Secuencias, Tuplas y Listas

Unidad 4

Apunte de cátedra
2do Cuatrimestre 2023

Pensamiento Computacional (90)
Cátedra: Camejo

.UBAXXI

1. Tipos de Estructuras de Datos

1.1. Secuencias (type sequence)

1.1.1. Concepto de Secuencia

Una secuencia es una serie de elementos que se suceden unos a otros y guardan relación entre sí.

Un ejemplo de secuencia es la sucesión de Fibonacci.

1.1.2. Secuencias en Python

Una **secuencia** en Python es un grupo de elementos con una organización interna; que se alojan de manera contigua en la memoria.

1.2. Tipos de Secuencias en Python

- Listas
- Tuplas
- String
- Rangos

1.2.1. Rangos (*range*)

Los Rangos son un tipo de dato que representa una secuencia de números inmutable.

Para crear secuencias numéricas utilizamos la función *range* que nos brinda Python, que se puede usar de diferentes formas:

- range(fin) : Nos crea una secuencia numérica desde el 0 hasta ‘fin’, el número que le indiquemos.
- range(comienzo, fin) : Nos crea una secuencia numérica que va desde ‘comienzo’ hasta ‘fin’.
- range(comienzo, fin, paso) : Nos crea una secuencia numérica que va desde ‘comienzo' hasta ‘fin’ y va aumentando en la cantidad que le indiquemos en ‘paso’.

Ejemplos:

- range(20) : Crea una secuencia que va de 0 al 19 (inclusives).
- range(10, 20) : Crea una secuencia que va del 10 al 19 (inclusives).
- rango(10, 20, 2) : Crea una secuencia que contiene los números [10, 12, 14, 16, 18]

Observaciones:

- Se crean en un intervalo de valores enteros.
- Los números de comienzo, fin y paso pueden ser negativos.
- El valor de ‘paso’ es 1 a menos que se indique lo contrario.

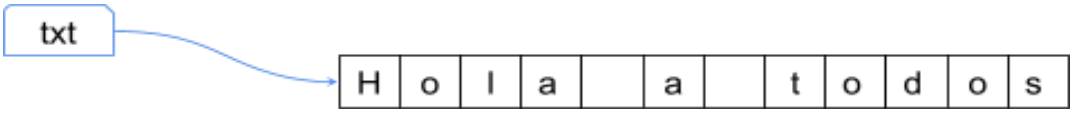
1.2.2. Cadenas de Texto (*String*)

Un **string** es un tipo de secuencia que sólo admite caracteres como elementos. Por eso podemos definir a un **string** como un conjunto de caracteres que se almacenan de manera contigua y tienen una organización interna.

¿Cómo se guarda internamente una **secuencia (sequence)**?

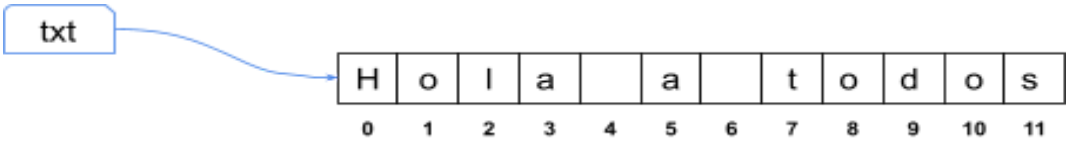
```
txt='Hola a todos'
```

Internamente se almacenará como 12 caracteres contiguos en la memoria:



Podemos observar que el orden de los caracteres importa. El primer caracter de **txt** es `'H'` y el último `'s'`.

Las secuencias enumeran sus posiciones de 0 en adelante. Es decir que el primer elemento ocupa la posición 0 (H), el segundo la 1 (o), y así sucesivamente.



1.2.3. Tuplas (*type tuple*)

Otra tipo de secuencia bastante empleada y muy sencilla de manejar son las **tuplas.**

Las **tuplas** son una **secuencia inmutable** (no se pueden editar) de objetos que pueden ser de cualquier tipo, a diferencia de los **string** que sólo pueden tener **caracteres**. Es decir que las **tuplas** son Estructuras de Datos Genéricas, porque con guardar sólo caracteres en diferentes cajoncitos de mi cajonera no alcanza: puede que quisiera guardar cosas diferentes en cada "cajón", en cada espacio o posición de la tupla, incluso otras estructuras.

