

Diplomatura en Python:

Python nivel intermedio

Módulo II:

Nivel Intermedio II

Unidad 2:

Uso de ORM



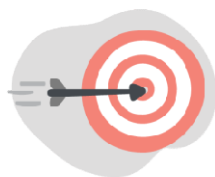
Presentación

Hasta ahora hemos trabajado con cadenas de texto sin entrar mucho en detalle en su trasfondo y nos hemos conectado con bases de datos como MySQL, SQLite3 o Shelves.

En el transcurso de esta unidad nos centraremos en dos puntos muy interesantes antes de comenzar a ver en las siguientes unidades como python funciona internamente.

En primer lugar utilizaremos un ORM (mapeo de objeto relacional) el cual permite convertir código de python en lenguaje de bases de datos. La ventaja fundamental es que podemos trabajar con python sin conocer el lenguaje de la base de datos. A modo de ejemplo podríamos trabajar con MySQL o SQLite3 sin tener que cambiar nuestro código.

En segundo lugar, profundizaremos en la forma en la cual python trabaja con diferentes formas de codificación de texto y en cómo nos permite pasar de una forma a la otra. Al final de la unidad comprenderemos que los strings pueden considerarse como una serie de caracteres, y que la forma en la cual estos caracteres son almacenados puede variar dependiendo del tipo de juego de caracteres que estemos considerando.



Objetivos

Logren aprender cómo trabajar con formatos diferentes utilizando Python 3.

Puedan pasar de un formato a otro y leer archivos con diferente tipo de contenido.



Bloques temáticos

- 1.- Uso de ORM
- 2.- Conceptos básicos.
- 3.- ASCII y Unicode.
- 4.- Strings.
- 5.- Argumentos de error en la decodificación.
- 6.- Argumentos de error durante la codificación.
- 7.- Trabajando con objetos de bytes.
- 8.- bytearray
- 9.- Uso de archivos de texto y binarios.

1.- Uso de ORM.

Un mapeador de objetos relacionales (ORM) es un programa que permite transformar datos almacenados en tablas de bases de datos relacionales en objetos de clases. En python existen varios ORM que podemos utilizar, como:

- El Django ORM
- Peewee
- SQLAlchemy
- PonyORM
- SQLAlchemy
- Tortoise ORM

En esta unidad veremos cómo trabajar con “Peewee”, el cual es un ORM de muy fácil implementación. Lo haremos mediante la creación de una aplicación que nos permita realizar un ABMC de datos. Esta aplicación tiene el objetivo de partir de una app conocida que hemos trabajado en el nivel intermedio y hacer un repaso de lo aprendido a la vez que introducimos nuevos conocimientos. Los puntos a repasar son:

- Uso de base de datos shelve
- Uso de base de datos SQLite3 o MySQL
- Creación de clases y objetos
- Uso de métodos y funciones.
- Importación de módulos.
- Estructuras de control y bucles

Recomendación Importante: El alumno puede utilizar esta aplicación para chequear sus conocimientos previos, si alguno de los puntos anteriores no ha quedado claro, se recomienda que el alumno cree una consulta en el foro y se lo haga saber al docente antes de seguir avanzando.

La vista de nuestra app

Nuestra app es bastante simple, consta de:

- Un título
- Una sección en la que se visualizan los datos
- Una sección de botones: Guardar, Eliminar, Modificar.
- Una sección destinada a modificar el tema de la aplicación.

Tarea POO

Ingrese sus datos

ID	Título	Fecha	Descripción	Estado
6	66	2019-12-17 23:28:43.809349	---	True
5	-qq	2019-12-17 23:26:52.047641	q	True
4	qq	2019-12-17 23:20:18.234462	---	True
3	---	2019-12-17 23:16:41.837949	---	True
2	Título 2	2019-12-17 22:59:50.495122	Descripción 2	True
1	Título1	2019-12-17 22:59:49.876334	Descripción 1	True

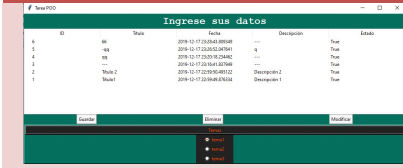
Guardar
Eliminar
Modificar

Temas

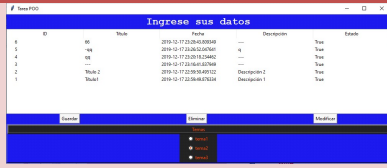
- tema1
- tema2
- tema3

Los diferentes temas se activan mediante botones del tipo “Radiobutton”:

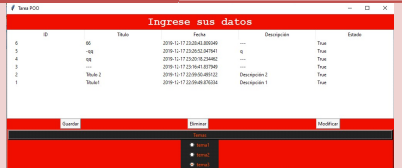
Tema 1



Tema 2



Tema 3



Los datos de los temas se han agregado en un módulo aparte y guardado en una base del tipo shelve.

