**CLASE II - CURSO DE PYTHON 2017**

**TEMARIO**:

* STRING
* LISTAS
* TUPLAS
* FILE I/O

**INTRODUCCIÓN**.-

El objetivo de la presente clase es obtener los fundamentos de las cadenas de texto, actualmente todo lo que se envia desde una computadora a otra es por medio de algún protocolo que para el entendimiento humano usa cadena de texto o “string”, de esta manera el programador y el usuario final puede entender la trama que se envia.

Una vez que en algún código de programación tengamos variables en abundancia y que presenten algo en comun es una buena practica de programación ordenarlos y para esto se deben de agrupar para que asi sea facil acceder a su información, este es el caso de las listas, tuplas y diccionarios que permitiran agrupar variables; debemos que tener en cuenta que todo en python son objetos y como tal presentan métodos que pueden acceder y modificar su contenido, en esta sección haremos uso de sus métodos para modificar y acceder a estas variables.

Luego de que el programador tenga listo su algoritmo y obtenga sus resultados dados generalmente surge la necesidad de almacenar su resultado y aqui es donde entra python y poderoso lenguaje de programación orientado para matematicos y cientificos que cuenta en su API ( interfaz de programación de aplicaciones) métodos para poder almacenar datos en diferentes formatos (.txt, .py, .csv, .xml, .json ,etc) y asi poder abri posteriormente estos archivos grabados por algun software determinado.

En la presente clase aprenderemos los fundamentos de las “cadenas”, “tuplas”, “listas” , “archivos” y desarrollaremos nuestro algoritmos y ejercicios en clase; sabemos que la documentación sobre estos es extensa y requiere de horas y horas para poder aprenderlos todos, la idea fundamental del curso es obtener la soltura para usar la sintaxis de python en cualquier algoritmo en el que se desea usar, la sintaxis de python nos iremos familiarizando poco a poco en clase.

**STRING EN PYTHON:**

## **Un tipo de datos compuesto**

Las cadenas son cualitativamente diferentes de los otros cuatro tipos (int, float, bool, NoneType) porque están hechas de piezas más pequeñas — los **caracteres**.

Los tipos que comprenden piezas mas pequeñas se llaman **tipos de datos compuestos**. Dependiendo de lo que hagamos, podemos tratar un tipo de datos compuesto como una única cosa o podemos acceder a sus partes. Esta ambigüedad es útil.

El operador corchete selecciona sólo un carácter de una cadena:

|  |
| --- |
| **>>>** fruta = "banana “  **>>>** letra = fruta[1] **>>> print** letra |

La expresión **fruta[1]** selecciona el carácter número 1 de **fruta**. La variable **letra** apunta al resultado. Cuando mostramos **letra**, nos encontramos “a” como resultado.

La primera letra de "**banana**" no es “**a”**. A no ser que usted sea un programador. Por perversas razones, los científicos de la computación siempre empiezan a contar desde cero. La 0-ésima letra (“cero- ésima”) de "**banana**" es “b”. La 1-ésima (“uno- ésima”) es **“a”**, y la 2-ésima (“dos-ésima”) letra es **“n”**.

Si quiere la cero-ésima letra de una cadena, simplemente pone 0, o cualquier expresión de valor 0, entre los corchetes:

|  |
| --- |
| **>>>** letra = fruta[0] **>>> print** letra b |

A la expresión entre corchetes se le llama **índice**. Un índice identifica a un miembro de un conjunto ordenado, en este caso el conjunto de caracteres de la cadena. El índice *indica* cuál quiere usted, de ahí el nombre. El índice puede ser cualquier expresión entera.

## **Longitud**

La función **len** devuelve el número de caracteres de una cadena:

|  |
| --- |
| **>>>** fruta = "banana" **>>>** len(fruta) 6 |

Para obtener la última letra de una cadena, puede sentirse tentado a probar algo como esto:

|  |
| --- |
| longitud = len(fruta) ultima = fruta[longitud] *# ¡ERROR!* |

Eso no funcionará. Esto provoca el error en tiempo de ejecución **IndexError: string index out of range**. La razón es que no hay una seis-ésima letra en "**banana**". Como empezamos a contar desde cero, las seis letras están numeradas del 0 al 5. Para obtener el último carácter tenemos que restar 1 a la **longitud**:

|  |
| --- |
| longitud = len(fruta)  ultima = fruta[longitud-1] |

Alternativamente podemos usar índices negativos, que cuentan hacia atrás desde el final de la cadena. La expresión **fruta[-1]** da la última letra, **fruta[-2]** da la penúltima, y así sucesivamente.

## **Recorrido y el bucle for**

Muchos cálculos implican procesar una cadena carácter por carácter. A menudo empiezan por el principio, seleccionan cada carácter por turno, hacen algo con él y siguen hasta el final. Este patrón de proceso se llama **recorrido**. Una forma de codificar un recorrido es con una sentencia **while**:

|  |
| --- |
| indice = 0 **while** indice < len(fruta):  letra = fruta[indice]  **print** letra  indice += 1 |

Este bucle recorre la cadena y muestra cada letra en una línea distinta. La condición del bucle es **indice &lt; len(fruta)**, de modo que cuando **indice** es igual a la longitud de la cadena, la condición es falsa y no se ejecuta el cuerpo del bucle. El último carácter al que se accede es el que tiene el índice **len(fruta)-1**, que es el último carácter de la cadena.

