1, modulo2, …, moduloN
from paquete import *
```

La diferencia de estas dos estructuras es la siguiente. En la primera estructura podemos importar algunos módulos del paquete, para ello ponemos los nombres de

los módulos que queremos importar separados por comas. Por su parte, la segunda estructura usa el asterisco (*) para indicar que importamos todos los módulos que existen dentro del paquete. Si usamos estas dos formas, ya no es necesario usar el nombre del paquete a la hora de ejecutar funciones de los módulos que hemos importado. Vamos a ver esto con nuestro ejemplo del paquete figuras :

```
from figuras import *


circunferencia.area(5)
poligono.area(10, 5)
```

También podemos usar los alias en esta estructura para importar módulos, aunque tendremos que importar los módulos de uno en uno. Para ello, incluiremos la palabra reservada as seguida del alias que queramos utilizar para nuestro módulo:

```
from paquete import modulo as mod
```

Veamos esta forma de importar los módulos con nuestro ejemplo de las figuras:

```
from figuras import circunferencia as cir
from figuras import poligono as pol


cir.area(5)
pol.area(10, 5)
```

Es posible crear paquetes dentro de otros paquetes. Para importar los módulos que hay dentro de paquetes incluidos en otros paquetes se utilizan las formas que hemos visto anteriormente, pero separando los nombres de la ruta de los paquetes por puntos (.).

```
from paquete.subpaquete import *
```

Documentar módulos y paquetes

Como hemos dicho en las funciones, es importante documentar nuestro código para que otros desarrolladores sepan cómo deben usar los objetos que hemos creado. Esto también se debe tener en cuenta en los módulos. Es posible documentar nuestros módulos para que otros desarrolladores sepan qué elementos están incluidos en los módulos que hemos desarrollado. Para ello, al igual que las funciones, podemos usar los *docstring.*

El *docstring* de un módulo debe estar en la primera línea del este. Para ello, escribiremos esta documentación usando las dobles comillas (“) tres veces para iniciar la documentación y otras tres veces para cerrarlo. Esta documentación puede ocupar más de una línea.

Por ejemplo, podemos crear el *docstring* del módulo circunferencia de la siguiente manera:

```
"""Modulo circunferencia:

Incluye las funciones que nos permiten obtener el perímetro o
área de una circunferencia.
"""

import math

def perimetro(radio):
    return 2 * math.pi * radio

def area(radio):
    return math.pi * radio ** 2
```

La gran ventaja de utilizar *docstring* consiste en que podemos consultar esta documentación con la instrucción `help()` e incluir el nombre del módulo del que queremos consultar la documentación.

```
from figuras import circunferencia
help(circunferencia)
```

```
Help on module figuras.circunferencia in figuras:

NAME
    figuras.circunferencia - Modulo circunferencia:

DESCRIPTION
    Incluye las funciones que nos permiten obtener el perímetro o
    área de una circunferencia.

FUNCTIONS
    area(radio)

