

# Proyecto gestor de clientes en Python

## Requisitos

Vamos a crear un programa poniendo a prueba lo que sabemos de Python:

- Listar los clientes del gestor.
- Consultar un cliente a partir del **dni**.
- Agregar un cliente con campos **nombre, apellido, dni**.
- Modificar el nombre y apellido de un cliente a partir del **dni**.
- Borrar un cliente a partir del **dni**.
- Salir del programa.

No deberá guardar los datos en el disco duro, siempre partirá de unos clientes de prueba iniciales y no podrá haber dos clientes con el mismo **dni**.

**Repositorio:** [https://github.com/rubences/Gestor\\_Clientes.git](https://github.com/rubences/Gestor_Clientes.git)

## La buena manera de estructurar un proyecto de Python

Ya sea que estés trabajando en un proyecto por su cuenta o colaborando con otros, hay **5 puntos simples para ayudarlo a mantener su espacio organizado**.

### Básicos

- #1: Cree un entorno virtual
- #2: Cree un directorio separado para las pruebas
- #3: Crea diferentes directorios de contenido
- #4: Documente su código
- #5: Use GitHub para el control de versiones

### Consejo #1: Cree un entorno virtual

Para mantener el espacio de su proyecto en funcionamiento, es una buena idea **crear un entorno virtual y mantener sus dependencias aisladas**.

Puede usar el venv de Python y especificar la versión de Python y el nombre del entorno. En el siguiente ejemplo, uso Python3 y llamo al entorno venv. Una vez que haya creado el directorio, debe ejecutar el script actívalo dentro de él.

```
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ ls

najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ python3 -m venv venv

najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ ls
venv

najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ source venv/bin/activate
(venv)
```

*Imagen por autor*

### Resumen de comandos:

1. `python3 -m venv venv` → *crea* un entorno virtual
2. `source venv/bin/activate` → *activa* el entorno
3. `deactivate` → *desactiva* el entorno

En pocas palabras, los entornos virtuales le permiten:

1. **Mantenga las dependencias aisladas.** Esto evita situaciones en las que tiene proyectos que usan diferentes versiones de paquetes y desinstala/reinstala globalmente lo que necesita cada vez que necesita ejecutar un proyecto.
2. **Comparte tus dependencias con otras personas.**

Una vez que haya instalado todos los paquetes que necesita su proyecto, puede ejecutar:

```
pip freeze > requirements.txt
```

pip está "congelando" todos los paquetes/versiones que se están utilizando actualmente. Luego, está canalizando (es decir, guardando) la salida de a pip freeze través de un archivo txt.

Otras personas que usen su programa ejecutarán:

```
pip install -r requirements.txt
```

y poder instalar todos los paquetes y las versiones correctas de una sola vez. ¿Increíble verdad? 🚀

```
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ pip freeze
attrs==21.4.0
importlib-metadata==4.10.1
iniconfig==1.1.1
packaging==21.3
pluggy==1.0.0
py==1.11.0
pyparsing==3.0.7
pytest==7.0.0
tomli==2.0.1
typing-extensions==4.0.1
zipp==3.7.0
(venv)
```

*Imagen por autor*

## Consejo #2: Cree un directorio separado para las pruebas

Ha escuchado que crear pruebas para su código es una buena idea. Así que estás escribiendo código Python y tratando de probarlo con Pytest.

*Ejemplo:* Digamos que creas un archivo llamado greetings.pyy otro para escribir unas pruebas llamado test\_greetings.py.

En este ejemplo, usaré Pytest como paquete de prueba. Pytest tiene muchas convenciones de nomenclatura. Comenzar el nombre del archivo con test\_es uno de ellos. Si lo sigue, Pytest buscará automáticamente en el directorio actual y en los que están debajo de él los archivos que comienzan con test\_y los ejecuta.

Para crear los nuevos archivos, puede usar el touch comando en su terminal:

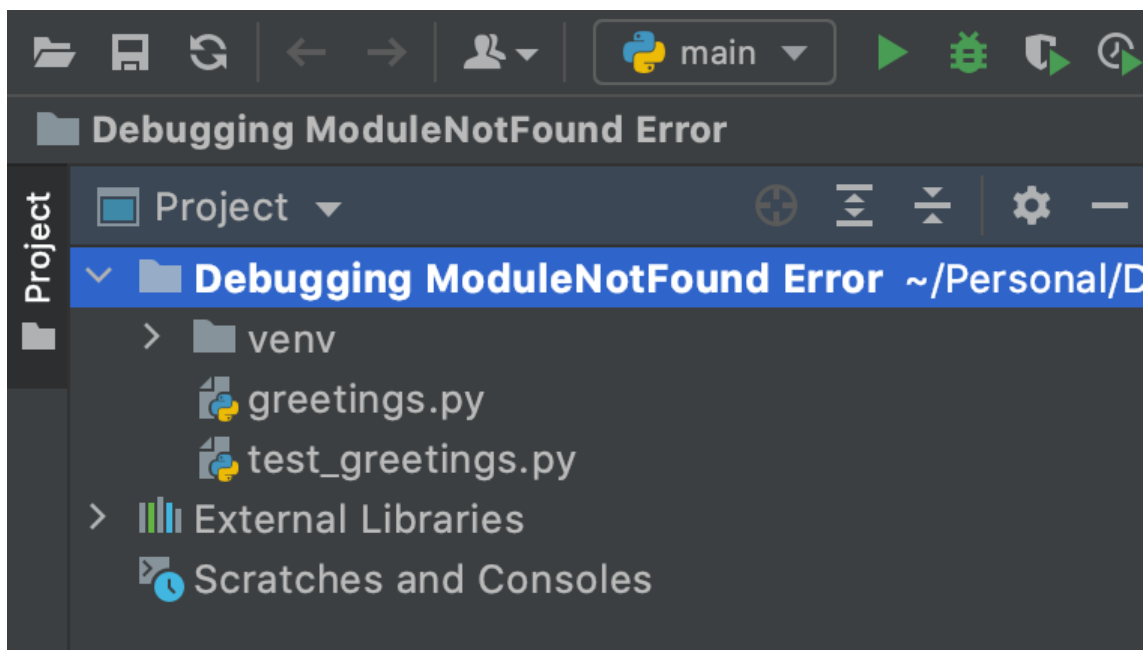
```

najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ touch greetings.py
(env)
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ ls
greetings.py      venv
(env)
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ touch test_greetings.py
(env)
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ ls
greetings.py      test_greetings.py      venv
(env)