1.1. Instalación de peewee

Para instalar “peewee” ejecutamos:

pip install peewee

1.2. Creación de estructura de base de datos

La estructura de la base de datos la realizamos en un módulo aparte:

1. En las primeras dos líneas de código importamos “peewee” y la librería “datetime” con la cual ya hemos trabajado en el nivel inicial.
2. La línea 5 nos permite utilizar el método “SqliteDatabase()” para crear una base de datos del tipo SQLite3, se puede consultar la bibliografía online, si deseamos trabajar con otro tipo de base de datos, desde manual online, disponible en el siguiente link:
<https://peewee.readthedocs.io/en/2.0.2/peewee/overview.html>
3. De la línea 7 a la línea 9, declaramos una clase dentro de la cual declaramos la base de datos con la cual vamos a trabajar.

4. De la línea 11 a la 15 declaramos la clase que va a ser mapeada por el ORM y transformada en una tabla de la base de datos. Cada atributo de la clase corresponde a una columna de la base de datos. Los distintos tipos de atributos los podemos encontrar en:

<https://peewee.readthedocs.io/en/2.0.2/peewee/fields.html>

Notar que con un simple parámetro como “unique = True” podemos hacer que un atributo del tipo “CharField()” el cual equivale a un campo de tipo string no pueda tener campos repetidos. Esto nos evita tener que implementar una regex para la validación del código.

5. En la línea 16 nos conectamos con la base de datos.
6. En la línea 17 agregamos la tabla a la base de datos.

```

base_datos.py
1  from peewee import *
2  import datetime
3
4  try:
5      db = SqliteDatabase('nivel_avanzado.db')
6
7      class BaseModel(Model):
8          class Meta:
9              database = db
10
11     class Noticia(BaseModel):
12         titulo = CharField(unique = True)
13         descripcion = TextField()
14         fecha = DateTimeField(default=datetime.datetime.now)
15         estado_de_publicacion = BooleanField(default=True)
16     db.connect()
17     db.create_tables([Noticia])
18
19 except:
20     print("mmmm")
  
```

1.3. Alta de registros

Realizar un alta de registros es muy fácil mediante el uso de “Peewee” ya que no necesitamos conocer el lenguaje de la base de datos que estamos utilizando, solo debemos crear un nuevo objeto de la clase que define la base de datos de la siguiente forma:

guardarModal.py

```
1 from tkinter import *
2 from guardar import *
3 from base_datos import *
4
5 def guarda(variables, popupGuardar, elobjeto):
6
7     popupGuardar.destroy()
8     lista = []
9     for variable in variables:
10         lista.append(variable.get())
11     noticia = Noticia()
12     noticia.titulo = lista[0]
13     noticia.descripcion = lista[1]
14     noticia.save()
15     elobjeto.mostrar()
```

Aquí estamos utilizando el módulo “guardarModal.py” de la aplicación creada en el módulo intermedio y estamos cambiando las líneas que van de la 11 a la 15. Básicamente lo que hacemos es crear un objeto de la clase Noticia, pasarle los datos a guardar y salvarlos mediante el método “save()”.

Notar que tanto el campo “fecha” como el campo “estado_de_publicacion” no es necesario que los agreguemos ya que le hemos puesto un valor por defecto en la declaración del atributo dentro de la clase.

