



Estructuras de programación



Laboratorio IPC 2
Unidad 4



Qué es una estructura de programación?

Una estructura es un tipo de dato compuesto que permite almacenar un conjunto de datos de diferente tipo. Los datos que contiene una estructura pueden ser de tipo simple (caracteres, números enteros o de coma flotante etc.) o a su vez de tipo compuesto (vectores, estructuras, listas, etc.).

A cada uno de los datos o elementos almacenados dentro de una estructura se les denomina miembros de esa estructura y éstos pertenecerán a un tipo de dato determinado.

Apuntadores

Son variables que ayudan a llevar el control de una estructura. De su manera más básica se pueden describir como una dirección de memoria física en donde se almacena un dato. En este caso, en una estructura de datos, el apuntador almacena una dirección de memoria de cierta posición importante de la estructura.

Por ejemplo, en el caso de una lista los apuntadores más comunes que se utilizan serán los que contienen las direcciones de memoria del principio de la lista y de la dirección de memoria del final de la lista, así como apuntadores internos que apuntarán hacia miembros específicos de la lista.

Lista simplemente enlazada.



Listas

Al igual que una cadena, una lista es una secuencia de valores. En una cadena, los valores son caracteres; en una lista, pueden ser de cualquier tipo. Los valores en las listas reciben el nombre de elementos, o a veces artículos.

En el lenguaje Python tendremos dos opciones para utilizar listas. Las listas que python incluye como librería y las que podemos generar utilizando clases de acuerdo al paradigma de la programación orientada a objetos en la cual Python está basado.

Listas nativas de Python

En este caso la forma más simple consiste en encerrar los elementos entre corchetes ([y]).

```
[10, 20, 30, 40]
```

```
['cadena1', 'cadena2', 'cadena3']
```

En el caso de Python los elementos en una lista no tienen porque ser todos del mismo tipo.

La lista siguiente contiene una cadena, un flotante, un entero y otra lista:

```
['spam', 2.0, 5, [10, 20]]
```

Cuando una lista se aloja dentro de otra lista se le llama **lista anidada**.

Una lista que no contiene elementos recibe el nombre de lista vacía. Se puede crear una simplemente con unos corchetes vacíos, [].

La forma de asignar valores a una lista es la siguiente:

```
numeros = [17, 123]
```

En donde 'numeros' es el nombre de la variable que guarda la lista y 17 y 123 son los valores de los miembros de la lista.

A diferencia de las cadenas, a las listas se les llama mutables (pueden mutar), porque se puede cambiar el orden de los elementos o reasignar un elemento dentro de la lista.

Cuando el operador corchete aparece en el lado izquierdo de una asignación, este identifica el elemento de la lista que será asignado.

```
numeros = [17, 123]  
numeros[1] = 5  
print numeros  
[17, 5]
```


Como podemos ver en el ejemplo anterior, las lista utilizan el método de indexación para acceder a los datos en cada posición. Siempre vale la pena recordar que los índices para acceso a datos empiezan en la posición 0, por lo que si escribimos `lista[1]` en realidad estamos accediendo a la segunda posición en la lista.

Cualquier número entero puede ser utilizado com índice. Si se intenta ingresar a un índice que no tiene ningún dato se obtiene un error 'IndexError'.

También podemos utilizar un valor negativo como índice. Esto hará que busque el elemento contando desde la última posición de la lista hacia atrás.

También podemos utilizar el operador **IN** para saber si algún elemento existe en la lista.

Luego de verificar la existencia del elemento nos devolverá un valor booleano.

```
quesos = ['Cheddar', 'Edam', 'Gouda']  
'Edam' in quesos  
True  
'Brie' in quesos  
False
```

Recorrer una lista

El modo más habitual de recorrer los elementos de una lista es con un bucle for.

La sintaxis es la misma que para las cadenas:

```
for queso in quesos:  
    print queso
```

A pesar de que una lista puede contener otra, la lista anidada sólo cuenta como un único elemento. La longitud de esta lista es cuatro:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

Operaciones con listas

El operador + concatena listas:

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print c
[1, 2, 3, 4, 5, 6]
```

De forma similar, el operador * repite una lista el número especificado de veces:

```
[0] * 4
[0, 0, 0, 0]
[1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

El primer ejemplo repite [0] cuatro veces. El segundo, repite la lista [1, 2, 3] tres veces.

Rebanado de listas

En listas podemos utilizar el operador ':' al cual llamaremos **Slice**.

```
t = ['a', 'b', 'c', 'd', 'e', 'f']  
t[1:3]  
['b', 'c']  
t[:4]  
['a', 'b', 'c', 'd']  
t[3:]  
['d', 'e', 'f']
```

Si se omite el primer índice, la rebanada comenzará al principio. Si se omite el segundo, la rebanada llegará hasta el final. De modo que si se omiten ambos, la rebanada será una copia de la lista completa.

Métodos de listas

Python proporciona varios métodos que operan con listas.

- **Append:** añade un nuevo elemento al final de la lista.

```
t = ['a', 'b', 'c']  
t.append('d')  
print t  
['a', 'b', 'c', 'd']
```

- **Extend:** toma una lista como argumento y añade al final de la actual todos sus elementos.

```
t1 = ['a', 'b', 'c']  
t2 = ['d', 'e']  
t1.extend(t2)  
print t1  
['a', 'b', 'c', 'd', 'e']
```


- **Sort:** ordena los elementos de una lista de menor a mayor.
-

```
t = ['d', 'c', 'e', 'b', 'a']  
t.sort()  
print t  
['a', 'b', 'c', 'd', 'e']
```

- **Borrado de elementos:** Si conocemos el índice del elemento que deseamos borrar, podemos utilizar la función **pop()**. Pop() modifica la lista y devuelve el valor que ha sido eliminado. Si a pop() no le proporcionamos argumento, entonces borrará el último elemento de la lista.

```
t = ['a', 'b', 'c']  
x = t.pop(1)  
print t  
['a', 'c']  
print x  
b
```

Si no necesitamos el valor que acaba de ser eliminado, podemos usar la función **del()** siempre indicando el índice del valor a eliminar.

```
t = ['a', 'b', 'c']  
del t[1]  
print t  
['a', 'c']
```

Si queremos eliminar un elemento pero en vez de tener el índice sabemos que valor tiene podemos utilizar **remove()** utilizando como argumento el valor del elemento. Esta función no devuelve nada.

Si queremos eliminar más de un elemento podemos utilizar la notación Slice utilizando índices de principio y de final.

```
t = ['a', 'b', 'c', 'd', 'e', 'f']  
del t[1:5]  
print t  
['a', 'f']
```

Funciones en listas

Son funciones inherentes a las listas que permiten realizar variedad de operaciones sin necesidad de algoritmos complicados.

- **len()**: muestra la cantidad de elementos que contiene la lista.
- **max()**: devuelve el elemento con valor máximo de la lista.
- **min()**: devuelve el elemento con valor mínimo de la lista.
- **sum()**: devuelve la suma de los elementos de la lista.

