ESTRUCTURA DE DATOS Y PROGRAMACIÓN ORIENTADA A OBJETOS

Tema Nº1:Módulos, Paquetes

Indicador de logro Nº1:Utiliza módulos existentes, crea módulos nuevos, los agrupa en paquete a través de un lenguaje de programación.

**TEMA 01 Teoría de los**

Imagen que contiene Icono

Descripción generada automáticamente

**TEMA Nº1:**

Módulos, Paquetes

**Subtema 1.1:**

El uso de módulos.

Si sales del intérprete de Python y vuelves a entrar, las definiciones que hayas hecho (funciones y variables) se pierden. Por ello, si quieres escribir un programa algo más largo, será mejor que utilices un editor de texto para preparar la entrada del intérprete y ejecutarlo con ese fichero como entrada. Esto se llama crear un guion. Según vayan creciendo los programas, puede que quieras dividirlos en varios ficheros para facilitar el mantenimiento. Puede que también quieras utilizar una función que has escrito en varios programas sin tener que copiar su definición a cada programa.

Para lograr esto, Python tiene un modo de poner definiciones en un fichero y utilizarlas en un guion o en una instancia interactiva del intérprete. Tal fichero se llama módulo; las definiciones de un módulo se pueden importar a otros módulos o al módulo principal (la colección de variables accesible desde un guion ejecutado desde el nivel superior y en el modo de calculadora).

Los paquetes son una manera de estructurar los espacios de nombres de Python usando “nombres de módulos con puntos”. Por ejemplo, el nombre de módulo A.B designa un submódulo llamado B en un paquete llamado A. Tal como el uso de módulos evita que los autores de diferentes módulos tengan que preocuparse de los respectivos nombres de variables globales, el uso de nombres de módulos con puntos evita que los autores de paquetes de muchos módulos, como NumPy o la Biblioteca de Imágenes de Python (Python Imaging Library, o PIL), tengan que preocuparse de los respectivos nombres de módulos.

**¿QUÉ ES UN MÓDULO?**

Un módulo es un fichero que contiene definiciones y sentencias Python. El nombre del fichero es el nombre del módulo con el sufijo **.py**. Dentro de un módulo, el nombre del módulo (como cadena) es accesible mediante la variable global **\_\_name\_\_**. Por ejemplo, utiliza tu editor de texto favorito para crear un fichero llamado **fibo.py** en el directorio actual, con el siguiente contenido:

# módulo de números Fibonacci

def fib(n): # escribe la serie Fibonacci hasta n

a, b = 0, 1

while b < n:

print(b, end = ' ')

a, b = b, a + b

print()

def fib2(n): # devuelve la serie Fibonacci hasta n

resultado = []

a, b = 0, 1

while b < n:

resultado.append(b)

a, b = b, a + b

return resultado

Ahora entra en el intérprete de Python e importa este módulo con la siguiente orden:

>>> import fibo

Esto no introduce los nombres de las funciones definidas en **fibo** directamente en la tabla de símbolos actual; sólo introduce el nombre del módulo **fibo**. Utilizando el nombre del módulo puedes acceder a las funciones:

>>> fibo.fib(1000)

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

>>> fibo.fib2(100)

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

>>> fibo.\_\_name\_\_

’fibo’

Si pretendes utilizar una función a menudo, la puedes asignar a un nombre local:

>>> fib = fibo.fib

>>> fib(500)

1 1 2 3 5 8 13 21 34 55 89 144 233 377

Esto importa todos los nombres, excepto los que empiezan por un guion bajo (\_).

* **IMPORTACIÓN DE MÓDULOS**

Un módulo puede contener tanto declaraciones ejecutables como definiciones de funciones. Estas declaraciones están pensadas para inicializar el módulo. Se ejecutan solamente la *primera* vez que el módulo se encuentra en una sentencia **import**. (Son también ejecutados si el archivo es ejecutado como un script).

Cada módulo tiene su propio espacio de nombres, el que es usado como espacio de nombres global por todas las funciones definidas en el módulo. Por lo tanto, el autor de un módulo puede usar variables globales en el módulo sin preocuparse acerca de conflictos con una variable global del usuario. Por otro lado, si sabes lo que estás haciendo puedes tocar las variables globales de un módulo con la misma notación usada para referirte a sus funciones, **nombremodulo.nombreitem**.

