UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES
Centro de e-Learning

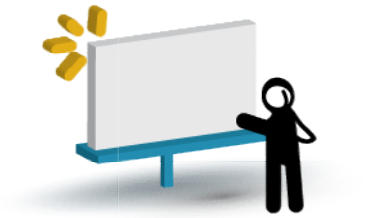
UNIDAD DIDÁCTICA I

DIPLOMATURA EN PYTHON

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo III – Nivel Intermedio

Unidad I – Módulos I.



Presentación:

En términos concretos, los módulos normalmente corresponden a los archivos de programa de Python. Cada archivo es un módulo, y los módulos importan otros módulos para usar los nombres que definen. Los módulos también pueden corresponder a extensiones codificadas en lenguajes externos como C, Java o C #, e incluso a directorios en la importación de paquetes.

En esta unidad comenzaremos a analizar cómo trabajar con módulos y paquetes en Python, desde cómo funciona la importación a la utilización de rutas relativas o absolutas.

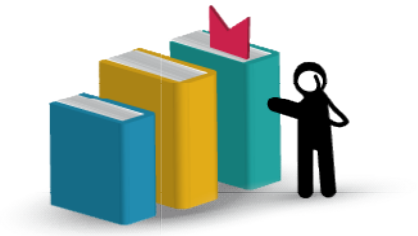
Objetivos:

Que los participantes:

Aprendan a incorporar módulos y paquetes a sus programas.

Comprendan la diferencia entre el uso de “import” y “from”.

Sepan en qué momento de la ejecucón del programa son importados los datos de un módulo o paquete.



Bloques temáticos:

1.- Introducción a Módulos.

2.- Diseño de Módulos.

3.- Releer Módulos.

Consignas para el aprendizaje colaborativo

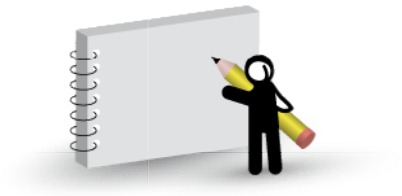
En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.



Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1.- Introducción a Módulos.

El termino modulo corresponde a un archivo de python, cada archivo es un módulo y los módulos pueden importar otros módulos para utilizar los nombres que estos definen.
Los módulos también pueden corresponder a extensiones de código de otros lenguajes como C, Java, o C# e incluso a directorios en la importación de paquetes.
Para importar los módulos se utiliza:

- **import**: Para importar todo el módulo. Import asigna un objeto módulo a un nombre, ya que el módulo en sí también es considerado un objeto.
- **from**: Se utiliza para importar un nombre específico, **from** asigna uno o más nombres a objetos con el mismo nombre en otro módulo

En los módulos definimos nombres conocidos como atributos que pueden ser referenciados externamente y de los cuales podemos utilizar sus herramientas.

Estructura de un programa.

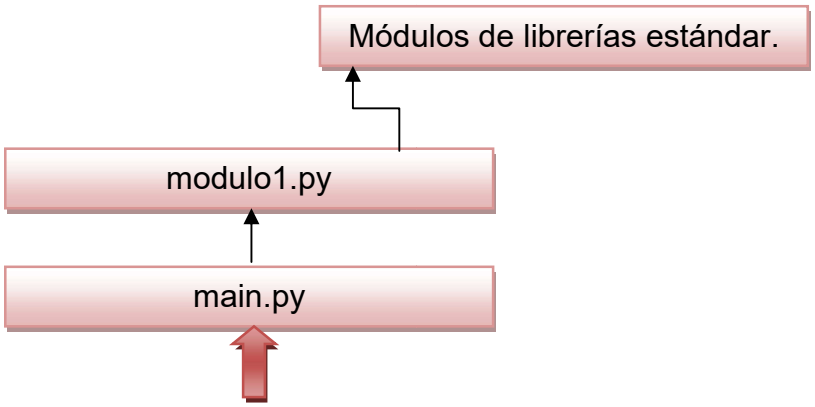
En un nivel básico python consiste de una serie de declaraciones dentro de un archivo principal que puede utilizar o no otros módulos complementarios. El archivo principal posee el control del flujo del programa, este es el archivo que se ejecuta al lanzar el programa.

Módulos por defecto (Standard Library)

Cada distribución de Python viene con un número de módulos por defecto, en las distribuciones actuales de python ya han superado los 200.