```

*Imagen por autor*

En la parte superior derecha de PyCharm, debería ver algo como:



*Imagen por autor*

Abra greetings.pyy escriba una función simple:

En su terminal, puede ejecutar `pip install pytest` para instalar el `pytest` módulo y luego simplemente `pytest`. Dado que aún no ha definido ninguna prueba, `pytest` se ejecutará pero recopilará 0 elementos.

```
Terminal: Local x +
Enabled auto-complete for datacli in your shell: bash.
(env)
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ pytest
===== test session starts =====
platform darwin -- Python 3.7.2, pytest-7.0.0, pluggy-1.0.0
rootdir: /Users/najma/Personal/Debugging ModuleNotFound Error
collected 0 items

===== no tests ran in 0.00s =====
(env)
```

Imagen por autor

Abra test\_greetings.pyy escriba una función de prueba:

Si se da cuenta, también el nombre de la función debe comenzar con test\_, por ejemplo, test\_say\_hello. Esta es otra convención de nomenclatura de Pytest.

## Cómo depurar el error ModuleNotFound

Otra buena idea es recopilar todas sus pruebas en un directorio.

Sin embargo, si simplemente hace esto, cuando ejecute pytest, verá que ninguna de sus pruebas está funcionando. Si miras más de cerca, probablemente obtengas un ModuleNotFoundError.

Veamos qué sucedió y cómo solucionarlo.

```
===== ERRORS =====
ERROR collecting test/test_greetings.py
ImportError while importing test module '/Users/najma/Personal/Debugging ModuleNotFound Error/test/test_greetings.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/usr/local/adyen/python/lib/python3.7/importlib/__init__.py:127: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
test/test_greetings.py:1: in <module>
    from greetings import say_hello
E   ModuleNotFoundError: No module named 'greetings'

===== short test summary info =====
ERROR test/test_greetings.py
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
===== 1 error in 0.08s =====
```

Imagen por autor

Pytest está intentando importar el greetingsmódulo pero está fallando. La forma más sencilla de arreglar esto es pretender que el testdirectorio es un paquete (es decir, una colección de módulos). En el directorio, cree un archivo llamado\_\_init\_\_.py

```
(venv)
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ cd test
(venv)
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error/test
$ ls
__pycache__          test_greetings.py
(venv)
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error/test
$ touch __init__.py
(venv)
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error/test
$
```

*Imagen por autor*

Este es un archivo completamente vacío:

```
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error/test
$ ls -l
total 8
-rw-r--r--  1 najma  staff   0 Feb  9 18:41 __init__.py
drwxr-xr-x  3 najma  staff  96 Feb  9 18:40 __pycache__
-rw-r--r--  1 najma  staff 126 Feb  9 18:39 test_greetings.py
(venv)
```

*Imagen por autor*

Sin embargo, el simple hecho de estar ahí hace que tu prueba vuelva a funcionar.

```
najma at mb-najma in ~/Personal/Debugging ModuleNotFound Error
$ pytest
===== test session starts =====
platform darwin -- Python 3.7.2, pytest-7.0.0, pluggy-1.0.0
rootdir: /Users/najma/Personal/Debugging ModuleNotFound Error
collected 1 item

test/test_greetings.py .
===== 1 passed in 0.01s =====
```

Imagen por autor

¡Voila! 🍷

**Consejo #3: Crea diferentes directorios de contenido**

Ahora que tiene la idea de crear un `__init__.py` archivo, puede crear tantos directorios como desee.

Por ejemplo:

Básicamente:

*`__init__.py` se utiliza para marcar directorios como directorios de paquetes de Python*

Si elimina el `__init__.py`, Python ya no buscará submódulos dentro de ese directorio. Por lo tanto, si intenta importar el módulo en otro lugar, fallará.

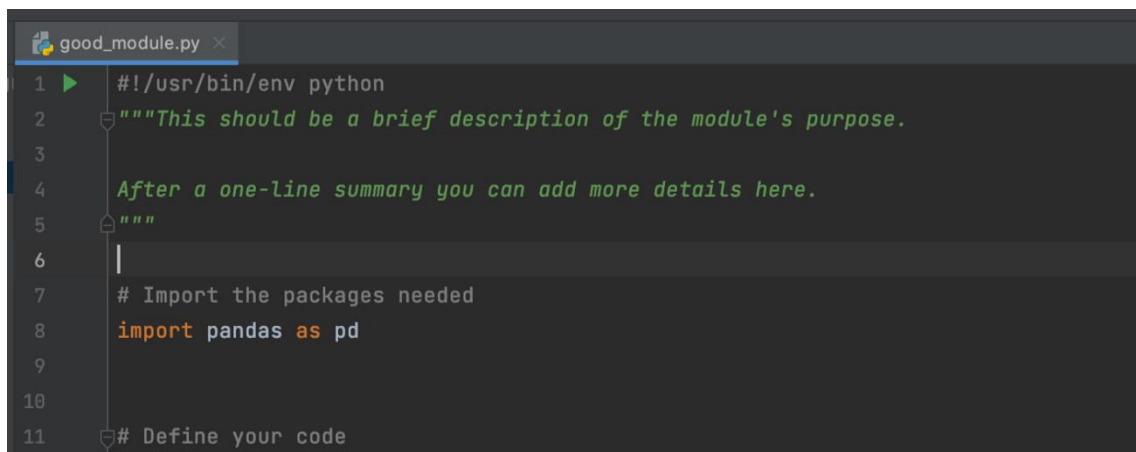
#### Consejo #4: Documente su código

Documentar su código es muy importante tanto para usted en el "futuro" como para otras personas que lean su proyecto. Algunos programadores incluso dicen:

*Si es difícil documentar su código, considere cambiar el diseño.*

Hay 3 cosas críticas que debe tener en cuenta:

