PythonBasicCourse-es006

October 13, 2022

1 Curso básico de Python

1.1 Apuntes

Curso básico de Python. Apuntes por Marcelo Horacio Fortino. Versión 2.3. Octubre 2022.

Versión	Autor/es	Fecha	Observaciones
1.0	Marcelo Horacio Fortino	2021/Marzo	Curso Python
1.1	Marcelo Horacio Fortino	2021/Junio	Convertido a markdown - ipynb
1.2	Marcelo Horacio Fortino	2021/Agosto	Actualizados contenidos
1.3	Marcelo Horacio Fortino	2021/Octubre	Agregado Flask microframework
1.4	Marcelo Horacio Fortino	2021/Noviembre	Agregado Pandas - datascience
1.5	Marcelo Horacio Fortino	2021/Diciembre	Agregado Devops - Ansible
2.0	Marcelo Horacio Fortino	2022/Abril	Nueva estructura: core / module
2.1	Marcelo Horacio Fortino	2022/Junio	Módulo apuntes intermedio
2.2	Marcelo Horacio Fortino	2022/Agosto	Actualizado temario y ejercicios
2.3	Marcelo Horacio Fortino	2022/Octubre	Actualizado despliegue a render.com

cluso sin la garantía MERCANTIL implícita o sin garantizar la CONVENIENCIA PARA UN PROPÓSITO PARTICULAR. El autor no asume ninguna responsabilidad si el lector hace un mal uso de la misma.

Estos apuntes se basan en: - La documentación oficial de Python, https://docs.python.org/es/3/tutorial/index.html, - La bibliografía presentada al final de este documento, y - Documentación propia recogida a lo largo de los años de diversas fuentes.

1.2 Objetivo del curso

- Objetivo general del curso:
- Aprender a usar Python para crear scripts y programas simples plenamente funcionales, desarrollar aplicaciones web y realizar análisis de datos.

- Objetivos específicos:

- Reconocer las características principales de Python y su utilidad práctica.
- Identificar tipos de datos (simples y compuestos) y operadores.
- Aplicar variables y estructuras de control de flujo.
- Construir funciones y clases (POO).
- Clasificar los módulos y paquetes por sus funcionalidades y objetivos.
- Comparar el lenguaje con otros similares de scripts, procedimentales y orientados a objetos.
- Probar bibliotecas para conexiones REST a aplicaciones web y bases de datos.
- Resolver problemas y errores en el código fuente proponiendo soluciones alternativas (refactoring).
- Utilizar la biblioteca pandas junto con matplotlib y numpy para realizar análisis estadísticos de datos y gráficos.

- Como resultado práctico al final del curso cada estudiante habrá creado una aplicación web utilizando el microframework de Python Flask.

1.3 Temario

- Introducción, instalación y compilación
- Datos, expresiones y sentencias
- Variables y funciones, control de flujo
- Clases y objetos, herencia, polimorfismo
- Entradas y salidas con Python
- Gestión de módulos, paquetes y bibliotecas
- Servicios y programas en red, REST API
- Desarrollo de aplicaciones web con Flask
- Análisis de datos con pandas, matplotlib y numpy

- Módulos opcionales
  - SQL ejemplos con pandas
  - Plotting con Python
  - Machine Learning con Python
  - DevOps con Ansible
  - SDK de GCP para Python
  - Kubernetes con Python y Docker

1.4 Bibliografía

- Downey, A., Elkner, J., Meyers, C. Aprenda a Pensar Como un Programador con Python. (2015).
  Recuperado de https://argentinaenpython.com/quiero-aprender-python/aprenda-a-pensar-como-un-programador-con-python.pdf
- Kent D. Lee. Python, Programming Fundamentals Second Edition. 2014.
- Marzal Varó, A., Gracia Luengo, I., García Sevilla, Pedro. Introducción a la programación con Python 3. (2014).
  Recuperado de http://repositori.uji.es/xmlui/handle/10234/102653
- Miller, B., Ranum, D. Solución de problemas con algoritmos y estructuras de datos usando Python. Traducido por Mauricio Orozco-Alzate, Universidad Nacional de Colombia - Sede Manizales.
  Recuperado de https://runestone.academy/ns/books/published/pythoned/index.html?mode=browsing
- Shaw, Z. A., Learn Python 3 the Hard Way. (2016).
  Recuperado de https://learnpythonthehardway.org/
- Van Rossum, G. and the Python development team. Documentación de Python en español. (2020).
  Recuperado de https://python-docs-es.readthedocs.io/es/3.10/