Usar un índice para recorrer un conjunto de valores es tan común que Python proporciona una sintaxis alternativa más simple — el bucle **for**:

|  |
| --- |
| **for** carac **in** fruta:  **print** carac |

Cada vez que recorremos el bucle, se asigna a la variable **carac** el siguiente carácter de la cadena. El bucle continúa hasta que no quedan más caracteres.

El ejemplo siguiente muestra cómo usar la concatenación y a un bucle *for* para generar una serie abecedaria. “**Abecedaria**” se refiere a la serie o lista en la que cada uno de los elementos aparece en orden alfabético. Por ejemplo, los mostremos los siguientes nombres Jack, Kack, Lack, Mack, Nack, Ouack, Pack, y Quack. Este bucle muestra esos nombres en orden:

|  |
| --- |
| prefijos = "JKLMNOPQ" sufijo = "ack"  **for** letra **in** prefijos:  **print** letra + sufijo |

La salida de este programa es:

|  |
| --- |
| Jack Kack Lack Mack Nack Oack Pack Qack |

Por supuesto, esto no es del todo correcto, porque “Ouack” y “Quack” no están correctamente escritos. Usted solucionará esto en un ejercicio más adelante.

## **Porciones de cadenas**

A la subcadena de una cadena se le llama **porción**. La selección de una porción es similar a la selección un carácter:

|  |
| --- |
| **>>>** s = "Pedro, Pablo, y Mario" **>>> print** s[0:5] Pedro **>>> print** s[7:12] Pablo **>>> print** s[16:21] Mario |

El operador **[n:m]** devuelve la parte de la cadena desde el “n-ésimo” carácter hasta el “m-ésimo”, incluyendo el primero pero excluyendo el último. Este comportamiento contradice a nuestra intuición; tiene más sentido si imagina los índices señalando *entre* los caracteres, como en el siguiente diagrama:

cadena 'banana'

Si omite el primer índice (antes de los dos puntos), la porción comienza al principio de la cadena. Si omite el segundo índice, la porción llega al final de la cadena. Así:

|  |
| --- |
| **>>>** fruta = "banana" **>>>** fruta[:3] 'ban' **>>>** fruta[3:] 'ana' |

¿Qué cree que significa **s[:]**?

## **Comparación de cadenas**

Los operadores de comparación trabajan sobre cadenas. Para ver si dos cadenas son iguales:

|  |
| --- |
| **if** palabra == "banana":  **print** "Sí, ¡no tenemos bananas!" |

Otras operaciones de comparación son útiles para poner palabras en orden alfabético:

|  |
| --- |
| **if** palabra < "banana":  **print** "Su palabra," + palabra + ", va antes de banana." **elif** word > "banana":  **print** "Su palabra," + palabra + ", va después de banana." **else**:  **print** "Sí, ¡no tenemos bananas!" |

Sin embargo, debería ser consciente de que Python no maneja las letras mayúsculas y minúsculas como lo hace la gente. Todas las letras mayúsculas van antes de las letras minúsculas. Por lo tanto:

**Su palabra, Cebra, va antes de banana.**

Una forma común de abordar este problema es convertir las cadenas a un formato estándar — como pueden ser las minúsculas— antes de realizar la comparación. Un problema mayor es hacer que el programa se de cuenta de que las cebras no son frutas.

## **Las cadenas son inmutables**

Es tentador usar el operador **[]** en el lado izquierdo de una asignación, con la intención de cambiar un carácter en una cadena. Por ejemplo:

|  |
| --- |
| saludo = "¡Hola todo el mundo!" saludo[2] = 'L' *# ¡ERROR!* **print** saludo |

*Nota del traductor:* De acuerdo con la codificación de caracteres en utf-8 ó latin-1 (véase la Nota del traductor del capítulo 2) el carácter de admiración, ‘¡’, en la variable **saludo** ocupa dos posiciones, de ahí que la letra ‘H’ esté localizada en el índice 2.

En lugar de presentar la salida **¡Lola todo el mundo!**, este código presenta el siguiente error en tiempo de ejecución **TypeError: 'str' object does not support itemassignment.**

Las cadenas son **inmutables**, lo que significa que no puede cambiar una cadena existente. Lo más que puede hacer es crear una nueva cadena que sea una variación de la original:

|  |
| --- |
| saludo = "¡Hola todo el mundo!" nuevoSaludo = '¡' + 'L' + saludo[3:] **print** nuevoSaludo |

Aquí la solución es concatenar la apertura del signo de admiración y una nueva primera letra a una porción de **saludo**. Esta operación no tiene efecto sobre la cadena original.