Los módulos pueden importar otros módulos. Es costumbre pero no obligatorio el ubicar todas las declaraciones **[import](http://docs.python.org.ar/tutorial/3/reference.html" \l "import)** al principio del módulo (o script, para el caso). Los nombres de los módulos importados se ubican en el espacio de nombres global del módulo que hace la importación.

Hay una variante de la declaración **[import](http://docs.python.org.ar/tutorial/3/reference.html" \l "import)** que importa los nombres de un módulo directamente al espacio de nombres del módulo que hace la importación. Por ejemplo:

>>> from fibo import fib, fib2

>>> fib(500)

1 1 2 3 5 8 13 21 34 55 89 144 233 377

Esto no introduce en el espacio de nombres local el nombre del módulo desde el cual se está importando (entonces, en el ejemplo, **fibo** no se define).

Hay incluso una variante para importar todos los nombres que un módulo define:

>>> from fibo import \*

>>> fib(500)

1 1 2 3 5 8 13 21 34 55 89 144 233 377

Esto importa todos los nombres excepto aquellos que comienzan con un subrayado (**\_**). La mayoría de las veces los programadores de Python no usan esto ya que introduce un conjunto de nombres en el intérprete, posiblemente escondiendo cosas que ya estaban definidas.

Nota que en general la práctica de importar **\*** de un módulo o paquete está muy mal vista, ya que frecuentemente genera un código poco legible. Sin embargo, está bien usarlo para ahorrar tecleo en sesiones interactivas.

**Nota:**

Por razones de eficiencia, cada módulo se importa una vez por sesión del intérprete. Por lo tanto, si modificas los módulos, tienes que reiniciar el intérprete o si es sólo un módulo que quieres probar interactivamente, usa **imp.reload()**, por ejemplo **import importlib**; **importlib.reload(modulename)**.

* **MÓDULOS ESTÁNDAR**

Python viene con una biblioteca de módulos estándar, descrita en un documento separado, la Referencia de la Biblioteca de Python (de aquí en más, “Referencia de la Biblioteca”). Algunos módulos se integran en el intérprete; estos proveen acceso a operaciones que no son parte del núcleo del lenguaje pero que sin embargo están integrados, tanto por eficiencia como para proveer acceso a primitivas del sistema operativo, como llamadas al sistema. El conjunto de tales módulos es una opción de configuración el cual también depende de la plataforma subyacente. Por ejemplo, el módulo **winreg** sólo se provee en sistemas Windows. Un módulo en particular merece algo de atención: **sys**, el que está integrado en todos los intérpretes de Python. Las variables **sys.ps1** y **sys.ps2** definen las cadenas usadas como cursores primarios y secundarios:

>>> import sys

>>> sys.ps1

'>>> '

>>> sys.ps2

'... '

>>> sys.ps1 = 'C> '

C> print('IDAT!')

IDAT!

C>

Estas dos variables están solamente definidas si el intérprete está en modo interactivo.

La variable **sys.path** es una lista de cadenas que determinan el camino de búsqueda del intérprete para los módulos. Se inicializa por omisión a un camino tomado de la variable de entorno **PYTHONPATH**, o a un valor predefinido en el intérprete si **PYTHONPATH** no está configurada. Lo puedes modificar usando las operaciones estándar de listas:

>>> import sys

>>> sys.path.append('/ufs/guido/lib/python')

**LA FUNCIÓN dir()**

La función interna **dir()** se utiliza para averiguar qué nombres define un módulo. Devuelve una lista de cadenas ordenada:

>>> import fibo, sys

>>> dir(fibo)

['\_\_name\_\_', 'fib', 'fib2']

>>> dir(sys)

['\_\_displayhook\_\_', '\_\_doc\_\_', '\_\_excepthook\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_stderr\_\_', '\_\_stdin\_\_', '\_\_stdout\_\_',

'\_clear\_type\_cache', '\_current\_frames', '\_debugmallocstats', '\_getframe', '\_home', '\_mercurial', '\_xoptions', 'abiflags', 'api\_version', 'argv', 'base\_exec\_prefix', 'base\_prefix', 'builtin\_module\_names', 'byteorder', 'call\_tracing', 'callstats', 'copyright', 'displayhook', 'dont\_write\_bytecode',