2 Funciones y módulos en Python

- Introducción, Funciones parametrizadas, La instrucción return, Palabra clave None, Funciones con listas, Recursividad, Secuencia y mutabilidad, Diccionarios.
- Módulos en Python: Módulos estándar, La declaración import, Módulos y paquetes, Herramientas de línea de comandos, Argparse, Módulo Platform, Python webserver, Conexión a bases de datos, Módulo sqlite3, SQLAlchemy ORM.

2.1 Introducción

- Se suele dividir el código fuente de un programa en partes, algunas de ellas se desarrollan como una función.
- A este proceso se lo denomina descomposición y permite probar cada parte del código por separado.
- En Python se utlilizan las funciones integradas *print()* para mostrar o imprimir algo en consola, e *input()* para leer el valor de una variable.
- Python también utiliza métodos, que son funciones declaradas de una manera muy específica.
- Si un fragmento de código se hace muy extenso es conveniente dividirlo en pequeños problemas por separado e implementar cada uno de ellos como una función independiente.
- Esto también permite dividir el trabajo entre varios programadores.
- En Python las funciones vienen integradas en el mismo lenguaje, provienen de módulos que se pueden importar al programa, o se pueden crear a propósito.

[ ]:
```python
def mensaje():    # Se define la función
    print("Hola mundo!")    # Cuerpo de la función

mensaje()    # Se invoca a la función
```

2.2 Funciones parametrizadas

- Una función permite aceptar parámetros que pueden modificar el comportamiento de la misma, haciéndola más flexible y adaptable.
- Un parámetro es una variable que existe dentro de la función en donde ha sido definido y se invoca especificando el argumento correspondiente.
- Los argumentos pasan los valores a los parámetros correspondientes y existen solamente fuera de las funciones.

- Se puede tener una variable con el mismo nombre del parámetro de la función.
- En ese caso se activa un mecanismo denominado sombreado:
  - El parámetro es una entidad completamente diferente de la variable.

[ ]:
```python
# Función con un argumento
def saludo(nombre):
    print("Hola,", nombre)

nombre = input("Ingresa tu nombre: ")

saludo(nombre)
print(id(saludo(nombre)))
print(nombre)
print(id(nombre))
```

- Una función puede tener n parámetros, que se pasan como parámetros posicionales.

[ ]:
```python
def mensaje(tipo, valor):
    print("Contenido: ", tipo, "= ", valor)

mensaje("teléfono", 644644644)
mensaje("email", "juan@mail.com")
mensaje("Ciudad", "Zaragoza")
```

- También se pueden pasar argumentos utilizando palabras clave.

[ ]:
```python
def saludo(nombre, apellidos):
    print("Hola, mi nombre es", nombre, apellidos)

saludo(nombre="Juan", apellidos="Perez Lopez")
saludo(apellidos="Perez Lopez", nombre="Juan")
```

- Los argumentos más frecuentes se pueden predefinir:

[ ]:
```python
def saludo(nombre, apellidos="Perez Lopez"):
    print("Hola, mi nombre es", nombre, apellidos)

saludo("Juan")
saludo(nombre="José")
saludo("Juan", apellidos="Garcia Lopez")
```

- Si se combinan, primero se deben especificar los argumentos posicionales y después los de palabras clave.

[ ]:
```python
def substraccion(a, b):
    print(a - b)

substraccion(5, b=2)
substraccion(5, 2)
```

2.3 La instrucción return

- La instrucción *return* (regresar o retornar) se utiliza para lograr que las funciones devuelvan un valor.
- Tiene dos variantes: la palabra reservada en sí y la palabra reservada en sí junto con una expresión.