## **El operador in**

El operador **in** prueba si una cadena es una subcadena de otra:

|  |
| --- |
| **>>>** 'z' **in** 'manzana' True **>>>** 'i' **in** 'manzana' False **>>>** 'nz' **in** 'manzana' True **>>>** 'az' **in** 'manzana' False |

Observe que una cadena es una subcadena de si misma:

|  |
| --- |
| **>>>** 'a' **in** 'a' True **>>>** 'manzana' **in** 'manzana' True |

Combinando el operador **in** con la concatenación de cadenas usando **+**, podemos escribir una función que elimina todas las vocales de una cadena:

|  |
| --- |
| **def** elimina\_vocales(s):  vocales = "aeiouAEIOU"  s\_sin\_vocales = ""  **for** letra **in** s:  **if** letra **not** **in** vocales:  s\_sin\_vocales += letra  **return** s\_sin\_vocales |

Pruebe esta función para confirmar que realiza lo que queremos que haga.

## **Una función encuentra**

¿Qué hace la siguiente función?

|  |
| --- |
| def encuentra(cadena, carac):  indice = 0  while indice < len(cadena):  if cadena[indice] == carac:  return indice  indice += 1  return -1 |

En cierto sentido, **encuentra** es lo contrario del operador **[]**. En lugar de tomar un índice y extraer el carácter correspondiente, toma un carácter y encuentra el índice donde aparece el carácter. Si el carácter no se encuentra, la función devuelve **-1**.

Este es el primer ejemplo que hemos visto de una sentencia **return dentro de un bucle. Si cadena[indice] == carac**, la función devuelve inmediatamente el índice, escapando del bucle prematuramente.

Si el carácter no aparece en la cadena, entonces el programa sale del bucle normalmente y devuelve **-1**.

Este patrón de computación se llama a veces un recorrido “eureka” porque tan pronto como encontramos lo que buscamos, podemos gritar “¡Eureka! ” y dejar de buscar.

**&lt; menor que: <**

**&gt; mayor que: >**

## **Iterando y contando**

El siguiente programa cuenta el número de veces que aparece la letra **a** en una cadena:

|  |
| --- |
| fruta = "banana" cuenta = 0 **for** carac **in** fruta:  **if** carac == 'a':  cuenta += 1 **print** cuenta |

## **Parámetros opcionales**

Para encontrar las posiciones de la segunda o tercera ocurrencia de un carácter en una cadena, podemos modificar la función **encuentra**, agregando un tercer parámetro para la posición inicial en la cadena de búsqueda:

|  |
| --- |
| **def** encuentra2(cadena, carac, inicio):  indice = inicio  **while** indice < len(cadena):  **if** cadena[indice] == carac:  **return** indice  indice += 1  **return** -1 |

La llamada a **encuentra2('banana', 'a', 2)** ahora devuelve **3**, el índice de la segunda ‘a’ en ‘banana’.

Mejor aún, podemos combinar **encuentra** y **encuentra2** usando un **parámetro opcional**:

|  |
| --- |
| **def** encuentra(cadena, carac, inicio=0):  indice = inicio  **while** indice < len(cadena):  **if** cadena[indice] == carac:  **return** indice  indice += 1  **return** -1 |

La llamada a **encuentra('banana', 'a', 2)** en esta versión de **encuentra** se comporta precisamente como **encuentra2**, mientras que, en la llamada a **encuentra('banana','a')**, el parámetro **inicio** se establecerá en el **valor por defecto**, esto es, **0**.

Agregar otro parámetro opcional a **encuentra** hace que busque tanto hacia adelante como hacia atrás:

|  |
| --- |
| **def** encuentra(cadena, carac, inicio=0, paso=1):  indice = inicio  **while** 0 <= indice < len(cadena):  **if** cadena[indice] == carac:  **return** indice  indice += paso  **return** -1 |

Asignar un valor de **-1** al **paso** hará que busque en dirección al inicio de la cadena en lugar de hacerlo hacia el final. Note que, para realizar este cambio, necesitamos verificar en el bucle **while** el límite inferior del **índice** así como el límite superior.

## **Formateo de cadenas**

La forma más concisa y poderosa de dar formato a una cadena en Python es usar el *operador de formateo de cadenas*, **%,** junto con las operaciones de formateo de cadenas de Python. Para observar cómo funciona, empecemos con algunos ejemplos:

|  |
| --- |
| **>>>** "Su nombre es *%s*." % "Arturo" 'Su nombre es Arturo.' **>>>** nombre = "Alicia" **>>>** edad = 10 **>>>** "Soy *%s* y tengo *%d* años de edad." % (nombre, edad) 'Soy Alicia y tengo 10 años de edad.' **>>>** n1 = 4 **>>>** n2 = 5 **>>>** "2\*\*10 = *%d* y *%d* \* *%d* = *%f*" % (2\*\*10, n1, n2, n1 \* n2) '2\*\*10 = 1024 y 4 \* 5 = 20.000000' >>> |

La sintaxis para las operaciones de formateo de cadenas luce así:

|  |
| --- |
| "<FORMATO>" % (<VALORES>) |

Ésta comienza con un *formato* que contiene una secuencia de caracteres y *especificaciones de conversión*. Las especificaciones de conversión inician con el operador **%.** Tras el formato de cadena está un solo **%** y después una secuencia de valores, *uno por especificación de conversión*, separados por comas y encerrados entre paréntesis. Los paréntesis son opcionales si hay solamente un valor.