1.4. Eliminación de registros

De forma análoga al punto anterior trabajamos ahora recuperando un objeto determinado y mediante el método “delete_instance()” lo borramos.

eliminarModal.py

```

1  from tkinter import *
2  from guardar import *
3  from base_datos import *
4
5  def elimina(variables, popupEliminar, elobjeto):
6      popupEliminar.destroy()
7      lista = []
8      for variable in variables:
9          lista.append(variable.get())
10
11     borrar = Noticia.get(Noticia.id == lista[0])
12     borrar.delete_instance()
13
14     elobjeto.mostrar()
```

1.5. Actualización de registros

actualizarModal.py

```

1  from tkinter import *
2  from guardar import *
3  from base_datos import *
4
5  def modifica(variables, popupModificar, elobjeto):
6      popupModificar.destroy()
7      lista = []
8      for variable in variables:
9          lista.append(variable.get())
10
11     actualizar = Noticia.update(titulo = lista[1], descripcion =
12                               lista[2]).where(Noticia.id == lista[0])
13     actualizar.execute()
14     elobjeto.mostrar()
```



De forma análoga a los pasos anteriores, ahora de la línea 11 a la 12 recuperamos el objeto a actualizar, le cambiamos los datos y luego mediante el método “execute()” actualizamos el registro.

2.- Conceptos básicos.

En esta unidad comenzaremos viendo algunos conceptos y definiciones básicas.

Byte

Un byte es un indicador de la cantidad memoria utilizada, la información es almacenada en ceros y unos, en donde cada elemento cero o uno es denominado bite. En la convención que vamos a ver, un byte podemos tomarlo como 8 bits. Dado que un bite posee solo dos estados (cero o uno) se utiliza con ellos aritmética de base 2. Antes de seguir con la definición de tipos de variables detengámonos un momento y aprendamos un poco sobre números binarios.

Números binarios.

Los números binarios se conforman con ceros y unos (0,1), en un circuito de lógica binaria un cero significa voltaje cero, y un uno voltaje máximo donde podemos simbolizar como apagado y prendido.

Los números son representados por potencia de dos (2), y veremos cómo se transforman estos en decimales. Partamos por escribir las potencias de dos en una tabla:

2 e+7	2 e+6	2 e+5	2 e+4	2 e+3	2 e+2	2 e+1	2 e+0	VALOR
128	64	32	16	8	4	2	1	

SUPONGAMOS QUE QUEREMOS CONVERTIR EL NÚMERO 220.

Colocaremos un 1 en el lugar de mayor valor que podamos sin pasarnos del valor pedido en este caso $2^7 = 128$, seguimos poniendo un 1 en el lugar de mayor valor siempre que no supere su suma $128 + 64 = 192$, si colocamos un 1 en la columna de $2^5 = 32$ nos pasaremos del valor pedido $128 + 64 + 32 = 224$, por lo tanto colocaremos un cero, tomamos el siguiente valor 16, su suma no alcanza el valor, su suma $128 + 64 + 16 = 208$ por lo tanto colocamos un 1, sumaremos la siguiente columna 8 tendremos $128 + 64 + 16 + 8 = 216$ no superamos el número colocamos un 1, el próximo es 4 sumándolo tendremos $128 + 64 + 16 + 8 + 4 = 220$ ya obtuvimos el valor no necesitamos sumar ningún valor, colocaremos un cero en la potencias siguientes.

2 e+7	2 e+6	2 e+5	2 e+4	2 e+3	2 e+2	2 e+1	2 e+0	VALOR
128	64	32	16	8	4	2	1	
1	1	0	1	1	1	0	0	220

A continuación podemos ver cuatro ejemplos:

2 e+7	2 e+6	2 e+5	2 e+4	2 e+3	2 e+2	2 e+1	2 e+0	VALOR
128	64	32	16	8	4	2	1	
1	0	0	1	0	1	1	0	150
0	1	1	1	0	0	0	1	113
1	0	1	1	0	1	1	1	183
0	0	1	0	0	1	0	1	37

Primer fila $128+16+4+2=150$

Segunda fila $64+32+16+1=113$

Tercera fila $128+32+16+4+2+1=183$

Cuarta fila $32+4+1=37$

Otra manera de obtener el binario es el siguiente, dividimos por dos el número y colocamos el resto (que puede ser cero o uno) a la derecha se repite la operación hasta que quede el ultimo resto que puede ser uno o cero.

220	0
110	0
55	1
27	1
13	1
6	0
3	1
1	1

Forma de leer el número binario

Nota: En internet podremos encontrar varios programas que convierten valores binarios a decimales, o decimales a binarios.