Una **tupla** se escribe entre ( ), y sus elementos van separados por coma.

Ejemplos de tuplas:

```
(8,)
```

```
()
```

```
(1,2, 'hola')
```

```
((a,2),b)
```

También se pueden escribir sin los paréntesis (salvo que estén dentro de otra estructura de datos). El último ejemplo se puede escribir como:

```
(a,2),b
```

Y

```
2,3,4,10
```

También es una **tupla**

```
(8,)
```

Observación:

```
(8,) es diferente de (8)
```

Es por la manera que tiene Python de saber qué es una **tupla;** y no la expresión aritmética cuyo valor es 8. Es algo importante, los tipos de datos son diferentes, es decir, (8) es un entero y (8,) es una tupla, por lo que uno permite operaciones que el otro no, y viceversa.

A diferencia de los string y los tipos de datos que vimos hasta acá, con las tuplas tenemos la limitación de no poder modificar su contenido ni su tamaño. Para cualquier modificación que se requiera, será necesario copiarla modificada en otra **tupla**. Y esto es porque son **Estructuras de Datos Inmutables**: una vez que se definen, no puedo cambiarlas. Entonces, si intentamos hacer una asignación a un elemento de una tupla, nos saltará un error. Por ejemplo, si tenemos la tupla:

```
a = (8, 2, 5)
```

```
a[0] = 1
```

Esto nos va a tirar un error cuando intentemos ejecutar el programa.

¿Cómo modificar una tupla?

Por ejemplo:

```
a=(1,2)
```

Si quisiera obtener una **tupla** con dos repeticiones más de esos elementos, podría aplicarle el operador repetición:

```
a=a*3
```

Por lo tanto, si quiero modificar el contenido original y que contenga las repeticiones debo **pisarlo**, es decir, asignar la nueva tupla.

Mostraremos un uso práctico de tuplas para generalizar la construcción de menús de opciones:

```
def menu(opciones):
    ''' arma menú de opciones y devuelve selección entera
    recibe tupla con opciones de menú'''

    print('Selecciona una opción')
    for i in range(1,len(opciones)):
        print(i,'-',opciones[i])
    opc=int(input())
    while opc not in range(1,len(opciones)):
        opc=int(input())
    return opc
```

```
quesos=('','cheddar','dambo','muzzarela','brie','cremoso','sin queso')
panes=('','semillas','árabe','centeno','pebete','francés')
carnes=('','hamburguesa','jamón cocido','jamón
                      crudo','lomito','salchicha','mortadela','veggie')
salsas=('','mayonesa','guacamole','ketchup','salsa golf','barbacoa',
                                  'ranch','sin salsa')
acomp=('','tomate','lechuga','pepinillos','verduras cocidas',
                            'berenjena escabeche','sin extras')
print('Armá tu sandwich')
op1=menu(panes)
op2=menu(carnes)
op3=menu(quesos)
op4=menu(acomp)
op5=menu(salsas)
print('Tu pedido de sandwich de pan',panes[op1],'saldrá pronto')
print('Detalle:',carnes[op2],quesos[op3],acomp[op4],salsas[op5],sep='\n')
```

1.2.4. Listas (*type list*)

Python hace una gran división en sus clases de objetos estructurados con respecto a la posibilidad que tiene cada una de ellos de cambiar dinámicamente, en partes.

Aprenderemos a usar una **secuencia** súper flexible y potente para organizar datos: la **lista**. Las listas son una secuencia genérica **mutable**, lo que marca una gran diferencia respecto a las tuplas.

Son **secuencias**, por lo tanto, **la organización interna importa** y admiten, como todas las secuencias, un acceso directo (a el o los elementos que deseo); son **mutables**, por ello pueden cambiar de tamaño y de contenido total, o por partes, y son **heterogéneas**, por lo que pueden contener cualquier tipo de objeto (otra lista, por ejemplo).