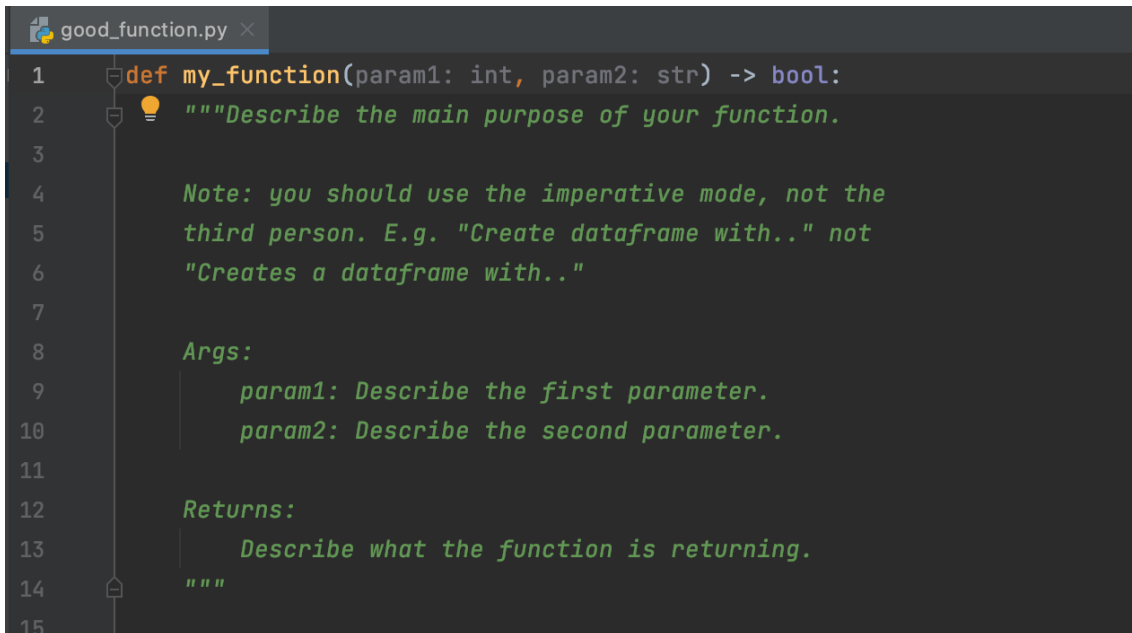
1. Agregue una cadena de documentación con una descripción al comienzo de cada archivo



```
1  #!/usr/bin/env python
2  """This should be a brief description of the module's purpose.
3
4  After a one-line summary you can add more details here.
5  """
6
7  # Import the packages needed
8  import pandas as pd
9
10
11 # Define your code
```

*Imagen por autor*

2. Agregue una cadena de documentación a cada función y clase



```

1  def my_function(param1: int, param2: str) -> bool:
2      """Describe the main purpose of your function.
3
4      Note: you should use the imperative mode, not the
5      third person. E.g. "Create dataframe with.." not
6      "Creates a dataframe with.."
7
8      Args:
9          param1: Describe the first parameter.
10         param2: Describe the second parameter.
11
12     Returns:
13         Describe what the function is returning.
14     """
15

```

*Imagen por autor*

3. Use sugerencias de tipo cada vez que defina una función o una clase



```

1
2  def greeting(name: str) -> str:
3      return 'Hello ' + name
4

```

*Imagen por autor*

Las sugerencias de tipo (1) hacen que las funciones y las clases sean más legibles, (2) le permiten omitir la especificación del tipo de parámetros en la documentación y (3) puede usarlas para verificar automáticamente su código con mypy. 🐍

**Hay muchas guías de estilo para escribir una buena documentación. Personalmente, me gusta Google Style Python Docstring .**

**#5: Use GitHub para el control de versiones**

Si está trabajando con otra persona, Git + GitHub es esencial hacer revisiones de código y poder evitar conflictos de fusión.

Si está trabajando solo, aún son útiles para guardar su trabajo y poder revertir los cambios y "retroceder en el tiempo".



# Organización

Empezaremos creando una carpeta **Gestor de clientes** con un fichero **requirements.txt** y otro **README.md** ambos vacíos, además de una carpeta llamada **gestor/** para contener los scripts de nuestro proyecto.

Esta organización es una buena práctica para ahorrar problemas en el futuro ya que permite añadir todo lo que necesitemos en la raíz externa sin molestar al código fuente, como por ejemplo documentación, pruebas, configuraciones, etc. Es la clave para mantener un proyecto organizado y extensible.

El programa lo vamos a desarrollar en 4 ficheros básicos:

- `run.py`: El script principal que lo pondrá todo en marcha.
- `menu.py`: La interfaz que mostrará por la terminal un menú.
- `database.py`: Encargado de manejar la gestión de los datos.
- `helpers.py`: Contendrá funciones auxiliares de uso general.

A medida que los necesitemos iremos añadiendo otros ficheros.

## Mock database

Empecemos con la base de datos del backend, que por ahora contendrá mock objects (objetos para pruebas), más adelante los refactorizaremos cambiándolos por persistencia en un fichero.

Primero la clase `Cliente` para manejar un cliente:

`gestor/database.py`

```
class Cliente:

    def __init__(self, dni, nombre, apellido):
        self.dni = dni
        self.nombre = nombre
        self.apellido = apellido

    def __str__(self):
        return f"({self.dni}) {self.nombre} {self.apellido}"
```

Y una clase `Cientes` para utilizar como fuente de datos y que implementará las funcionalidades de buscar, crear, actualizar y borrar clientes. Es una clase especial con funciones estáticas, eso significa que no se utilizará para crear instancias, sino que la usaremos directamente como origen único de la información:

```
class Cientes:

    # Lista de clientes
    lista = []

    @staticmethod
    def buscar(dni):
        for cliente in Cientes.lista:
            if cliente.dni == dni:
                return cliente
```

```

@staticmethod
def crear(dni, nombre, apellido):
    cliente = Cliente(dni, nombre, apellido)
    Clientes.lista.append(cliente)
    return cliente

@staticmethod
def modificar(dni, nombre, apellido):
    for i, cliente in enumerate(Clientes.lista):
        if cliente.dni == dni:
            Clientes.lista[i].nombre = nombre
            Clientes.lista[i].apellido = apellido
            return Clientes.lista[i]

@staticmethod
def borrar(dni):
    for i, cliente in enumerate(Clientes.lista):
        if cliente.dni == dni:
            cliente = Clientes.lista.pop(i)
            return cliente

```

¿Cómo probamos si todo funciona? Pues con unas pruebas unitarias, lo hacemos en la siguiente lección.