Importar un atributo.

Tomemos dos archivos main.py y modulo1.py, en donde main.py es el archivo de nivel más alto y el otro conforma un módulo.



Si ejecutamos main.py

a/main.py	a/modulo1.py
import modulo1 modulo1.imprimir('Mensaje a imprimir')	def imprimir(mensaje): print(mensaje)

Nos retorna:

```
Mensaje a imprimir
```

Nota: El módulo a importar se importa sin agregar la extensión del archivo.

¿Cómo funciona la importación?

Durante la ejecución del programa se producen tres pasos la primera vez que un programa importa un archivo.
1.- Encuentra el archivo.
2.- Lo compila a código de bit (si es necesario).
3.- Ejecutar el código del módulo para construir los objetos que define.
Las posteriores importaciones acceden simplemente a la memoria. Python almacena los módulos en una tabla llamada **sys.modules** y chequea ahí al inicio de una operación de importación. Si el módulo no se encuentra, se ejecutan los tres pasos previos.
Para ver el contenido podemos simplemente modificar el código anterior:

a/main.py

```
import modulo1
import sys

modulo1.imprimir('Mensaje a imprimir')
print(sys.modules)
```

Lo cual nos retorna:

```
Mensaje a imprimir
{'sys': <module 'sys' (built-in)>, 'builtins': <module 'builtins' (built-in)>, '_frozen_importlib': <module '_frozen_importlib' (frozen)>, ......
, 'modulo1': <module 'modulo1' from 'C:\\Users\\juanb\\Desktop\\a\\modulo1.py'>}
```

Nota: Se ha limitado el mensaje retornado para facilitar la comprensión.

¿Dónde se guarda el código de bit?.

Hasta la versión 3.1 se guardaba en archivos con extensión .pyc dentro del directorio en el cual se encontraba el módulo. Luego de la versión 3.2 se crea un directorio __pycache__ dentro del directorio en donde se encuentra el módulo y dentro se ubican los archivos .pyc. El nombre del archivo incluye el nombre de la versión de python que le da origen, así el archivo **models.py** puede convertirse en **models.cpython-37.pyc**

¿En que path se buscan los módulos?

Los módulos se buscan en los siguientes lugares según el orden que se presenta:

1.- El directorio del programa.
2.- Los directorios del PYTHONPATH si existen. Recordar que en este caso podemos ver todas las rutas importando el módulo “sys” y ejecutando la línea: print(sys.path)
3.- Direcciones de librerías Standard en la máquina.
4.- El contenido de cualquier archivo .pth (si existen). Archivos con los cuales se puede extender el punto 2. Estos son archivos que se pueden colocar dentro de la raíz de la distribución que estamos utilizando o dentro del directorio site-packages, en mi caso se encuentra en: C:\Python37-32\Lib\site-packages
5.- El directorio site-packages de extensiones de terceros.

Import vs From modulo import *

Al usar “from modulo import *” en lugar de un nombre específico, obtenemos copias de todos los nombres asignados en el nivel más alto del módulo referenciado. Técnicamente tanto import como from invocan la misma operación de importación, la operación from * agrega un paso extra en el cual se copia todos los nombres en el módulo importado. En esencia combina un namespace dentro del otro de forma de que la segunda opción requiere menos tipeo de código.

La importación se da una sola vez.

Tanto “import” como “from” se cargan y ejecutan solo la primera vez, dado que la importación es algo costoso desde el punto de vista computacional. Las importaciones posteriores solamente hacen referencia a los objetos ya importados de cada módulo.

Tomemos un ejemplo, si ejecutamos main.py

b/main.py	b/modulo2.py

import modulo2 #Importamos y ejecutamos el código del archivo
print(modulo2.color) # La asignación crea un atributo
modulo2.color = "verde" # Modificamos el atributo en el módulo importado
import modulo2 # Solo obtenemos el módulo ya cargado
print(modulo2.color) # El código no se recarga el atributo no ha retornado a su valor original.
	print('hola')
	color = 1

Nos retorna:

```
hola
rojo
verde
```

Cambiando objetos mutables en los módulos.

Los nombres copiados con from se convierten en referencias a objetos dados, pero la modificación de un objeto mutable en un nombre copiado puede modificar su valor en el módulo del cual se ha copiado.

