Secuencias, Tuplas y Listas

Unidad 4

Apunte de cátedra

Pensamiento Computacional (90)
Cátedra: Camejo

.UBA XXI

Tipos de Estructuras de Datos

Secuencias (*type sequence*)

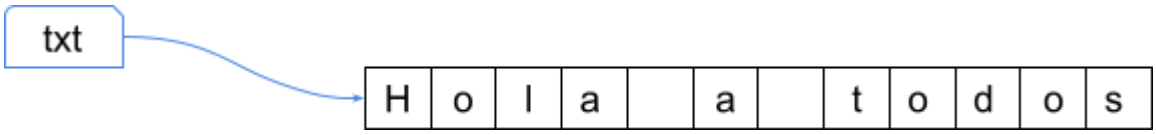
Una **secuencia** en Python es un grupo de elementos con una organización interna; que se alojan de manera contigua en la memoria.

Una **string** es una **Subclase de secuencia** que sólo admite caracteres como elementos. Por eso podemos definir a una **string** como un conjunto de caracteres que se almacenan de manera contigua y tienen una organización interna. Muchas de las particularidades que hemos conocido y empleado para las **string** (**str**) en realidad son aplicaciones de facilidades, funciones y operadores definidos para la **Clase Secuencia**, que es más abarcativa (o sea, su superclase).

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

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

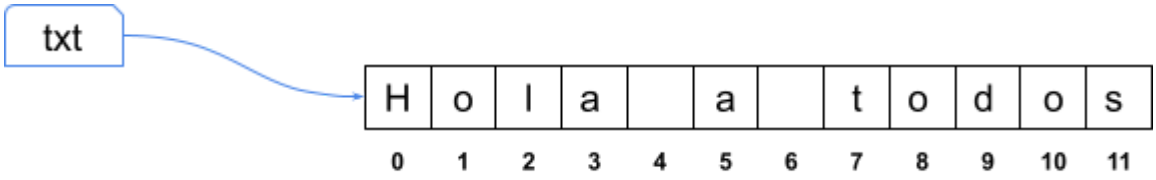
Internamente se almacenará como 12 caracteres contiguos en la memoria:



La variable **txt** apunta a la primera posición.

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

Las secuencias numeran sus posiciones de 0 en adelante (una herencia del lenguaje C). Es decir que el primer elemento ocupa la posición 0, el segundo la 1, y así sucesivamente.



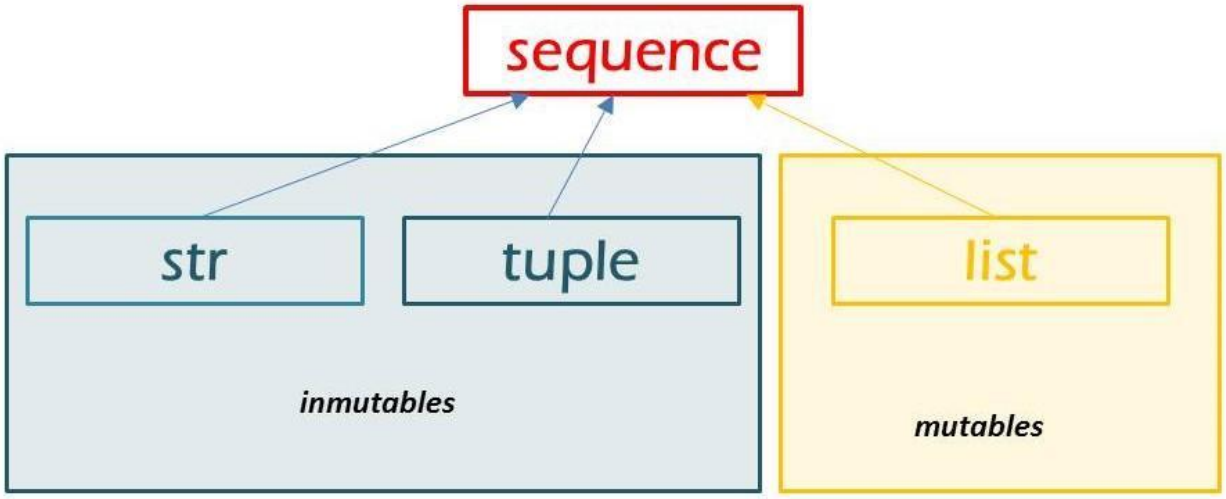
Tenemos varias cosas interesantes predefinidas para **secuencias**, conocerlas resultará beneficioso a la hora de programar. Además, es importante saber que todas las Estructuras de Datos que veamos que sean del tipo **secuencia** (es decir, que sean subclase de **sequence**) dispondrán de todas estas facilidades.

