

Trabajo Práctico Entrega Final

Python Nivel Intermedio

Integrantes:

- Cairola, Sebastián
- De Bonis, Christian
- Ibarra, Yoel
- Medina, Ivan
- Rodriguez, Leonel
- Samudio, Leandro

Requisitos de la entrega:

- La funcionalidad de interacción con la base de datos y validación de campos debe ubicarse en módulos aparte.
- La app debe realizarse según el paradigma de POO
- Se debe implementar el patrón MVC
- Se debe agregar el trabajo con excepciones

Para llevar a cabo los requisitos de esta entrega, decidimos modificar la estructura de directorios que usamos para la entrega pasada.

La nueva estructura quedó de esta manera:

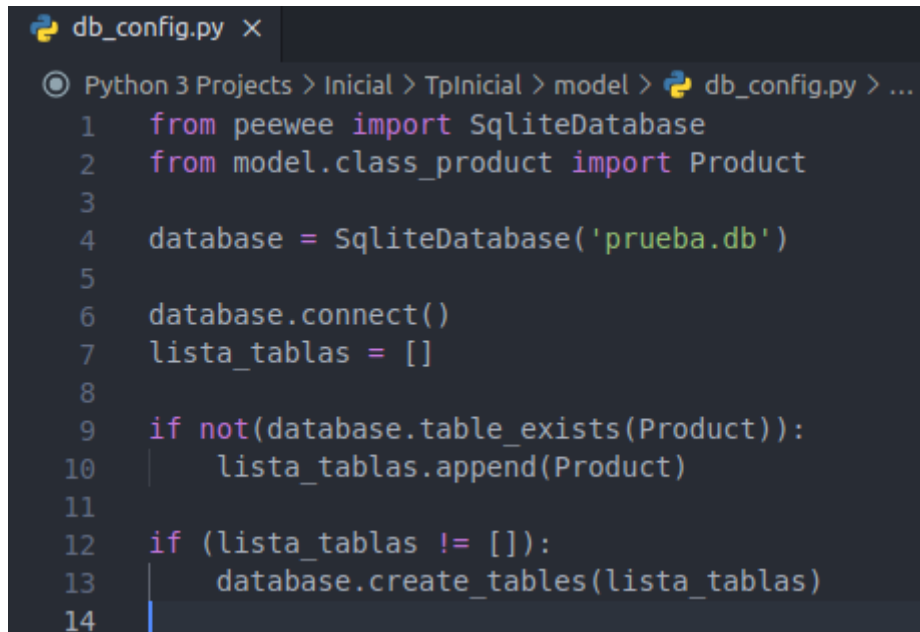
```
– controller
    |– controller.py
– model
    |– base_model.py
    |– class_product.py
    |– db_config.py
– view
    |– view.py
– utils
    |– validator.py
– inicio.py
```

A continuación pasaremos a detallar como fué que resolvimos cada uno de los requerimientos.

La funcionalidad de interacción con la base de datos y validación de campos debe ubicarse en módulos aparte

Para este requisito hemos creado un módulo para la interacción con la base de datos (model/db_config.py) y otro para la validación de los campos (utils/validator.py)

db_config.py



```
db_config.py x
Python 3 Projects > Inicial > TpInicial > model > db_config.py > ...
1  from peewee import SqliteDatabase
2  from model.class_product import Product
3
4  database = SqliteDatabase('prueba.db')
5
6  database.connect()
7  lista_tablas = []
8
9  if not(database.table_exists(Product)):
10     lista_tablas.append(Product)
11
12  if (lista_tablas != []):
13     database.create_tables(lista_tablas)
14
```

En la línea 1 importamos la clase SqliteDatabase del módulo peewee.

En la línea 2 importamos la clase Product del módulo class_product.

En la línea 4 creamos una instancia de SqliteDatabase pasándole al constructor el nombre del archivo de la base de datos como parámetro. Esta instancia es referenciada por la variable database.

En la línea 6 iniciamos la conexión con la base de datos a través del método connect().

En la línea 7 creamos una variable (lista_tablas) que almacena una lista vacía.

En la línea 9 hacemos una condición para agregar la clase Product a la lista lista_tablas, en caso de que esta no se encuentre presente.