## Pruebas unitarias

Para añadir las pruebas crearemos un directorio `tests`, en plural, con un `__init__.py` para indicar que es un paquete, así Python podrá autodescubrir los ficheros de pruebas.

Nosotros crearemos uno llamado `test_database.py` con las siguientes pruebas unitarias:

gestor/tests/test\_database.py

```

import copy
import unittest
import database as db

```

```

class TestDatabase(unittest.TestCase):

```

```

    def setUp(self):
        # Se ejecuta antes de cada prueba
        db.Clientes.lista = [
            db.Cliente('15J', 'Marta', 'Pérez'),
            db.Cliente('48H', 'Manolo', 'López'),
            db.Cliente('28Z', 'Ana', 'García')
        ]

    def test_buscar_cliente(self):
        cliente_existente = db.Clientes.buscar('15J')
        cliente_no_existente = db.Clientes.buscar('99X')
        self.assertIsNotNone(cliente_existente)
        self.assertIsNone(cliente_no_existente)

```

```

    def test_crear_cliente(self):
        nuevo_cliente = db.Clientes.crear('39X', 'Héctor', 'Costa')
        self.assertEqual(len(db.Clientes.lista), 4)
        self.assertEqual(nuevo_cliente.dni, '39X')

```

```

        self.assertEqual(nuevo_cliente.nombre, 'Héctor')
        self.assertEqual(nuevo_cliente.apellido, 'Costa')

    def test_modificar_cliente(self):
        cliente_a_modificar = copy.copy(db.Clientes.buscar('28Z'))
        cliente_modificado = db.Clientes.modificar('28Z', 'Mariana',
        'Pérez')
        self.assertEqual(cliente_a_modificar.nombre, 'Ana')
        self.assertEqual(cliente_modificado.nombre, 'Mariana')

    def test_borrar_cliente(self):
        cliente_borrado = db.Clientes.borrar('48H')
        cliente_rebuscado = db.Clientes.buscar('48H')
        self.assertNotEqual(cliente_borrado, cliente_rebuscado)

if __name__ == '__main__':
    unittest.main()

```

Para probar las pruebas necesitamos instalar el paquete `pytest`:

```
pip install pytest
```

Ahora dejamos que `pytest` autodescubra las pruebas y las ejecute haciendo:

```
pytest -v
```

Perfecto, esto nos dará la seguridad de que el módulo `database` está funcionando.

## Estructurando el menú

Vamos a construir el menú, en lugar de añadirlo en el `run.py` vamos a hacerlo en su propio módulo. Recordad, cuanto más separado y organizado esté el código más reutilizable y extensible será el programa:

```

gestor/menu.py
import os

def iniciar():
    while True:
        os.system('clear') # cls en Windows

        print("=====")
        print("  BIENVENIDO AL Manager ")
        print("=====")
        print("[1] Listar clientes      ")
        print("[2] Buscar cliente       ")
        print("[3] Añadir cliente       ")
        print("[4] Modificar cliente     ")
        print("[5] Borrar cliente       ")
        print("[6] Cerrar el Manager    ")
        print("=====")

        opcion = input("> ")
        os.system('clear') # cls en Windows

        if opcion == '1':
            print("Listando los clientes...\n")
        if opcion == '2':
            print("Buscando un cliente...\n")
        if opcion == '3':

```

```

        print("Añadiendo un cliente...\n")
    if opcion == '4':
        print("Modificando un cliente...\n")
    if opcion == '5':
        print("Borrando un cliente...\n")
    if opcion == '6':
        print("Saliendo...\n")
        break

    input("\nPresiona ENTER para continuar...")

```

Para probarlo podemos añadirlo al fichero `run.py` impórtándolo cómodamente:

```

gestor/run.py
import menu

if __name__ == "__main__":
    menu.iniciar()

```

## Funciones auxiliares

Es un buen momento para crear unas funciones de ayuda.

La primera será una versión mejorada para limpiar la terminal porque la que tenemos no detecta automáticamente el sistema operativo y uno de los puntos fuertes de Python es que es multiplataforma:

```

gestor/helpers.py
import os
import platform

def limpiar_pantalla():
    os.system('cls') if platform.system() == "Windows" else
os.system('clear')

```

Ahora podemos cargar el módulo cómodamente y ejecutar la función en el menú en lugar de llamar directamente al módulo `os`:

```

gestor/menu.py
import helpers

helpers.limpiar_pantalla()

```

La segunda función de ayuda será para leer un texto cómodamente:

```

gestor/helpers.py
def leer_texto(longitud_min=0, longitud_max=100, mensaje=None):
    print(mensaje) if mensaje else None
    while True:
        texto = input("> ")
        if len(texto) >= longitud_min and len(texto) <= longitud_max:
            return texto

```

Esta función la utilizaremos para leer los campos dni, nombre y apellido del cliente que vayamos a gestionar.