Una **lista** se escribe entre [ ]

Por ejemplo:

```
[1,2]
[4]
[True,'Hola',56,(1,2)]
```

Y se puede asignar como cualquier objeto de datos

```
a=[1,2,3]
```

Cuando precisamos una **lista** vacía, la creamos de la siguiente manera:

```
lis=[]
```

Para tener en cuenta:

```
a=[1,2,3]
b=[1,2,3]
```

No es igual a:

```
a=[1,2,3]
b=a
```

En el primer caso, si bien **a** y **b** tienen el mismo tamaño y los mismos valores, son dos variables completamente diferentes. En cambio, en el segundo caso la variable **b** decimos que es una referencia a la variable **a,** lo que implica que si modificamos una de las dos, el cambio se ve reflejado en ambas variables.

Si efectivamente queremos tener dos estructuras separadas para que al modificar los datos de una se preserven los datos originales en otra, debemos forzar su copia.

Una forma práctica de hacerlo es emplear el método **copy**()

```
b=a.copy()
```

Ahora sí copiará los contenidos de la lista **a** en otra parte de la memoria.

La operación:

```
lis=[1,2,'ya']
lis[2]=3
```

A diferencia de con las tuplas, es válida.

Si hacemos:

`print(lis)` la salida será `[1,2,3]`

A **lis** podremos agregarle elementos empleando métodos que realizan dos trabajos: agregan un cajoncito a la estructura y permiten colocar algo dentro (**append**(), **insert**(),**extend**()) o quitárselos también (**pop**(), **remove**(),**clear**()).

```
lis.append(4)
print(lis) #la salida sera [1,2,3,4]
```

Observación: *El método* ***append()*** *modifica directamente* ***lis*** *y no requiere de volver a asignar a la lista.*

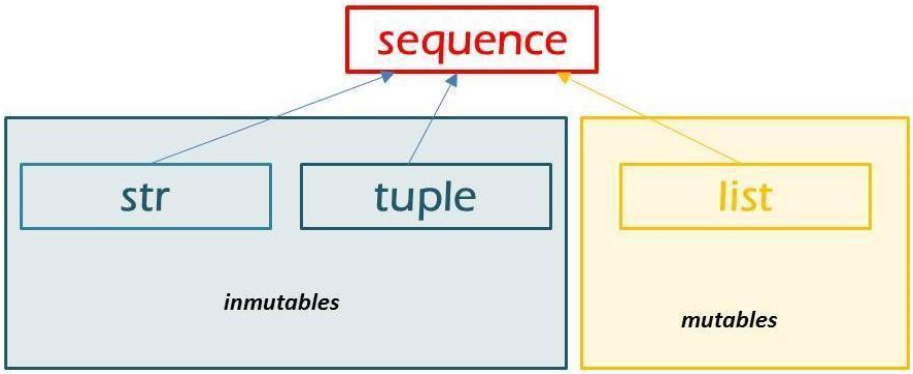
Si bien esa operación de reasignación es válida, estaríamos guardando en la variable ***lis*** *un objeto nulo (None), que es lo que el método* ***append****() devuelve: nada. Y es que* ***append****()* ***sólo modifica, no retorna ningún valor****. Y por definición en Python, todas las funciones y métodos que no devuelven nada (no hay* ***return*** *en su cuerpo) devuelven la constante nula* ***None****.*

Existe una gran variedad de trabajos necesarios para realizar sobre **listas** de objetos. Y también una buena oferta de métodos predefinidos en la clase para simplificar la tarea. Lo vamos a ver en la siguiente parte del apunte.