La función `sum()` solamente funciona cuando los elementos de la lista son números. Las otras funciones (`max()`, `len()`, etc.) funcionan también con listas de cadenas y otros tipos que se puedan comparar.

Listas y cadenas

Aunque una cadena es una secuencia de caracteres y una lista es una secuencia de elementos, una cadena no es una lista de caracteres. Para convertir una cadena a una lista de caracteres utilizamos la función **list()**.

```
s = 'spam'
t = list(s)
print t
['s', 'p', 'a', 'm']
```

La función `list()` divide una cadena en letras individuales. Si se desea dividir una cadena en palabras, se puede usar el método **`split()`**.

```
s = 'suspirando por los fiordos'
t = s.split()
print t
['suspirando', 'por', 'los', 'fiordos']
print t[2]
los
```

Si en vez de necesitar dividir el string en palabras, se necesita dividirlo basado en algún caracter delimitador, podemos pasar un argumento con el caracter al método split y éste dividirá el string en partes divididas por el caracter definido devolviendo una lista con las palabras luego de dividir las.

```
s = 'spam-spam-spam'  
delimitador = '-'  
s.split(delimitador)  
['spam', 'spam', 'spam']
```


La operación inversa de dividir sería unir y para eso podemos utilizar la función [join\(\)](#). Toma una lista de cadenas y las concatena a un nuevo string. Se debe definir un delimitador para la unión de las cadenas y se le pasa una lista como parámetro.

```
s = 'spam-spam-spam'
delimitador = '-'
s.split(delimitador)
['spam', 'spam', 'spam']
```

En caso de que el delimitador sea el caracter espacio, entonces join coloca un espacio entre las palabras. Para concatenar cadenas sin espacios, se puede usar la cadena vacía "" como delimitador.

Listas con POO

En python podemos utilizar las listas ya definidas por el lenguaje pero también tenemos la opción de crear listas personalizadas. Esto debido a la existencia de las clases y los apuntadores. Cada clase tendrá como atributos los datos que desee guardar así como apuntadores que servirán de referencia de memoria hacia otras clases. Estas clases las llamaremos **Nodos**.

Dependiendo de la clase de lista que creamos, dependerá la cantidad de nodos que poseerá o hacia donde apuntan.

Ejemplo de una clase nodo

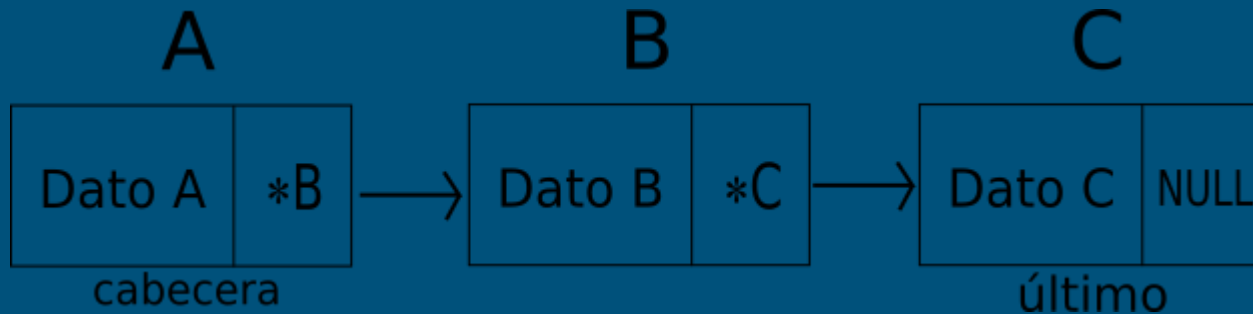
```
class node:
    def __init__(self, data = None, next = None):
        self.data = data
        self.next = next
```

En donde 'data' es la información que deseamos guardar y 'next' es el apuntador hacia el siguiente miembro de la lista.

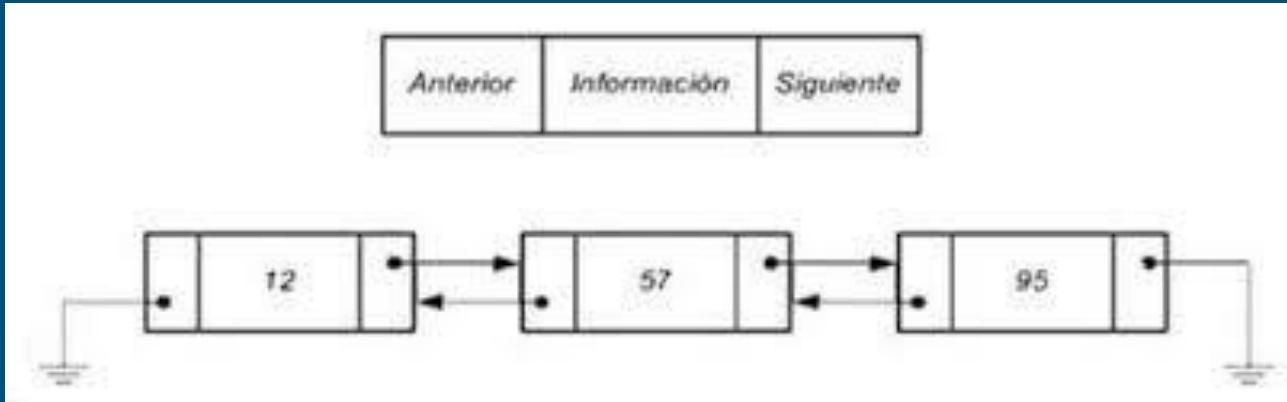
Utilizamos el método `__init__` para inicializar los valores del nodo. Algo que en otros lenguajes de programación se conoce como el constructor de la clase.

Tipos de listas

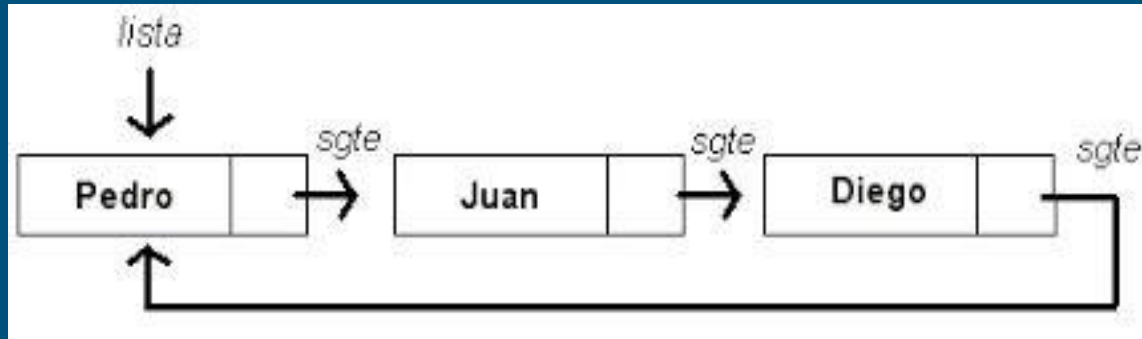
- **Lista simplemente enlazada:** Cada nodo tiene un apuntador que apunta hacia el nodo siguiente en la lista. Generalmente se instancian dos referencias hacia la cabecera de la lista y al final de la lista. El nodo siguiente del último nodo de la lista tendrá un valor de nulo.