En la línea 12 hacemos una condición para crear las tablas a partir de la lista lista_tablas, en caso de que esta no se encuentre vacía.

validador.py

```
validator.py x
Python 3 Projects > Inicial > TpInicial > utils > validator.py > Validator > has_numbers_v
1  import re
2
3
4  class Validator():
5
6      @staticmethod
7      def empty_field_validation(field, label):
8          if not len(field.strip()):
9              label['text'] = 'No puede estar vacío'
10             return bool(len(field.strip()))
11
12     @staticmethod
13     def has_numbers_validation(field, label):
14         if not bool(re.match(re.compile("^[a-zA-Z]+$"), field)):
15             label['text'] = 'No puede ingresar números'
16         return bool(re.match(re.compile("^[a-zA-Z]+$"), field))
17
```

En la línea 1 importamos la clase re que nos permite utilizar expresiones regulares.

En la línea 4 declaramos la clase Validator

En la línea 6 declaramos un método static “empty_field_validation” que toma dos parámetros.

El primero, “field”, hace referencia al texto que se quiere validar, el segundo parámetro “label” hace referencia un label que se encuentre en la vista. De esta manera, el método valida que el parámetro “field” ingresado no se encuentre vacío, y en caso de estarlo, actualiza el texto del label con la leyenda “No puede estar vacío”.

En la línea 12 tenemos otro método static muy similar al anterior, pero este valida a que el parámetro “field” no contenga números utilizando una expresión regular, y en caso de que haya alguno, actualiza el texto del label con la leyenda “No puede ingresar números”.

La app debe realizarse según el paradigma de POO

Para este requisito utilizamos clases para modelar cada uno de los módulos utilizados para implementar el patrón MVC.

Se debe implementar el patrón MVC

Para este requisito cambiamos la estructura con respecto a la entrega pasada, donde teníamos toda la funcionalidad en dos módulos, donde uno se encargaba de crear la ventana y darle funcionalidad a la misma, y el otro se encargaba de la comunicación con la base de datos.

Al implementar el patrón MVC, creamos un módulo que se encarga de crear la vista (view.py), otro que realiza la comunicación con la base de datos sobre la información que esta última necesita (class_product.py) y otro que coordina ambos módulos para evitar el acoplamiento entre ellos (controller.py).

Vista (view.py)



```
view.py 9 X
Python 3 Projects > Inicial > Tplnicial > view > view.py > ...
1 import tkinter as tk
2 from tkinter import ttk
3 from tkinter import Label
4 from tkinter import Entry
5 from tkinter import StringVar
6 from tkinter import Button
7 from utils.validator import Validator
8 import tkinter.font as tkFont
9
10
11 class View():
12
13     def __init__(self, controller):
14
15         # Attributes
16         self.controller = controller
17         self.root = tk.Tk()
18         self.validator = Validator()
19         self._init_root()
20
21     def show_view(self):
22         self.root.mainloop()
23
24 > def _init_root(self): ...
128 >
129 > def _on_tree_row_clicked(self, event): ...
136 >
137 > def _resetear_inputs(self): ...
141 >
142 > def _resetear_labels(self): ...
146 >
147 > def _update_treeview(self): ...
165 >
166 > def _create_product(self): ...
191 >
192 > def _delete_product(self): ...
202 >
203 > def _edit_product(self): ...
```

