Listas

March 30, 2022

Ejecuta el siguiente bloque de código siempre antes de ejecutar el resto del notebook.

[ ]:
```
from IPython.core.magic import  Magics, magics_class, cell_magic, line_magic

@magics_class
class Helper(Magics):

  def __init__(self, shell=None,  **kwargs):
    super().__init__(shell=shell, **kwargs)

  @cell_magic
  def debug_cell_with_pytutor(self, line, cell):
    import urllib.parse
    url_src = urllib.parse.quote(cell)
    str_begin = '<iframe width="1000" height="500" frameborder="0" src="https://
↪pythontutor.com/iframe-embed.html#code='
    str_end   = '&cumulative=false&py=3&curInstr=0"></iframe>'
    import IPython
    from google.colab import output
    display(IPython.display.HTML(str_begin+url_src+str_end))

get_ipython().register_magics(Helper)
```

1 Estructuras de datos

Hasta ahora hemos trabajado almacenando datos en variables, esto es útil cuando necesitamos hacer operaciones sobre estos y reportar los resultados. Pero ¿Qué pasa si queremos trabajar con una cantidad de datos mayor? En este caso podríamos asignar una variable para cada dato, pero podría ocurrir que sean tantos que se vuelva una tarea tediosa, o incluso, que ni siquiera sepamos cuantos datos necesitamos. Para remediar este problema podemos usar las estructuras de datos, estas son una forma de organizar los datos en la computadora, de tal manera que nos permita realizar unas operaciones con ellas de forma muy eficiente. Python ofrece 4 tipos de estructuras de forma nativa, con las cuales podemos crear otras mucho más complejas según la necesidad de nuestro programa.

1.1 Listas

Consideremos una situación en la que necesitamos almacenar cinco números enteros. Si utilizamos los conceptos de variables, entonces necesitamos cinco variables y el programa será el siguiente:

[ ]:
```
num1 = 10
num2 = 20
num3 = 30
num4 = 40
num5 = 50

print(num1,num2,num3,num4,num5)
```

```
10 20 30 40 50
```

Ahora regresemos a la discusión anterior, que pasa si necesitamos 5000 números, ¿creamos 5000 variables? Por supuesto que no. Para manejar estas situaciones, casi todos los lenguajes de programación proporcionan un concepto llamado array o en nuestro caso una lista. Una lista es una estructura de datos que puede almacenar una colección de tamaño indeterminado de elementos.

```
lista = [] #lista vacía
lista[2] = 5 #asignación de un valor a una posición
```

En lugar de declarar variables individuales, como `num1`, `num2`, ..., `num99`, sólo tenemos que declarar una lista y utilizar la notación de corchetes `num[0]` que vimos en la sección de string en los tipos de datos: `num[0], num[1]`,..., `num[99]` para representar las variables individuales. Aquí, 0, 1, 2, ...99 son los índices asociados a la variable `num` y se utilizan para representar elementos individuales disponibles en la lista. Todas las listas inician el conteo de sus índices en la posición 0, de este modo, la posición 5 de la lista `num` sería `num[4]`. Los índices son los identificadores numéricos que diferencian cada posición de la lista, este valor siempre es un número entero. Todas las listas están formadas por posiciones de memoria contiguas dentro de la memoria RAM, La dirección más baja corresponde al primer elemento y la más alta al último.

Si queremos definir una lista con algunos valores en su interior, basta con escribirlos separados por coma dentro de los corchetes `[ ]`:

[1]:
```
lista = [1,2,3,4]
print(lista)
print(lista[2])
```

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

[ ]:
```
print(lista[1.5])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-806c4651355a> in <module>()
----> 1 print(lista[1.5])
```

```
TypeError: list indices must be integers or slices, not float
```

Las listas además nos permiten modificar los valores ya existentes en su interior, si conocemos la posición del dato que queremos modificar, basta con usar la notación de corchetes `[ ]` e indicar el índice del dato dentro de la operación de asignación.

```
lista[2] = 5 #asignación de un valor a una posición
```

[3]:
```
print('Lista original', lista)
lista[2] = 5 #asignación de un valor a una posición
print('Lista modificada', lista)
```

```
Lista original [1, 2, 3, 4]
Lista modificada [1, 2, 5, 4]
```

Las listas en su interior pueden almacenar cualquiera de los tipos de datos que estudiamos anteriormente, además también puede almacenar otras listas internamente y vale la pena resaltar que también podemos almacenar las demás estructuras de datos que veremos luego:

[ ]:
```
lista2 = ['a',0,5.14,[1,2,3]]
print(lista2)
```

```
['a', 0, 5.14, [1, 2, 3]]
```

Si necesitamos conocer cuántos valores tenemos almacenados en su interior, podemos usar la función `len`, si la aplicamos a la `lista2` obtendremos 4 items, esto quiere decir, que las listas al interior de otras, solo se cuentan como una sola entidad para la lista exterior.