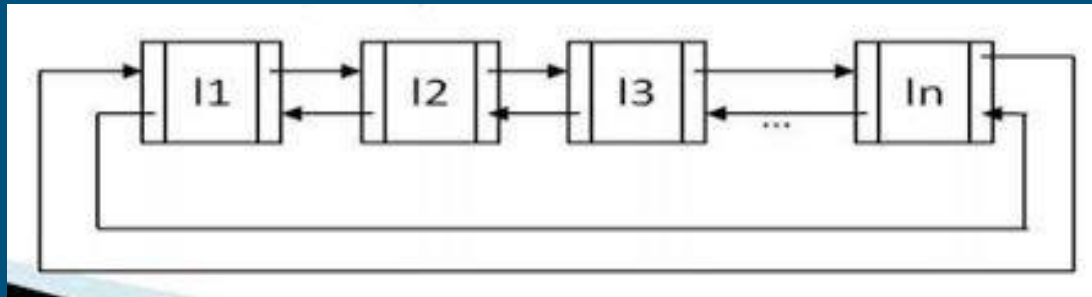
- **Lista doblemente enlazada:** En esta lista, cada nodo tendrá dos apunadores. Uno al nodo siguiente y el otro hacia el nodo anterior. También es recomendable utilizar las instancias de cabecera y final de lista. Tanto el apuntador anterior del primer nodo de la lista como el apuntador siguiente del último nodo de la lista tendrán valor nulo.



- **Lista enlazada circular:** En este tipo de lista se tendrá un solo apuntador hacia el siguiente nodo de la lista (como en la lista enlazada simple). La diferencia será que el apuntador siguiente del último nodo apuntará al primer nodo de la lista por lo que ninguno de los apuntadores tendrá valor nulo.



- **Lista circular doblemente enlazada:** sigue los mismo principios de la lista circular simple, la diferencia será que cada nodo tendrá dos apuntadores, el nodo anterior que apunta al nodo anterior de la lista y el nodo siguiente apunta al nodo siguiente de la lista. La diferencia será que el apuntador siguiente del último valor de la lista apunta al primer nodo de la lista y el apuntador anterior del primera elemento de la lista apunta al último valor de la lista.



Operaciones sobre listas enlazadas

Todas las operaciones siguientes pueden realizarse en los diferentes tipos de listas. La diferencia será el algoritmo a utilizar.

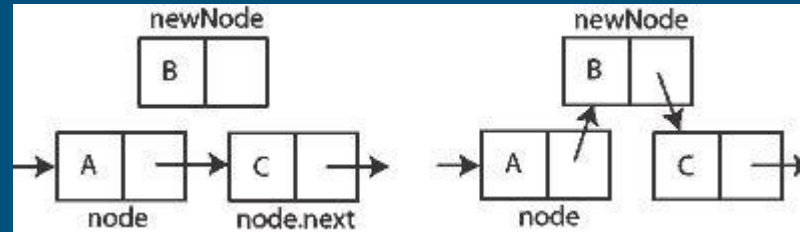
Las operaciones básicas sobre listas son:

- Agregar nuevo nodo
- Borrar nodo
- Recorrer lista
- Buscar elementos
- Borrar lista entera

Agregar nodo en listas enlazadas no circulares

- Si queremos agregar un nodo en una lista vacía, tanto el apuntador de la cabecera de la lista como el del final apuntarán al nuevo nodo y el nodo siguiente del nuevo nodo será nulo. Si es una lista doblemente enlazada el apuntador anterior también será nulo.
- Si queremos agregar un nodo al principio de una lista, asignaremos el apuntador siguiente del nuevo nodo a la cabecera de la lista y reasignaremos al nuevo nodo como cabecera de la lista. Si es una doblemente enlazada, el apuntador anterior de la cabecera apuntará hacia el nuevo nodo y luego reasignaremos el nuevo nodo como nueva cabecera.

- Si queremos agregar un nodo al final de lista, haremos que el apuntador siguiente del nodo final apunte al nuevo nodo y reasignaremos el final de la lista al nuevo nodo. Si es una doblemente enlazada, el nodo anterior del nuevo nodo apuntará al antiguo final de la lista y luego reasignaremos el nuevo nodo como final de la lista.



Agregar nodo en listas circulares

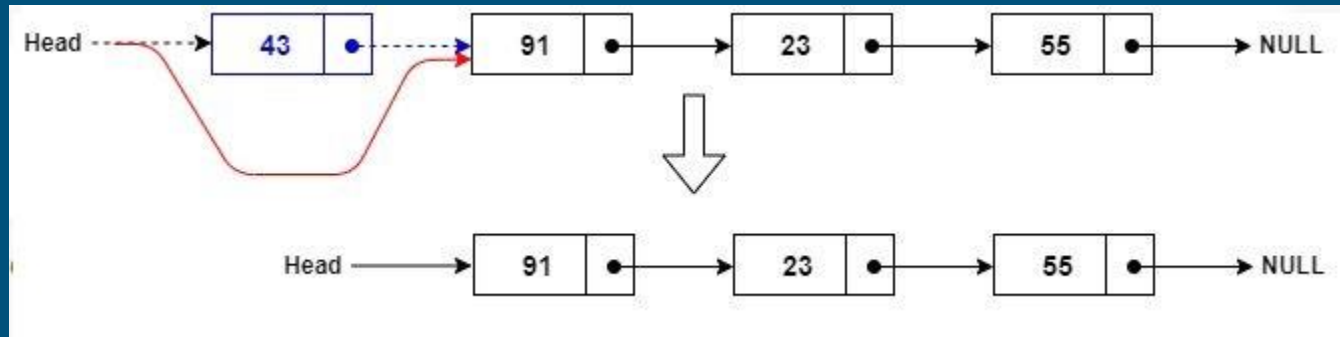
- Si queremos agregar un nodo en una lista vacía, tanto el apuntador de la cabecera de la lista como el del final apuntarán al nuevo nodo y el nodo siguiente del nuevo nodo será apuntará a sí mismo. Si es una lista doblemente enlazada el apuntador anterior también apuntará a sí mismo.
- Si queremos agregar un nodo al principio de una lista, asignaremos el apuntador siguiente del nuevo nodo a la cabecera de la lista y el apuntador siguiente del final de la lista apuntará al nuevo nodo. Luego reasignaremos al nuevo nodo como cabecera de la lista. Si es una doblemente enlazada, el apuntador anterior de la cabecera apuntará hacia el nuevo nodo y luego reasignaremos el nuevo nodo como nueva cabecera.