# Implementando el menú

Por fin ha llegado el momento de conectar la base de datos y el menú, tendremos que envolver las funcionalidades de nuestra `database` en las diferentes opciones del menú:

```
gestor/menu.py
if opcion == '1':
    print("Listando los clientes...\n")
    for cliente in db.Clientes.lista:
        print(cliente)

if opcion == '2':
    print("Buscando un cliente...\n")
    dni = helpers.leer_texto(3, 3, "DNI (2 ints y 1 char)").upper()
    cliente = db.Clientes.buscar(dni)
    print(cliente) if cliente else print("Cliente no encontrado.")

if opcion == '3':
    print("Añadiendo un cliente...\n")
    dni = helpers.leer_texto(
        3, 3, "DNI (2 ints y 1 char)").upper()
    nombre = helpers.leer_texto(
        2, 30, "Nombre (de 2 a 30 chars)").capitalize()
    apellido = helpers.leer_texto(
        2, 30, "Apellido (de 2 a 30 chars)").capitalize()
    db.Clientes.crear(dni, nombre, apellido)
    print("Cliente añadido correctamente.")

if opcion == '4':
    print("Modificando un cliente...\n")
    dni = helpers.leer_texto(3, 3, "DNI (2 ints y 1 char)").upper()
    cliente = db.Clientes.buscar(dni)
    if cliente:
        nombre = helpers.leer_texto(
            2, 30, f"Nombre (de 2 a 30 chars)
[{cliente.nombre}]).capitalize()
        apellido = helpers.leer_texto(
            2, 30, f"Apellido (de 2 a 30 chars)
[{cliente.apellido}]).capitalize()
        db.Clientes.modificar(cliente.dni, nombre, apellido)
        print("Cliente modificado correctamente.")
    else:
        print("Cliente no encontrado.")

if opcion == '5':
    print("Borrando un cliente...\n")
    dni = helpers.leer_texto(3, 3, "DNI (2 ints y 1 char)").upper()
    print("Cliente borrado correctamente.") if db.Clientes.borrar(
        dni) else print("Cliente no encontrado.")

if opcion == '6':
    print("Saliendo...\n")
    break

input("\nPresiona ENTER para continuar...")
```

## Validación del campo DNI

Hay dos cosas que tenemos comprobar para el campo DNI antes de añadir un nuevo cliente. La primera es que el DNI sea válido, es decir, cumpla un formato de dos números y una letra. La segunda es que no haya ningún otro cliente con ese DNI, así que pongámonos a ello.

Vamos a crear la función auxiliar en `helpers.py`:

```
gestor/helpers.py
import re

def dni_valido(dni, lista):
    if not re.match('[0-9]{2}[A-Z]$', dni):
        print("DNI incorrecto, debe cumplir el formato.")
        return False
    for cliente in lista:
        if cliente.dni == dni:
            print("DNI utilizado por otro cliente.")
            return False
    return True
```

Vamos a añadir una prueba para nuestra nueva función:

```
gestor/tests/test_database.py
import helpers

def test_dni_valido(self):
    self.assertTrue(helpers.dni_valido('00A', db.Clientes.lista))
    self.assertFalse(helpers.dni_valido('23223S', db.Clientes.lista))
    self.assertFalse(helpers.dni_valido('F35', db.Clientes.lista))
    self.assertFalse(helpers.dni_valido('48H', db.Clientes.lista))
```

Las probamos para ver si todo es correcto:

```
pytest -v
```

Con la seguridad de que todo está bien la utilizaremos en un bucle para leer el dni hasta que sea válido:

```
gestor/menu.py
if opcion == '3':
    print("Añadiendo un cliente...\n")

    # Comprobación de DNI válido
    while 1:
        dni = helpers.leer_texto(3, 3, "DNI (2 ints y 1
char)").upper()
        if helpers.dni_valido(dni, db.Clientes.lista):
            break

    nombre = helpers.leer_texto(
        2, 30, "Nombre (de 2 a 30 chars)").capitalize()
    apellido = helpers.leer_texto(
        2, 30, "Apellido (de 2 a 30 chars)").capitalize()
    db.Clientes.crear(dni, nombre, apellido)
```

## Persistencia en fichero CSV

En esta lección vamos a implementar el módulo CSV, que ya vimos anteriormente en el curso, para almacenar los clientes del gestor.

Cuando el programa se ponga en marcha, la clase `database.Clientes` cargará los clientes de un fichero y los irá serializando a medida que se realicen cambios.

Para que todo funcione correctamente, trasladaremos los mock objects al fichero CSV desde el principio:

```
gestor/clientes.csv
15J;Marta;Perez
48H;Manolo;Lopez
28Z;Ana;Garcia
```

Cargaremos los clientes del fichero en la lista de `Clientes`:

```
class Clientes:
    # Creamos la lista y cargamos los clientes en memoria
    lista = []
    with open("clientes.csv", newline="\n") as fichero:
        reader = csv.reader(fichero, delimiter=";")
        for dni, nombre, apellido in reader:
            cliente = Cliente(dni, nombre, apellido)
            lista.append(cliente)
```

Esta parte ya funcionará, ahora debemos implementar el método para guardar los `Clientes` de vuelta en el fichero después de modificarlos.

Podemos crear el método:

```
@staticmethod
def guardar():
    with open("clientes.csv", "w", newline="\n") as fichero:
        writer = csv.writer(fichero, delimiter=";")
        for c in Clientes.lista:
            writer.writerow((c.dni, c.nombre, c.apellido))
```

Lo llamamos después de crear, modificar o borrar un cliente:

```
@staticmethod
def crear(dni, nombre, apellido):
    cliente = Cliente(dni, nombre, apellido)
    Clientes.lista.append(cliente)
    Clientes.guardar() # new
    return cliente

@staticmethod
def modificar(dni, nombre, apellido):
    for i, cliente in enumerate(Clientes.lista):
        if cliente.dni == dni:
            Clientes.lista[i].nombre = nombre
            Clientes.lista[i].apellido = apellido
            Clientes.guardar() # new
            return Clientes.lista[i]

@staticmethod
def borrar(dni):
    for i, cliente in enumerate(Clientes.lista):
        if cliente.dni == dni:
            cliente = Clientes.lista.pop(i)
            Clientes.guardar() # new
            return cliente
```

Listo, el programa ya debería sincronizar automáticamente los cambios realizados en la memoria también en el fichero CSV.

Sin embargo los tests de database.py...  
`pytest -v`

Nos están modificando la información original del fichero y esto no puede ser.

Los arreglamos en la próxima lección.

## Arreglando las pruebas unitarias

El problema que tenemos es que las pruebas modifican la base de datos del fichero `clientes.csv`. Esto es un fallo importante porque perderemos los datos cada vez que ejecutemos las pruebas, tenemos que conseguir manejar dos ficheros CSV independientes, uno para las pruebas y otro para el programa.