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

Métodos de la Clase sequence (Secuencia)

s.***count****(elemento|substring[,desde[,hasta]])* devuelve la cantidad de veces que aparece el elemento|substring en la secuencia s
s.***index****(elemento|substring[,desde[,hasta]])* devuelve la posición de la primera ocurrencia elemento o del comienzo de substring en s



Tuplas (*type tuple*)

Otra subclase de las secuencias bastante empleada y muy sencilla de manejar son las **tuplas.**

Las **tuplas** son una **secuencia** inmutable y heterogénea de objetos que pueden ser de cualquier tipo, a diferencia de las **string** que sólo pueden tener **caracteres**. Es decir que las **tuplas** son Estructuras de Datos Genéricas. ¡Claro! Porque con guardar sólo caracteres en diferentes cajoncitos de mi cajonera no alcanza. Puede que quisiera guardar cosas diferentes en cada cajón; incluso otras Estructuras.

*Por ser una subclase de la clase **secuencia** son aplicables las funciones, métodos, operadores y modos de acceso que están definidos para las **secuencias**.*

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,)` ¿Esto se ve raro no?

Si! Pero es la manera que tiene Python de saber qué ésa es una **tupla;** y no la expresión aritmética cuyo valor es 8. Es algo importante, porque no es el mismo uso el que le dará a cada uno de esos datos. Incluso se almacenarán en memoria de manera diferente. Lo bueno de emplear esta notación exótica (que sólo es obligatoria para **tuplas** de un único elemento) es que si te quedó colgando una coma al final en una **tupla** de más elementos Python te retribuirá siendo comprensivo y entenderá que se te pasó por error sin decir nada.

Al igual que con las string y los tipos de datos que vimos hasta acá tenemos la limitación de no poder modificar su contenido ni su tamaño. Cualquier modificación que se requiera será necesario copiarla modificada en otra **tupla**. Y esto es porque son Estructuras de Datos **Inmutables**, como su nombre lo indica, una vez que se alojan en la memoria, no puedo cambiarlas. Si necesito cambiar su valor o modificar parte de él se guardará la versión útil en otra parte de la memoria y la etiqueta (el nombre de la variable) apuntará al inicio de la nueva región de memoria. La versión anterior quedará abandonada y descolgada a la espera de que pase el proceso de recolección de basura (el packman de memoria que recupera como libres los espacios de memoria ocupados por datos descolgados e inaccesibles).

Por ejemplo:

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

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

	Salida
`print(a*3)`	`(1,2,1,2,1,2)`

Pero, si hago:

	Salida
`print(a*3)`	`(1,2,1,2,1,2)`
`print (a)`	`(1,2)`

Por lo tanto, si quiero modificar el contenido original y que contenga las repeticiones debo pisarlo (igual que con una variable simple o una **string**).

```
a=a*3
```

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')
```

Recapitulando:

```
num=(0,)
t=('ana',23, 'y',False,0.5,-6, 'yo',num)
t2=()
```

Recordemos que las tuplas son **secuencias inmutables**, por eso las siguientes aplicaciones son válidas:

```
nombre=t[0]
ultimos=t[5:]
ultConstNum=t[-3]
t=t[0:3:2]+t[6:7]
i=8
t2+=(i,)
t2+=(i+1,)
```

Pero…

~~`t[1]='*'`~~
~~`t2+=(10)`~~
No!

En el primer caso la operación es inválida porque no puedo modificar una estructura inmutable (las tuplas lo son)

En el segundo caso porque el operador + está definido entre números (adición aritmética) y entre secuencias (concatenación), pero no se puede aplicar a una mezcla de tipos. No puedo agregarle un número (como (10) en el ejemplo) a una tupla, podría concatenarle la tupla (10,), eso sería otra cosa.

Listas (*type list*)

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

Vamos a incorporar a nuestro conjunto de herramientas la favorita de los programadores. Aprenderemos a usar una **secuencia** super flexible y potente para organizar datos: la **lista**. Las **listas** son las primas con onda de las tuplas.

Son una secuencia genérica heterogénea **mutable**. Y en esto último radica un abismo de diferencia!

Son **secuencias**, por lo tanto, la organización interna **importa** y admiten, como todas las secuencias, un acceso directo (tomo directamente él o los elementos que deseo, si conozco en qué cajoncito se aloja, Python va directo a su ubicación); 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).

Reúne las ventajas de tipos de estructuras contrapuestas habitualmente en la mayoría de los Lenguajes de Programación. Tiene la facilidad de acceso de los arreglos (**array**), lo que permite utilizar algoritmos más sencillos de manipulación de datos; sin perder la flexibilidad dinámica, como el resto de las estructuras dinámicas tradicionales (c**olas, pilas, listas propiamente dichas**, etc.). Quizás lo más cercano a ellas sean los **arreglos dinámicos** de C; que son un híbrido propio de este lenguaje. Pero con el adicional que permiten almacenamiento heterogéneo (lo que les brinda la utilidad de una estructura **record**). Y además de todo, las maneja Python.

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 hay **dos regiones diferentes en memoria**, mientras que en el segundo **a** y **b** comparten la zona de memoria y son el **mismo objeto**.

Primer Atajo:

Cada vez que se asigna a una **lista** (no a un elemento) un objeto constante (como `[1,2,3],[],['hola']`), la **lista** apunta a una **región de memoria propia**.

Pero como es un objeto **mutable** también admite que se modifique un elemento en particular, o un subgrupo. Al ser una **secuencia**, admite todas las reglas de referenciación de un elemento o parte de elementos que conocemos para **secuencias**.

Segundo Atajo:

Si quiero obtener una **copia efectiva** de una lista (en otra región de memoria) no alcanza con asignar esa lista a otra variable:

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

Esto solo produciría dos maneras diferentes de acceder a los mismos datos. Como cuando te ponen un apodo. Te llamas José, Pepe para los amigos. Al final, si llaman a José o a Pepe viene la misma persona.

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 adentro (**append**(), **insert**(),**extend**()) o quitárselos también (**pop**(), **remove**(),**clear**()).

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

¡Importante!
*Observemos que el método **append()** modifica directamente **lis** y no requiere*

~~***lis=lis.append(4)***~~

De hecho, esa operación es válida, pero guardaríamos en ***lis*** *un objeto nulo (NONE), que es lo que el método* ***append****() devuelve: nada. Y es que* ***append****()* ***sólo*** *modifica, no entrega 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. Te copio una lista de ayuda, aunque puedes buscar más en la documentación del Lenguaje.

Métodos de la Clase list (Lista)

miLista.***append(****valor****)*** agrega el elemento *valor* al final de miLista
miLista.***insert****(posic,valor)* inserta el elemento *valor* en la posición *posic*
miLista.***remove****(valor)* quita de miLista el elemento *valor*
miLista.***pop****([índice])* quita de miLista el elemento de la posición *índice*. Si no se usa este parámetro, quita el último elemento. Este método devuelve el valor retirado
miLista.***extend****(otraLista)* agrega al final de miLista *otraLista*
miLista.***sort****([****reverse=True****][,****key****=función])* ordena miLista. Si se emplea el parámetro **reverse** en orden descendente, si se usa **key**, con criterio de ordenamiento *función*
miLista.***reverse()*** invierte el orden de miLista (el primero pasa a ser el último)
miLista.***count(****valor****)*** cuenta la cantidad de apariciones de *valor* en miLista

¿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()
```

Podemos armar una lista con cualquier colección de objetos. Claramente podemos colocar como elementos alguna otra estructura. Este recurso es el que podemos emplear para representar tablas de datos de manera sencilla. Podemos armar listas de filas, donde cada fila es una lista (si necesitaré modificar los datos) o una tupla (si no habrá cambios una vez que se haya agregado la fila). Organizar y manipular tablas o estructuras más complejas es sencillo, sólo debemos prestar atención a cómo guardamos la información para no perdernos al armar el camino de acceso a un dato particular. El problema no es complejo, simplemente no es apto para distraídos.