Veamos un ejemplo:

c/main.py	c/modulo3.py

from modulo3 import x, y
 	
x = 42 # cambia solo esta x no la del módulo importado esta x es diferente a la del módulo.
y[0] = 42
print(x)
print(y[0])
print(y)
from modulo3 import x, y #Al hacer referencia a la x del módulo y ver su valor veo que es igual a la que tenía en el módulo. Pero el valor de la "y" del módulo ha cambiado.
print(x)
print(y[0])
print(y)
	x = 1
	y = [1, 2]

La salida nos retorna:

```
42
```

```
84
[84, 2]
1
84
[84, 2]
```

Es decir que tanto el objeto en el modulo como el atributo que es copia del objeto están apuntando al mismo objeto.
Notar que el valor de la x en el archivo original no ha cambiado ya que no hay un link con el valor de la x del módulo. Para cambiar un valor en otro archivo se debe utilizar import.

d/main.py

```
import modulo4

modulo4.x = 42          # cambia el valor de la x del módulo.
print(modulo4.x)

import modulo4
print(modulo4.x)
from modulo4 import x
print(x)
```

d/modulo4.py

```
x = 1
```

Retorna:

```
42
42
42
```

Notar la diferencia de que el cambio de y[0] a diferencia de el cambio en modulo4.x solo cambia el objeto no el nombre, y el nombre en ambos módulos referencia al mismo objeto.

Equivalencia entre import y from.

```
from module import nombre1, nombre2  #solo copia dos nombres del modulo module
Es equivalente a:
import module                  #busca el módulo
nombre1 = module.nombre1       #Copia los nombres por asignación
nombre2 = module.nombre2
del module                     # Eliminamos el nombre de módulo
```

Como toda asignación, from crea nuevas variables que inicialmente refieren a objetos con el mismo nombre en el módulo del cual provienen. Solo el nombre es copiado, no el objeto referenciado, ni tampoco el nombre del módulo. Al usar:

from module import *

Es lo mismo, pero todos los nombres del módulo son copiados.
Los módulos en python son código de byte en python, no código de máquina.

From debe usarse con mucho cuidado.

From tiene la potencialidad de corromper los espacios de nombre (namespaces) generando que sin que nos demos cuenta estemos pisando los nombres del módulo por lo que para importar un modulo en lugar de usar:

from module import *

Es mejor usar:

import module

El uso de import se prefiere en lugar de from cuando debemos utilizar los mismos nombres en dos módulos diferentes y debemos tener ambas versiones de los módulos en nuestro script. El uso de import evita colisiones al asignar un namespaces.
Nota: Toda sentencia import le asigna a un objeto del módulo un nombre, generando un atributo de módulo.

Diccionarios Namespaces: __dict__

Internamente los namespaces son guardados dentro de diccionarios, por lo que podemos acceder a los namespaces de módulo mediante el uso de __dict__.

e/main.py	e/modulo5.py
import modulo5 **print(list(modulo5.__dict__.keys()))** **print(modulo5.__dict__['__file__'])** **print(modulo5.__dict__['__name__'])**	print('Hola') nombre = 'Juan' def funcion1():pass class Clase1(): pass print('Chau')

Retorna:

```
Hola
Chau
['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__file__', '__cached__',
'__builtins__', 'nombre', 'funcion1', 'Clase1']
Diplomatura\Modulo-2\Unidad-1-Modulo-1-Diplomatura-python - copia\e\modulo5.py
modulo5
```

Como vemos en los nombres python agrega algunos nombres al namespace de módulo, por ejemplo, __file__ nos da el nombre del archivo del cual el módulo es cargado y __name__ retorna el nombre del módulo

Import vs alcance.

El alcance para “modulo6.f()” y “modulo6.g()” es siempre el módulo dentro del cual se encuentra independientemente desde donde este el llamado.

f/main.py	f/modulo6.py

X = 11
Y = 11
import modulo6
modulo6.f()
print(X, modulo6.X)
modulo6.g(1)
print(Y, modulo6.Y)
	X = 88
	Y = 88
	
	def f():
	global X
	X = 99
	
	def g(Z):
	global Y
	Y = 99 + Z

Retorna:

```
11 99
11 100
```

Uso de reload.