Para ello vamos a crear un fichero de configuración con una constante que contendrá la localización del CSV:

```
gestor/config.py
import sys
DATABASE_PATH = 'clientes.csv'
```

Simplemente substituiremos la ruta en crudo por esta constante:

```
gestor/database.py
class Clientes:
    lista = []
    with open(config.DATABASE_PATH, newline="\n") as fichero: # ...

    @staticmethod
    def guardar():
        with open(config.DATABASE_PATH, "w", newline="\n") as fichero:
# ...
```

El truco consistirá en definir una constante diferente si trabajamos con las pruebas y eso podemos saberlo analizando el primer argumento del script:

```
gestor/config.py
import sys
DATABASE_PATH = 'clientes.csv'

if 'pytest' in sys.argv[0]:
    DATABASE_PATH = 'tests/clientes_test.csv'
```

Con esto las pruebas deberían ir a buscar ese nuevo fichero, que no existirá:

```
pytest -v
```

Solo tenemos que crearlo:

```
gestor/tests/clientes_tests.csv
```



Y ya tendremos dos ficheros diferentes.

Para redondearlo podemos añadir una prueba para confirmar que el contenido del fichero cambia al realizar modificaciones y asegurarnos de que la persistencia funciona correctamente:

```
import csv

def test_escritura_csv(self):
    db.Clientes.borrar('48H')
    db.Clientes.borrar('15J')
    db.Clientes.modificar('28Z', 'Mariana', 'Pérez')

    dni, nombre, apellido = None, None, None
    with open(config.DATABASE_PATH, newline="\n") as fichero:
        reader = csv.reader(fichero, delimiter=";")
        dni, nombre, apellido = next(reader) # Primera línea del
iterador

    self.assertEqual(dni, '28Z')
    self.assertEqual(nombre, 'Mariana')
    self.assertEqual(apellido, 'Pérez')
```

Si etodo está correcto las pruebas deberían pasar:

```
pytest -v
```

## GUI (1): Ventana principal

Durante las próximas lecciones vamos a ver la ventaja de haber programado modularmente el programa.

Al tener bien separado el menú de la base de datos podemos substituir la terminal por una interfaz gráfica sin modificar el código base. De hecho no tenemos ni que prescindir de él, podemos proveerlo como alternativa para un entorno sin gráficos, ya veréis que curioso queda.

Para hacer más sencilla la gestión de la ventana y subventanas del programa, vamos a desarrollarlo utilizando clases y objetos, por lo que partiremos de un programa básico de Tkinter manejado en una clase que a mi me gusta

llamar MainWindow:

gestor/ui.py

```
from tkinter import *

class MainWindow(Tk):
    def __init__(self):
        super().__init__()
        self.title('Gestor de clientes')
        self.build()

    def build(self):
        button = Button(self.root, text="Hola", command=self.hola)
        button.pack()

    def hola(self):
        print(";Hola mundo!")
```

```
if __name__ == "__main__":
    app = MainWindow()
    app.mainloop()
```

## GUI (2): Mixin para centrar widgets

Antes de continuar vamos a tomarnos un momento para invertir en calidad de vida y es que cuando se abre una ventana de tkinter no se posiciona automáticamente en el centro de la pantalla, por lo menos en MAC OS, que es el sistema que utilizo actualmente.

Os voy a enseñar una forma genial de solucionar este problema mediante el uso de un mixin. Un **mixin** es una clase que contiene una o varias definiciones. Por sí mismos los mixins no tienen utilidad, pero al heredarlos en otra clase conseguiremos su funcionalidad y eso es precisamente lo que vamos a hacer, un mixin para centrar un widget en la pantalla (en nuestro caso la ventana principal):

```
class CenterWidgetMixin:
    def center(self,):
        self.update()
        w = self.winfo_width()
        h = self.winfo_height()
        ws = self.winfo_screenwidth()
        hs = self.winfo_screenheight()
        x = int((ws/2) - (w/2))
        y = int((hs/2) - (h/2))
        self.geometry(f"{w}x{h}+{x}+{y}")
```

Si heredamos de este mixin conseguiremos el método `center` que podemos utilizar al final del constructor para autocentrar la ventana:

```
class MainWindow(Tk, CenterWidgetMixin): # edited
    def __init__(self):
        super().__init__()
        self.title('Gestor de clientes')
        self.build()
        self.center() # new
```

## GUI (3): Widget Treeview

Mixins a parte, vamos a utilizar el método `build` para construir y configurar todos los widgets de la ventana principal. Si queremos comunicar alguno con otro método queramos comunicar con otros métodos lo haremos en atributos de clase.

Ahora, para mostrar los clientes podemos utilizar algo como una tabla. El widget extendido `Treeview` del subpaquete `ttk` sirve para eso así que vamos a utilizarlo:

```
from tkinter import ttk

def build(self):
    # Top Frame
    frame = Frame(self)
```

```

frame.pack()

# Treeview
treeview = ttk.Treeview(frame)
treeview['columns'] = ('DNI', 'Nombre', 'Apellido')
treeview.pack()

# Column format
treeview.column("#0", width=0, stretch=NO)
treeview.column("DNI", anchor=CENTER)
treeview.column("Nombre", anchor=CENTER)
treeview.column("Apellido", anchor=CENTER)

# Heading format
treeview.heading("#0", anchor=CENTER)
treeview.heading("DNI", text="DNI", anchor=CENTER)
treeview.heading("Nombre", text="Nombre", anchor=CENTER)
treeview.heading("Apellido", text="Apellido", anchor=CENTER)

# Pack
treeview.pack()

```

Por defecto el `Treeview` no tiene una barra de scroll, deberíamos crear una manualmente y configurarla por si en algún momento hay tantos registros que falta espacio verticalmente:

```

# Scrollbar
scrollbar = Scrollbar(frame) # new
scrollbar.pack(side=RIGHT, fill=Y) # new

# Treeview
treeview = ttk.Treeview(frame, yscrollcommand=scrollbar.set) # edited
treeview['columns'] = ('DNI', 'Nombre', 'Apellido')
treeview.pack()