'exc\_info', 'excepthook', 'exec\_prefix', 'executable', 'exit', 'flags', 'float\_info', 'float\_repr\_style', 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount', 'gettrace', 'hash\_info', 'hexversion', 'implementation', 'int\_info', 'intern', 'maxsize', 'maxunicode', 'meta\_path', 'modules', 'path', 'path\_hooks', 'path\_importer\_cache', 'platform', 'prefix', 'ps1',

'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr']

Sin argumentos, **dir()** lista los nombres que tienes actualmente definidos:

>>> a = [1, 2, 3, 4, 5]

>>> import fibo

>>> fib = fibo.fib

>>> dir()

['\_\_builtins\_\_', '\_\_name\_\_', 'a', 'fib', 'fibo', 'sys']

Nota que lista todos los tipos de nombres: variables, módulos, funciones, etc.

**dir()** no lista los nombres de las funciones y variables integradas. Si quieres una lista de esos, están definidos en el módulo estándar **builtins**:

>>> import builtins

>>> dir(builtins)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',

'ValueError', 'Warning', 'ZeroDivisionError', '\_', '\_\_build\_class\_\_', '\_\_debug\_\_', '\_\_doc\_\_', '\_\_import\_\_', '\_\_name\_\_', '\_\_package\_\_', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',

'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

**Actividad:**

**CASO PRÁCTICO 1:**

Realizar un programa que simule tirar dos dados y luego muestre los valores que aparecieron. Si la suma de dichos números es igual a 9 mostrar un mensaje de “has ganado” sino mostrar “has perdido”.

# importamos el módulo random

import random

dado1 = random.randint(1, 6)

dado2 = random.randint(1, 6)

suma = dado1 + dado2

print("El primer dado vale: ", dado1)

print("El segundo dado vale: ", dado2)

print("La suma de los dados es: ", suma)

if suma == 9:

print("Has ganado")

else:

print("Has perdido")

Otra opción, sería implementar sólo aquella funcionalidad que necesitemos para llevar a cabo nuestro programa. Para ello, vamos a realizar el mismo ejemplo anterior pero cargando sólo la función **randint()** del módulo **random**.

from random import randint

dado1 = randint(1, 6)

dado2 = randint(1, 6)

suma = dado1 + dado2

print("El primer dado vale: ", dado1)

print("El segundo dado vale: ", dado2)

print("La suma de los dados es: ", suma)

if suma == 9:

print("Has ganado")

else:

print("Has perdido")

También podemos asignarle un alias a la función cargada de un módulo de la siguiente manera

from random import randint as rnd, uniform as unf

Utilizando la palabra clave as podemos cambiar el nombre de dicha función haciendo más sencillo y personalizado nuestro programa.

**CASO PRÁCTICO 2:**

Además de importar un módulo oficial de Python, que pasa si yo quiero importar un módulo o un archivo mío. En este caso lo haríamos de la siguiente forma:

**Fichero operaciones.py**

# crearemos una primera función para cargar un valor entero

def cargar():

return int(input("Introduce un valor entero: "))

# creamos una segunda función para sumar dos valores

def suma(num1, num2):

return num1 + num2

**Fichero resultado.py**

import operaciones

num1 = operaciones.cargar()

num2 = operaciones.cargar()

suma = operaciones.suma(num1, num2)

print("El primer valor es: ", num1)

print("El segundo valor es: ", num2)

print("La suma de los valores es: ", suma)

En el ejemplo anterior, creamos un primer archivo llamado **operaciones.py** en el cual incluimos las funciones necesarias para la ejecución de nuestro programa. Por otro lado, creamos el fichero **resultado.py** donde llamaremos al primer archivo para obtener las funciones que contiene.

**EJERCICIOS**:

**EJERCICIO 1:**

Desarrollar un programa que cargue una lista con 10 valores enteros. Cargar los valores aleatoriamente con números comprendidos entre 0 y 1000. Mostrar la lista obtenida por pantalla. Dividir el programa en dos archivos, por un lado, las funciones y por otro lado la ejecución de las mismas.