Veremos ahora como basado en la numeración binaria transformaremos estos para hacer una conversión hexadecimal. Partamos de la siguiente tabla en donde tenemos una equivalencia entre los números hexadecimales básicos y sus correspondientes números.

binario	hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Para formar el número hexadecimal a partir de un número binario, agruparemos el binario de a cuatro, veamos aprovechando los valores anteriores como quedaría su representación en hexadecimal:

Decimal	Binario	Hexadecimal
150	10010110	96 H
113	01110001	71 H
183	10110111	B7 H
37	00100101	25 H

3.- ASCII y Unicode

En 1968, se estandarizó el Código Estándar Americano para el Intercambio de Información, más conocido por su acrónimo ASCII. ASCII definió códigos numéricos para varios caracteres, con los valores numéricos que van del 0 al 127. Por ejemplo, la letra minúscula "a" se asigna a 97 como su valor de código.

ASCII era un estándar desarrollado en los Estados Unidos, por lo que sólo definía los caracteres sin acento. Había una "e", pero no una "é". Esto significa que los idiomas que requieren caracteres acentuados no se pueden representar fielmente en ASCII.

Durante un tiempo, las personas solo escribieron programas que no mostraban acentos.

En la década de 1980, casi todas las computadoras personales eran de 8 bits, lo que significa que los bytes podían contener valores que oscilaban entre 0 y 255. Los códigos ASCII solo subían a 127, por lo que algunas máquinas asignaban valores entre 128 y 255 a caracteres acentuados. Sin embargo, diferentes máquinas tenían códigos diferentes, lo que llevó a problemas para intercambiar archivos. Finalmente, surgieron varios conjuntos de valores de uso común para el rango de 128–255. Algunos eran estándares verdaderos, definidos por la Organización Internacional para la Estandarización, y otros eran convenciones de facto que fueron inventadas por una compañía u otra y lograron ponerse al día.

255 caracteres no son muchos. Por ejemplo, no se puede ajustar los caracteres acentuados utilizados en Europa occidental y el alfabeto cirílico utilizado para el ruso en el rango de 128–255 porque hay más de 128 caracteres de este tipo.

Puede escribir archivos usando diferentes códigos (todos sus archivos rusos en un sistema de codificación llamado KOI8, todos sus archivos franceses en un sistema de codificación diferente llamado Latin1), pero ¿qué sucede si desea escribir un documento francés que cita algún texto en ruso? En la década de 1980, la gente comenzó a querer resolver este problema y comenzó el esfuerzo de estandarización de Unicode.

Unicode comenzó utilizando caracteres de 16 bits en lugar de caracteres de 8 bits. 16 bits significa que tiene $2^{16} = 65,536$ valores distintos disponibles, lo que hace posible representar muchos caracteres diferentes de muchos alfabetos diferentes; un objetivo inicial era que Unicode contuviera los alfabetos de cada idioma humano. Resulta que incluso 16 bits no es suficiente para cumplir ese objetivo, y la especificación moderna de Unicode utiliza una gama más amplia de códigos, de 0 a 1.114.111 (0x10FFFF en la base 16).

Existe una norma ISO relacionada, ISO 10646. Unicode e ISO 10646 fueron originalmente esfuerzos separados, pero las especificaciones se fusionaron con la revisión 1.1 de Unicode.

Definiciones

Un carácter es el componente más pequeño posible de un texto. "A", "B", "C", etc., son caracteres diferentes. Así son 'È' y 'Í'. Los caracteres son abstracciones y varían según el idioma o el contexto del que estés hablando. Por ejemplo, el símbolo para ohmios (Ω) generalmente se dibuja de manera muy similar a la letra mayúscula omega (Ω) en el alfabeto griego (incluso pueden ser iguales en algunas fuentes), pero estos son dos caracteres diferentes que tienen significados diferentes.

El estándar de Unicode describe cómo se representan los caracteres mediante puntos de código. Un punto de código es un valor entero, generalmente denotado en la base 16.

Un carácter está representado en una pantalla o en papel por un conjunto de elementos gráficos que se llaman **glifos**. El glifo para una A mayúscula, por ejemplo, es dos trazos diagonales y un trazo horizontal, aunque los detalles exactos dependerán de la fuente que se use. La mayoría del código Python no necesita preocuparse por los glifos; averiguar el glifo correcto para mostrar es generalmente el trabajo de un kit de herramientas GUI (Interfaz gráfica) o un procesador de fuentes de una terminal.