```

Vamos a cargar los datos de la base de datos en el `treeview` justo antes de empaquetarlo:

```

import database as db

# Fill treeview data
for cliente in db.Clientes.lista:
    treeview.insert(
        parent='', index='end', iid=cliente.dni,
        values=(cliente.dni, cliente.nombre, cliente.apellido))

# Treeview repack with scrollbar
treeview.pack()

```

Vamos a añadir tres botones en un nuevo frame que nos permitan, crear, modificar y borrar registros:

```

# Bottom Frame
frame = Frame(self)
frame.pack(pady=20)

# Buttons
Button(frame, text="Crear", command=None).grid(row=1, column=0)
Button(frame, text="Modificar", command=None).grid(row=1, column=1)
Button(frame, text="Borrar", command=None).grid(row=1, column=2)

```

Para terminar la interfaz base, necesitamos exportar como atributo de clase el `treeview` ya que lo necesitaremos para realizar tareas como recuperar la información de la fila seleccionada:

```
# Export treeview to the class
self.treeview = treeview
```

## GUI (4): Diálogo de borrado

Ahora hay que programar los botones de acción, empezaremos por el de **borrar** ya es el más sencillo y lo implementaremos sobre un cuadro de diálogo por defecto:

```
from tkinter.messagebox import askokcancel, WARNING

def delete(self):
    cliente = self.treeview.focus()
    if cliente:
        campos = self.treeview.item(cliente, 'values')
        confirmar = askokcancel(
            title='Confirmación',
            message=f'¿Borrar a {campos[1]} {campos[2]}?',
            icon=WARNING)
        if confirmar:
            # remove the row
            self.treeview.delete(cliente)
```

Lo llamamos al presionar el botón:

```
Button(frame, text="Borrar", command=self.delete).grid(row=1,
column=2)
```

## GUI (5): Subventana de creación

En cuanto a las opciones de `crear` y `modificar` son más complejas, tendremos que programar nuestras propias ventanas secundarias con sus campos de texto, botones y validaciones.

Este es el diseño de la subventana de creación:

```
class CreateClientWindow(Toplevel, CenterWidgetMixin):
    def __init__(self, parent):
        super().__init__(parent)
        self.title('Crear cliente')
        self.build()
        self.center()
        # Obligar al usuario a interactuar con la subventana
        self.transient(parent)
        self.grab_set()

    def build(self):
        # Top frame
        frame = Frame(self)
        frame.pack(padx=20, pady=10)

        # Labels
```

```

        Label(frame, text="DNI (2 ints y 1 upper char)").grid(row=0,
column=0)
        Label(frame, text="Nombre (2 a 30 chars)").grid(row=0,
column=1)
        Label(frame, text="Apellido (2 a 30 chars)").grid(row=0,
column=2)

        # Entries
        dni = Entry(frame)
        dni.grid(row=1, column=0)
        nombre = Entry(frame)
        nombre.grid(row=1, column=1)
        apellido = Entry(frame)
        apellido.grid(row=1, column=2)

        # Bottom frame
        frame = Frame(self)
        frame.pack(pady=10)

        # Buttons
        crear = Button(frame, text="Crear",
command=self.create_client)
        crear.configure(state=DISABLED)
        crear.grid(row=0, column=0)
        Button(frame, text="Cancelar", command=self.close).grid(row=0,
column=1)

    def create_client(self):
        pass

    def close(self):
        self.destroy()
        self.update()

```

Crearemos una instancia desde un nuevo método en la ventana principal:

```

Button(frame, text="Crear",
command=self.create_client_window).grid(row=1, column=0)

def create_client_window(self):
    CreateClientWindow(self)

```

## GUI (6): Validación en tiempo real

La interfaz de la subventana de creación está lista, ahora vamos a configurar las validaciones en los campos antes de recuperar la información y añadir el nuevo cliente a la tabla:

```

# Entries and validations
dni = Entry(frame)
dni.grid(row=1, column=0)
dni.bind("<KeyRelease>", lambda ev: self.validate(ev, 0))
nombre = Entry(frame)
nombre.grid(row=1, column=1)
nombre.bind("<KeyRelease>", lambda ev: self.validate(ev, 1))
apellido = Entry(frame)
apellido.grid(row=1, column=2)
apellido.bind("<KeyRelease>", lambda ev: self.validate(ev, 2))

```

Los métodos bindeados con las validaciones quedarán:

```
def validate(self, event, index):
    valor = event.widget.get()
    # Validar como dni si es el primer campo o textual para los otros
dos
    valido = helpers.dni_valido(valor, db.Clientes.lista) if index ==
0 \
        else (valor.isalpha() and len(valor) >= 2 and len(valor) <=
30)
    event.widget.configure({"bg": "Green" if valido else "Red"})
```

## GUI (7): Manejando el botón crear

Para controlar las validaciones y activar o desactivar el botón de creación utilizaremos una lista con booleanos que cambiaremos en tiempo real en las validaciones:

```
# Create button activation
self.validaciones = [0, 0, 0] # False, False, False

# Class exports
self.crear = crear

def validate(self, event, index):
    valor = event.widget.get()
    # Validar el dni si es el primer campo o textual para los otros
dos
    valido = helpers.dni_valido(valor, db.Clientes.lista) if index ==
0 \
        else (valor.isalpha() and len(valor) >= 2 and len(valor) <=
30)
    event.widget.configure({"bg": "Green" if valido else "Red"})
    # Cambiar estado del botón en base a las validaciones
    self.validaciones[index] = valido
    self.crear.config(state=NORMAL if self.validaciones == [1, 1, 1]
                        else DISABLED)
```

Finalmente para crear el registro en la tabla haremos referencia al `treeview` de la ventana principal mediante el accesor `master`, pero antes necesitaremos exportar los campos para poder acceder a los valores entre métodos de la clase:

```
# Class exports
self.crear = crear
self.dni = dni
self.nombre = nombre
self.apellido = apellido
```

Con ellos podemos crear el cliente en el `treeview` y cerrar la subventana:

```
def create_client(self):
    self.master.treeview.insert(
        parent='', index='end', iid=self.dni.get(),
        values=(self.dni.get(), self.nombre.get(),
self.apellido.get()))
    self.close()
```

Listo, vamos a por la subventana de modificación.