Borrar elementos

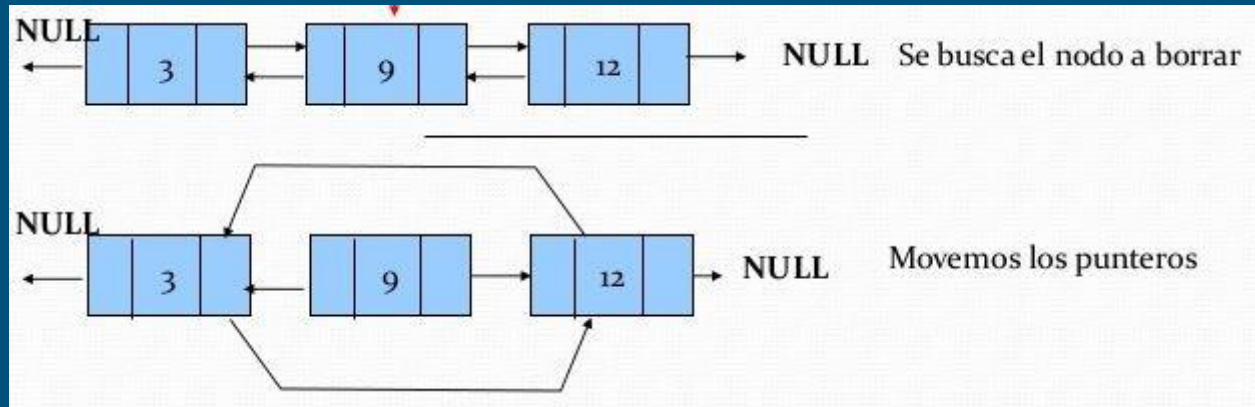
Tanto en la listas simples como circulares, los algoritmos de borrado de elementos son similares, la única diferencia será la manera en que se manejan sus apuntadores.

- Si se desea borrar un elemento al principio de la lista, los apuntadores del nodo siguiente y anterior (dependiendo del tipo de lista que sea) se reasignarán entre ellos y luego se asignará como cabecera de la lista al nodo siguiente del nodo eliminado.

- Para borrar un nodo al final de lista lista se utilizará el mismo proceso que el borrar al principio de la lista, la diferencia es que se asignará al nodo anterior al eliminado como nuevo nodo final de la lista y sus apuntadores se manejan de manera similar al borrado de primer elemento.



- Para borrar un nodo en cualquier otra posición de la lista, se necesita ya sea un índice, o el valor que se desea borrar. La lista debe recorrer desde su primer elemento hasta encontrar una coincidencia y luego aplicar el algoritmo anterior para borrar el elemento.

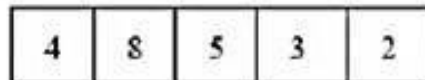


Recorrer listas y buscar elementos

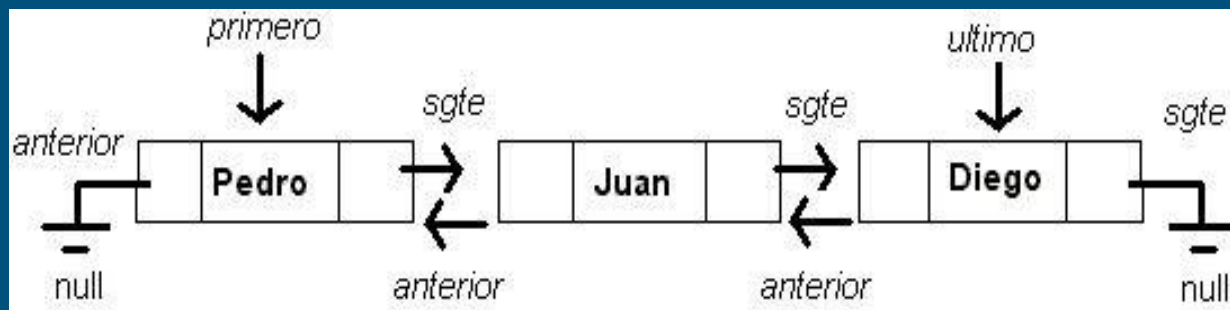
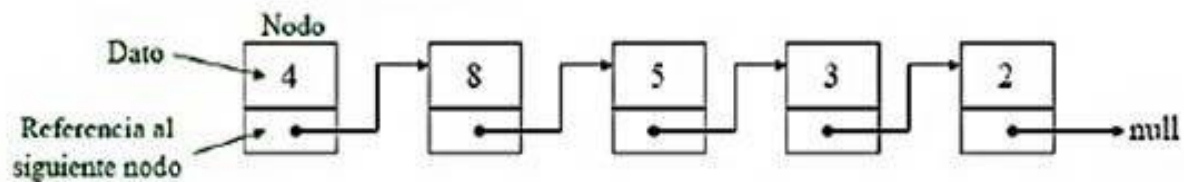
Tal como la operación de borrar, la operación de recorrer la lista es igual en listas enlazadas no circulares y circulares y va de la mano con la operación buscar. También necesita de un índice o de un valor para recorrer la lista en el caso de que se desee buscar un elemento.

- Se toma la cabecera de la lista y se utiliza una instrucción cíclica como el `while` que utilice el apuntador siguiente de la lista para pasar de nodo en nodo hasta cumplir con la condición de búsqueda (si la existiese). En el caso de la lista circular debe existir esta condición. De no hacerlo, la lista se recorrerá en un ciclo interminable, esto debido a las características de sus apuntadores.

Representación secuencial:



Representación enlazada:



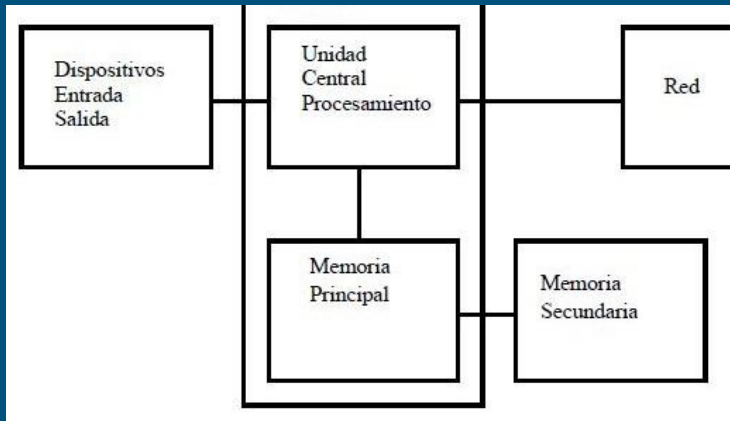
Borrar lista entera

La operación de borrar la lista entera es bastante simple. Solo basta con tomar los apuntadores de la cabecera y del nodo final y asignar todos sus apuntadores a valores nulos.