- La palabra reservada en sí: termina la función si se emplea dentro de la misma y realiza un retorno instantáneo al punto de invocación.
- No es obligatorio emplearla, se ejecuta implícitamente al final de la función.

[ ]:
```python
def feliz_navidad(deseos=True):
    print("jo, jo, jo...")
    if not deseos:
        return
    print("¡Feliz navidad!")


feliz_navidad()

print("Sin deseos de navidad:")
feliz_navidad(False)
```

- La palabra reservada en sí junto con una expresión: Evalua el valor de la expresión y lo devuelve como resultado de la función finalizándola.

[ ]:
```python
def hola_mundo():
    return print("Hola Mundo!")


x = hola_mundo()
print(f"La función hola_mundo ha devuelto: {x}")
```

2.4 Palabra clave None

- Solo existen dos tipos de circunstancias en las que None se puede usar de manera segura:
  - Cuando se le asigna a una variable (o se devuelve como el resultado de una función).
  - Cuando se compara con una variable para diagnosticar su estado interno.

[1]:
```
def funcion_none(n):
    if(n % 2 == 0):
        return True


print(funcion_none(2))
print(funcion_none(1))
```

```
True
None
```

[2]:
```
def funcion_none(n):
    if(n % 2 == 0):
        return True
    else:
        return False


print(funcion_none(2))
print(funcion_none(1))
```

```
True
False
```

[1]:
```
variable_none = None
if variable_none is None:
    print("La variable no contiene ningún valor")
```

```
La variable no contiene ningún valor
```

2.5 Funciones con listas

- Se puede usar una lista como argumento de una función:

[ ]:
```
def hola(lista):
    for nombre in lista:
        print("Hola", nombre)


hola(["María", "Juan", "Marta", "José"])
```

- Una lista también puede ser un resultado de función:

[ ]:
```
def crear_lista(n):
    lista = []
    for i in range(n):
        lista.append(i)
    return lista


print(crear_lista(10))
```

2.6 Recursividad

- En programación la recursividad es una técnica donde una función se invoca a si misma.
- Funciona de forma similar a las iteraciones, por lo que se debe evitar caer en bucles (*loops*) infinitos.

- Ejemplo de como escribir utilizando while una parte de la serie de Fibonacci.
- Fuente: https://en.wikipedia.org/wiki/Fibonacci_number

[ ]:
```
# Ejemplo serie Fibonacci
# La suma de dos elementos adyacentes define el siguiente
a, b = 0, 1
while a < 1000:
print(a)
a, b = b, a + b
```

- La función factorial utiliza signo de exclamación `n!`
- Es igual al producto de todos los números naturales previos al número especificado.
- Ejemplos y explicación: https://es.wikihow.com/calcular-factoriales.

[ ]:
```
0! = 1
4! = 1 * 2 * 3 * 4
n! = 1 * 2 ** 3 * 4 * ... * n-1 * n
```

[ ]:
```
# Ejemplo Recursividad
def funcion_factorial(numero):
    if numero < 0:
        return None
    if numero > 1:
        numero = numero * funcion_factorial(numero - 1)
    print("Valor final ->", numero)
    return numero


funcion_factorial(5)
```

[ ]:
```
def funcion(a):
    if a > 30:
        return 3
```

```
    else:
        return a + funcion(a + 3)


print(funcion(25))
```

2.7 Secuencia y mutabilidad

- La secuencia es un tipo de dato en Python que puede almacenar cero o más de un valor (por ejemplo una lista) y que son secuencialmente examinados.
- El bucle *for* es una herramienta especialmente diseñada para iterar a través de las secuencias.
- La mutabilidad es una propiedad de cualquier tipo de dato en Python que permite cambiar libremente durante la ejecución de un programa.
- Existen dos tipos de datos en Python: mutables e inmutables.
- Los datos mutables pueden ser actualizados *in situ*:

[ ]:
```
list.append(1)
```

- Los datos inmutables no pueden ser modificados en el momento. Una tupla es un ejemplo de esto.
- Las tuplas utilizan paréntesis, pero es posible crear una tupla solo separando los valores por comas.