Codificaciones

Las reglas para convertir una cadena Unicode en una secuencia de bytes se denominan codificación. UTF-8 es probablemente la codificación más comúnmente soportada.

4.- String

Los strings pueden considerarse como una serie de caracteres, la forma en la cual estos caracteres son almacenados puede variar dependiendo del tipo de juego de caracteres que estemos considerando. Cuando por ejemplo el texto es almacenado en archivos, el conjunto de caracteres determina su formato, estos conjuntos de caracteres son estándares que asignan códigos enteros a caracteres individuales para que puedan representarse en la memoria de la computadora.

A modo de ejemplo el estándar ASCII le asigna el valor 97 a la letra 'a', y el valor 97 se puede expresar en hexadecimales como el número 61 (0x61), el cual se puede almacenar en un solo byte en la memoria. Python cuenta con las funciones internas `ord()` y `chr()` para pasar de carácter a unicode y de unicode a carácter, y la función `hex()` para encontrar el número hexadecimal de un número decimal.

Nota: Los caracteres de la tabla ascii son considerados como un caso reducido de Unicode.

ascii.py

```
1 print(ord('a'))  
2 print(chr(97))  
3 print(hex(97))
```

Retorna:

97

a
0x61

Algunos alfabetos definen tantos caracteres que es imposible representar cada uno de ellos como un byte. Unicode permite más flexibilidad. El texto Unicode a veces se denomina cadenas de "caracteres amplias", porque los caracteres se pueden representar con varios bytes si es necesario. Unicode se usa normalmente en programas internacionalizados, para representar conjuntos de caracteres europeos, asiáticos y otros que no están en inglés y que tienen más caracteres de los que pueden representar los bytes de 8 bits. Los códigos de caracteres menores de 128 se representan como un solo byte; los códigos entre 128 y 0x7ff (2047) se convierten en 2 bytes, donde cada byte tiene un valor entre 128 y 255; y los códigos superiores a 0x7ff se convierten en secuencias de 3 o 4 bytes con valores entre 128 y 255.

Desde Python 3.0, el lenguaje presenta un tipo "str" que contiene caracteres Unicode, lo que significa que cualquier cadena creada con comillas simples, dobles o triples se almacena como Unicode. La codificación por defecto de Python es UTF-8, por lo que simplemente se puede incluir un carácter Unicode en un literal de cadena. Para cambiar el tipo de codificación se debe incluir en la primera o segunda línea del programa el siguiente código:

```
# -*- coding: <encoding name> -*-
```

En python a diferencia de otros lenguajes es posible utilizar caracteres especiales en identificadores:

unicode.py

```
1 canción = "Mi canción preferida"  
2 print(canción)
```

Otras codificaciones permiten conjuntos de caracteres más ricos de diferentes maneras. UTF-16 y UTF-32, por ejemplo, presentan un formato de texto con un tamaño fijo de 2 y 4 bytes por cada esquema de caracteres, respectivamente, incluso para caracteres que de otro modo podrían caber en un solo byte. Algunas codificaciones también pueden insertar prefijos que identifican el orden de los bytes. Para verlo podemos ejecutar el método de codificación de una cadena, que proporciona su formato de bytstring codificado bajo un esquema con nombre; una cadena ASCII

de dos caracteres es de 2 bytes en ASCII, Latin-1 y UTF-8, pero es mucho más amplia en UTF - 16 y UTF-32, e incluye bytes de encabezado:

formatos.py

```

1  mi_string = "as"
2
3  print(mi_string.encode('ascii'))
4  print(mi_string.encode('latin1'))
5  print(mi_string.encode('utf8'))
6  print(mi_string.encode('utf16'))
7  print(mi_string.encode('utf32'))
8
9  var = b'\xff\xfe\x00\x00a\x00\x00\x00s\x00\x00\x00'
10 print(var.decode('utf32'))
11
12 print("\N{GREEK CAPITAL LETTER DELTA}")
13 print("\u0394")
14 print("\U00000394")
  
```

Retorna:

```

b'as'
b'as'
b'as'
b'\xff\xfea\x00s\x00'
b'\xff\xfe\x00\x00a\x00\x00\x00s\x00\x00\x00'
Δ
Δ
Δ
  
```

Nota: La b inicial representa cadena de byte. Es posible utilizar también una B mayúscula.