En el primero de los anteriores ejemplos, hay una única especificación de conversión, **%s**, que indica una cadena. El único valor, "**Arturo**", le corresponde a la especificación de conversión, y no está entre paréntesis.

En el segundo ejemplo, **nombre** tiene el valor de cadena, "**Alicia**", y **edad** tiene el valor entero, **10**. Estos corresponden a las especificaciones de conversión **%s y %d**. La **d** en la segunda especificación de conversión indica que el valor es un entero decimal.

En el tercer ejemplo las variables **n1** y **n2** tienen los valores enteros **4** y **5** respectivamente. Hay cuatro especificaciones de conversión en el formato de la cadena: tres con **%d** y una con **%f**. La **f** indica que el valor debe ser representado como un número de punto flotante. Los cuatro valores que corresponden a las cuatro especificaciones de conversión son: **2\*\*10, n1, n2, y n1 \* n2.**

**s**, **d**, y **f** son todos los tipos de conversión que necesitaremos.

El siguiente ejemplo ilustra la verdadera utilidad del formateo de cadenas:

|  |
| --- |
| i = 1 **print** "i**\t**i\*\*2**\t**i\*\*3**\t**i\*\*5**\t**i\*\*10**\t**i\*\*20" **while** i <= 10:  **print** i, '**\t**', i\*\*2, '**\t**', i\*\*3, '**\t**', i\*\*5, '**\t**', i\*\*10, '**\t**', i\*\*20  i += 1 |

Este programa muestra una tabla de varias potencias de los números del 1 al 10. En su forma actual, la tabla depende del carácter de tabulación (**\t**) para alinear las columnas de valores, pero esto se distorsiona cuando los valores en la tabla son más largos que el ancho de 8 del carácter de tabulación:

|  |
| --- |
| i i\*\*2 i\*\*3 i\*\*5 i\*\*10 i\*\*20 1 1 1 1 1 1 2 4 8 32 1024 1048576 3 9 27 243 59049 3486784401 4 16 64 1024 1048576 1099511627776 5 25 125 3125 9765625 95367431640625 6 36 216 7776 60466176 3656158440062976 7 49 343 16807 282475249 79792266297612001 8 64 512 32768 1073741824 1152921504606846976 9 81 729 59049 3486784401 12157665459056928801 10 100 1000 100000 10000000000 100000000000000000000 |

Una posible solución sería cambiar el ancho del carácter de tabulación, pero la primera columna ya tiene más espacio del que necesita. La mejor solución es sería establecer el ancho de cada columna independientemente. Como probablemente ya ha adivinado, el formateo de cadenas proporciona la solución:

|  |
| --- |
| i = 1 **print** "*%-4s%-5s%-6s%-8s%-13s%-15s*" % \  ('i', 'i\*\*2', 'i\*\*3', 'i\*\*5', 'i\*\*10', 'i\*\*20') **while** i <= 10:  **print** "*%-4d%-5d%-6d%-8d%-13d%-15d*" % (i, i\*\*2, i\*\*3, i\*\*5, i\*\*10, i\*\*20)  i += 1 |

La ejecución de esta versión produce la siguiente salida:

|  |
| --- |
| i i\*\*2 i\*\*3 i\*\*5 i\*\*10 i\*\*20 1 1 1 1 1 1 2 4 8 32 1024 1048576 3 9 27 243 59049 3486784401 4 16 64 1024 1048576 1099511627776 5 25 125 3125 9765625 95367431640625 6 36 216 7776 60466176 3656158440062976 7 49 343 16807 282475249 79792266297612001 8 64 512 32768 1073741824 1152921504606846976 9 81 729 59049 3486784401 12157665459056928801 10 100 1000 100000 10000000000 100000000000000000000 |

El **-** después de cada **%** en la especificación de conversión indica justificación a la izquierda. Los valores numéricos especifican la longitud mínima, así que **%-13d** es un número justificado a la izquierda de por lo menos 13 caracteres de ancho.

**Listas**

Presentaremos ahora una nueva estructura de datos: la *lista*. Usaremos listas para poder modelar datos compuestos pero cuya cantidad y valor varían a lo largo del tiempo. Son secuencias *mutables* y vienen dotadas de una variedad de operaciones muy útiles.

La notación para lista es una secuencia de valores encerrados entre corchetes y separados por comas. Por ejemplo, si representamos a los alumnos mediante su número de padrón, se puede tener una lista de inscritos en la materia como la siguiente: [78455, 89211, 66540, 45750]. Al abrirse la inscripción, antes de que hubiera inscritos, la lista de inscritos se representará por una lista vacía: [].