De esta manera todas las referencias hacia los espacios de memoria que ocupaban los nodos pertenecientes a la lista se perderán y ya no podrán ser referenciados. La memoria que estos nodo ocupaban será limpiada al finalizar el programa.

FICHEROS (Capítulo 7)

Persistencia: Hasta ahora, hemos aprendido cómo escribir programas y comunicar nuestras intenciones a la Unidad Central de Procesamiento usando ejecución condicional, funciones e iteraciones. Hemos aprendido como crear y usar estructuras de datos en la Memoria Principal.



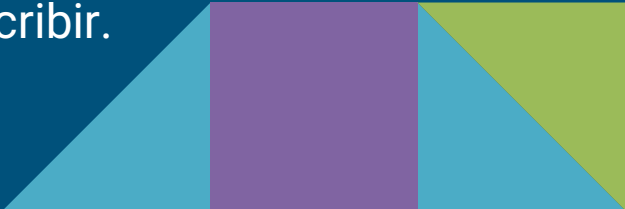
En este capítulo, comenzaremos a trabajar con la Memoria Secundaria (o ficheros, o archivos). La memoria secundaria no se borra aunque se interrumpa la corriente. Incluso, en el caso de una unidad flash USB, los datos que escribamos desde nuestros programas pueden ser retirados del sistema y transportados a otro equipo.

Apertura de Ficheros

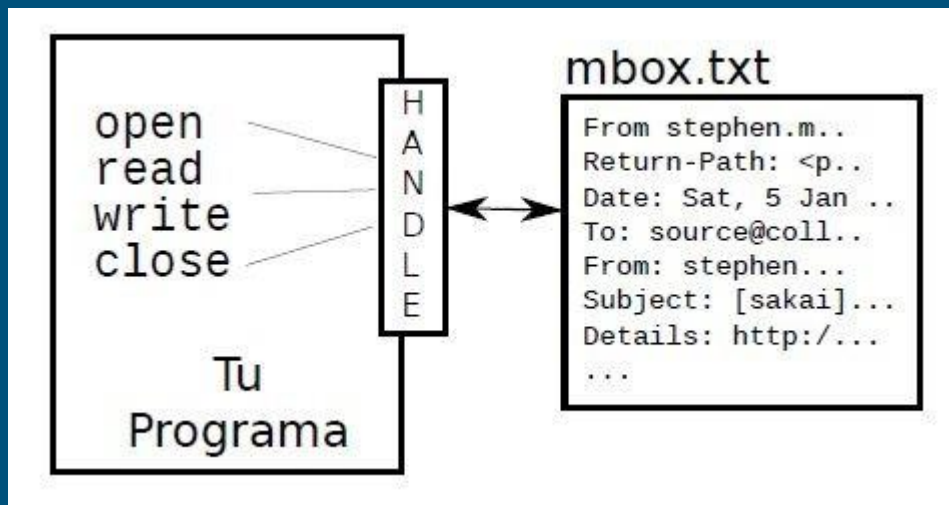
Cuando se desea leer o escribir en un archivo (nos referimos en el disco duro), primero debemos abrir el fichero. Al abrir el fichero nos comunicamos con el sistema operativo, que sabe dónde se encuentran almacenados los datos de cada archivo. Cuando se abre un fichero, se está pidiendo al sistema operativo que lo busque por su nombre y se asegure de que existe. En este ejemplo, abrimos el fichero mbox.txt, que debería estar guardado en la misma carpeta en la que te encontrabas cuando iniciaste Python.

```
>>> manf = open('mbox.txt')
>>> print manf
<open file 'mbox.txt', mode 'r' at 0x1005088b0>
```

Modos

- `r` Solo lectura. El fichero solo se puede leer. Es el modo por defecto si no se indica.
 - `w` Sólo escritura. En el fichero solo se puede escribir. Si ya existe el fichero, machaca su contenido.
 - `a` Adición. En el fichero solo se puede escribir. Si ya existe el fichero, todo lo que se escriba se añadirá al final del mismo.
 - `x` Como 'w' pero si existe el fichero lanza una excepción.
 - `r+` Lectura y escritura. El fichero se puede leer y escribir.
- 

Si la apertura tiene éxito, el sistema operativo nos devuelve un manejador de fichero (file handle). El manejador de fichero no son los datos que contiene en realidad el archivo, sino que se trata de un “manejador” (handle) que se puede utilizar para leer esos datos.



Si el fichero no existe, open fallará con un traceback, y no obtendrás ningún manejador para poder acceder a su contenido:

```
>>> manf = open('cosas.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'cosas.txt'
```

Ficheros de texto y líneas

Un fichero de texto puede ser considerado una secuencia de líneas, de igual modo que una cadena en Python puede ser considerada una secuencia de caracteres. Por ejemplo, esta es una muestra de un fichero de texto que guarda la actividad de correos de varias personas en el equipo de desarrollo de un proyecto de código abierto:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

En Python, el carácter salto de línea se representa por barra invertida-n en las cadenas. A pesar de que parezcan dos caracteres, se trata en realidad de uno sólo. Cuando revisamos la variable introduciendo “cosa” en el intérprete, nos mostrará el `\n` en la cadena, pero cuando usemos `print` para mostrar la cadena, veremos cómo esta aparece dividida en dos líneas por el carácter de salto de línea.

```
>>> cosa = '¡Hola\nMundo!'
>>> cosa
'¡Hola\nMundo!'
>>> print cosa
¡Hola
Mundo!
>>> cosa = 'X\nY'
>>> print cosa
X
Y
>>> len(cosa)
3
```


Lectura de Ficheros:

A pesar de que el manejador de fichero no contiene los datos del archivo, es bastante fácil construir un bucle for para ir leyendo y contabilizando cada una de las líneas de un fichero:

```
manf = open('mbox.txt')
contador = 0
for linea in manf:
    contador = contador + 1
print 'Líneas contabilizadas:', contador

python open.py
Line Count: 132045
```

Si sabes que el fichero es relativamente pequeño comparado con el tamaño de tu memoria principal, puedes leer el fichero completo en una cadena usando el método `read` sobre el manejador del fichero.

```
>>> manf = open('mbox-short.txt')
>>> ent = manf.read()
>>> print len(ent)
94626
>>> print ent[:20]
From stephen.marquar
```



Se imprime
desde el caracter
0 al 20

Búsqueda dentro de un fichero:

Cuando se buscan datos dentro de un fichero, un diseño muy común consiste en ir leyendo el archivo completo, ignorando la mayoría de las líneas y procesando únicamente aquellas que cumplen alguna condición particular. Es posible combinar ese diseño de lectura de ficheros con los métodos de cadena para construir un mecanismo simple de búsqueda.

```
manf = open('mbox-short.txt')
for linea in manf:
    if linea.startswith('From:') :
        print linea
```