1.3. Operadores de Secuencias

Para todos los tipos de secuencias que vimos, tenemos definidos ciertos operadores que van a ser nuestras herramientas para programar.

x **in** s	Devuelve **True** si x pertenece a s, **False**, en caso contrario
s+t	Concatena la secuencia s y la t en ese orden
s*n	Concatena n veces la secuencia s
s[i] s[-k]	Referencia el elemento de la posición i de la secuencia s Referencia el elemento que está k posiciones antes del final
s[i:j]	Referencia la porción de la secuencia s que va del elemento i al j-1
s[i:j:k]	Referencia la porción de la secuencia s que va del elemento i al j-1, con paso k
>,<,>=,<=,!=,==	Se pueden comparar dos secuencias comparables
len(s)	Devuelve la longitud de la secuencia s
min(s)	Devuelve el mínimo elemento de s
max(s)	Devuelve el máximo elemento de s
sorted(s)	Devuelve s ordenada en forma de lista
reversed(s)	Devuelve s invertida
s.count(x)	Devuelve el número total de ocurrencias de x en s
s.index(x[, i[, j]])	Devuelve el índice de la primera ocurrencia de x en s (en la posición i o superior, y antes de j)



1.3.1. Métodos Específicos de la clase list (Lista)

La clase lista tiene sus propios métodos (funciones predefinidas) que se pueden utilizar para manipular sus contenidos. Dejamos aquí varios de ellos, los más genéricos:

Nombre	Descripción	Ejemplo de Uso
append(valor)	Agrega el elemento valor al final de la lista.	`miLista.append(5)`
insert(posic,valor)	Inserta el elemento valor en la posición posic.	`miLista.insert(2, 10)`
remove(valor)	Quita de la lista el elemento valor.	`miLista.remove(7)`
pop([índice])	Quita de la lista el elemento de la posición índice.	`valor = miLista.pop(3)`
extend(otraLista)	Agrega los elementos de otraLista al final de la lista.	`miLista.extend([8, 9, 10])`
sort()	Ordena la lista.	`miLista.sort()`
sorted()	Devuelve una nueva lista ordenada (sin modificar la original).	`lista_ordenada = sorted(miLista)`
reverse()	Invierte el orden de la lista. * puede que este método no esté habilitado para su uso en la herramienta replit. En ese caso, se	`miLista.reverse()`

	recomienda este sustituto: `miLista[::-1]`	
count(valor)	Cuenta la cantidad de apariciones de valor en la lista. Este método es válido para cualquier secuencia.	`cantidad = miLista.count(5)`
index(valor)	Devuelve el índice de la primera aparición de valor. Este método es válido para cualquier secuencia.	`indice = miLista.index(8)`
len()	Devuelve la longitud de la lista. Este método es válido para cualquier secuencia.	`longitud = len(miLista)`
clear()	Elimina todos los elementos de la lista. Otra opción es pisar el valor de la lista con una lista vacía: `miLista = []`	`miLista.clear()`
copy()	Crea una copia superficial de la lista.	`copia = miLista.copy()`
max()	Devuelve el valor máximo de la lista.	`maximo = max(miLista)`
min()	Devuelve el valor mínimo de la lista.	`minimo = min(miLista)`
sum()	Devuelve la suma de los elementos de la lista.	`suma = sum(miLista)`

1.4. Ejemplos

¿Cómo cargamos una **lista** con datos ingresados por el usuario? ¿Cómo mostramos el contenido de una **lista**?

```
#Ej de carga e impresión de listas
nombres=[]
nom=input('Ingrese un nombre, * para salir:  ')
while nom!='*':
    nombres.append(nom)
    nom=input('Ingrese un nombre, * para salir:  ')
print('Lista de Nombres')
print(nombres)
print('Salida detallada')
for n in nombres:
    print(n)
```

Considerando que las listas son secuencias podemos usar sólo partes de ellas, por ejemplo, para un for.

```
#Ejemplo con Listas
# for con una parte de lista
from random import randint
a=[]
for i in range(20):
    a.append(randint(1,35))
print('Segunda Mitad')
for n in a[10:]:
    print(n,end=' ')
print()
print('Primera Mitad')
for n in a[:10]:
    print(n,end=' ')
print()
```

En este caso creamos una lista de 20 números generados aleatoriamente entre 1 y 35, y utilizamos el for para imprimir los primeros 10 números de la lista.