**ejer1func.py**

# importamos el modulo random

import random

# función para cargar valores aleatorios

def cargar():

lista = []

for x in range(10):

num = random.randint(1, 1000)

lista.append(num)

return lista

# función para imprimir la lista

def imprimir(lista):

print(lista)

**ejer1main.py**

# importamos nuestro fichero de funciones

import ejer1func as func

# bloque principal

lista = func.cargar()

func.imprimir(lista)

**EJERCICIO 2:**

Realizar el juego de “adivina el numerito”. Se deberá crear un programa que genere un valor aleatorio entre 1 y 100 y no se muestre. Después solicitar la carga de valores por teclado de un número y mostrar el mensaje “Has ganado” si el número es correcto. En caso contrario mostrar un mensaje informando si el número aleatorio es superior o inferior al introducido. Dividir el programa en dos archivos, por un lado, las funciones y por otro lado la ejecución de las mismas.

**ejer2func.py**

# importamos el modulo random

import random

# función para cargar un valor entero

def cargar():

return int(input("Ingrese un valor entero: "))

# función para obtener un número aleatorio

def aleatorio():

return random.randint(1, 100)

# función para jugar a adivina el numerito

def adivina():

num = aleatorio()

imprimir("Bienvenido al juego de adivina el numerito")

imprimir("A ver si eres capaz de adivinarlo")

correcto = False

while correcto == False:

n = cargar()

if num == n:

imprimir("Has ganado")

correcto = True

elif num < n:

imprimir("El valor introducido es menor")

else:

imprimir("El valor introducido es mayor")

# función para imprimir mensajes

def imprimir(texto):

print(texto)

**ejer2main.py**

# importamos nuestro fichero de funciones

import ejer2func

# bloque principal

# llamamos a la función de nuestro juego

ejer2func.adivina()

**EJERCICIO 3:**

Realizar un programa que solicite la carga de un valor entero. Después mostrar por pantalla la raíz cuadrada y el valor elevado al cuadrado y al cubo de dicho número. (Utilizar el módulo math de python).

# primero importaremos los modulos necesarios

from math import sqrt, pow

# cargaremos el valor entero

def cargar():

return int(input("Ingrese un valor entero: "))

# calcularemos la raíz cuadrada

def raiz(num):

return sqrt(num)

# calculamos el exponente

def exponente(num, exp):

return pow(num, exp)

# bloque principal

num = cargar()

print("El valor utilizado es ", num)

r = raiz(num)

print("La raíz cuadrada da el resultado de ", r)

print("El valor elevado al cuadrado da el resultado de ", exponente(num, 2))

print("El valor elevado al cubo da el resultado de ", exponente(num, 3))

**CASO PRÁCTICO 3:**

¿Eres capaz de desarrollar un reloj de horas, minutos y segundos utilizando el módulo datetime con la hora actual? Hazlo en un script llamado **reloj.py** y ejecútalo en la terminal:

**Tener en cuenta lo siguiente:**

El módulo **time** integra una función llamada **sleep(segundos)**que puede pausar la ejecución de un programa durante un tiempo. Así mismo el módulo **os** integra la función **system('cls')** y **system('clear')** para limpiar la pantalla de la terminal en sistemas Windows y Linux/Unix respectivamente.

**SOLUCIÓN:**

reloj.py

import datetime

import time

import os

# Completa el código aquí

while(True):

os.system('cls') # Limpiamos la pantalla

dt = datetime.datetime.now()

print( "{}:{}:{}".format( dt.hour, dt.minute, dt.second ) )

time.sleep(1) # Esperar 1 segundo

**Resultado**

21:29:36

.

.

.

**Subtema 1.2:**

Los paquetes

**¿QUÉ ES UN PAQUETE?**

Los paquetes son ficheros que contienen definiciones que se pueden importar en otros scripts para reutilizar sus funcionalidades.

Utilizar paquetes nos ofrece varias ventajas. En primer lugar, nos permite unificar distintos módulos bajo un mismo nombre de paquete, pudiendo crear jerarquías de módulos y submódulos, o también subpaquetes.

Por otra parte, nos permiten distribuir y manejar fácilmente nuestro código como si fueran librerías instalables de Python. De esta forma se pueden utilizar como módulos estándar desde el intérprete o scripts sin cargarlos previamente.