En primer lugar, la sintaxis de los literales de bytes es prácticamente la misma que la de los literales de cadena, excepto que se agrega un prefijo b:

Comillas simples: **b'still permite "comillas dobles" incrustadas'**

Comillas dobles: **b "todavía permite comillas 'simples' incrustadas".**

Comillas triples: **b "'3 comillas simples' ", b "" "3 comillas dobles" ""**

Solo se permiten los caracteres ASCII en bytes literales (independientemente de la codificación del código fuente declarado). Cualquier valor binario superior a 127 debe ingresarse en bytes literales usando la secuencia de escape apropiada.

Al igual que con los literales de cadena, los literales de bytes también pueden usar un prefijo `r` para deshabilitar el procesamiento de secuencias de escape.

Nota: Para codificar un string utilizamos `string.encode()`, el paso inverso lo obtenemos con `byte.decode()`.

Nota: También puedes usar secuencias de escape en cadenas literales. En el ejemplo anterior hemos recuperado el carácter delta a modo de ejemplo.

Nota: Algunas codificaciones utilizan secuencias de bytes incluso más grandes para representar caracteres. Cuando sea necesario, puede especificar valores de punto de código Unicode de 16 y 32 bits para los caracteres en sus cadenas; como se mostró anteriormente, podemos usar `"\u ..."` con cuatro dígitos hexadecimales para el primero, y `"\U. ..."` con ocho dígitos hexadecimales para el último, y puede mezclarlos en literales con caracteres ASCII más simples libremente.

Estas codificaciones se aplican cuando el texto se almacena o transfiere externamente, en archivos y otros medios, en tanto que en la memoria Python siempre almacena cadenas de texto decodificadas en un formato de codificación neutral, que puede o no usar múltiples bytes para cada carácter. La forma en que Python realmente almacena el texto en la memoria es propensa a cambiar con el tiempo de forma de hacer el proceso más eficiente y veloz.

Python 3.X viene con tres tipos de objetos para representar cadenas de caracteres (strings): uno para datos textuales y dos para datos binarios:

- **str** para representar texto Unicode decodificado (incluido ASCII) (**IMMUTABLE**)
- **bytes** para representar datos binarios (incluido texto codificado) (**IMMUTABLE**)
- **bytearray**, una variante de bytes. (**MUTABLE**)

Los objetos de cadena de Python 3.X se originan cuando se llama a una función incorporada como `str` o `bytes`.

Todas las formas literales de cadenas (sea utilizando comillas simples, dobles o triples) generan un texto Unicode, pero si le agregamos una `b` o `B` delante, generamos un objeto de bytes de 8 bits. Si utilizamos la función `type()` podemos ver que el primer objeto es del tipo `'bytes'` y el segundo del tipo `'str'` en el caso de que queramos pasar

string_literal.py

```
1 Fruta1 = b'Manzana'
2 Fruta2 = 'Pera'
3 print(type(Fruta1))
4 print(type(Fruta2))
5 print(Fruta1.decode('ascii'))
6 print(type(Fruta1.decode('ascii')))
7 print(Fruta2.encode('ascii'))
8 print(type(Fruta2.encode('ascii')))
```

Retorna:

```
<class 'bytes'>
<class 'str'>
Manzana
<class 'str'>
b'Pera'
<class 'bytes'>
```

Nota: Como ya hemos visto antes hemos utilizado `encode()` y `decode()` para pasar de un tipo al otro.

Hay un par de puntos que debemos notar:

Primero: En el caso de trabajar con código de byte, cada uno de sus elementos son tratados como el correspondiente número en la tabla ASCII, es decir que si intentamos obtener el primer elemento de “Fruta1” nos retornará el número correspondiente a la letra “M” el cual es el 77.

Segundo: Ambos casos son inmutables por lo que no es posible por ejemplo asignarle un valor a una posición dada de “Fruta1” o “Fruta2”, ya que daría un error.

string_literal2.py

```
1 fruta1 = b'Manzana'
2 fruta2 = 'Pera'
```

```
3 print(fruta1[0])
4 print(fruta2[0])
```

Retorna:

```
77
P
```

5.- Argumentos de error en la decodificación.