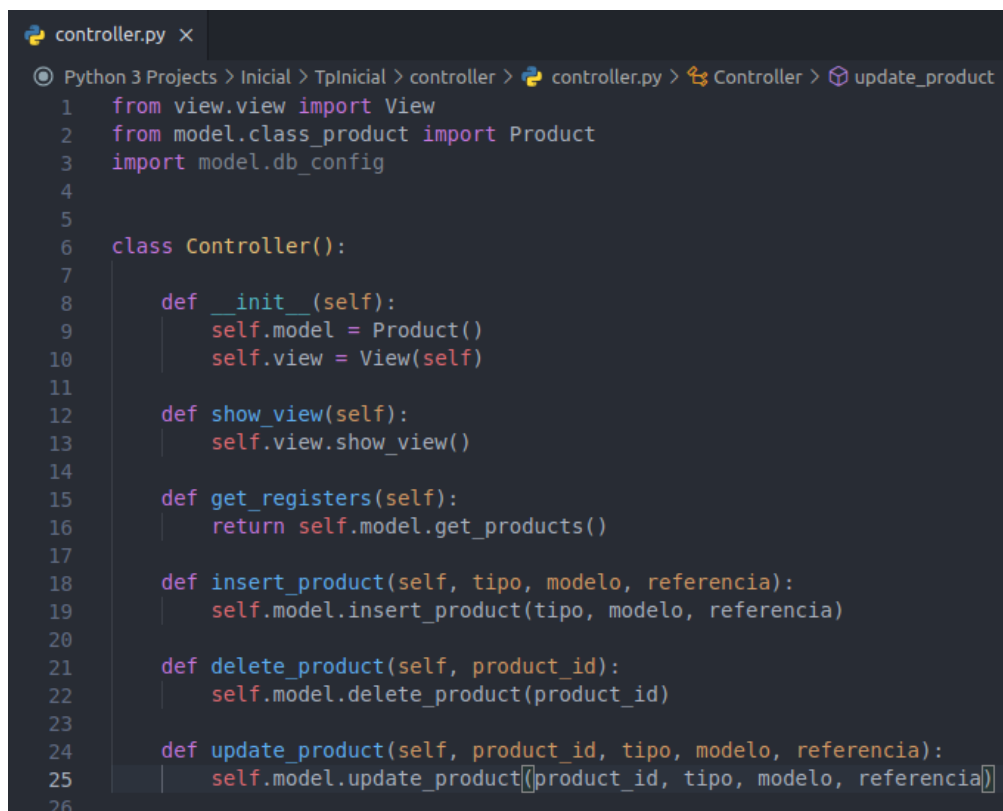
La vista la desarrollamos utilizando tkinter.

Por eso en las líneas 1 a 8 se importan elementos de tkinter, salvo en la línea 7 que se importa el validador.

En la línea 13 se define el constructor de la clase View el cual recibe una instancia de Controller (“controller”). Dentro del constructor se establece una referencia al controller que recibimos como parámetro, otra para una instancia de tk.Tk(), y otra mas para una instancia de Validator. Por último se llama al método propio init_root() que es el que se encarga de inicializar la ventana de tkinter.

Del resto de los métodos, los más relevantes son “_create_product”, “_delete_product”, “_edit_product” y “_update_treeview” que se comunican con la instancia de controller para crear, eliminar, actualizar y consultar los productos.

Controlador (controller.py)



```
controller.py x
Python 3 Projects > Inicial > Tpnicial > controller > controller.py > Controller > update_product
1  from view.view import View
2  from model.class_product import Product
3  import model.db_config
4
5
6  class Controller():
7
8      def __init__(self):
9          self.model = Product()
10         self.view = View(self)
11
12     def show_view(self):
13         self.view.show_view()
14
15     def get_registers(self):
16         return self.model.get_products()
17
18     def insert_product(self, tipo, modelo, referencia):
19         self.model.insert_product(tipo, modelo, referencia)
20
21     def delete_product(self, product_id):
22         self.model.delete_product(product_id)
23
24     def update_product(self, product_id, tipo, modelo, referencia):
25         self.model.update_product(product_id, tipo, modelo, referencia)
26
```

En la línea 1 se importa la clase View.

En la línea 2 se importa la clase Product, que hace el papel de modelo.