Para crear un paquete lo que tenemos que hacer es crear un fichero especial **init** vacío en el directorio donde tengamos todos los módulos que queremos agrupar. De esta forma cuando Python recorra este directorio será capaz de interpretar una jerarquía de módulos:

paquete/

\_\_init\_\_.py

saludos.py

script.py

Ahora, si utilizamos un script desde el mismo directorio donde se encuentra el paquete podemos acceder a los módulos, pero esta vez refiriéndonos al paquete y al módulo, así que debemos hacerlo con from import:

from paquete.saludos import Saludo

s = Saludo()

Esta jerarquía se puede expandir tanto como queramos creando subpaquetes, pero siempre añadiendo el fichero **init** en cada uno de ellos:

script.py

paquete/

\_\_init\_\_.py

hola/

\_\_init\_\_.py

saludos.py

adios/

\_\_init\_\_.py

despedidas.py

**paquete/hola/saludos.py**

def saludar():

print("Hola, te estoy saludando desde la función saludar() " \

"del módulo saludos")

class Saludo():

def \_\_init\_\_(self):

print("Hola, te estoy saludando desde el \_\_init\_\_" \

"de la clase Saludo")

**paquete/adios/despedidas.py**

def despedir():

print("Adiós, me estoy despidiendo desde la función despedir() " \

"del módulo despedidas")

class Despedida():

def \_\_init\_\_(self):

print("Adiós, me estoy despidiendo desde el \_\_init\_\_ " \

"de la clase Despedida")

Ahora de una forma bien sencilla podemos ejecutar las funciones y métodos de los módulos de cada subpaquete:

script.py

from paquete.hola.saludos import saludar

from paquete.adios.despedidas import Despedida

saludar()