El argumento de errores especifica la respuesta cuando la cadena de entrada no se puede convertir de acuerdo con las reglas de la codificación. Los valores legales para este argumento son:

- **strict** : Lanza una excepción del tipo `UnicodeDecodeError`.
- **replace**: Utiliza U+caracteres de reemplazo.
- **backslashreplace**: Inserta una secuencia de escape (`\`)
- **ignore**: Quita el carácter que da error del resultado.

argumentos_error.py

```
1 try:
2     print(b'\x80abc'.decode("utf-8", "strict"))
3 except UnicodeDecodeError:
4     print("Ha existido un error en la decodificación")
5
6 print(b'\x80abc'.decode("utf-8", "replace"))
7 print(b'\x80abc'.decode("utf-8", "backslashreplace"))
8 print(b'\x80abc'.decode("utf-8", "ignore"))
```

Retorna:

Ha existido un error en la decodificación

abc
 \x80abc
 abc

6.- Argumentos de error durante la codificación.

El método opuesto de `bytes.decode()` es `str.encode()`, que devuelve una representación de bytes de la cadena Unicode, codificada en la codificación solicitada.

El parámetro de errores es el mismo que el del método `decode()` pero admite algunos controladores más aparte de `strict`, `replace`, e `ignore`:

- **xmlcharrefreplace**: inserta una referencia de carácter XML
- **backslashreplace**: inserta un \ secuencia de escape uNNNN
- **namereplace**: inserta una secuencia de escape \ N {...}

Por ejemplo el `chr(40960)` retorna el carácter “Yi Syllable It” que no puede ser representado en código ascii por lo que al intentar pasar el símbolo a ascii nos retornará un error.

Unicode Decimal Code ꀀ



Symbol Name: Yi Syllable It
 Html Entity:
 Hex Code: ꀀ
 Decimal Code: ꀀ
 Unicode Group: Yi Syllables

<http://www.codetable.net/decimal/40960>



argumentos_error2.py

```
1 u = chr(40960) + 'abcd'
2 print(u)
3 print(u.encode('utf-8'))
4 try:
5     print(u.encode('ascii'))
6 except UnicodeEncodeError:
7     print('Se ha dado un error en la codificación')
8
9 print(u.encode('ascii', 'ignore'))
10 print(u.encode('ascii', 'replace'))
11 print(u.encode('ascii', 'xmlcharrefreplace'))
12 print(u.encode('ascii', 'backslashreplace'))
13 print(u.encode('ascii', 'namereplace'))
```

Retorna:

```
␣ abcd
b'\xea\x80\x80abcd'
Se ha dado un error en la codificación
b'abcd'
b'?abcd'
b'&#40960;abcd'
b'\ua000abcd'
b'\N{YI SYLLABLE IT}abcd'
```

Nota: Las rutinas de bajo nivel para registrar y acceder a las codificaciones disponibles se encuentran en el módulo “**codecs**”. La implementación de nuevas codificaciones también requiere la comprensión del módulo de codecs.

7. Trabajando con objetos de bytes

Formateo de datos

Los objetos de “bytes” admiten operaciones de formateo al igual que los “str” pero la forma en la cual trabajamos con ambos puede variar de versión en versión, en python 3.7 los objetos de “bytes” pueden ser formateados de la misma forma que str, pero debemos tener cuidado ya que al python nos retorna un str al realizar el formateo. Veamos como al imprimir en la línea 14 el tipo de objeto señalado por “a” nos retorna que es del tipo str aún cuando en el formateo mezclamos objetos de “bytes”:

argumentos_error2.py

```
1  mistr = 'Pera'
2  mistr2 = 'Coco'
3  print('%s y %s' %(mistr, mistr2))
4  print('{0} y {1}'.format(mistr, mistr2))
5
6  mibyte = b'Manzana'
7  mibyte2 = b'Papas'
8  print(type(mibyte))
9  print('%s y %s' %(mibyte, mibyte2))
10 print('{0} y {1}'.format(mibyte, mibyte2))
11 print(type(mibyte))
12 a = '%s y %s' %(mibyte, mibyte2)
13 print(a)
14 print(type(a))
```

Retorna:

```
Pera y Coco
Pera y Coco
<class 'bytes'>
b'Manzana' y b'Papas'
b'Manzana' y b'Papas'
```



```
<class 'bytes'>  
b'Manzana' y b'Papas'  
<class 'str'>
```