[4]:
```
tupla_vacia = ()
tupla_1 = (1, 3, 5, 10, 22)
tupla_2 = 10, 2, 5, 325, 250
tupla_1valor = (10, )
tuple_2valor = 10.,
```

- También se puede crear una tupla utilizando la función integrada *tuple()*.
- Es útil para convertir un iterable (lista, rango, cadena, etc.) en una tupla.

[ ]:
```
tupla = tuple((1, 2, 3, "cadena"))
print(tupla)

lista = [4, 5, 6]
print(lista)
print(type(lista))

tup = tuple(lista)
print(tup)
print(type(tup))
```

[ ]:
```
# Convertir tupla en lista
lista = list(tup)
print(type(lista))
```

2.8 Diccionarios

- Los diccionarios son colecciones indexadas de datos, mutables y (eran) desordenadas.
- A partir de Python 3.6 los diccionarios están ordenados de manera predeterminada por orden de inserción.
- Para comprobar si una clave existe en un diccionario, se emplea la palabra clave reservada *in*.
- Fuente: https://docs.python.org/es/3/library/stdtypes.html#dict

[ ]:
```
diccionario = {
    "nombre": "José",
    "apellido1": "Rodriguez",
    "apellido2": "López"
    }

# Buscar clave
if "nombre" in diccionario:
    print("Si")
else:
    print("No")
```

[ ]:
```
# Buscar valor
print("Rodriguez" not in diccionario.values())

# Buscar clave y valor
print(("nombre", "José") in diccionario.items())
```

- Para eliminar todos los elementos del diccionario:

[ ]:
```
diccionario.clear()
print(len(diccionario))

# Elimina el diccionario
del diccionario
```

- Para copiar un diccionario:

[1]:
```
diccionario = {"nombre": "José", "apellido1": "Rodriguez", "apellido2": "López"}
copia_diccionario = diccionario.copy()
diccionario.clear()
print(copia_diccionario)
```

```
{'nombre': 'José', 'apellido1': 'Rodriguez', 'apellido2': 'López'}
```

- Para unir dos diccionarios:

[ ]:
```
d1 = {"fruta": "naranja", "cantidad": "5", "precio": "3"}
d2 = {"nombre": "María", "apellido1": "Castillo", "apellido2": "Rodriguez"}
d3 = {}
```

```
for item in (d1, d2):
    d3.update(item)

print(d3)
```

[ ]:
```
d4 = {**d1, **d2}
print(d4)
```

[ ]:
```
d1 = {"nombre": "José", "apellido1": "Rodriguez", "apellido2": "López"}
d2 = {"fruta": "naranja", "cantidad": "5", "precio": "3"}
d5 = {}

d5.update(d1)
d5.update(d2)

print(d5)
```

- Fuente: https://peps.python.org/pep-0584/.

3 Módulos en Python

- Cuando se escribe un programa se suele dividir el mismo en varios ficheros para poder luego gestionarlo más fácilmente.
- Es común también escribir funciones que serán reutilizadas en diversas partes del programa.
- Python permite crear ficheros llamados módulos que se podrán utilizar en scripts o en las instancias interactivas del intérprete del lenguaje.

- Un módulo puede ser escrito en Python,
- Escrito en lenguaje C y cargado dinámicamente en tiempo de ejecución como por ejemplo *re (regular expression)*, o
- Ser un módulo integrado (*built-in*) que está intrínsecamente contenido en el intérprete, como el módulo *itertools*.

- En los tres casos es accedido de la misma manera utilizando la declaración *import*.

[ ]:
```
import NombreDelModulo
```

- Fuente: https://docs.python.org/es/3/tutorial/modules.html.