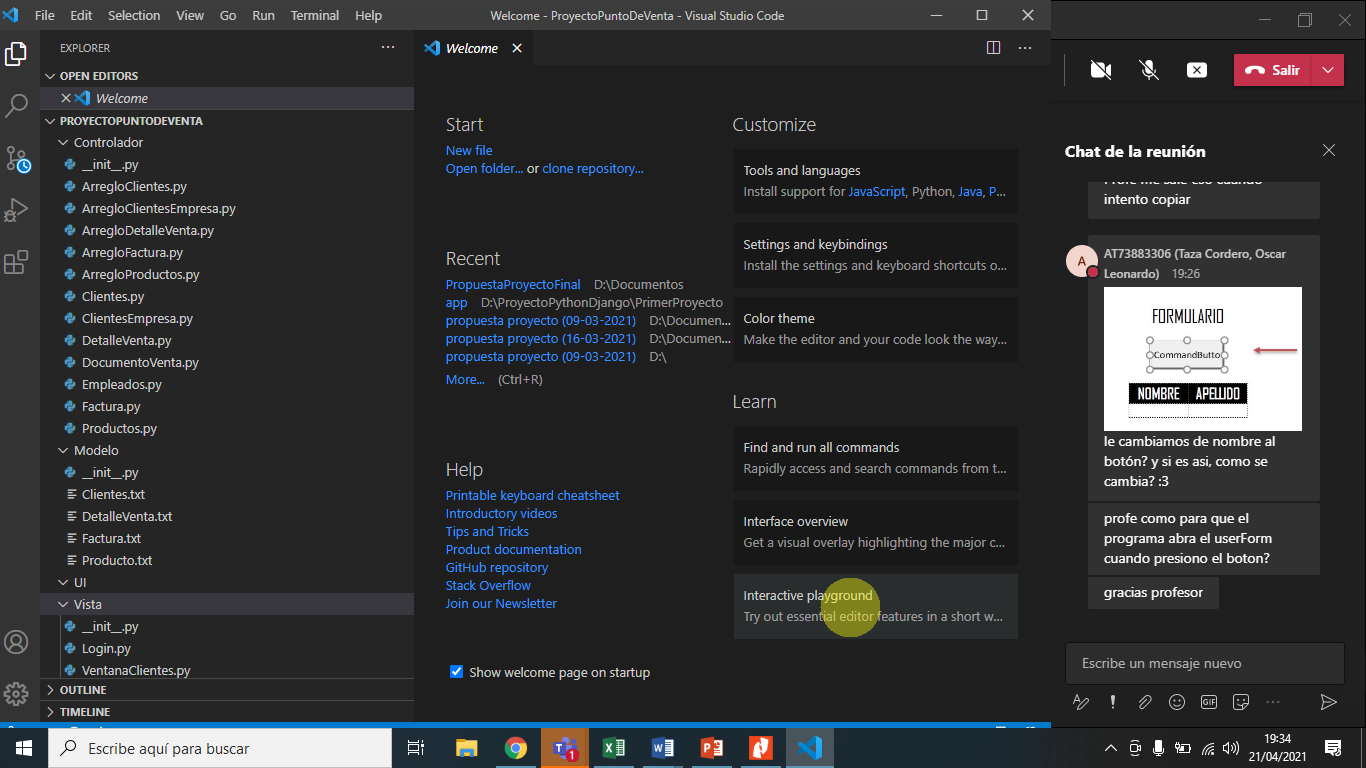
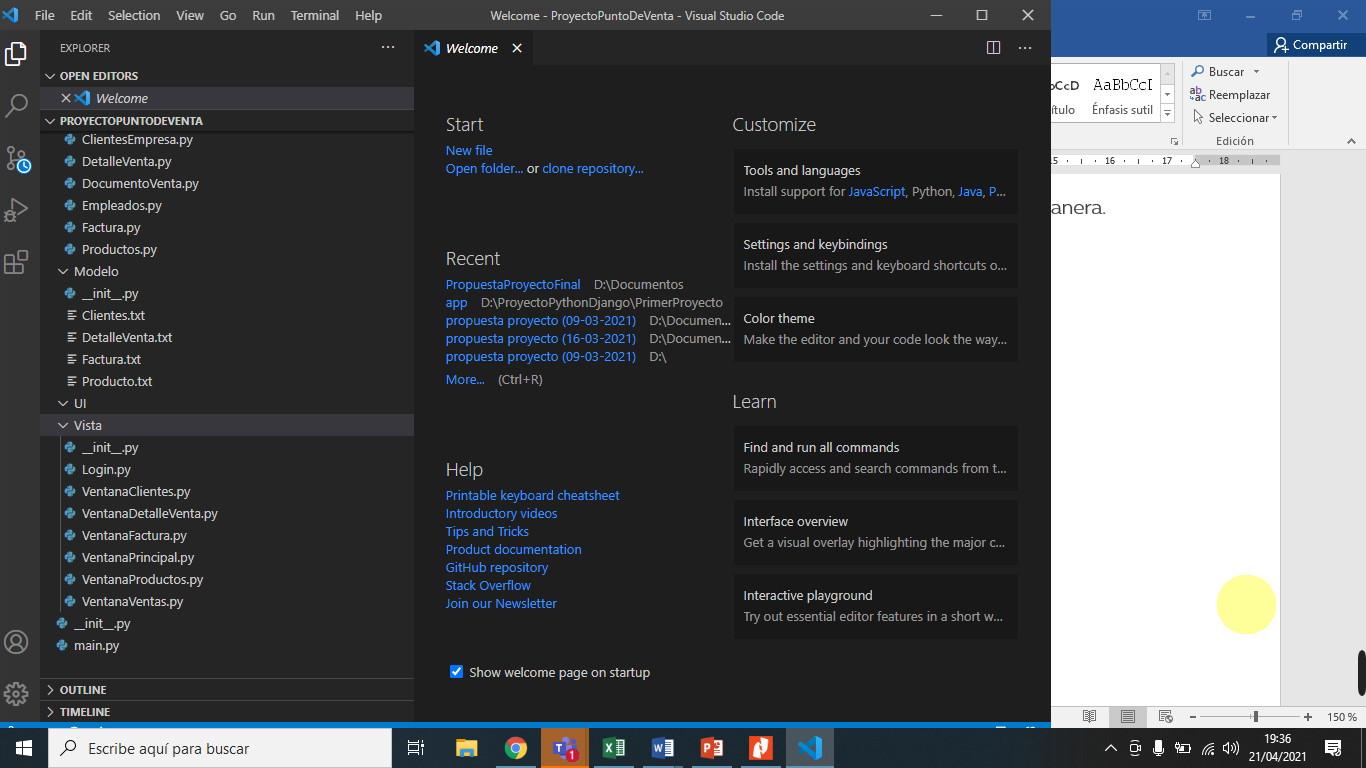