## GUI (8): Subventana de modificación

Para modificar un cliente vamos a reutilizar en gran parte lo que tenemos en la subventana de creación, la ventaja es que ahora no necesitamos aplicar validación al campo DNI porque éste no es editable, por lo que lo desactivaremos.

La implementación de esta subventana es mayormente una copia recortada de la de creación, con la peculiaridad de que al cargarla buscaremos el cliente seleccionado en la `treeview` y escribiremos los valores inicialmente en los campos.

Además esta vez dejaremos activado el botón de `Actualizar` por defecto:

```
class EditClientWindow(Toplevel, CenterWidgetMixin):
    def __init__(self, parent):
        super().__init__(parent)
        self.title('Actualizar cliente')
        self.build()
        self.center()
        # Obligar al usuario a interactuar con la subventana
        self.transient(parent)
        self.grab_set()

    def build(self):
        # Top frame
        frame = Frame(self)
        frame.pack(padx=20, pady=10)

        # Labels
        Label(frame, text="DNI (no editable)").grid(row=0, column=0)
        Label(frame, text="Nombre (2 a 30 chars)").grid(row=0,
column=1)
        Label(frame, text="Apellido (2 a 30 chars)").grid(row=0,
column=2)

        # Entries
        dni = Entry(frame)
        dni.grid(row=1, column=0)
        nombre = Entry(frame)
        nombre.grid(row=1, column=1)
        nombre.bind("<KeyRelease>", lambda ev: self.validate(ev, 0))
        apellido = Entry(frame)
        apellido.grid(row=1, column=2)
        apellido.bind("<KeyRelease>", lambda ev: self.validate(ev, 1))

        # Set entries initial values
        cliente = self.master.treeview.focus()
        campos = self.master.treeview.item(cliente, 'values')
        dni.insert(0, campos[0])
        dni.config(state=DISABLED)
        nombre.insert(0, campos[1])
        apellido.insert(0, campos[2])

        # Bottom frame
        frame = Frame(self)
        frame.pack(pady=10)

        # Buttons
```

```

        actualizar = Button(frame, text="Actualizar",
command=self.update_client)
        actualizar.grid(row=0, column=0)
        Button(frame, text="Cancelar", command=self.close).grid(row=0,
column=1)

    # Update button activation
    self.validaciones = [1, 1] # True, True

    # Class exports
    self.actualizar = actualizar
    self.dni = dni
    self.nombre = nombre
    self.apellido = apellido

    def validate(self, event, index):
        valor = event.widget.get()
        valido = (valor.isalpha() and len(valor) >= 2 and len(valor)
<= 30)
        event.widget.configure({"bg": "Green" if valido else "Red"})
        # Cambiar estado del botón en base a las validaciones
        self.validaciones[index] = valido
        self.actualizar.config(state=NORMAL if self.validaciones ==
[1, 1] else DISABLED)

    def update_client(self):
        pass

    def close(self):
        self.destroy()
        self.update()

```

Actualizar los valores en la función `update_client` es tan fácil como hacer referencia a la fila activa del `treeview` y sobreescribir sus campos:

```

def update_client(self):
    cliente = self.master.treeview.focus()
    # Sobreescribimos los datos de la fila seleccionada
    self.master.treeview.item(
        cliente, values=(self.dni.get(), self.nombre.get(),
self.apellido.get()))
    self.close()

```

Perfecto, lo tenemos todo preparado a falta de sincronizar los cambios de la tabla en el fichero.

## GUI (9): Sincronización de datos

Para volcar los datos de la tabla al fichero CSV, solo tenemos que enlazar los momentos de creación, actualización y borrado a los métodos de la clase `Cientes` de nuestro módulo `database`. Es decir, a parte de modificar los cambios en la `treeview` visualmente, también haremos lo propio en la clase `db.Cientes`.

El **borrado**:

```

def delete(self):
    cliente = self.treeview.focus()

```



```

    if cliente:
        campos = self.treeview.item(cliente, 'values')
        confirmar = askokcancel(
            title='Confirmación', message=f'¿Borrar a {campos[1]}
{campos[2]}?', icon=WARNING)
        if confirmar:
            self.treeview.delete(cliente)
            # !!! Borrar también en el fichero
            db.Clientes.borrar(campos[0])

```

### La creación:

```

def create_client(self):
    self.master.treeview.insert(
        parent='', index='end', iid=self.dni.get(),
        values=(self.dni.get(), self.nombre.get(),
self.apellido.get()))
    # !!! Crear también en el fichero
    db.Clientes.crear(self.dni.get(), self.nombre.get(),
self.apellido.get())
    self.close()

```

### Y la actualización:

```

def update_client(self):
    cliente = self.master.treeview.focus()
    # Sobreescibir los datos
    self.master.treeview.item(
        cliente, values=(self.dni.get(), self.nombre.get(),
self.apellido.get()))
    # !!! Modificar también en el fichero
    db.Clientes.modificar(self.dni.get(), self.nombre.get(),
self.apellido.get())
    self.close()

```

Con esto hemos terminado la interfaz pero todavía nos falta un pequeño detalle...

## GUI (10): Modo terminal e interfaz

Cuando empezamos a crear la interfaz os comenté que no era necesario descartar el programa para la terminal, sino que podíamos implementar ambos.

Para lograrlo sería tan fácil como pasar un argumento al ejecutar el script:

```

import sys
import menu
import ui

if __name__ == "__main__":
    # Si pasamos un argumento -t lanzamos el modo terminal
    if len(sys.argv) > 1 and sys.argv[1] == "-t":
        menu.iniciar()
    # En cualquier otro caso lanzamos el modo gráfico
    else:
        app = ui.MainWindow()
        app.mainloop()

```

Listo, ahora si ejecutamo el script con `-t` lanzaremos el modo terminal:

```
python3 run.py -t
```

Y si lo ejecutamos normalmente, el modo gráfico:

```
python3 run.py
```

Con esto acabamos, espero que os haya gustado y hayáis aprendido mucho.