* **Longitud de la lista. Elementos y segmentos de listas**

Como a las secuencias ya vistas, a las listas también se les puede aplicar la función **len()** para conocer su longitud.

Para acceder a los distintos elementos de la lista se utilizará la misma notación de índices de cadenas y tuplas, con valores que van de 0 a la longitud de la lista −1.

|  |
| --- |
| **>>>** xs=[78455, 89211, 66540, 45750] **>>>** xs[0] 78455 **>>>** len(xs) 4 **>>>** xs[4] Traceback (most recent call last):  File "<stdin>", line 1, in <module> IndexError: list index out of range **>>>** xs[3] 45750 |

Para obtener una sublista a partir de la lista original, se utiliza la notación de rangos, como en las otras secuencias.

Para obtener la lista que contiene sólo a quién se inscribió en segundo lugar podemos escribir:

|  |
| --- |
| **>>>** xs[1:2] [89211] |

Para obtener la lista que contiene al segundo y tercer inscritos podemos escribir:

|  |
| --- |
| **>>>** xs[1:3] [89211, 66540] |

Para obtener la lista que contiene al primero y segundo inscritos podemos escribir:

|  |
| --- |
| **>>>** xs[:2] [78455, 89211] |

* **Cómo mutar listas**

Dijimos antes que las listas son secuencias mutables. Para lograr la mutabilidad Python provee operaciones que nos permiten cambiarle valores, agregarle valores y quitarle valores.

Para cambiar una componente de una lista, se selecciona la componente mediante su índice y se le asigna el nuevo valor:

|  |
| --- |
| **>>>** xs[1]=79211 **>>>** xs [78455, 79211, 66540, 45750] |

Para agregar un nuevo valor al final de la lista se utiliza la operación **append()**. Escribimos xs.append(47890) para agregar el padrón 47890 al final.

|  |
| --- |
| **>>>** xs.append(47890) **>>>** xs [78455, 79211, 66540, 45750, 47890] |

Para insertar un nuevo valor en la posición cuyo índice es k (y desplazar un lugar el resto de la lista) se utiliza la operación **insert()**.

Escribimos xs.insert(2, 54988) para insertar el padrón 54988 en la tercera posición de xs.

|  |
| --- |
| **>>>** xs.insert(2, 54988) **>>>** xs [78455, 79211, 54988, 66540, 45750, 47890] |

**Nota** Las listas no controlan si se insertan elementos repetidos, si necesitamos exigir unicidad, debemos hacerlo mediante el código de nuestros programas.

|  |
| --- |
| **>>>** xs.insert(1,78455) **>>>** xs [78455, 78455, 79211, 54988, 66540, 45750, 47890] |

Para eliminar un valor de una lista se utiliza la operación **remove**().

Escribimos xs.remove(45750) para borrar el padrón 45750 de la lista de inscritos:

|  |
| --- |
| **>>>** xs.remove(45750) **>>>** xs [78455, 78455, 79211, 54988, 66540, 47890] |

Si el valor a borrar está repetido, se borra sólo su primera aparición:

|  |
| --- |
| **>>>** xs.remove(78455) **>>>** xs [78455, 79211, 54988, 66540, 47890] |

**Advertencia** Si el valor a borrar no existe, se produce un error:

* **Cómo buscar dentro de las listas**

Queremos poder formular dos preguntas más respecto de la lista de inscritos:

* ¿Está la persona cuyo padrón es v inscripta en esta materia?
* ¿En qué orden se inscribió la persona cuyo padrón es v?.

Veamos qué operaciones sobre listas se pueden usar para lograr esos dos objetivos:

Para preguntar si un valor determinado es un elemento de una lista usaremos la operación **in**:

|  |
| --- |
| **>>>** xs [78455, 79211, 54988, 66540, 47890] **>>>** 78 in xs False **>>>** 66540 in xs True |

**Nota** Esta operación se puede utilizar para todas las secuencias, incluyendo tuplas y cadenas

Para averiguar la posición de un valor dentro de una lista usaremos la operación **index**().

|  |
| --- |
| **>>>** xs.index(78455) 0 **>>>** xs.index(47890) 4 |

**Advertencia** Si el valor no se encuentra en la lista, se producirá un error:

Si el valor está repetido, el índice que devuelve es el de la primera aparición:

|  |
| --- |
| **>>>** ys=[10,20,10] **>>>** ys.index(10) 0 |

**Nota** Esta operación está disponible en cadenas, pero no en tuplas.

Para iterar sobre todos los elementos de una lista usaremos una construcción **for**:

|  |
| --- |
| **>>>** zs = [5, 3, 8, 10, 2] **>>>** for x in zs: ... print x ... 5 3 8 10 2 |

**tuplas**

Una tupla es una lista inmutable. Una tupla no puede modificarse de ningún modo después de su creación.