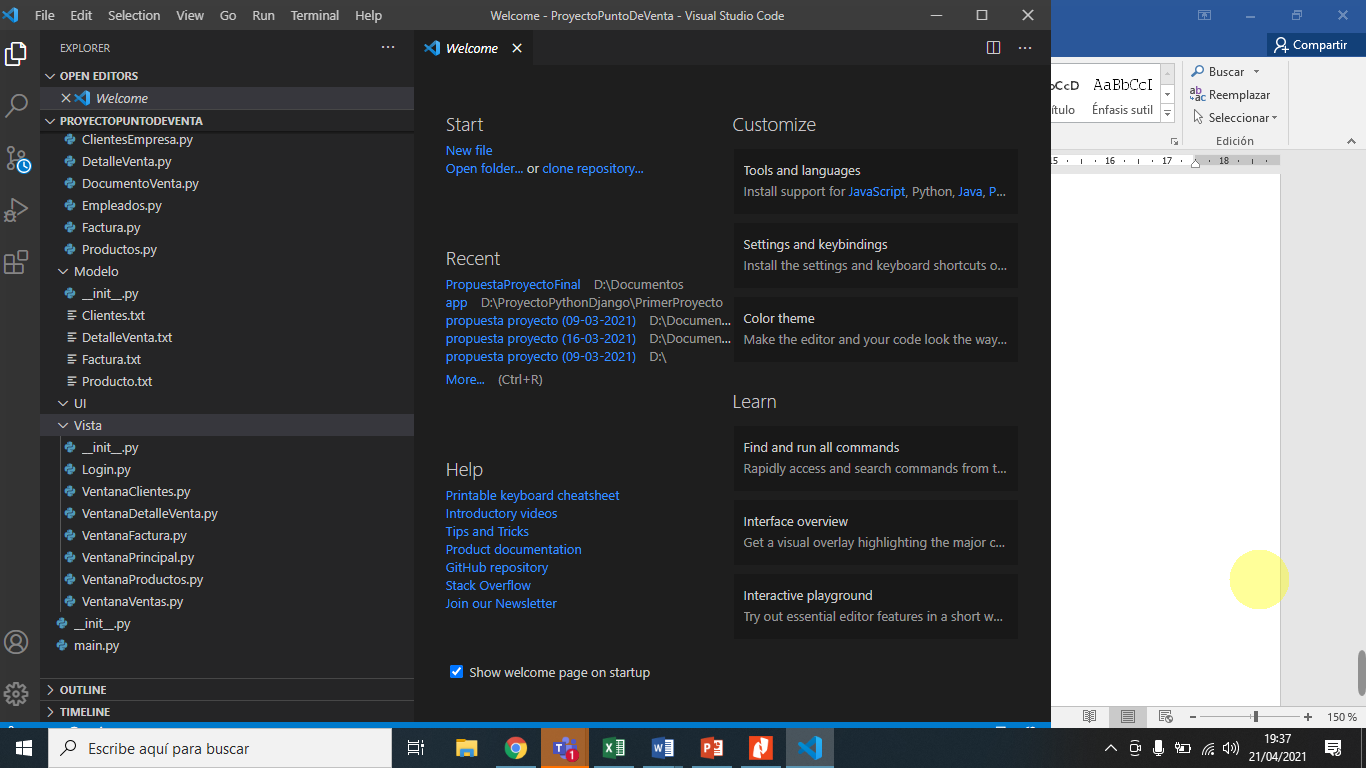
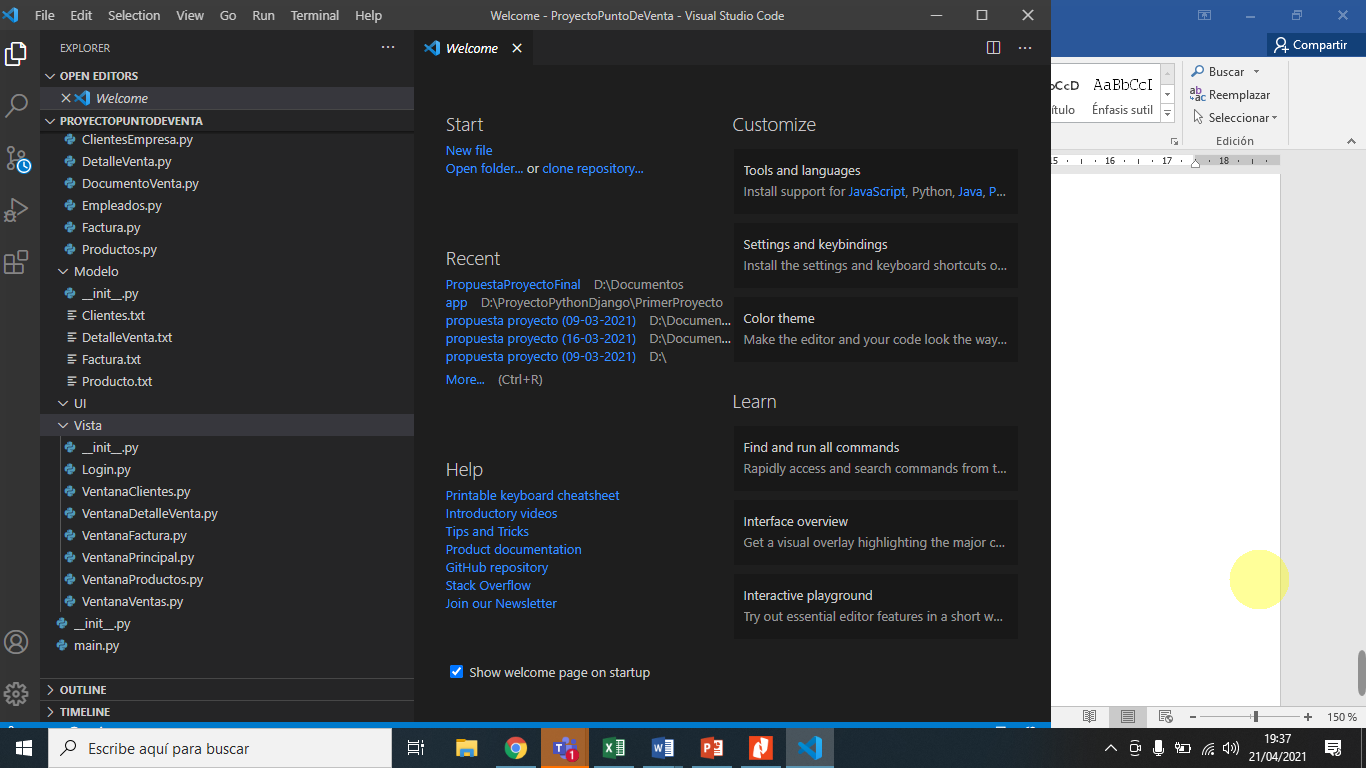
Despedida()