[ ]:
```
print(len(lista2))
```

```
4
```

Intentemos acceder a los datos de la lista:

[ ]:
```
print(lista2[0])
print(lista2[1])
print(lista2[2])
print(lista2[3])
```

```
a
0
5.14
[1, 2, 3]
```

Si quisiéramos acceder a los datos de la lista interna debemos usar la notación de corchetes dobles `[ ][ ]` ¿por qué? En realidad es muy simple, la variable `lista[3]` es una lista y Python la reconoce como tal, por esta razón, al añadir los segundos corchetes, estamos indicandole a python que acceda a un valor en el interior de esta nueva lista.

[ ]:
```
print(lista2[3][2])
```

```
3
```

Es importante que tengamos en cuenta que esta notación sólo funciona en este tipo de casos y nos arrojará un error al intentar aplicarla en una posición de la lista que no sea indexable (es decir, que el dato no contenga valores con índice en su interior) De momento hemos estudiado dos tipos de datos indexables, las listas y los strings, ya que en las cadenas de texto podemos acceder a un carácter en una posición específica.

[ ]:
```
print(lista2[2][0])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-e493425a069d> in <module>()
----> 1 print(lista2[2][0])

TypeError: 'float' object is not subscriptable
```

[ ]:
```
s = 'soy un string indexado'
print(s[5],s[8])
```

```
n t
```

1.2 Inserción de elementos

Añadir nuevos elementos dentro de la lista es un proceso bastante sencillo, contamos con dos métodos, el primero consiste en usar la función `append` de las listas, escribiendolo luego del nombre de la lista y un punto:

```
lista.append(dato)
```

[ ]:
```
print('lista antes de inserción:',lista)
lista.append(5)
print('lista luego de inserción:',lista)
```

```
lista antes de inserción: [1, 2, 3, 4]
lista luego de inserción: [1, 2, 3, 4, 5]
```

El segundo método es la concatenación, es decir, unir dos listas pegando la nueva lista al final de la otra, usando el operador de inserción `+=`.

```
lista+=[dato]
```

[ ]:
```
print('lista antes de inserción:',lista)
lista+=[6]
print('lista luego de inserción:',lista)
```

```
lista antes de inserción: [1, 2, 3, 4, 5]
lista luego de inserción: [1, 2, 3, 4, 5, 6]
```

Este proceso se puede realizar con listas de cualquier longitud, no necesariamente se debe concatenar un solo dato a la vez.

[ ]:
```
print('lista antes de inserción:',lista)
lista+=[7,8,9,10]
print('lista luego de inserción:',lista)
```

```
lista antes de inserción: [1, 2, 3, 4, 5, 6]
lista luego de inserción: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

1.3 Eliminar elementos de una lista

Python cuenta con varios métodos para eliminar datos de la lista, cada uno cumple una función distinta.

1.3.1 `del`

La sentencia `del` nos permite eliminar directamente de la memoria una lista completa o el valor en una posición específica:

[ ]:
```
print('lista antes de eliminar un dato:',lista)
del lista[3]
print('lista luego de eliminar un dato:',lista)
```

```
lista antes de eliminar un dato: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
lista luego de eliminar un dato: [1, 2, 3, 5, 6, 7, 8, 9, 10]
```

[ ]:
```
del lista
print(lista)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-a365b7d8c882> in <module>()
      1 del lista
----> 2 print(lista)

NameError: name 'lista' is not defined
```

Como vemos en el bloque de arriba, al usar `del` con la lista completa, al tratar de operar nuevamente con ella, el código nos retorna un error.

[ ]:
```
lista3 = [5,1,2,8,6,4,3,9,10,56,10,9,5,1]
```

1.3.2 `remove`

Si no conocemos la posición del dato que queremos eliminar, podemos usar la función `remove` (la cual tiene una sintaxis similar a `append`)

```
lista.remove(dato)
```

`remove` recorre la lista de derecha a izquierda (o sea partiendo del índice 0) y elimina el primer valor que coincida con el dato que especificamos, si el dato no se encuentra, la función retornará un error.

[ ]:
```
print('lista antes de eliminar un dato:',lista3)
lista3.remove(8)
print('lista luego de eliminar un dato:',lista3)
```