|  |
| --- |
| **>>>** t = ("a", "b", "mpilgrim", "z", "example") **>>>** t ('a', 'b', 'mpilgrim', 'z', 'example') **>>>** t[0]  'a' **>>>** t[-1]  'example' **>>>** t[1:3]  ('b', 'mpilgrim') |

|  |  |
| --- | --- |
|  | * Una tupla se define del mismo modo que una lista, salvo que el conjunto se encierra entre paréntesis en lugar de entre corchetes. |
|  | * Los elementos de una tupla tienen un orden definido, como los de una lista. Las tuplas tienen primer índice 0, como las listas, de modo que el primer elemento de una tupla no vacía es siempre t[0], |
|  | * Los índices negativos cuentan desde el final de la tupla, como en las listas. |
|  | * Las porciones funcionan como en las listas. Advierta que al extraer una porción de una lista, se obtiene una lista nueva; al extraerla de una tupla, se obtiene una tupla nueva. |

* **Las tuplas no tienen métodos**

|  |
| --- |
| **>>>** t ('a', 'b', 'mpilgrim', 'z', 'example')  **>>>** t.append("new")  Traceback (innermost last):  File "<interactive input>", line 1, in ? AttributeError: 'tuple' object has no attribute 'append'  **>>>** t.remove("z")  Traceback (innermost last):  File "<interactive input>", line 1, in ? AttributeError: 'tuple' object has no attribute 'remove'  **>>>** t.index("example") Traceback (innermost last):  File "<interactive input>", line 1, in ? AttributeError: 'tuple' object has no attribute 'index'  **>>>** "z" in t  1 |

|  |  |
| --- | --- |
|  | * No pueden añadirse elementos a una tupla. Las tuplas no tienen los métodos append ni extend. |
|  | * No pueden eliminarse elementos de una tupla. Las tuplas no tienen los métodos remove ni pop. |
|  | * No pueden buscarse elementos en una tupla. Las tuplas no tienen el método index. |
|  | Se puede, no obstante, usar in para ver si un elemento existe en la tupla. |

* **Entonces, ¿para qué sirven las tuplas?**

Las tuplas son más rápidas que las listas. Si está usted definiendo un conjunto constante de valores y todo lo que va a hacer con él es recorrerla, utilice una tupla en lugar de una lista.pueden utilizarse como claves en un diccionario, pero las listas no.

# **Manejo de Archivos**

## **¿Cómo abrimos un archivo?**

Antes de que podamos leer o escribir un archivo, hay que abrirlo con la función de Python **open**(). Esta función crea un objeto **archivo** que se utiliza para llamar a otros métodos de apoyo asociados. Veamos cual es la sintaxis:

|  |
| --- |
| objecto\_archivo = **open**(file\_name [, access\_mode][, buffering]) |

Describamos un poco los argumentos de la función:

* **file\_name** - Nombre del archivo que se desea acceder.
* **access\_mode** - Determina el modo en el que el archivo tiene que ser abierto, es decir. leer, escribir, etc. El modo de acceso de archivos por defecto es de **lectura**.
* **buffering** - Si el valor de búfer se establece en 0, ningún almacenamiento temporal se llevará a cabo. Si el valor es 1, el búfer se realizara por linea.

Aquí está una lista con los diferentes modos de apertura de un archivo:

|  |  |
| --- | --- |
| **MODOS** | **Descripción** |
| r | Abre un archivo de sólo lectura. El puntero del archivo se coloca en el principio del archivo. Este es el modo predeterminado. |
| rb | Abre un archivo de sólo lectura en formato binario. |
| r+ | Abre un archivo para lectura y escritura. El puntero del archivo estará en el principio del archivo. |
| rb+ | Abre un archivo para la lectura y la escritura en formato binario. El puntero del archivo estará en el principio del archivo. |
| w | Abre un archivo para escribir solamente. Sobrescribe el archivo si el archivo existe. Si el archivo no existe, se crea un nuevo archivo para escritura. |
| wb | Abre un archivo para escribir sólo en formato binario. Sobrescribe el archivo si el archivo existe. Si el archivo no existe, se crea un nuevo archivo para escritura. |
| w+ | Abre un fichero para escritura y lectura. Sobrescribe el archivo existente si existe el archivo. Si el archivo no existe, se crea un nuevo archivo para la lectura y la escritura. |
| wb+ | Abre un archivo, tanto para la escritura y la lectura en formato binario. Sobrescribe el archivo existente si existe el archivo. Si el archivo no existe, se crea un nuevo archivo para la lectura y la escritura. |

## **¿Cuáles son los atributos más importantes de un archivo?**

Una vez que hemos abierto el archivo, podemos obtener la información relacionada a el mismo. Veamos una lista con todos los atributos más importantes relacionados al tipo de objeto:

|  |  |
| --- | --- |
| **Atributo** | **Descripción** |
| objeto\_archivo.closed | Devuelve true si el archivo está cerrado, false en caso contrario. |
| objeto\_archivo.mode | Devuelve el modo de acceso con el que se abrió el archivo. |
| objeto\_archivo.name | Devuelve el nombre del archivo. |

Veamos un ejemplo sobre como utilizar todo lo que hemos visto hasta ahora:

|  |
| --- |
| fo = open("labotec.txt", "wb") print "Nombre del archivo : ", fo.name print "Cerrado o no : ", fo.closed print "Modo de apertura : ", fo.mode |

Si ejecutamos el script anterior debemos obtener el siguiente resultado:

|  |
| --- |
| **Nombre del archivo:** labotec.txt **Cerrado o no :** False **Modo de apertura :** wb |

* **¿Cómo cerramos un archivo?**

Para cerrar un archivo usamos el método close(), el cual limpia cualquier información que no haya sido escrita y cierra el archivo. Después que este método es ejecutado no podemos ejecutar ninguna escritura.