Cuando se hace funcionar el programa, obtenemos la siguiente salida:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
...
```

Permitiendo al usuario elegir el nombre del fichero:

Lo más probable es que no nos apetezca editar nuestro código Python cada vez que queramos procesar un archivo diferente. Sería más útil pedir al usuario que introdujera una cadena con el nombre del fichero cada vez que el programa funcione, de modo que se pueda usar nuestro programa sobre ficheros diferentes sin tener que cambiar el código Python. Esto es bastante sencillo de hacer, pidiendo el nombre del fichero al usuario mediante el uso de `raw_input`, como se muestra a continuación:

```
nombref = raw_input('Introduzca el nombre del fichero: ')
manf = open(nombref)
contador = 0
for linea in manf:
    if linea.startswith('Subject:'):
        contador = contador + 1
print 'Hay', contador, 'líneas subject en', nombref
```

Uso de try, except, y open

¿Qué ocurre si nuestro usuario escribe algo que no es un nombre de fichero?

Debemos asumir que la llamada a open puede fallar y añadir código de recuperación para ese fallo, como se muestra a continuación:

```
nombref = raw_input('Introduzca el nombre del fichero: ')
try:
    manf = open(nombref)
except:
    print 'No se pudo abrir el fichero:', nombref
    exit()

contador = 0
for linea in manf:
    if linea.startswith('Subject:') :
        contador = contador + 1
print 'Hay', contador, 'líneas subject en', nombref
```

Escritura en ficheros

Para escribir en un fichero, debes abrirlo usando el modo 'w' (de write) como segundo parámetro:

```
>>> fsal = open('salida.txt', 'w')  
>>> print fsal  
<open file 'salida.txt', mode 'w' at 0xb7eb2410>
```

El método write del objeto manejador del fichero pone datos dentro del archivo.

```
>>> lineal = "Aquí está el zarzo,\n"  
>>> fsal.write(lineal)
```

Depuración

Cuando estés leyendo y escribiendo archivos, puedes tener problemas con los espacios en blanco. Estos errores pueden ser difíciles de depurar porque los espacios, tabulaciones y saltos de línea normalmente son invisibles:

```
>>> s = '1 2\t 3\n 4'
>>> print s
1 2 3
 4
```

La función interna `repr` puede ayudarnos. Toma cualquier objeto como argumento y devuelve una representación de cadena del objeto. En el caso de las cadenas, representa los caracteres en blanco con secuencias de barras invertidas:

```
>>> print repr(s)
'1 2\t 3\n 4'
```



ESTRUCTURAS DE PROGRAMACIÓN



Laboratorio IPC 2
Unidad 4



Diccionarios

En Python un diccionario es un conjunto desordenado de pares clave-valor, en el que la clave debe ser única y puede ser cualquier tipo de dato inmutable como enteros, cadenas, booleanos, etc. Para los valores que se pueden tener dentro de un diccionario no hay restricciones pueden ser de cualquier tipo como listas, cadenas, enteros, tuplas, incluso otro diccionario.

Sintaxis de un diccionario

D = {'ENERO':1,'ABRIL':4,'AGOSTO':8}



Par clave:valor



clave



valor



separador
de pares



operador
de vinculo

Creación de diccionarios

Un diccionario que no contiene elementos recibe el nombre de diccionario vacío y se crea solamente con unas llaves vacías, {}.

```
d = {}  
print(type(d))  
>>> <class 'dict'>
```

Otra forma de crear diccionarios es de manera directa, asignando los elementos.

```
d = {"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12}  
print(type(d))  
>>> <class 'dict'>
```

En este ejemplo las claves son representadas por los meses y el valor asociado es representado por el número que le corresponde a cada mes.

Insertar elementos

La forma de asignar un nuevo par de clave-valor es la siguiente:

```
d = {"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12}  
d["Marzo"] = 3  
print (d)  
>>> {'Enero': 1, 'Abril': 4, 'Agosto': 8, 'Diciembre': 12, 'Marzo': 3}
```

Dónde “Marzo” representa la nueva clave que se agregara y 3 representa el valor que tendrá asociado.

Métodos de diccionarios

Python proporciona varios métodos que operan con diccionarios

- `clear()`: remueve todos los elementos del diccionario.

```
d = {"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12}
print (d)
>>> {'Enero': 1, 'Abril': 4, 'Agosto': 8, 'Diciembre': 12}
d.clear()
print(d)
>>> {}
```

Métodos de diccionarios

- `get()`: Devuelve el valor en base a una coincidencia de búsqueda mediante una clave, de no encontrar la clave devuelve el objeto `None`

```
d = {"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12}
print(d.get("Abril"))
>>> 4
print(d.get("Noviembre"))
>>> None
```

Métodos de diccionarios

- `items()`: Devuelve una lista de pares clave:valor

```
d = {"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12}  
print(d.items())  
>>> dict_items([('Enero', 1), ('Abril', 4), ('Agosto', 8), ('Diciembre', 12)])
```

Métodos de diccionarios

- `keys()`: Devuelve una lista de las claves del diccionario

```
d = {"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12}  
print(d.keys())  
>>> dict_keys(['Enero', 'Abril', 'Agosto', 'Diciembre'])
```


Método de diccionarios

- `values()`: Devuelve una lista de los valores del diccionario

```
d = {"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12}  
print(d.values())  
>>> dict_values([1, 4, 8, 12])
```

Métodos de diccionarios

- `pop()`: Elimina específicamente una clave de diccionario y devuelve el valor que le corresponde, de no encontrar la clave lanza una excepción `KeyError`.

```
d = {"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12}
print (d.pop("Agosto"))
>>> 8
print(d)
>>> {'Enero': 1, 'Abril': 4, 'Diciembre': 12}
```

```
d = {"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12}
d.pop("Febrero")
>>> KeyError: 'Febrero'
```

Métodos de diccionarios

- `update()`: Actualiza un diccionario agregando los pares clave-valor.