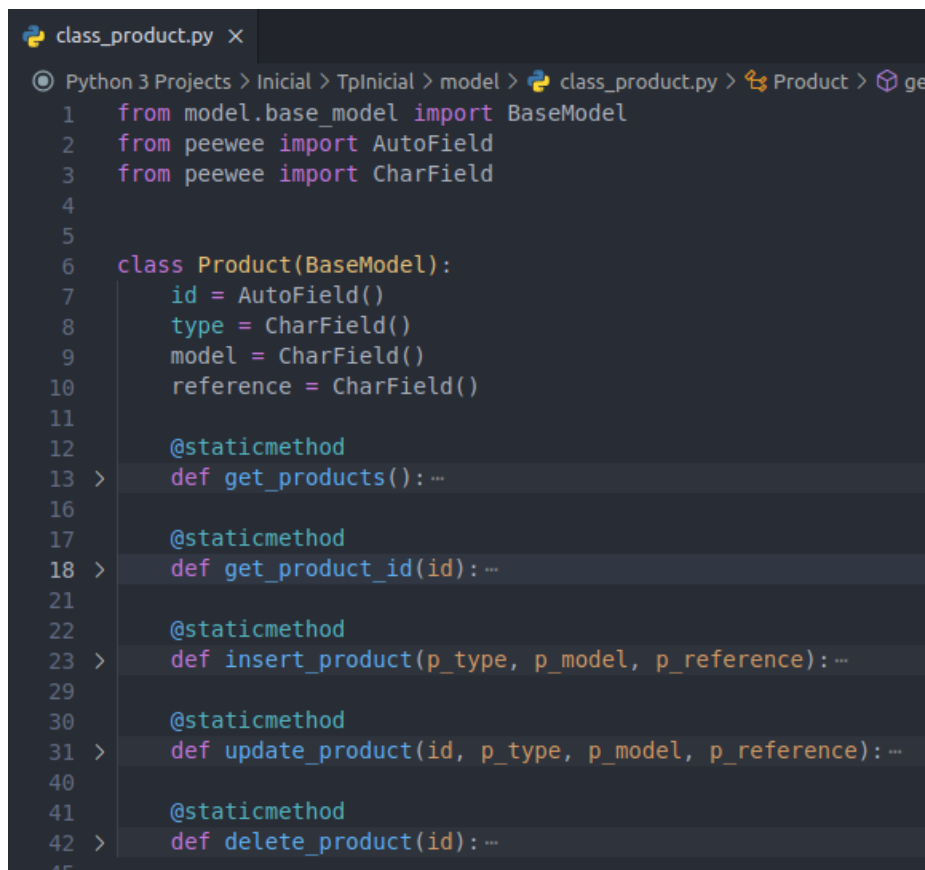
En la línea 3 se importa el módulo db_config que quien setea la conexión con la base de datos.

En la línea 8, el constructor inicializa dos atributos, uno que mantiene una referencia al modelo, y el otro a la vista que son instanciados.

En la línea 12, el método show_view es quien le dá la orden a la vista de mostrar la ventana.

El resto de los métodos se comunican con el modelo delegándole las operaciones de crud.

Modelo (class_product.py)



```
class_product.py x
Python 3 Projects > Inicial > Tplnicial > model > class_product.py > Product > ge
1  from model.base_model import BaseModel
2  from peewee import AutoField
3  from peewee import CharField
4
5
6  class Product(BaseModel):
7      id = AutoField()
8      type = CharField()
9      model = CharField()
10     reference = CharField()
11
12     @staticmethod
13 >     def get_products(): ...
16
17     @staticmethod
18 >     def get_product_id(id): ...
21
22     @staticmethod
23 >     def insert_product(p_type, p_model, p_reference): ...
29
30     @staticmethod
31 >     def update_product(id, p_type, p_model, p_reference): ...
40
41     @staticmethod
42 >     def delete_product(id): ...
45
```

En la línea 1 se importa la clase padre de Product (BaseModel), la cual es requerida por el orm peewee. En esta entrega decidimos tener la clase BaseModel en un archivo aparte y no declararla en el archivo class_product.py, por el caso de que en una futura entrega debamos crear otros modelos.

En las líneas 2 y 3 se importan los tipos necesarios por peewee para realizar el mapeo con la base de datos.

En el resto de las líneas creamos métodos static que son los que se comunican con la base de datos y realizan las operaciones necesarias para llevar a cabo el crud.

Se debe agregar el trabajo con excepciones

Decidimos agregar las excepciones en la vista, ya que de esta manera, al realizarse una excepción en la base de datos por alguna operación de crud, esta sería atrapada en la vista y podría mostrar el mensaje pertinente en la ventana. Agregamos bloques de try except en los métodos “_update_treeview”, “_create_product”, “_delete_product” y “_edit_product” ya que estos terminan recibiendo la respuesta de la base de datos ante las operaciones de crud.