- Los módulos contienen definiciones y declaraciones de Python que son programadas en un fichero con extensión *.py*.
- Cuando el intérprete de Python lee el fichero fuente de un módulo primero establece algunas variables especiales como por ej. `__name__` y luego ejecuta el código fuente del mismo.
- Dentro del módulo, el nombre del mismo está disponible como el valor de la variable global `__name__`.

[ ]:
```
>>>__name__
'__main__'
```

- La variable local `__name__` en el módulo principal del programa es `__main__`.
- Cuando creamos un módulo, `__name__` toma el nombre del mismo.

[ ]:
```
# -*- coding: utf-8 -*-
# Ejemplo variables __name__ y __main__
# Fichero foo.py
import math


print("Antes import")
print("Antes función A")

def funcionA():
    print("Función A")


print("Antes función B")

def funcionB():
    print("Función B: {}".format(math.sqrt(100)))


print("Antes del control __name__")

if __name__ == '__main__':
    funcionA()
    funcionB()
print("Después del control __name__")
```

[ ]:
```
# Ejemplo variables __name__ y __main__
# Fichero foo_import.py
import foo
```

- Ejemplo traducido y adaptado de la pregunta *What does if* ***name*** *==* ***"main"****: do?*
- https://stackoverflow.com/questions/419163/what-does-if-name-main-do?page=1&tab=votes#tab-top

- Cuando un módulo es importado el intérprete primero busca por los módulos integrados de Python con ese nombre.
- Si no lo encuentra, busca por un fichero con ese nombre y extensión *.py* en una lista de directorios proporcionada por la variable *sys.path*.

- *Sys.path* es inicializada desde:
  - El directorio donde reside el script ejecutable (o el directorio actual si no se especifica un fichero),
  - PYTHONPATH (una lista de nombres de directorios, con la misma sintaxis que la variable de la shell PATH),
  - El valor predeterminado que depende de la instalación.

[ ]:
```
# Muestra la ruta
>>> import sys
>>> sys.path
```

- Una vez que el módulo se ha importado, se puede determinar la ubicación donde fue encontrado mediante el atributo `__file__`:

[ ]:
```
>>> import os
>>> os.__file__
```

- Un listado de los paquetes, módulos y bibliotecas más populares es ofrecido por la PSF:
- https://wiki.python.org/moin/UsefulModules.
- Los *Top 10* módulos del *Python Package Wiki*:
- https://package.wiki/.
- Índice de módulos de Python:
- https://docs.python.org/es/3/py-modindex.html.

3.1 Módulos estándar

- Python provee una biblioteca de módulos estándar denominada "Python Library Reference".
- Algunos módulos están integrados en el intérprete para proporcionar acceso a operaciones que no forman parte del núcleo del lenguaje.
- Estos módulos dependen de la plataforma subyacente, por ejemplo, el módulo *winreg* solo se proporciona en sistemas Windows.

[ ]:
```
# Muestra una lista de todos los módulos disponibles
>>> help('modules')
```

- La función integrada *dir()* es utilizada para descubrir que nombres son definidos por un módulo.
- Ella devuelve una lista ordenada de strings:

[ ]:
```
import sys
dir(sys)
```

- La función *dir()* de todas maneras no devuelve la lista de las funciones y variables integradas.
- Para esto, se utiliza el módulo estándar *builtins*:

[ ]:
```
import builtins
dir(sys)
```

- Para mostrar la ayuda de un módulo:

[ ]:
```
help(os)
```

3.2 La declaración import

- Para importar un módulo se puede:
- Utilizar la declaración *import* como se hizo anteriormente: *import sys*

- Luego se puede llamara al método utilizando un punto: *sys.path*
- Importar los distintos métodos del módulo individualmente: *from sys import platform*.

[ ]:
```
from sys import platform
platform
```

- Al importar un módulo se le puede agregar un alias:

[ ]:
```
# Importa el módulo Pandas con el alias 'pd'
import pandas as pd
serie = pd.Series([1, 3, 5, 8, 13])
print(serie)
```

3.3 Módulos y paquetes

- Un módulo es un conjunto de código fuente relacionado (variables, funciones y/o clases) guardado en un fichero con extensión *.py*.
- Un paquete es una colección de módulos.
- Una biblioteca o librería es una colección de paquetes.