```
lista antes de eliminar un dato: [5, 1, 2, 8, 6, 4, 3, 9, 10, 56, 10, 9, 5, 1]
lista luego de eliminar un dato: [5, 1, 2, 6, 4, 3, 9, 10, 56, 10, 9, 5, 1]
```

[ ]:
```
print('lista antes de eliminar un dato:',lista3)
lista3.remove(77)
print('lista luego de eliminar un dato:',lista3)
```

```
lista antes de eliminar un dato: [5, 1, 2, 6, 4, 3, 9, 10, 56, 10, 9, 5, 1]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-050a43da3327> in <module>()
      1 print('lista antes de eliminar un dato:',lista3)
----> 2 lista3.remove(77)
      3 print('lista luego de eliminar un dato:',lista3)

ValueError: list.remove(x): x not in list
```

1.3.3 pop

La función pop() también se puede utilizar para eliminar y devolver un elemento de la lista, pero por defecto sólo elimina el último elemento de la lista, para eliminar un elemento de una posición específica de la Lista, debemos pasar el índice del elemento a la función.

```
lista.pop() #elimina el último dato
lista.pop(5) #elimina el dato en el índice 5
x = lista.pop(4) #elimina el dato en el índice 4 y lo almacena en la variable x
n = lista.pop() #elimina el último dato y lo almacena en la variable n
```

[ ]:
```
print('lista antes de eliminar un dato:',lista3)
lista3.pop()
print('lista luego de eliminar un dato:',lista3)
```

```
lista antes de eliminar un dato: [5, 1, 2, 6, 4, 3, 9, 10, 56, 10, 9, 5, 1]
lista luego de eliminar un dato: [5, 1, 2, 6, 4, 3, 9, 10, 56, 10, 9, 5]
```

[ ]:
```
print('lista antes de eliminar un dato:',lista3)
lista3.pop(0) #eliminemos el primer dato
print('lista luego de eliminar un dato:',lista3)
```

```
lista antes de eliminar un dato: [5, 1, 2, 6, 4, 3, 9, 10, 56, 10, 9, 5]
lista luego de eliminar un dato: [1, 2, 6, 4, 3, 9, 10, 56, 10, 9, 5]
```

[ ]:
```
print('lista antes de eliminar un dato:',lista3)
x = lista3.pop()
print('lista luego de eliminar un dato:',lista3)
print('dato eliminado: ',x)
```

```
lista antes de eliminar un dato: [1, 2, 6, 4, 3, 9, 10, 56, 10, 9, 5]
lista luego de eliminar un dato: [1, 2, 6, 4, 3, 9, 10, 56, 10, 9]
dato eliminado:  5
```

1.4 Iteración en una lista

Como vemos, imprimir, añadir o eliminar datos de una lista, dependiendo de su tamaño, se puede convertir en una tarea compleja, pero **repetitiva**. Por este motivo Python nos permite recorrer listas usando el ciclo `for`.

Tenemos dos alternativas en este caso, la primera consiste en usar la función `len` para conocer el tamaño de la lista y crear el ciclo con un `range` que use este valor como límite.

```
for i in range(0,len(lista)):
    lista[i] #acceso a la i-ésima posición de la lista
```

De este modo, podemos acceder a las posiciones de la lista usando el contador del ciclo (en el caso del ejemplo, `i`) Dentro de este ciclo se puede usar cualquiera de las operaciones que hemos visto hasta ahora para manipular la lista.

[ ]:
```
%%debug_cell_with_pytutor
# Creemos una lista con los números del 1 al 15
l = [] #lista vacía
for i in range(1,16):
    l.append(i)
print(l)
```

[ ]:
```
%%debug_cell_with_pytutor
# Creemos una lista con los números del 1 al 15
l = [] #lista vacia
for i in range(1,16):
    l.append(i)
print(l)
# Eliminemos los valores entre 5 y 10
for i in range(5,11):
    l.remove(i)
print(l)
```

La segunda alternativa es reemplazar `range` directamente por la lista dentro del ciclo `for`