```
d = {"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12}
d.update({"Junio":6})
print(d)
>>> {'Enero': 1, 'Abril': 4, 'Agosto': 8, 'Diciembre': 12, 'Junio': 6}
```

Funciones

Los diccionarios tienen disponible una serie de funciones integradas por el intérprete Python las cuales son:

- `len(diccionario)` : devuelve cuantos elementos posee un diccionario

```
d = {"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12}  
print (len(d))  
>>> 4
```

Funciones

- `dict()` : Es el constructor del tipo diccionario, es usada para crear un diccionario.

Usando `dict` se pueden crear diccionarios de la siguiente manera:

```
d = dict(Enero=1,Abril=4,Agosto=8,Diciembre=12)
print(type(d))
>>> <class 'dict'>
print(d)
>>> {'Enero': 1, 'Abril': 4, 'Agosto': 8, 'Diciembre': 12}
```

```
d = dict({"Enero":1,"Abril":4,"Agosto":8,"Diciembre":12})
print(type(d))
>>> <class 'dict'>
```

Tuplas

En python como parte de las estructuras internas que maneja existen las llamadas tuplas.

Son estructuras capaces de poseer elementos con distintos tipos de datos, estas estructuras se dicen que son ordenadas e inmutables, dado que una vez definida la estructura somos incapaz de cambiar el orden de los elementos o de cambiar los valores de los elementos.

Dentro de python, para la creación y manejo de las tuplas utiliza la clase “tuple”

```
print(type(tupla))  
- <class 'tuple'>
```

Definición

Para crear una tupla dentro de python es parecido a las listas, son elementos separados por comas, solo que ahora en lugar de utilizar los delimitadores “[” y “]” utilizamos los paréntesis

```
tupla = (1,"Hola",3,"Mundo",5, True)
print(tupla)
- (1, 'Hola', 3, 'Mundo', 5, True)
```

Podemos omitir los paréntesis pero para evitar confusiones es mejor siempre utilizarlos

Definición

Existen también las tuplas que no poseen elementos dentro de ellas, a estas se les conoce como tuplas vacías.

```
tupla = ()  
print(type(tupla))  
- <class 'tuple'>
```

Acá se puede ejemplificar el por que es recomendable siempre usar paréntesis, pues si en este caso los omitimos el parser de python lanzará error.

Definición

Podemos crear tuplas de un solo valor, para ello nos apoyaremos de una coma al final del elemento, ya que si no la utilizamos python toma como valor en lugar de una tupla.

```
tupla = (1)
print(type(tupla))
- <class 'int'>
```

```
tupla = (1,)
print(type(tupla))
- (1,)
```

Nota: cuando imprimimos la tupla de un solo elemento, nos damos cuenta que también añade la coma al final, esto solo es para que nosotros identifiquemos qué es una tupla, ya que internamente no existe un elemento nulo.

Definición

```
tupla = tuple((1,2,3))  
print(type(tupla))  
- <class 'tuple'>
```

```
tupla = tuple([1,2,3])  
print(type(tupla))  
- <class 'tuple'>
```

```
tupla = tuple("Hello")  
print(tupla)  
- ('H', 'e', 'l', 'l', 'o')  
print(type(tupla))  
- <class 'tuple'>
```

Podemos utilizar el constructor tuple() para definir tuplas o convertir secuencias de datos a tuplas como por ejemplo listas a tuplas o convertir una cadena de caracteres a tupla.

Operaciones sobre tuplas

Con las tuplas podemos ejecutar casi todas las operaciones que podíamos usar con las listas tomando en cuenta que las tuplas son inmutables y por ende cada acción que intente cambiar los valores de las tuplas después de su definición resultará en error o nos generará un objeto completamente distinto.

- Concatenación (+)
- Multiplicación o repetición (*)
- Buscar miembro (in)
- Notación Slice

Concatenación

Si tenemos 2 tuplas, en las cuales es necesario el unir las en una sola tupla, utilizaremos el operador de concatenación que nos generará un nuevo objeto tupla.

```
tupla1 = ("a","b","c")  
tupla2 = ("d","e","f")  
tupla3 = tupla1 + tupla2  
print(tupla3)  
- ('a', 'b', 'c', 'd', 'e', 'f')
```

Repetición

Al igual que en las listas podemos utilizar el operador (*) para repetir n cantidad de veces una tupla, el resultado es otro objeto tupla.

```
tupla1 = ("a","b","c")  
tupla2 = tupla1*3  
print(tupla2)  
- ('a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c')
```

Buscar miembro

Con el operador (in) podemos saber fácilmente si un valor es un elemento contenido dentro de una tupla, el resultado será True o False.

Podemos recorrer cada uno de los elementos si a así lo deseamos, esto lo realizamos con un for.

```
tupla1 = ("a","b","c")  
print("b" in tupla 1)  
- True
```

```
tupla1 = ("a","b","c")  
for elemnto in tupla1:  
    print(elemento)  
  
- a  
- b  
- c
```

Slice

Debido a que las tuplas son una estructura indexada podemos utilizar todos los tipos de notación slice que existen además de los índices negativos, como resultado tendremos un nuevo objeto tupla.

```
tupla = ("a","b","c","d","e","f","g", "h")  
print(tupla[2:6])  
- ('c', 'd', 'e', 'f')
```

```
tupla = ("a","b","c","d","e","f","g", "h")  
print(tupla[:4])  
- ('a', 'b', 'c', 'd')
```

```
tupla = ("a","b","c","d","e","f","g", "h")  
print(tupla[5:])  
- ('f', 'g', 'h')
```

```
tupla = ("a","b","c","d","e","f","g", "h")  
print(tupla[-2])  
- g
```

Funciones

De igual manera, tenemos funciones internas, las cuales son:

- `len(tupla)` devuelve cuantos elementos posee una tupla
- `max(tupla)` devuelve el elemento con mayor peso dentro de la tupla
- `min(tupla)` devuelve el elemento con menor peso dentro de la tupla

```
tupla1 = (1,2,3,6)
print(len(tupla1))
- 4
```

```
tupla1 = (1,2,3,6)
print(max(tupla1))
- 6
```

```
tupla1 = (1,2,3,6)
print(min(tupla1))
- 1
```


Expresiones regulares

Las expresiones regulares también son llamadas regex. Son una secuencia de caracteres que forman un patrón de búsqueda que están definidos con una sintaxis formal.

Los patrones se interpretan como un conjunto de instrucciones, que luego se ejecutan con una cadena como entrada para producir un subconjunto de coincidencia o una versión modificada del original.

Las expresiones regulares son útiles para verificar las cadenas que coinciden con un patrón y para realizar sustituciones en una cadena.

Componentes de las Expresiones Regulares

Todas las expresiones regulares están compuestas por dos tipos de caracteres que son los metacaracteres y literales.

- Metacaracteres: Es un carácter o secuencias de caracteres que tienen un significado especial, son la esencia de las expresiones regulares. Sirven para representar un patrón que se repite dentro de una secuencia. Existen diferentes tipos de metacaracteres:
 - Delimitadores
 - Iteradores
 - Alternativos

Componentes de las Expresiones Regulares

Metacaracteres

- metacaracteres delimitadores: Permiten delimitar donde queremos buscar los patrones de búsqueda.

METACARACTER	DESCRIPCIÓN
^	inicio de línea
.	cualquier carácter en la línea
\$	limita la búsqueda al final de una línea
\A	inicio de texto
\Z	fin de texto

Componentes de las Expresiones Regulares

Metacaracteres

- metacaracteres - clases predefinidas: Estos son clases predefinidas que facilitan la utilización de las expresiones regulares.

METACARACTER	DESCRIPCIÓN
\w	Carácter alfanumérico
\W	Carácter no alfanumérico
\d	Carácter numérico
\D	Carácter no numérico
\s	Cualquier espacio

Componentes de las Expresiones Regulares

Metacaracteres

- metacaracteres iteradores: Con estos caracteres se puede especificar el número de ocurrencias del caracter previo de un metacaracter o de una subexpresión.

METACARACTER	DESCRIPCIÓN
*	cero o más, indica que la expresión antes del signo puede ser aleatorio.
+	una o más, indica que la expresión antes del signo debe aparecer como mínimo una vez.
?	cero o una, indica que la expresión antes del signo podría aparecer al menos una vez.

Componentes de las Expresiones Regulares

Metacaracteres

- Metacaracteres alternativos: Se puede especificar una serie de alternativas utilizando “|” este metacaracter es el que se utiliza para separa las alternativas, las alternativas son evaluadas de izquierda a derecha resulta positivo si cualquiera de las dos condiciones cumple la expresión.

Componentes de las Expresiones Regulares Literales

Cualquier carácter que se encuentra a sí mismo, a menos que se trate de un metacaracter, una serie de caracteres encuentra esa misma serie en el texto de entrada. Por ejemplo "ipc2" encontrará todas las apariciones de "ipc2" en el texto.

Rangos

Dentro de las expresiones regulares se pueden definir rangos de las siguiente manera:

- `[A-Z]` : Cualquier carácter alfabético en mayúscula
- `[a-z]` : Cualquier carácter alfabético en minúscula
- `[0-9]` : Cualquier carácter numérico.
- `[a-zA-Z0-9]` : Cualquier carácter alfanumérico.