Para forzar a que un módulo sea recargado debemos usar reload.

```
from imp import reload          # Obtenemos reload en python (in 3.X)

reload(modulo)
```

Paquetes de módulos.

Cuando importamos un paquete en lugar de un módulo utilizamos notación de punto:

```
import dir1.dir2.mod
from dir1.dir2.mod import x
```

Esto está representando la siguiente estructura:

```
dir0\
        dir1\
        __init__.py
                dir2\
                __init__.py
                        mod.py
```

En donde dir0 es algún directorio de los que encontramos en el path de python.

ARCHIVO DE PAQUETE __init__.py

Cada directorio importado debe poseer un archivo __init__.py o la importación fallará.

Inicialización de paquete: La primera vez que un programa de python se importa desde un paquete, se ejecuta todo el código que se encuentra dentro del archivo __init__.py

Declaración de uso del módulo: Evita problemas no intencionales entre paquetes

Inicialización del namespaces del módulo: Asocia los nombres de objetos que se encuentran dentro de los archivos con los directorios, construye sus namespaces

Comportamiento de la sentencia from *: Al usar la lista __all__ dentro de __init__.py podemos determinar que se va a exportar cuando desde un script importamos el directorio con from *.

Veamos un ejemplo:

dir0	dir1	dir2
main.py	__init__.py	__init__.py
import dir1.dir2.mod	print('dir1 init') x = 1	print('dir2 init') y = 2
dir1	dir2	mod.py
		print('en modulo') z = 3

Retorna:

```
dir1 init
dir2 init
en modulo
```

Cada nombre de directorio en la ruta es una variable asignada al objeto de módulo cuyo namespaces es inicializado por todas las asignaciones en el directorio __init__.py.
dir1.x hace referencia a la variable x asignada en el archivo dir1/__init__.py

Importación relativa utilizando punto.

Al utilizar punto, estamos indicando que la importación es relativa solo al paquete contenedor. Esta importación buscará módulos dentro del directorio de paquete solamente y no buscará el mismo nombre de módulo localizado en la ruta de búsqueda de python (sys.path)

Ejemplos:

from . import spam **# relativo a este paquete**

Le estamos indicando a python que importe un módulo llamado “spam” localizado en el mismo directorio del archivo en donde aparece la instrucción.

from .spam import nombre

Significa que del módulo llamado “spam” localizado en el mismo directorio de paquete que aparece la declaración, importe la variable “nombre”

IMPORT SIN PUNTO

En python 3.x al no utilizar punto le estamos diciendo que busque un paquete en alguna de las rutas de (sys.path) en lugar del modulo con el mismo nombre en el paquete.

import string

Mientras que en python 2.x buscará un módulo con el mismo nombre dentro del paquete.

From con doble punto.

El doble punto busca una importación relativa desde el paquete padre del paquete en el cual estamos parados.

dir0
    dir1
    dirx
        dir2
        main.py

main.py

```
from .. import dirx
from ..dirx impor nombre
```

Relative - __init__.py con estructura

Para que la importación relativa funcione debemos incluir los módulos o paquetes que componen un paquete dentro de cada __init__.py
No se puede ejecutar directamente un modulo que incluye una ruta relativa.
Error que sale si lo intentamos en python 2.x

ValueError: Attempted relative import in non-package

Error que sale si lo intentamos en python 3.x

SystemError: Parent module '' not loaded, cannot perform relative import

Usar un paquete.

Para usar un paquete debemos agregarlo al path el paquete:

```
import sys
sys.path.append('ruta')
```

Ejemplo de rutas relativas.

En el siguiente ejemplo considero que el directorio "ejemploRelativo" se encuentra en mi escritorio en la ruta "**'C:/Users/Juan/Desktop/ejemploRelativo**"

ejemploRelativo

dir0

main.py

```
import sys
sys.path.append('C:/Users/Juan/Desktop/ejemploRelativo/dir0/')

import mi_paquete
mi_paquete.sub_paquete1.module_sub1_a
mi_paquete.sub_paquete1.module_sub1_a.main()
```

mi_paquete

main.py

```
import sys
sys.path.append('C:/Users/Juan/Desktop/ejemploRelativo/dir0/')
import mi_paquete
mi_paquete.sub_paquete1.module_sub1_a
mi_paquete.sub_paquete1.module_sub1_a.main()
```