    perimetro(radio)
```

Figura 1. Instrucción help() con *docstring.*

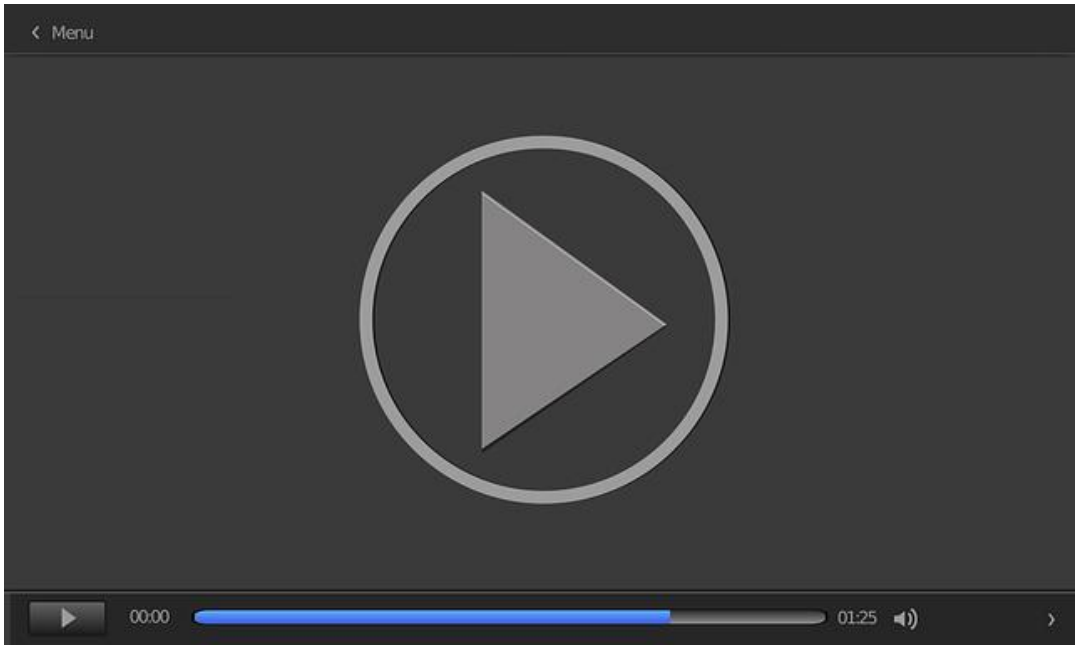
De la misma manera podemos incluir esta documentación en los paquetes. Para ello, el *docstring* debe estar incluido en la primera línea del archivo `__init__.py` del paquete. Al ejecutar la instrucción `help()` con el nombre del paquete nos devolverá la documentación que hemos escrito.

```
import figuras
help(figuras)
```

```
Help on package figuras:

NAME
    figuras - Paquete figuras:

DESCRIPTION
    Incluye las funciones para calcular el área y el perímetro de diferentes formas.

PACKAGE CONTENTS
    circunferencia
    poligono
```

Figura 2. Instrucción help() en paquetes.

A continuación veremos el vídeo titulado *Módulos y paquetes*.



Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=5f0d5827-e0b0-4f85-be74-af4e00be52c1

5.3. Programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma de programación en el que se encapsulan los datos en forma de objeto y, a cada uno de ellos, se les define un conjunto de funcionalidades para trabajar con los datos que están almacenados en los objetos. En este epígrafe vamos a ver todos los conceptos relacionados con POO.

Objetos y clases

En POO, los elementos más importantes son los **objetos.** Un objeto es una abstracción de los datos con tres características:

- **Tienen un estado**. Es decir, el objeto tiene un valor concreto en un momento determinado.
- **Tienen un comportamiento**. A partir de un conjunto de métodos podemos modificar el estado de un objeto.
- **Tienen una identidad**. Los objetos tienen un identificador que permite diferenciarlos entre ellos.

Los objetos pueden ser cualquier cosa que podamos definir con las categorías que hemos visto antes. Por ejemplo, un objeto puede ser un empleado, un animal, un libro, etc. Todos estos elementos podemos definirlos con un conjunto de características, un comportamiento y una identidad. Por ejemplo, podemos definir un libro, el cual puede tener las siguientes características:

- Título.
- Autor.

- ISBN.
- Editorial.
- Páginas.
- Edición.

Y podemos definir diferentes métodos que permitan modificar esos atributos, como modificar la edición o la editorial. Todo esto lo podemos definir en Python. Para ello, lo que implementaremos es la clase `Libro`. Una clase es, digamos, una estructura que tienen que seguir los objetos de dicha clase. En una clase se les define qué operaciones pueden hacer y qué características tienen. Para crear una clase en Python utilizaremos la palabra reservada `class`, seguida del nombre de la clase y de dos puntos (:) al final de la sentencia:

```
class Libro:
```

Aunque no es obligatorio, los nombres de las clases deben comenzar por una mayúscula. Una vez que hemos comenzado por definir la clase, podemos incluir diferentes elementos según los necesitemos.

Atributos

Los atributos son un conjunto de valores que almacenan las características de un objeto en un estado concreto. En el ejemplo del libro, los atributos serán las características que hemos definido antes para un libro. Estos atributos se incluyen en la clase y pueden almacenar cualquier tipo de dato o estructura:

```
class Libro:
    titulo = 'Don Quijote de la Mancha'
    autor = 'Miguel de Cervantes'
    isbn = '0987-7489'
    editorial = 'Mi Editorial'
    paginas = 934
    edicion = 34
```

Como se puede observar, los atributos comienzan por una letra minúscula. Aunque no es obligatorio, se trata de un estándar en el estilo de programación. En este ejemplo, la clase incluye un valor inicial a cada uno de los atributos; sin embargo, esto no siempre es necesario, como veremos más adelante. Una vez que hemos creado la clase, podemos crear objetos de dicha clase. Para ello, asignaremos a un identificador una clase seguida de los paréntesis:

```
variable = Clase()
```

Por ejemplo, si queremos crear una instancia (es decir, un nuevo objeto) de la clase Libro, usaríamos la siguiente instrucción:

```
mi_libro = Libro()
```

Esto hará que tengamos un nuevo objeto `Libro` asignado a la variable `mi_libro`. Una vez que tenemos creado el objeto, podemos acceder a sus atributos para conocer su valor. Para ello, escribiremos el nombre de la variable seguido de un punto (.) y del nombre del atributo que queremos leer. Por ejemplo, si queremos acceder al título del libro, ejecutaríamos la siguiente sentencia:

```
mi_libro.titulo # Devolverá 'Don Quijote de la Mancha'
```

Los atributos de un objeto solo deben ser modificados a partir de las operaciones válidas que se hayan definido en la clase. Esto se hace con la definición de los métodos.

Métodos

Los métodos son las funciones que incluiremos en las clases y que definirán qué operaciones podemos realizar sobre los objetos. Algunos métodos pueden modificar el estado de un objeto, es decir, modificar el valor de los atributos.

Los métodos deben estar dentro del bloque de código de la clase, lo que significa que deben estar alineados correctamente con la sangría de 4 espacios dentro de la clase. Se definen igual que las funciones normales; sin embargo, deben incluir como primer parámetro la instancia del objeto que le llama, así Python sabe identificar que dicho método pertenece a una clase concreta. Para ello, el primer parámetro de los métodos de una clase será la palabra reservada self :

```
class MiClase:

    def metodo(self, param1, param2):
        return resultado
```

En el mismo ejemplo del libro vamos a definir un método que nos imprima un mensaje estándar con su título y autor. Nuestra clase quedará de la siguiente manera:

```
class Libro:
    titulo = 'Don Quijote de la Mancha'
    autor = 'Miguel de Cervantes'
    isbn = '0987-7489'
    editorial = 'Mi Editorial'
    paginas = 934
    edicion = 34

    def imprime(self):
        print(self.titulo + " - " + self.autor)
```

Dentro del método, para obtener o modificar el valor de los atributos en el momento de ejecutar el método, usaremos el parámetro `self`, ya que es el objeto desde el cual se está ejecutando el método. Este método lo podremos ejecutar desde la variable que almacena el objeto de la siguiente manera:

```
mi_libro.imprime() # Devolverá Don Quijote de la Mancha - Miguel de Cervantes
```

Como se puede observar, no es necesario pasarle por parámetro nada a la función, ya que el parámetro `self` hace referencia a la instancia del propio objeto, en este caso, `mi_libro`.

Un método especial en las clases es el método `__init__()`. Este método se ejecuta en el momento en que creamos un nuevo objeto. El comportamiento de esta función es muy similar al de los constructores en otros lenguajes de programación. Los parámetros que se incluyan en este método deberán hacerlo como argumentos a la hora de crear una instancia de dicha clase. Por ejemplo, imaginemos que, al construir un objeto de la clase `Libro`, queremos pedir al usuario el valor de cada uno de los atributos. Para ello, crearíamos el siguiente método `init`:

```
class Libro:

    def __init__(self, titulo, autor, isbn, editorial, paginas, edicion):
        self.titulo = titulo
        self.autor = autor
        self.isbn = isbn
        self.editorial = editorial
        self.paginas = paginas
        self.edicion = edicion

    def imprime(self):
        print(self.titulo + " - " + self.autor):
```

Al haber definido el método __init__() , en el momento en que queramos crear la instancia de un Libro, tendremos que introducir todos esos argumentos:

```
mi_libro = Libro('Don Quijote de la Mancha', 'Miguel de Cervantes', '0987-
7489', 'Mi Editorial', 934, 34)
```

Esto nos permite crear diferentes objetos que pertenecen a una misma clase. En nuestro ejemplo podemos crear una colección de libros donde cada uno de estos sea un objeto diferente.

Herencia

La herencia es una de las características más importantes que hay en la programación orientada a objetos. La herencia es un mecanismo que nos permite que las clases puedan heredar métodos y atributos de otras clases. Esto nos permite definir nuevas clases a partir de otras que ya existen y a las que queremos ampliar su funcionalidad o hacer que una funcionalidad concreta sea más específica.

Para explicar el funcionamiento de la herencia vamos a suponer que queremos modelar los siguientes datos para crear una aplicación para la universidad:

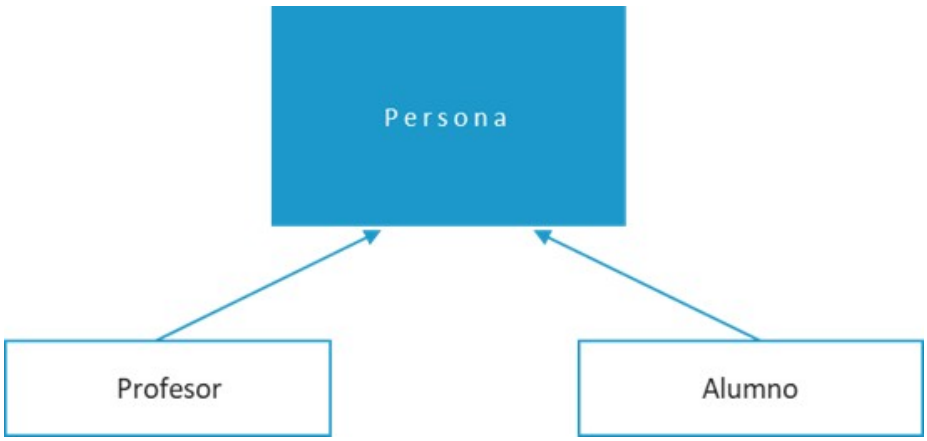


Figura 3. Esquema de clases para el ejemplo de una aplicación para la universidad.

En este caso, tenemos dos clases (`Alumno` y `Profesor`) que almacenarán la información concreta de cada uno de ellos y, además, tendrán diferentes funcionalidades. Sin embargo, existe cierta información que es común a ambas, como puede ser el nombre, la fecha de nacimiento o el domicilio. En este caso, para no duplicar la información, crearemos una clase `Persona` , la cual tendrá los atributos y métodos comunes de `Profesor` y `Alumno` . Esta clase tendrá la siguiente implementación:

```
class Persona:

    def __init__(self, nombre, fecha_nacimiento, domicilio):
        self.nombre = nombre;
        self.fecha_nacimiento = fecha_nacimiento
        self.domicilio = domicilio

    def cambiar_domicilio(self, nuevo_domicilio):
        self.domicilio = nuevo_domicilio
```

En este ejemplo los atributos comunes serán el nombre, la fecha de nacimiento y el domicilio. Además, incluiremos el método `cambiar_domicilio` para que ambos roles (alumnos y profesores) puedan cambiar de domicilio. A continuación, implementamos

las clases que heredarán los atributos y métodos de Personas. Para ello debemos hacer lo siguiente:

- En la primera línea escribimos la palabra reservada `class`, seguida del nombre de la nueva clase y, a continuación, la clase de la que se van a heredar los atributos y métodos entre paréntesis:

```
class NuevaClase(ClasePadre):
```

- Definir el método `__init__()`. Este método debe incluir los parámetros que se han definido en la clase padre y se incluirán los nuevos parámetros. Dentro de este método se llamará al método `__init()__` de la clase padre para crear el objeto:

```
def __init__(self, param_padre1, …, param_padreN, nuevo_param1, …
nuevo_param2):
    ClasePadre.__init__(self, param_padre1, …, param_padreN)
    self.nuevo_attr1 = nuevo_param1
    …
    self.nuevo_attrN = nuevo_paramN
```

- A continuación, implementamos los métodos específicos de la nueva clase.

Veamos esto con el ejemplo de profesores y alumnos. Vamos a implementar la clase `Alumno` a la que, además de los atributos y métodos de la clase `Persona`, incluiremos en qué asignatura está matriculada y su calificación. También añadiremos un método para incluir la calificación de un alumno. Esta clase quedaría de la siguiente manera:

```
class Alumno(Persona):

    def __init__(self, nombre, fecha_nacimiento, domicilio,
        asignatura_matriculada):

        Persona.__init__(self, nombre, fecha_nacimiento, domicilio)

        # Nuevos atributos
        self.asignatura_matriculada = asignatura_matriculada
        self.calificacion = None

    def calificar(self, calificacion):
        self.calificacion = calificacion
```

Por otra parte, tendremos la clase `Profesor` que incluye su especialidad y una lista donde se incluirán las asignaturas que imparte. Además, incluimos un método para ir sumando nuevas asignaturas en su lista de asignaturas impartidas. La clase `Profesor` quedaría de la siguiente manera:

```
class Profesor(Persona):

    def __init__(self, nombre, fecha_nacimiento, domicilio,
        especialidad):

        Persona.__init__(self, nombre, fecha_nacimiento, domicilio)

        # Nuevos atributos
        self.especialidad = especialidad
        self.asignaturas_impartidas = []

    def anyadir_asignatura(self, asignatura):
        self.asignaturas_impartidas.append(asignatura)
```

Ahora vamos a ver las propiedades de la herencia creando un objeto de la clase Alumno y un objeto de la clase Profesor. Para ello usaremos sus correspondientes constructores:

```
alumno = Alumno('Juan', '27/09/1992', 'C/ Gran Vía 25', 'Inteligencia Artificial')
profesor = Profesor('Roberto', '12/03/1976', 'Plaza Mayor 1', 'Sistemas Inteligentes')
```

Ahora ambos métodos, como heredan las propiedades de la clase Persona, nos permiten acceder a los atributos y el método definido en la clase padre (en este caso Persona):

```
alumno.nombre # Devolverá el nombre del alumno
profesor.fecha_nacimiento # Devolverá la fecha de nacimiento del profesor

alumno.cambiar_domicilio('C/ Mayor 35')
alumno.domicilio # Devolverá el nuevo domicilio del alumno
```

Pero, además, las clases Alumno y Profesor tienen sus propios atributos y métodos que no pueden ser accesibles desde otra clase:

```
alumno.asignatura_matriculada # Devolverá la asignatura matriculada del alumno
profesor.asignatura_matriculada # Devolverá un error de atributo
```

Documentar clases

Python también nos permite documentar las clases y sus métodos para que esta documentación pueda ser accesible desde la función help(). Para documentar las clases, pondremos un comentario de bloque usando tres comillas dobles (“) para abrirlo y para cerrarlo, justo debajo de la sentencia donde definimos la clase y su nombre. Los métodos se documentan de la misma forma que vimos en el tema de las funciones. A continuación, se muestra un ejemplo sobre cómo documentar la clase Persona que hemos visto en este tema:

```
class Persona:
    """Clase que encapsula la información básica de una persona"""

    def __init__(self, nombre, fecha_nacimiento, domicilio):
        """ Constructora de la clase Persona.

        Argumentos:
        nombre -- nombre de la persona
        fecha_nacimiento -- fecha de nacimiento de la persona
        domicilio -- domicilio de la persona
        """
        self.nombre = nombre;
        self.fecha_nacimiento = fecha_nacimiento
        self.domicilio = domicilio

def cambiar_domicilio(self, nuevo_domicilio):
    """
    Función que permite modificar el domicilio de una persona.