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)]
```

donde claramente tenemos una lista con dos elementos. Cada elemento de la lista 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 secuencia. Cuántos niveles debes indicar para referenciar la **u** de **uno** ?

3

¿Cómo sería eso?

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

[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'`)

Tercer Atajo:

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. Como son estructuras que no cambiarán durante la ejecución del programa las mantuvimos como tuplas de tuplas, pero podrían haber sido listas también. Se operarán igual. 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. Claro que podríamos haber calculado 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))
```

Métodos de la Clase str (String)

s.**capitalize**() devuelve una copia del string con la primera letra en mayúscula y el resto en minúscula

s.**center**(ancho[,relleno]) string centrado con ese relleno a los costados

s.**find**(substring[,desde[,hasta]]) devuelve la primera posición de comienzo del substring en s

s.**rfind**(substring[,desde[,hasta]]) devuelve la última posición de comienzo del substring en s

s.**format**(args,*) devuelve s formateada sustituyendo dinámicamente un texto

Ejemplo1:

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

Retorna:
Bienvenido a mi aplicación en Python

Ejemplo2:

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

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

s.**rindex**(substring[,desde[,hasta]]) idem a s.rfind

s.**join**(iterable) arma una string uniendo los elementos de iterable e intercalándolos con s

Ejemplo:

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

Retorna:
a-b-c

s.**ljust**(ancho[,relleno]) justifica hacia la izquierda

s.**rjust**(ancho[,relleno]) justifica a derecha

s.**lower**() devuelve s en minúsculas

s.**upper**() devuelve s en mayúsculas

s.**maketrans**(x[,y[,z]]) asocia en un diccionario los correspondientes caracteres de las cadenas x e y

Ejemplo:

```
vocales="aeiou"
numeros="12345"
```

```
texto="murcielagos"
print(texto.maketrans(vocales,numeros))
```
Retorna:
{97: 49, 111: 52, 117: 53, 101: 50, 105: 51}

s.**translate**(pares) devuelve s con los caracteres asociados en el diccionario pares remplazados

Ejemplo:
```
vocales="aeiou"
acentos="áéíóú"
texto="murcielagos"
parejas=texto.maketrans(vocales,acentos)
print(texto.translate(parejas)
```
Retorna:
múrcíélágós

s.**replace**(antes,ahora[,cantidad]) Reemplaza el substring de antes por el de ahora

s.**strip**() elimina los espacios del inicio y fin del string

s.**lstrip**()elimina los espacios del inicio

s.**rstrip**() elimina los espacios del fin

s.**swapcase**() devuelve s con las mayúsculas convertidas en minúsculas y viceversa

s.**split**([separador[,maximaDivision]]) devuelve una lista cuyos elementos son las partes del texto separadas por separador. Si se omite separador toma blancos

s.**rsplit**([separador[,maximaDivision]])ídem a derecha

s.**startswith**(prefijo[,desde[,hasta]) devuelve True si s comienza con prefijo, si no False

s.**endswith**(sufijo[,desde[,hasta]) devuelve True si s termina con sufijo, si no False

s.**zfill**(ancho) Rellena con ceros a la izquierda hasta el ancho

s.**title**() devuelve s en minúsculas con cada palabra inicializada en mayúsculas

s.**isnumeric**() devuelve True si s es numérico, si no False

s.**isalnum**() devuelve True si s es alfanumérico, si no False

s.**isalpha**() devuelve True si s es alfabético, si no False

s.**isdecimal**() devuelve True si s tiene sólo dígitos decimales (0-9), si no False

s.**isdigit**() devuelve True si s tiene sólo números (los dígitos decimales de 0 a 9), si no False

s.**isprintable**() devuelve True si s es un carácter imprimible, si no False

s.**isspace**() devuelve True si s es espacio, si no False

s.**istitle**() devuelve True si s es un título, si no False

s.**isidentifier**() devuelve True si s es un nombre válido de objeto, si no False

s.**isupper**() devuelve True si s está en mayúsculas, si no False

s.**islower**() devuelve True si s está en minúsculas, si no False

Bibliografía Adicional

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