Como ves los paquetes son muy útiles, pero si intentamos ejecutarlos desde un directorio distinto a donde se encuentran, pasa lo mismo que los módulos y no funcionan.

**Subtema 1.2:**

Desarrollo del Proyecto

**PROYECTO: PUNTO DE VENTA**

* En esta primera parte crearemos la estructura de nuestro proyecto, el cual estará compuesto de 4 paquetes, todos ellos dentro de un paquete principal llamado ProyectoPuntoDeVenta:
  + Controlador
  + Modelo
  + UI
  + Vista
* En el paquete Controlador crearemos los siguientes módulos:
  + \_\_init\_\_.py
  + ArregloClientes.py
  + ArregloClientesEmpresa.py
  + ArregloDetalleVenta.py
  + ArregloFactura.py
  + ArregloProductos.py
  + Clientes.py
  + ClientesEmpresa.py
  + DetalleVenta.py
  + DocumentoVenta.py
  + Empleados.py
  + Factura.py
  + Productos.py
* En el paquete Modelo crearemos los archivos de texto:
  + Clientes.txt
  + DetalleVenta.txt
  + Factura.txt
  + Producto.txt
* En el paquete UI crearemos los formularios:
  + Login.ui
  + VentanaClientes.ui
  + VentanaDetalleVenta.ui
  + VentanaFactura.ui
  + VentanaPrincipal.ui
  + VentanaProductos.ui
  + VentanaVentas.ui
* En el paquete Vista crearemos los siguientes módulos:
  + \_\_init\_\_.py
  + Login.py
  + VentanaClientes.py
  + VentanaDetalleVenta.py
  + VentanaFactura.py
  + VentanaPrincipal.py
  + VentanaProductos.py
  + VentanaVentas.py
* En el paquete ProyectoPuntoDeVenta crearemos los siguientes módulos:
  + \_\_init\_\_.py
  + main.py