    Argumentos:
    nuevo_domicilio -- nuevo domicilio de la persona.
    """
    self.domicilio = nuevo_domicilio
```

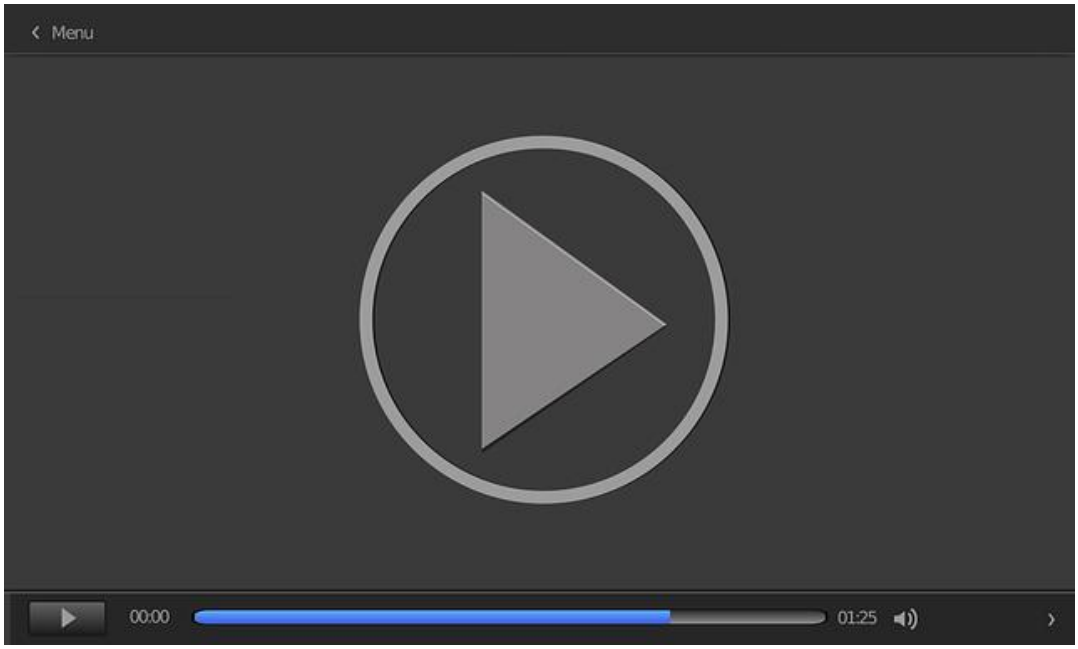
Al estar nuestra clase Persona documentada, podemos aplicar la función `help()` para que nos muestre información de la clase y de los métodos:

```
Help on class Persona in module Herencia:

class Persona(builtins.object)
 |  Persona(nombre, fecha_nacimiento, domicilio)
 |
 |  Clase que encapsula la información básica de una persona
 |
 |  Methods defined here:
 |
 |  __init__(self, nombre, fecha_nacimiento, domicilio)
 |      Constructora de la clase Persona.
 |
 |      Argumentos:
 |      nombre -- nombre de la persona
 |      fecha_nacimiento -- fecha de nacimiento de la persona
 |      domicilio -- domicilio de la persona
 |
 |  cambiar_domicilio(self, nuevo_domicilio)
 |      Función que permite modificar el domicilio de una persona.
 |
 |      Argumentos:
 |      nuevo_domicilio -- nuevo domicilio de la persona.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Figura 4. Función help() en nuestra clase Persona.

Ideas clave

A continuación veremos el vídeo titulado *Programación orientada a objetos*.



Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=f8c47ddf-3e31-4ad0-a486-af4e00c00680

Documentación módulos

Python. (28 de septiembre de 2020). Modules. *The Python Tutorial* [Página web]. https://docs.python.org/3/tutorial/modules.html

En este tutorial de Python se detalla cómo crear módulos y cómo importarlos en nuestros proyectos para usar su contenido de diferentes maneras.