Reemplazar elementos

En el caso de reemplazar elementos de un objeto de tipo bytes, debemos anteponer la letra b o B,

reemplazar_en_objetos_byte.py

```
1 mibyte = b'Manzana'  
2 print(mibyte.replace(b'an',b'un'))
```

Retorna:

```
b'Munzuna'
```

Concatenar str y bytes

Si queremos concatenar objetos del tipo str y objetos de tipo bytes es necesario previamente convertir alguno de los dos al otro formato ya que no, no es posible realizar la operación.

concatenar.py

```
1 mistr = 'Pera'  
2 mibyte2 = B'Manzana'  
3 try:  
4     print(mibyte2 + mistr)  
5 except:  
6     print('No puedo concatenar un objeto de tipo str con uno de tipo bytes')
```

Retorna:

```
No puedo concatenar un objeto de tipo str con uno de tipo bytes
```



8.- bytearray

Trabajar con objetos del tipo bytearray es similar a trabajar con objetos del tipo bytes, con la diferencia de que los objetos bytearray son mutables y necesitan que especifiquemos el tipo de codificación que estamos utilizando. En el siguiente ejemplo utilizamos la codificación “latin1” y sustituimos “é” por “á”. Debemos conocer el valor numérico según la codificación para realizar la sustitución.

bytearray.py

```
1  mi_string = "Compré una Manzana"
2  mi_bytearray = bytearray(mi_string, 'latin1')
3  print(mi_bytearray)
4  print(mi_bytearray[5])
5  mi_bytearray[5] = 225
6  print(mi_bytearray)
7  print(mi_bytearray.decode('latin1'))
```

Retorna:

```
bytearray(b'Compr\xe9 una Manzana')
233
bytearray(b'Compr\xe1 una Manzana')
Comprá una Manzana
```

9.- Uso de archivos de texto y binarios.

El modo en el que se abre un archivo es crucial, ya que se determina qué tipo de objeto se utilizará para representar el archivo.

El modo de texto implica objetos de cadena y el modo binario implica objetos de bytes:

- Los archivos de modo de texto interpretan el contenido de los archivos de acuerdo con una codificación Unicode, ya sea el valor predeterminado o el nombre del tipo de codificación que le asignemos explícitamente.
- Los archivos de modo de texto también realizan traducciones de fin de línea universales: de forma predeterminada, todos los formularios de fin de línea se asignan al único carácter '\n' en su script, independientemente de la plataforma en la que lo ejecute.
- Los archivos de modo binario, en cambio, retornar el contenido del archivo sin formato, como una secuencia de enteros que representan valores de bytes, sin codificación o decodificación y sin traducciones de fin de línea. El segundo argumento determina si se desea texto o procesamiento binario, agregar una b a esta cadena implica un modo binario (por ejemplo, "rb" para leer archivos de datos binarios). El modo por defecto es "rt"; esto es lo mismo que "r", que significa entrada de texto.
- Los archivos de texto devuelven una cadena para las lecturas y esperan una para las escrituras, pero los archivos binarios devuelven bytes para las lecturas y esperan uno (o un bytearray) para las escrituras.

Veamos un ejemplo en el cual escribamos un archivo de texto y otro de bytes y recuperemos ambos casos para la lectura.



archivo1.py

```
1 import os
2 archivo1 = os.path.dirname(os.path.abspath(__file__))+"\\archivo1.txt"
3
4 open(archivo1, 'w').write('Escribo algo de texto\n')
5 print(open(archivo1, 'r').read())
6 print(open(archivo1, 'rb').read())
7
8 print('-----')
9 archivo2 = os.path.dirname(os.path.abspath(__file__))+"\\archivo2.txt"
10
11 open(archivo2, 'wb').write(b'Escribo algo de texto\n')
12 print(open(archivo2, 'r').read())
13 print(open(archivo2, 'rb').read())
```

Retorna:

Escribo algo de texto

b'Escribo algo de texto\r\n'

Escribo algo de texto

b'Escribo algo de texto\n'

Nota: Si queremos guardar la información en el archivo en un tipo específico de codificación, como podría ser latin1, debemos indicarlo en el momento de guardar los cambios así:

```
open('archivo3', 'w', encoding='latin-1').write('texto a grabar')
```



Bibliografía utilizada y sugerida

Libros y otros manuscritos:

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>