```
(?=.*[0-9]{2})[0-9a-zA-Z]{8,12}
```

- `?` Busca que la siguiente expresión se encuentre en el texto
- `.*[0-9]` Un texto que contenga al menos un número
- `[0-9]{2}` El número debe ser de dos dígitos
- `(?=.*[0-9]{2})` En lo que siga a esta expresión, se valida que exista un número de al menos dos dígitos.
- `[0-9a-zA-Z]` Un carácter entre números y letras mayúsculas o minúsculas
- `[0-9a-zA-Z]{8,12}` Será una cadena de longitud mínima 8 y máxima 12.

Módulo Re

re es la librería estándar de Python que soporta operaciones de coincidencia de expresiones regulares. Se podrán crear objetos de tipo patrón y generar objetos de tipo matcher, que son los que contienen la información de la coincidencia del patrón en la cadena.

Para utilizar este módulo se debe de importar.

```
import re
```

Funciones - Search

Una vez tengamos importado el módulo `re`, podemos utilizar la función `search(patron, cadena)` en donde `patron` es la coincidencia que necesitamos encontrar y `cadena` es la fuente en donde python buscará nuestro patrón.

- Si existe coincidencia, entonces nos retorna un objeto de tipo `Match` con información de la primera coincidencia.
- Si no existe coincidencia, entonces retorna `"None"`

```
cadena = "Buscaremos un patrón en esta cadena"
patron = "cadena"
resultado = re.search(patron,cadena)
print(resultado)
- <re.Match object; span=(29, 35), match='cadena'>
```

Funciones - Objeto Match

El objeto match posee ciertas funciones que podrían ser de utilidad.

- `start()` : retorna la posición en la cual inicia la coincidencia.
- `end()` : retorna la posición en la cual finaliza la coincidencia.
- `span()` : retorna una tupla con la posición inicial y final de la coincidencia.
- `string()` : retorna la cadena de origen en donde se buscó el patrón.

```
print(resultado.start())
print(resultado.end())
print(resultado.span())
print(resultado.string())
- 29
- 35
- (29, 35)
- Buscaremos un patrón en esta cadena
```

Funciones - Match

La función “match(patron, cadena)” buscará el patrón que le indiquemos únicamente al inicio de la cadena, si no lo encuentra al inicio retornará “None”.

```
cadena = "Buscaremos Buscaremos un patrón en esta cadena"
patron = "Buscaremos"
resultado = re.match(patron,cadena)
print(resultado)
- <re.Match object; span=(0, 10), match='Buscaremos'>
```

```
cadena = "Buscaremoos Buscaremos un patrón en esta cadena"
patron = "Buscaremos"
resultado = re.match(patron,cadena)
print(resultado)
- None
```

Funciones - Split

La función “split(patron, cadena)” buscará dentro de la cadena el patrón y dividirá la cadena en una lista donde los elementos están conformados por los caracteres, antes de encontrar el patrón.

```
cadena = "Buscaremos un patrón en esta cadena"  
patron = " "  
resultado = re.match(patron,cadena)  
print(resultado)  
- ['Buscaremos', 'un', 'patrón', 'en', 'esta', 'cadena']
```

Funciones - Sub

La función “sub(patron, valor, cadena)” buscará dentro de toda la cadena el patrón especificado y sustituirá cada una de las coincidencias por el “valor” que le pasemos, como resultado tendremos la nueva cadena con los patrones sustituidos.

```
cadena = "Hola mundo, Hola gente"  
patron = "Hola"  
valor = "Hello"  
resultado = re.sub(patron, valor, cadena)  
print(resultado)  
- Hello mundo, Hello gente
```

Funciones - FindAll

La función “findall(patrón, valor, cadena)” buscará dentro de toda la cadena el patrón especificado retorna una lista con todas las cadenas que coinciden con el patrón que especificamos.

```
cadena = "Hola mundo, Hola gente"  
patron = "Hola"  
resultado = re.findall(patron,cadena)  
print(resultado)  
- ['Hola', 'Hola']
```


Funciones con expresión regular - Search

Por simplicidad en los ejemplos de las funciones se utilizó el parámetro patrón como una simple cadena de texto, pero este parámetro puede ser una expresión regular con metacaracteres, rangos, etc.

```
cadena = "Hola mundddddo, Hola gente"  
patron = "mund*o"  
resultado = re.search(patron, cadena)  
print(resultado)  
- <re.Match object; span=(5, 14), match='mundddddo'>
```

Funciones con expresión regular

Función Match

```
cadena = "12345678 es mi numero telefonico"  
patron = "[1-9]+"
```

resultado = re.match(patron, cadena)
print(resultado)
- <re.Match object; span=(0, 8), match='12345678'>

Función Split

```
cadena = "Esta8cadena88sera dividida8en888una8888lista"  
patron = "8+"
```

resultado = re.split(patron, cadena)
print(resultado)
- ['Esta', 'cadena', 'sera dividida', 'en', 'una', 'lista']

Funciones con expresión regular

Función Sub

```
cadena = "h0la, hla, hoola, hooooooooola"  
patron = "ho*la"  
valor = "cambio"  
resultado = re.sub(patron, valor, cadena)  
print(resultado)  
- h0la, cambio, cambio, cambio
```

Función findall

```
cadena = "h0la, hla, hoola, hooooooooola"  
patron = "ho*la"  
resultado = re.findall(patron, cadena)  
print(resultado)  
- ['hla', 'hoola', 'hooooooooola']
```