- Para organizar los módulos de modo jerárquico Python utiliza el concepto de paquetes (packages).
- Una manera de entender este concepto es pensar en paquetes como directorios y módulos como ficheros:

[ ]:
```
paquete/                        Paquete Top-level
    __init__.py                 Inicializa el paquete
    subpaquete/                 Subpaquete para funcionalidad x
              __init__.py
              script_x1.py
              script_x2.py
              ...

    subpaquete2/                Subpaquete para funcionalidad y
              __init__.py
              script_y1.py
              script_y2.py
              ...
```

- Es importante notar que todos los paquetes son módulos pero no todos los módulos son paquetes.
- Los paquetes son un tipo especial de módulo: específicamente cualquier módulo que contiene un atributo `__path__` lo es.
- Todos los subpaquetes tienen un nombre separado por su paquete padre mediante un punto:

[ ]:
```
paquete.subpaquete
```

- Python cuenta además con un repositorio de paquetes mantenido por la PSF, el *Python Package Index (PyPI)*.

- Permite encontrar e instalar software desarrollado por la comunidad utilizando el comando *pip*.
- Se accede al mismo en la URL https://pypi.org/.
- Python Packaging User Guide: https://packaging.python.org/.
- Fuente: https://docs.python.org/es/3/tutorial/modules.html.

3.4 Herramientas de línea de comandos

- El soporte que tiene Python para trabajar con los comandos y argumentos de la línea de comando es *sys.argv* incluido en la biblioteca *sys*.
- https://docs.python.org/3/library/sys.html#sys.argv
- El módulo *getopt* a su vez, ayuda a *parsear* opciones de la línea de comando y sus argumentos.
- Los scripts de Python pueden ejecutarse con la orden: *python fichero.py* o agregar argumentos y opciones o párámetros como en: *python fichero.py argumento parámetros*

[ ]:
```
import sys


print("Nombre del script: ", (sys.argv[0]))
print("Argumentos del script: ", (sys.argv[1:]))
print('Cantidad de argumentos:', len(sys.argv), 'argumentos.')
print('Lista de argumentos:', str(sys.argv))
```

3.5 Argparse

- Python utiliza la biblioteca *argparse (argument parse)* para programar una interfaz utilizable en la línea de comando.
- Esta biblioteca permite usar argumentos de posición, personalizar caracteres, y utilizar sub-comandos entre otras cosas.
- La documentación se encuentra en: https://docs.python.org/es/3/library/argparse.html.
- El tutorial de la misma: https://docs.python.org/es/3/howto/argparse.html#id1.
- En la línea de comando se trabaja con argumentos separados por espacios que contienen, como en el siguiente ejemplo, un comando (*ls*), sus argumentos (*-la*), y parámetros (*/etc*):

[ ]:
```
ls -la /etc
```

[ ]:
```
# Fichero ejemplo_argparse.py
import argparse
import os
import sys


# Crea el parser
mi_parser = argparse.ArgumentParser(prog = 'ejemplo_argparse.py',
```

```
                                    description = 'Listado del contenido del␣
 ↪directorio',
                                    epilog = 'Muchas gracias.')

# Agrega los argumentos
mi_parser.add_argument('Path',
                       metavar = 'Ruta',
                       type = str,
                       help = 'La ruta al directorio')
```

[ ]:
```
# Ejecuta el método parse_args()
args = mi_parser.parse_args()

dir_ruta = args.Path

if not os.path.isdir(dir_ruta):
    print('Este directorio no existe')
    sys.exit()

print('\n'.join(os.listdir(dir_ruta)))
```

[ ]:
```
python ejemplo_argparse.py directorio
python ejemplo_argparse.py -h
```

3.6 Módulo Platform

- El módulo platform nos permite obtener información sobre el sistema operativo en el cual se está ejecutando python.
- La página de la documentación: https://docs.python.org/3/library/platform.html