Python automáticamente cierra el archivo una vez que la referencia del objeto es asignada a un nuevo archivo.

Es una buena práctica hacer uso del método **close()** para cerrar los archivos una vez que terminemos de trabajar con ellos.

Veamos la sintaxis que debemos usar:

|  |
| --- |
| objecto\_archivo.close(); |

Veamos como usar **close**() con un ejemplo:

|  |
| --- |
| fo = open("labotec.txt", "wb") print "Nombre del archivo: ", fo.name  fo.close() |

Si ejecutamos el script anterior obtenemos:

|  |
| --- |
| Nombre del archivo: labotec.txt |

## **¿Cómo escribir un archivo?**

En Python para escribir información en un archivo usamos el método write().

Es importante tener en cuenta que las cadenas de caracteres(strings) de Python pueden tener datos binarios y no sólo de texto.

El método **write**() no añade un carácter de nueva línea (**'\ n'**) al final de la cadena.

Veamos la sintaxis para poder escribir en un archivo:

|  |
| --- |
| objecto\_archivo.write(string); |

Observemos que string es el texto que queremos escribir en el archivo.

Veamos como usar **write**() con un ejemplo:

|  |
| --- |
| fo = open("/tmp/labotec.txt", "wb") fo.write( "Quiero aprender a programar Python.\nRapido ");  fo.close() |

Si ejecutamos el script anterior obtenemos:

|  |
| --- |
| Quiero aprender a programar Python.  Rapido |

## **¿Cómo leer un archivo?**

En Python para leer información de un archivo usamos el método **read**().

Es importante tener en cuenta que las cadenas de caracteres(strings) de Python pueden tener datos binarios y no sólo de texto.

Veamos la sintaxis para poder leer un archivo:

|  |
| --- |
| objecto\_archivo.read([count]); |

Observemos que count es el numero de bytes a leer del archivo.

Veamos como usar **read**() con un ejemplo:

|  |
| --- |
| fo = open("/tmp/labotec.txt", "r+") str = fo.read(10); print "La lectura es : ", str  fo.close() |

Si ejecutamos el script anterior obtenemos:

|  |
| --- |
| La lectura es : Quiero apr |

Es posible, además, obtener todas las líneas del archivo utilizando una sola llamada a función:

|  |
| --- |
| líneas = archivo.readlines() |

En este caso, la variable líneas tendrá una lista de cadenas con todas las líneas del archivo, también se puede usar **readline()** lectura linea a linea.

## **¿Cómo renombramos un archivo?**

En Python para renombrar un archivo usamos el método rename().

Veamos la sintaxis para poder renombrar un archivo:

|  |
| --- |
| os.rename(current\_file\_name, new\_file\_name) |

Observemos que current\_file\_name es el nombre del archivo que queremos renombrar y new\_file\_name es el nuevo nombre que le queremos dar.

Veamos como usar **rename**() con un ejemplo:

|  |
| --- |
| import os  os.rename( "labotec.txt", "labotec.txt" ) |

## **¿Cómo removemos un archivo?**

En Python para remover un archivo usamos el método remove().

Veamos la sintaxis para poder leer un archivo:

|  |
| --- |
| os.remove(file\_name) |

Observemos que file\_name es el nombre del archivo que queremos remover.

Veamos como usar remove() con un ejemplo:

|  |
| --- |
| import os  os.remove("labotec.txt") |

**APÉNDICE**

Éstos son los métodos de cadena que soportan las cadenas de caracteres:

**capitalize** ()

Devuelve una copia de la cadena con el primer carácter en mayúscula.

**center** (width)

Devuelve la cadena centrada en una cadena de longitud width. Se rellena con espacios.

**count** (sub[, start[, end]])

Devuelve cuántas veces aparece sub en la cadena S[start:end]. Los argumentos opcionales start y end se interpretan según la notación de corte.

**encode** ([encoding[,errors]])

Devuelve una versión codoficada de la cadena. La codificación predeterminada es la codificación predeterminada de cadenas. El parámetro opcional errors fija el esquema de gestión de errores. Su valor predeterminado es 'strict', que indica que los errores de codificación harán saltar ValueError. Otros valores posibles son 'ignore' (ignorar) y 'replace' (reemplazar).

**endswith** (suffix[, start[, end]])

Devuelve verdadero si la cadena finaliza con el sufijo suffix especificado, en caso contrario falso. Si se da valor al parámetro opcional start, la comprobación empieza en esa posición. Si se da valor al parámetro opcional end, la comprobación finaliza en esa posición.