1.5. Accediendo a los elementos de una secuencia

Dado que podemos armar una lista con cualquier colección de objetos, podríamos crear una lista de listas. Este recurso es el que podemos emplear para representar matrices o tablas de datos de manera sencilla; se puede armar listas de filas, donde cada fila es una lista (si necesitaré modificar los datos), y también puede ser cada fila una tupla (si no habrá cambios una vez que se haya agregado la fila).

Cuando en una **lista** (o estructura de datos en general) hay otro objeto estructurado, para poder acceder a objetos individuales que están dentro de otros, se debe referenciar cada nivel por separado, de izquierda a derecha, siguiendo el orden desde el más externo al más interno.

Ej:

```
lis=[1,2,3]
```

Para referenciar al **2** basta con hacer:

```
lis[1]
```

En la siguiente lista:

```
lis=[1,(1,0,2),3]
```

Para referenciar el **2** (que está como tercer elemento de una tupla, que a su vez es el segundo elemento de la lista):

```
lis[1][2]
```

Y si tuviéramos:

```
lis=[('uno', 'dos', 'tres'),(1,2,3)]
```

Observamos que tenemos una lista con dos elementos y cada uno es una tupla con tres elementos. La segunda tupla tiene elementos simples (no son estructuras o iterables), números; mientras que la primera tiene tres elementos que son una secuencia cada uno.

¿Cuántos niveles debes indicar para referenciar la **u** de **uno** ?

3

¿Cómo sería eso?

```
lis[0][0][0]
```

El **[0]** de la izquierda referencia la tupla `('uno', 'dos', 'tres').` El siguiente **[0]** (siempre sentido izquierda-derecha) referencia la string `'uno'.` El último **[0]** referencia `'u'` (de `'uno'`)

Para no perdernos en el intento de utilizar estructuras complejas, sólo debemos tener un esquema gráfico a mano para individualizar en qué nivel de profundidad se encuentra el dato que queremos alcanzar. Así se va bajando de nivel en nivel, referenciándolos uno por uno con un par de [] que se agregarán de izquierda a derecha por cada nivel que descendamos.

Volvamos a nuestro ejemplo inicial de armar sándwiches a elección. A diferencia de la versión anterior (pondremos menos opciones para no tener un programa tan largo) agregaremos los precios de cada componente, así sabremos cuánto nos costará el sándwich elegido. Hemos decidido armar estructuras tipo tablas para las alternativas de los menús, agregando el precio de cada opción. Cómo son estructuras que no cambiarán durante la ejecución del programa, las establecimos como tuplas de tuplas, pero podrían haber sido listas también (en caso de ser lista, la

forma en que accedemos es la misma). También armaremos una lista de los precios que vayamos incluyendo en nuestro sándwich, así al final podremos saber cuánto pagar. En este último caso, sí empleamos una lista, ya que partiremos con una lista de gastos vacía y le iremos agregando valores a medida que vamos seleccionando. Otra opción podría haber sido calcular el precio paso a paso en cada elección, pero optamos por emplear una lista en este caso, para ejemplificar su uso.

```
def menu(opciones):
    ''' arma menú de opciones y devuelve selección entera
    recibe tupla con opciones de menú'''

    print('Selecciona una opción')
    for i in range(1,len(opciones)):
        print(i,'-',opciones[i][0])
    opc=int(input())
    while opc not in range(1,len(opciones)):
        opc=int(input())
    return opc


quesos=('',('cheddar',75),
        ('dambo',60),
        ('sin queso',0))
panes=('',('árabe',70),
       ('pebete',70),
       ('francés',60))
carnes=('',('hamburguesa',200),
        ('jamón cocido',150),
        ('lomito',200),
        ('salchicha',100),
        ('veggie',0))
sand=[]
print('Armá tu sandwich')
op1=menu(panes)
sand.append(panes[op1][1])
op2=menu(carnes)
sand.append(carnes[op2][1])
op3=menu(quesos)
sand.append(quesos[op3][1])
print('Tu pedido de sandwich de pan',panes[op1][0],'saldrá pronto')
print('Detalle:',carnes[op2][0],quesos[op3][0],sep='\n')
print('Abonar: $%.2f'%sum(sand))
```