__pycache__

__init__.cpython-37.py
modulo_a.cpython-37.py

sub_paquete1

__pycache__

__init__.cpython-37.py
modulo_sub1_a.cpython-37.py
modulo_sub1_b.cpython-37.py

__init__.py

```
from . import modulo_sub1_a
from . import modulo_sub1_b
```

modulo_sub1_a.py

```
from .modulo_sub1_b import Imprimir as impresion
from ..modulo_a import variable
from ..sub_paquete2 import modulo_sub2_a
def main():
    impresion()
print(variable)
print(modulo_sub2_a.variable2)
```

modulo_sub1_b.py

```
def Imprimir():
    print('Juan ' * 2)
```

sub_paquete2

__pycache__

__init__.cpython-37.py

modulo_sub2_a.cpython-37.py

__init__.py

```
from . import sub_paquete1
from . import sub_paquete2
```

modulo_sub2_a.py

```
print('hola modulo_sub2_a')
variable2 = 'Soy una variable dentro del modulo_sub2_a que esta dentro de sub_paquete2'
```

modulo_a.py

```
print('hola modulo a')
variable = 'Soy una variable dentro de modulo_a'
```

__init__.py

```
from . import sub_paquete1
from . import sub_paquete2
from . import modulo_a
```

2. Diseño de módulos.

Ocultación no existe en python

En python la ocultación de datos en módulos es una convención no una restricción. Se puede restringir la importación de variables o funciones por su nombre cuando utilizamos:

from modulo import *

Para evitar pisar el nombre de una variable que cause un daño que no queremos que se dé.
Esta restricción no se encuentra para import ya que con import incluimos el namespaces.

__all__

Este método se utiliza para declarar aquellas variables o métodos que queremos que se vean desde el archivo que importamos el módulo, todo no que no está declarado en __all__ no se importa cuando se importa con el uso de:

from privadoall import *

Y solo pueden ser importados si se especifica solo el nombre de lo que queremos importar.

from privadoall import variablePrivada1

Nota: las variable y métodos que no queremos importar, en realidad no son privados (como se consideraría en lenguajes como JAVA, PHP, entre otros) ya que vasta indicar su nombre explícitamente para poder acceder a ellos.

Veamos un ejemplo para aclarar los conceptos.

privado__all__/ privadoall.py

```
__all__ = ['variableACompartir1', 'funcionACompartir1']

variablePrivada1 = "Hola variable privada 1"
variableACompartir1 = "Hola variable pública 1"

def funcionACompartir1():
    return 'Hola función Privada 1'

print("i----------------------")
print(variablePrivada1)
print(variableACompartir1)
print(funcionACompartir1())
print("f----------------------")
```

Retorna:

```
i----------------------
Hola variable privada 1
Hola variable pública 1
Hola función pública 1
f----------------------
```

privadoall/ recuperarall.py

```
from privadoall import *
#from privadoall import variablePrivada1
print(variableACompartir1)
print(funcionACompartir1())

# variablePrivada1 no es exportado por el módulo ya que no está definido dentro de __all__
# no es accesible mediante : from privadoall import *
# pero si es accesible mediante :
print(variablePrivada1)
```

Retorna:

```
¡----------------------
Hola variable privada 1
Hola variable pública 1
Hola función pública 1
f----------------------
Hola variable pública 1
Hola función pública 1

Traceback (most recent call last):

File "C:/Users/juanb/Documents/000-TRABAJOS-2018/000-MEDRANO-2019/004-Python-Diplomatura/Modulo-2/Unidad-1-Modulo-1-Diplomatura-python - copia/privado__all__/recuperarall.py", line 10, in <module>

print(variablePrivada1)

NameError: name 'variablePrivada1' is not defined
```

Al estar comentada la línea

#from privadoall import variablePrivada1

Python nos va a decir que no existe la variablePrivada1 retornándonos un error. Sin embargo si descomentamos la línea, estamos explícitamente llamando a la variable, con lo cual en este caso si podríamos trabajar con dicho valor.