Documentación paquetes

Python. (28 de septiembre de 2020). Modules. *The Python Tutorial* [Página web]. https://docs.python.org/3/tutorial/modules.html#packages

Esta sección del tutorial de Python describe qué son los paquetes, cómo crearlos y las diferentes maneras que tenemos para importar el contenido de estos paquetes.

Programación orientada a objetos

Python. (28 de septiembre de 2020). Classes. *The Python Tutorial* [Página web]. https://docs.python.org/3/tutorial/classes.html

Este capítulo del tutorial de Python describe todos los conceptos de la programación orientada a objeto aplicado a Python, mostrando con ejemplos cada uno de estos conceptos.

Test

1. ¿Cuál es la principal característica de los módulos?

A. Poder hacer pruebas en nuestro desarrollo.

B. Organizar el código de manera que podamos agrupar los diferentes elementos que desarrollemos.

C. Implementar la interfaz gráfica de nuestra aplicación.

D. Adaptar nuestro código de programación a un entorno de desarrollo.

2. ¿Qué utilidad tiene el alias a la hora de importar un módulo?

A. El alias permite asignar otro identificador a dicho módulo.

B. El alias se debe poner para módulos que ocupan mucho espacio.

C. El alias no se pueden usar al importar módulos, solo al importar paquetes.

D. El alias mejora la eficiencia de un módulo.

3. ¿Cómo debemos crear un paquete en Python?

A. Creando una carpeta cuyo nombre empiece por `pkg` .

B. Creando un módulo que solo incluya instrucciones de importación.

C. Creando una carpeta donde exista un fichero llamado `init.py` .

D. Asignando módulos a un objeto de tipo paquete.

4. En la siguiente importación de un paquete, ¿cómo debemos acceder a la función `test()` del módulo `mi_modulo` ?

```
from paquete import mi_modulo as mod
```

A. `paquete.mimodulo.test()` .

B. `mimodulo.mod.test()` .

C. `paquete.mod.test()` .

D. `mod.test()` .

5. ¿Qué es un objeto?

A. Un conjunto de funciones para nuestro código.

B. Un módulo que se puede incluir a Python.

C. Una abstracción de los datos que tiene un estado, un comportamiento y una identidad.

D. Una estructura de datos.

6. Cuando implementamos un método en una clase, ¿para qué sirve el parámetro self ?

A. Para dar al método la instancia del objeto que va a utilizar.

B. Se trata de un parámetro anónimo.

C. Este parámetro indica que la función es un método.

D. Este parámetro no hay que incluirlo en los métodos.

7. Para la siguiente clase, ¿cómo podríamos crear una instancia?

```
class MiClase:
    def __init__(self, nombre, puntuacion=0.0):
        self.nombre = nombre
        self.puntuacion = puntuacion
```

A. objeto = new MiClase() .

B. objeto = MiClase() .

C. objeto = MiClase.init('Hola', 10) .

D. objeto = MiClase('Hola') .

8. Cuando implementamos una clase que es una herencia de otra clase, ¿a qué elementos de la clase padre podemos acceder desde la clase hija?

A. Solo a los atributos de la clase padre.

B. Solo a los métodos de la clase padre.

C. A los atributos y métodos de la clase padre.

D. Solo a la constructora de la clase padre.

9. Supongamos que hemos implementado las siguientes clases. ¿Qué se mostrará en la consola si ejecutamos el método imprimir del objeto?

```
class ClasePadre:
    def imprimir(self):
        print('Hola, soy la clase padre')

class ClaseHija(ClasePadre):
    def imprimir(self):
        print('Hola, soy la clase hija')
objeto = ClaseHija()
```

A. Mostrará un error al no saber a qué método nos referimos.

B. «Hola, soy la clase hija».

C. «Hola, soy la clase padre».

D. No se puede imprimir porque no le hemos pasado el argumento `self`.

10. ¿Para qué sirven los *docstring*?

A. Para obtener información del módulo, paquete o clase, como la fecha de creación, autor, etc.

B. Para definir variables de tipo cadena de caracteres.

C. Para definir constantes en un módulo.

D. Para escribir documentación que luego puede ser consultada por la instrucción `help()`.