[ ]:
```
import platform


print("Sistema operativo: ", platform.system())
print("Versión plataforma:", platform.release())
print("Versión SO: ", platform.version())
print("Identificación del SO: ", platform.release())
print("Arquitectura: ", platform.machine())
print("Procesador: ", platform.processor())
print("Versión del Kernel: ", platform.platform())
print("-------")

if platform.system() == 'Linux':
    print("Linux Rocks")
elif platform.system() == 'Darwin':
    print("Mac")
elif platform.system() == 'Win':
```

```
    print("Windows")
print("-------")
```

- Muestra también información sobre Python:

```
[ ]: print("Versión de Python: ", platform.python_version())
print("Compilación: ", platform.python_build())
print("Compilador: ", platform.python_compiler())
print("Implementación: ", platform.python_implementation())
print("-------")
```

3.7 Python webserver

- Python cuenta con un servidor web básico que permite hacer pruebas en un servidor web local sin tener que instalar Apache https://httpd.apache.org/ o Nginx https://www.nginx.com/.
- Para ejecutarlo, simplemente se escribe la orden:

```
[ ]: python3 -m http.server 8000
```

```
[ ]: python -m SimpleHTTPServer 8000 # Python 2
```

- Este servidor no es útil para ambientes de producción dado que no cuenta con funcionalidades para establecer una seguridad adecuada.

3.8 Conexión a bases de datos

3.9 Módulo sqlite3

- Python viene con la base de datos SQLite instalada por defecto.
- Sitio web y documentación de SQLite: https://www.sqlite.org/index.html.
- Para poder ver la base de datos en modo gráfico se puede descargar e instalar https://sqlitebrowser.org/
- En GNU/Linux o Mac viene ya instalado, para ver la versión:

```
[ ]: sqlite3 --version
```

- En las distribuciones Debian/Ubuntu si no está SQLite ya instalado, se instala con:

```
[ ]: sudo apt install sqlite3 libsqlite3-dev
```

- En Windows primero se debe descargar de https://www.sqlite.org/download.html los ficheros *sqlite-dll-win64-x64-3360000.zip* (o superior) y sqlite-tools-win32-x86-3360000.zip (o superior).
- Luego se los descomprimime en un directorio creado a tal fin, por ejemplo: `C:\sqlite3`.
- Finalmente se agrega la variable de entorno a la ruta (PATH):
- Presionar las teclas (Windows + E) para abrir el explorador de archivos.
- Clic DERECHO en “Este equipo” y seleccionar propiedades.

- Seleccionar Configuración avanzada del sistema.
- Clicar en Variables de entorno: Buscar la variable PATH, seleccionarla y Editar: Nuevo.
- Agregar la ruta del directorio anteriormente creado y aceptar los cambios.

- Ejemplo crear una base de datos y una tabla dentro de la misma:

[ ]:
```
import sqlite3


# Establecer la conexión a la base de datos
connection = sqlite3.connect('ejemplo.sqlite')

# Crear el acceso a los datos
cursor = connection.cursor()

# Crear la tabla
cursor.execute('CREATE TABLE IF NOT EXISTS ejemplo(nombre TEXT, ciudad TEXT,␣
 ↪telefono INTEGER)')

# Guardar los cambios y salir
connection.commit()
connection.close()
```

- Ejemplo conectarse a la base de datos y agregar datos a la tabla:

[ ]:
```
import sqlite3


# Establecer la conexión a la base de datos
connection = sqlite3.connect('ejemplo.sqlite')

# Crear el acceso a los datos
cursor = connection.cursor()

# Ejeccutar la sentencia SQL
cursor.execute("INSERT INTO ejemplo VALUES ('Carmen', 'Alicante', 456456456)")
cursor.execute("INSERT INTO ejemplo VALUES ('Juan', 'Zamora', 744522411)")

# Guardar los cambios y salir
connection.commit()
connection.close()
```

- Seleccionar todos los datos de una tabla mediante SQL:

[ ]:
```
import sqlite3


# Establecer la conexión a la base de datos
```

```
connection = sqlite3.connect('ejemplo.sqlite')

# Crear el acceso a los datos
cursor = connection.cursor()
cursor_object = connection.execute("SELECT * FROM ejemplo")

print(cursor_object.fetchall())
connection.close()
```

3.10 SQLAlchemy ORM

- SQLAlchemy https://www.sqlalchemy.org/ es una biblioteca de código abierto con herramientas SQL.
- Contiene su propio mapeador relacional de objetos para Python (ORM - *Object Relational Mapper*).
- Permite a los desarrolladores de aplicaciones conectarse a una base de datos relacional (PostgreSQL, Oracle, MariaDB, etc.).

- SQLAlchemy utiliza la API creada para Python DBAPI (DataBase API) para especificar cómo los módulos que se integran con bases de datos deben exponer sus interfaces a ellas.
- La documentación de esta API se encuentra en https://www.python.org/dev/peps/pep-0249/ y https://docs.sqlalchemy.org/en/14/index.html
- La lista de bases de datos soportadas en https://wiki.python.org/moin/DatabaseInterfaces.

[ ]:
```
pip install SQLAlchemy # Instalación de SQLAlchemy
```

[ ]:
```
# Fichero para crear DB: fichero_sql_tablas.py
from sqlalchemy import create_engine
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
# Métodos para relacionar la tabla
# from sqlalchemy.orm import relationship, ForeignKey


engine = create_engine('sqlite:///estudiantes.db', echo=True)
Base = declarative_base()


class Estudiante(Base):
    __tablename__ = "estudiante"

    id = Column(Integer, primary_key=True)
    usuario = Column(String)
    nombre = Column(String)
    apellido1 = Column(String)
    apellido2 = Column(String)
    universidad = Column(String)
```

```
    def __init__(self, usuario, nombre, apellido1, apellido2, universidad):
        self.usuario = usuario
        self.nombre = nombre
        self.apellido1 = apellido1
        self.apellido2 = apellido2
        self.universidad = universidad

# crea la tabla
Base.metadata.create_all(engine)
```

[ ]:
```
# Fichero para inserir datos en la DB: fichero_sql_datos.py
from sqlalchemy.orm import sessionmaker
from fichero_sql_tablas import Estudiante, create_engine


engine = create_engine('sqlite:///estudiantes.db', echo=True)

# crea una sesión
Session = sessionmaker(bind=engine)
session = Session()

# Crea las instancias
usuario = Estudiante("juan", "Juan", "Perez", "Lopez", "Complutense")
session.add(usuario)

usuario = Estudiante("Maria", "María", "García", "Gomez", "UPC")
session.add(usuario)

usuario = Estudiante("Beatriz", "Beatriz", "Suarez", "Gonzalez", "Carlos III")
session.add(usuario)

# commit a la base de datos
session.commit()
session.close()
```

[ ]:
```
# Fichero para consultar datos en la DB: fichero_sql_consulta.py
from sqlalchemy.orm import sessionmaker
from fichero_sql_tablas import Estudiante, create_engine


engine = create_engine('sqlite:///estudiantes.db', echo=True, future=True)

# Crea una sesion
Session = sessionmaker(bind=engine)
session = Session()

# Consulta a la base de datos
```

```
for Estudiante1 in session.query(Estudiante).order_by(Estudiante.id):
    print(Estudiante1.nombre, Estudiante1.apellido1, Estudiante1.apellido2)


session.close()
```

[ ]:
```
# Ejemplo filtrar base de datos
for Estudiante2 in session.query(Estudiante).filter(
        Estudiante.universidad == 'UPC'):
    print(Estudiante2.nombre, Estudiante2.apellido1, Estudiante2.apellido2)
```

- Tutoriales en el sitio web de SQLAlchemy:
  - ORM Quick Start https://docs.sqlalchemy.org/en/14/orm/quickstart.html.
  - Object Relational Tutorial (1.x API) https://docs.sqlalchemy.org/en/14/orm/tutorial.html.