Al utilizar import en la importación y llamar a la variable de forma explícita nos retorna el valor.

privadoall/ recuperarallimport.py

```
import privadoall

print(privadoall.variablePrivada1)
```

Retorna:

```
¡----------------------
Hola variable privada 1
```

```
Hola variable pública 1
Hola función pública 1
f----------------------
Hola variable privada 1
```

Anteponiendo un guión bajo a la variable: _X
Logramos un efecto similar, la variable no puede ser importada con el uso de:

```
from modulo import *
```

Pero si puede ser importado con import ya que no se van a pisar al usar namespaces.

Ejemplo

privado_x/privado_x.py	privado_x/recuper_x.py
a, _b, c, _d = 1, 2, 3, 4	import privado_x print(privado_x._b)

Retorna:

```
2
```

Uso de __main__

Una de las ventajas que incorpora Python con el uso de __main__ es poder llamar un módulo directamente, con un código de prueba que es leído y ejecutado únicamente cuando este se invoca. Esto permite testear nuestro script y dejar en el mismo el código de prueba.

OJO NO PUEDE SER USADO CON IMPORTACIONES RELATIVAS, DA ERROR.

3. Releer módulos.

El nombre del módulo en import o from es un nombre de variable, en algunos casos un programa puede necesitar importar un módulo por su nombre como string en tiempo de ejecución, desafortunadamente Python no puede utilizar una declaración como “import” aplicada a un nombre de módulo como string. Por tal motivo la siguiente línea de código nos dará un error:

```
import ‘string’
```

Un error se dará también si intentamos asignar el nombre a una variable y pasarle el nombre:

```
x = ‘string’
import x
```

En este caso python intentará cargar el módulo x.py y no string.py
Para realizar la carga de un módulo a partir de su nombre como string podemos realizar lo siguiente:

recargar_Modulos/main1.py

```
modname = 'b'
exec('import ' + modname)
b.imprimir('texto desde a -')
```

recargar_Modulos/modulo.py

```
def imprimir(texto):
print(texto, 'texto desde b')
```

Retorna:

```
texto desde a - texto desde modulo
```

La función exec compila un código de string y le pasa esto al intérprete de Python para que sea ejecutado. En Python la compilación del código de byte se encuentra disponible en tiempo de ejecución, por lo que se puede escribir un programa que construya y ejecute otro programa como este.

La función exec() debe compilar el import cada vez que se ejecuta y la compilación puede resultar lenta, por lo que pre compilar a código de byte mediante el compilador de built-in puede ayudar al ejecutar un código a partir de un string muchas veces, sin embargo resulta más rápido utilizar la función del built-in: __import__ obteniendo un resultado similar.

recargar_Modulos/main2.py	recargar_Modulos/modulo.py
nombreModulo = 'modulo' **modulo = __import__(nombreModulo)** **modulo.imprimir('texto desde a -')**	def imprimir(texto): print(texto, 'texto desde b')

Retorna:

```
texto desde a - texto desde modulo
```

De acuerdo a la documentación oficial es preferible el uso de importlib.import_module()

recargar_Modulos/main3.py	recargar_Modulos/modulo.py
import importlib **nombreModulo = 'modulo'** **modulo =** **importlib.import_module(nombreModulo)** **modulo.imprimir('texto desde a -')**	def imprimir(texto): print(texto, 'texto desde b')

Retorna:

```
texto desde a - texto desde modulo
```

Cuando realizo una recarga de módulos, esta se realiza solo sobre el modulo invocado, no sobre los módulos de los cuales depende el módulo invocado.
Es decir que si realizo una recarga del módulo A, el cual está a su vez importando a los módulos B y C, la recarga se realiza solamente sobre A.
Una solución podría ser realizar una recarga recursiva que busque todos los módulos utilizados.

TRABAJO PARA LA APROBACIÓN DEL CURSO

Cada unidad presenta un trabajo a entregar, la aprobación del nivel inicial se realiza mediante la entrega de todos los trabajos propuestos (8 en total).