1.6. Métodos de la clase str (String)

Algunos métodos más comunes para la clase String son:

Nombre	Descripción	Ejemplo de Uso
capitalize()	Devuelve el string con la primera letra en mayúscula.	`"hola".capitalize()`
center(ancho[,relleno])	Devuelve el string centrado con relleno a los costados.	`"Python".center(10, "*")`
count(valor)	Devuelve la cantidad de veces que aparece "valor" en el string.	`"banana".count("a")`
find(substring[,desde[,hasta]])	Devuelve la primera posición de comienzo del substring en el string.	`"python".find("th")`
rfind(substring[,desde[,hasta]])	Devuelve la última posición de comienzo del substring en el string.	`"python programming".rfind("ing")`
format(args,*)	Devuelve el string formateado con valores sustituidos.	`"Mi nombre es {} y tengo {} años".format("Juan", 25)`
upper()	Devuelve el string en mayúsculas.	`"hola".upper()`

lower()	Devuelve el string en minúsculas.	`"Hola".lower()`
strip()	Devuelve el string sin espacios en blanco al inicio y al final.	`" Python ".strip()`
replace(viejo, nuevo)	Devuelve el string con todas las apariciones de "viejo" reemplazadas por "nuevo".	`"Hello, World!".replace("Hello", "Hi")`
split([separador])	Devuelve una lista de substrings separados por "separador".	`"apple,banana,grape".split(",")`
join(iterable)	Devuelve un string que es la concatenación de los elementos en el iterable.	`",".join(["apple", "banana", "grape"])`
isdigit()	Devuelve True si todos los caracteres son dígitos.	`"12345".isdigit()`
isalpha()	Devuelve True si todos los caracteres son letras.	`"Python".isalpha()`

startswith(substring)	Devuelve True si el string comienza con "substring".	`"Hello, World!".startswith("Hello")`
rindex(substring[,desde[,hasta]])	Devuelve la última posición de comienzo del substring en el string.	`"python programming".rindex("ing")`
join(iterable)	Devuelve un string que es la concatenación de los elementos en el iterable, intercalados con el string.	`",".join(["apple", "banana", "grape"])`
ljust(ancho[,relleno])	Justifica el string hacia la izquierda con relleno.	`"Python".ljust(10, "-")`
rjust(ancho[,relleno])	Justifica el string hacia la derecha con relleno.	`"Python".rjust(10, "-")`
lower()	Devuelve el string en minúsculas.	`"Hola".lower()`
upper()	Devuelve el string en mayúsculas.	`"hola".upper()`
maketrans(x[,y[,z]])	Asocia en un diccionario los correspondientes caracteres de las cadenas x e y.	`str.maketrans("aeiou", "12345")`

translate(pares)	Devuelve el string con los caracteres asociados en el diccionario pares reemplazados.	`"hello".translate(str.maketrans("aeiou", "12345"))"`

1.6.1. Ejemplos

Ejemplo 1:

```
texto="Bienvenido a mi aplicación{0}"
print(texto.format(" en Python"))
```

Retorna:
Bienvenido a mi aplicación en Python

Ejemplo 2:

```
texto="Bruto: ${bruto} + IVA: ${iva} = Neto: ${neto}"
print(texto.format(bruto=100,iva=21,neto=121))
```

Retorna:
Bruto: $100 + IVA: $21 = Neto: $121

Ejemplo 3:

```
tup=('a','b','c')
print('-'.join(tup))
```

Retorna:
a-b-c

Ejemplo 4:

```
vocales="aeiou"
numeros="12345"
texto="murcielagos"
print(texto.maketrans(vocales,numeros))
```

Retorna:
{97: 49, 111: 52, 117: 53, 101: 50, 105: 51}

Ejemplo 5:

```
vocales="aeiou"
acentos="áéíóú"
texto="murcielagos"
parejas=texto.maketrans(vocales,acentos)
```

```
print(texto.translate(parejas)
```

Retorna:

múrcíélágós

Bibliografía Adicional

Wachenchauzer, R. (2018). *Aprendiendo a programar usando Python como herramienta*. Capítulo 7.