**expandtabs** ([tabsize])

Devuelve una copia de la cadena con todos los tabuladores expandidos a espacios. Si no se indica el paso de tabulación tabsize se asume 8.

**find** (sub[, start[, end]])

Devuelve el menor índice de la cadena para el que sub se encuentre, de tal modo que sub quede contenido en el rango [start, end). Los argumentos opcionales start y end se interpretan según la notación de corte. Devuelve -1 si no se halla sub.

**index** (sub[, start[, end]])

Como find(), pero lanza ValueError si no se encuentra la subcadena.

**isalnum** ()

Devuelve verdadero si todos los caracteres de la cadena son alfanuméricos y hay al menos un carácter. En caso contrario, devuelve falso.

**isalpha** ()

Devuelve verdadero si todos los caracteres de la cadena son alfabéticos y hay al menos un carácter. En caso contrario, devuelve falso.

**isdigit** ()

Devuelve verdadero si todos los caracteres de la cadena son dígitos y hay al menos un carácter. En caso contrario, devuelve falso.

**islower** ()

Devuelve verdadero si todos los caracteres alfabéticos de la cadena están en minúscula y hay al menos un carácter susceptible de estar en minúsculas. En caso contrario, devuelve falso.

**isspace** ()

Devuelve verdadero si todos los caracteres de la cadena son espacio en blanco (lo que incluye tabuladores, espacios y retornos de carro) y hay al menos un carácter. En caso contrario, devuelve falso.

**istitle** ()

Devuelve verdadero la cadena tiene forma de título (anglosajón) y hay al menos un carácter. En caso contrario, devuelve falso. Se considera que una cadena tiene formato de título si todas sus palabras están en minúsculas a excepción de la primera letra de cada una, que debe ser mayúscula.

**isupper** ()

Devuelve verdadero si todos los caracteres alfabéticos de la cadena están en mayúscula y hay al menos un carácter susceptible de estar en mayúsculas. En caso contrario, devuelve falso.

**join** (seq)

Devuelve una cadena formada ppor la concatenación de todos los elementos de la secuencia seq. Los elementos se separan por la cadena que proporciona el método. Se lanza TypeError si alguno de los elementos no es una cadena.

**ljust** (width)

Devuelve la cadena justificada a la izquierda en una cadena de longitud width. Se rellena la cadena con espacios. Se devuelve la cadena original si width es menor que len(s).

**lower** ()

Devuelve una copia de la cadena convertida en minúsculas.

**lstrip** ()

Devuelve una copia de la cadena con el espacio inicial eliminado.

**replace** (old, new[, maxsplit])

Devuelve una copia de la cadena en la que se han sustituido todas las apariciones de old por new. Si se proporciona el argumento opcional maxsplit, sólo se sustituyen las primeras maxsplit apariciones.

**rfind** (sub [,start [,end]])

Devuelve el índice máximo de la cadena para el que se encuentra la subcadena sub, tal que sub está contenido en cadena[start,end]. Los argumentos opcionales start y end se interpretan según la notación de corte. Devuelve -1 si no se encuentra sub.

**rindex** (sub[, start[, end]])

Como rfind() pero lanza ValueError si no se encuentra sub.

**rjust** (width)

Devuelve la cadena justificada a la derecha en una cadena de longitud width. Se rellena la cadena con espacios. Se devuelve la cadena original si width es menor que len(s).

**rstrip** ()

Devuelve una copia de la cadena con el espacio al final suprimido.

**split** ([sep [,maxsplit]])

Devuelve una lista de las palabras de la cadena, usando sep como delimitador de palabras. Si se indica maxsplit, se devolverán como mucho maxsplit valores (el último elemento contendrá el resto de la cadena). Si no se especifica sep o es None, cualquier espacio en blanco sirve de separador.

**splitlines** ([keepends])

Devuelve una lista de las líneas de la cadena, dividiendo por límites de línea. No se incluyen los caracteres limitadores en la lista resultante salvo que se proporcione un valor verdadero en keepends.

**startswith** (prefix[, start[, end]])

Devuelve verdadero si la cadena comienza por prefix, en caso contrario, devuelve falso. Si se proporciona el parámetro opcional start, se comprueba la cadena que empieza en esa posición. Si se proporciona el parámetro opcional end, se comprueba la cadena hasta esa posición.

**strip** ()

Devuelve una copia de la cadena con el espacio inicial y final suprimido.

**swapcase** ()

Devuelve una copia de la cadena con las mayúsculas pasadas a minúsculas y viceversa.

**title** ()

Devuelve una versión con formato título, es decir, con todas las palabras en minúsculas excepto la primera letra, que va en mayúsculas.

**translate** (table[, deletechars])

Devuelve una copia de la cadena donde se han eliminado todos los caracteres de deletechars y se han traducido los caracteres restantes según la tabla de correspondencia especificada por la cadena table, que debe ser una cadena de longitud 256.

**upper** ()

Devuelve una copia de la cadena en mayúsculas.