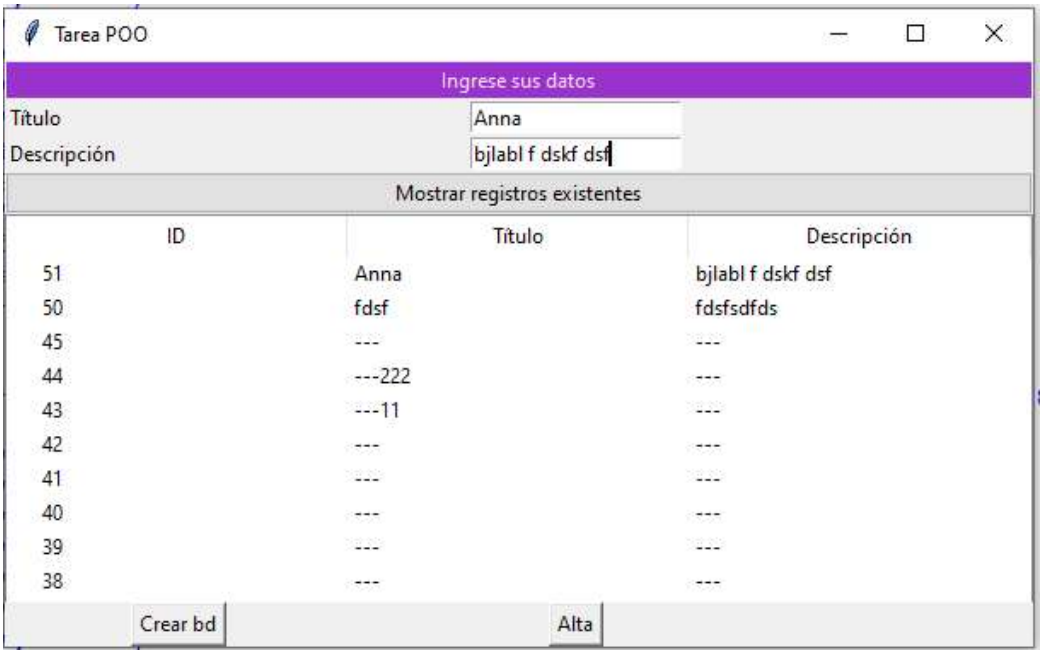
La siguiente tarea tiene como finalidad:

1. Hacer un repaso de lo estudiado en el curso de python nivel inicial.
2. Nivelar el conocimiento de los alumnos.
3. Comenzar a aplicar el trabajo con módulos separando nuestro código en varios scripts.

Nota: La siguiente tarea requiere conocimientos previos de tkinter, los cuales han sido adquiridos en el curso previo de python nivel inicial, si el alumno no ha realizado el curso de python nivel inicial y no ha visto cómo trabajar con tkinter, podrá solicitar material sobre la plataforma al docente y realizar la aplicación sin el uso de la interfaz gráfica.

TAREA

Cree una aplicación con tkinter que se vea como en la siguiente imagen:



Nota: Si el alumno no ha realizado el curso de python nivel inicial y no ha visto como trabajar con tkinter, podrá solicitar material sobre la plataforma al docente y realizar la aplicación sin el uso de la interfaz gráfica.

La misma debe poder:

1. Dar de alta un título y una descripción.

2. Guardar los registros en una base de datos del tipo MySQL mediante una función que se encuentre en un módulo aparte.

3. Presentar los registros en pantalla mediante treeView de tkinter.

4. Presentar un botón para crear la base de datos o crearla directamente al cargar el primer registro.

5. Validar el código del campo título para admitir solo alfanuméricos.

6. El código de validación debe encontrarse en un modulo aparte y ser llamado mediante una función de nombre “Validar”

Ayudas:

Instrucción para la creación de la base de datos:

CREATE DATABASE baseprueba3

Instrucción para la creación de la estructura de tablas:

CREATE TABLE producto(id int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT, titulo VARCHAR(128) COLLATE utf8_spanish2_ci NOT NULL, descripcion text COLLATE utf8_spanish2_ci NOT NULL)

Instrucción para la validación de código:

patron="^[A-Za-z]+(?:[_-][A-Za-z]+)*$"

Bibliografía utilizada y sugerida

Libros

Programming Python 5th Edition – Mark Lutz – O'Reilly 2013

Programming Python 4th Edition – Mark Lutz – O'Reilly 2011

Manual online

https://docs.python.org/3.7/tutorial/
https://docs.python.org/3.7/library/index.html
https://docs.python.org/3/distutils/introduction.html

Lo que vimos

En esta unidad hemos comenzado a trabajar con módulos y paquetes.

Lo que viene:

En la siguiente unidad avanzaremos en la construcción de la distribución analizando el archivo setup.py.