```
for i in lista:
    instrucciones
```

En este caso `i` deja de ser un contador y pasa a recibir uno a uno los valores que estén en la lista, esto es útil cuando no nos interesa conocer el índice de cada dato, por ejemplo, cuando queremos simplemente realizar una operación con los valores. Calculemos la media aritmética de los valores en `lista3`

[ ]:
```
%%debug_cell_with_pytutor
lista3 = [5,1,2,8,6,4,3,9,10,56,10,9,5,1]
media = 0
for i in lista3:
    print(i) #veamos qué hay internamente en i
    media+=i #sumamos el dato al acumulado
media/=len(lista3) #dividimos por el total de datos
print("la media de los datos es: ",media)
```

1.5 Indexado negativo

Si por algún motivo necesitamos recorrer las listas de forma inversa, es decir, del dato en el indice de mayor valor hacia el cero, Python nos permite hacerlo de forma simple usando el indexado negativo, del mismo modo que cada posición tiene un valor positivo asociado y partiendo del 0, las listas cuentan con un indice negativo que parte de `-1`, el cual siempre se le asigna al último dato de la lista.

Esto es algo muy útil ya que siempre podremos acceder a la última posición de la lista sin necesidad de conocer el tamaño de la misma.

[ ]:
```
print(lista3)
print('último dato: ',lista3[-1])
lista3.pop()
print(lista3)
print('último dato: ',lista3[-1])
lista3.pop()
print(lista3)
print('último dato: ',lista3[-1])
```

```
[5, 1, 2, 8, 6, 4, 3, 9, 10, 56, 10, 9, 5, 1]
ultimo dato:  1
[5, 1, 2, 8, 6, 4, 3, 9, 10, 56, 10, 9, 5]
ultimo dato:  5
[5, 1, 2, 8, 6, 4, 3, 9, 10, 56, 10, 9]
ultimo dato:  9
```

1.6 "Slicing"

En las listas, hay múltiples formas de imprimir una lista completa con todos los elementos, pero para imprimir un rango específico de elementos de la lista, utilizamos la operación **Slice**. La operación **Slice** se realiza en las Listas con el uso de dos puntos (`:`).

`lista[índice_inicial:índice_final:variación]`

La operación de slicing se comporta del mismo modo que la función `range` que estudiamos en la

sección anterior, podemos definir un valor inicial y final entre los cuales se extraerán las posiciones de la lista y además podemos definir un incremento en el parámetro `variación`, es decir, la cantidad que saltará entre cada índice (si usamos un incremento de 2, la sublista tomará los datos de 2 en 2). Para imprimir los elementos desde el principio hasta un rango se utiliza `[: índice]`, para imprimir los elementos desde el final se utiliza `[:-índice]`, para imprimir los elementos desde un Índice específico hasta el final se utiliza `[índice:]`, para imprimir los elementos dentro de un rango se utiliza `[índice_inicial:índice_final]` y para imprimir la lista en su totalidad podemos usar el operador `slice` por sí solo `[:]`. Además, para imprimir toda la lista en orden inverso, usamos `[::-1]` ya que un incremento de -1 le indica a Python que debe recorrer en reversa la lista.

Como vimos en la imagen anterior, la operación de slice se puede realizar también en sobre los índices negativos.

[ ]:
```
texto = ['M', 'O', 'N', 'T', 'Y', ' ', 'P', 'Y', 'T', 'H', 'O', 'N']
print(texto[:])
```

```
['M', 'O', 'N', 'T', 'Y', ' ', 'P', 'Y', 'T', 'H', 'O', 'N']
```

[ ]:
```
print(texto[6:10])
```

```
['P', 'Y', 'T', 'H']
```

[ ]:
```
print(texto[-12:-7])
```

```
['M', 'O', 'N', 'T', 'Y']
```

[ ]:
```
print(texto[::-1])
```

```
['N', 'O', 'H', 'T', 'Y', 'P', ' ', 'Y', 'T', 'N', 'O', 'M']
```

[ ]:
```
texto[2:8:2]
```

[ ]:
```
['N', 'Y', 'P']
```

1.7 Similitudes entre String y Listas

Como habrás notado, tanto las cadenas de caracteres como las listas tienen algunos comportamientos similares, podemos obtener el tamaño de ambos podemos usar la función `len`, ambas tienen una estructura ordenada y podemos acceder a sus datos con la notación de corchetes `[ ]` y de este mismo modo, podemos recorrer los strings con un ciclo for:

```
for s in string
    intrucción
```

En este caso, la variable s recibirá en cada iteración uno a uno los caracteres que conforman el string.

[ ]:
```
%%debug_cell_with_pytutor
cadena = 'parangaricutirimícuaro'
for s in cadena:
```

```
    print(s)
```

[ ]:
```
%%debug_cell_with_pytutor
cadena = 'parangaricutirimícuaro'
for i in range(len(cadena)):
    print(cadena[i])
```