Algoritmos con Listas

Iteración

Iteración - El estatuto while

Se utiliza para repetir la ejecución de un bloque de instrucciones, mientras la evaluación de una condición sea verdadera.

La sintaxis es:

```
while expresion: #Cuántas veces o hasta cuándo?
    bloque-1     #Qué se repite?
```

Iteración sobre Secuencias

Secuencias

Colecciones de elementos que pueden ser accesados por un índice

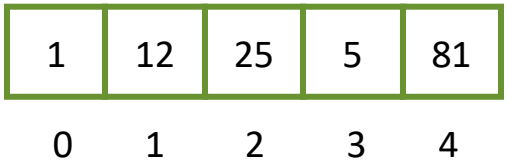
Tipos

- Mutables (listas)
- Inmutables (tuplas y strings)

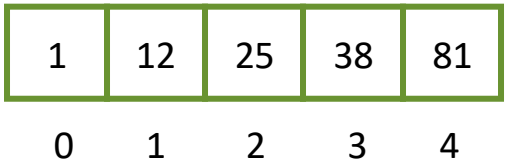
Listas

- Elementos entre [ ]
- Separados por coma
- Heterogéneas: pueden tener diferentes tipos de datos
- Lista nula: [ ]

```
lista = [1,12,25,5,81]
```




```
lista[3] = 38
```




```
>>>lista= [1,3,5,9]
>>>lista = [4, "hola", 5.0, [3]]
```

Strings

Colecciones de elementos entre comillas o apóstrofes.

```
>>>cadena = "Esta es una cadena de caracteres."
>>>cadena[5] = "E"
TypeError: str' object does not support item
  assignment
```

String nulo: "" o '' => ¡Sin espacios!

Funciones:

```
>>> variable = 'Hola'
>>>  variable.
```

Funciones de Python para el string

Upper, lower, Trim, EndsWith, Find, Index, Replace, Split

Tuplas

- Colecciones de elementos entre paréntesis redondos.

```
>>> tupla = (1,3,5,9)

>>> tupla[5] = "E"
TypeError: 'tuple' object does not
  support item assignment
```

Una tupla nula: `()`

Función len

- Retorna la cantidad de elementos de una secuencia.

```
>>> tira = "hola"
>>> lista = [1,4]
>>> tupla = (1,3,5,9,13)
>>> len(tira)
4
>>> len(lista)
2
>>> len(tupla)
5
```

Slicing

- Forma diferente de accesar a las posiciones de una secuencia.

`s[i:j]` -> retorna una secuencia del mismo tipo de la original con las posiciones contenidas entre `i` y `j-1`.

Los argumentos `i,j` pueden ser opcionales.

- Si no se indica i, se asume que la posición inicial es cero.
- Si no se indica j, se assume que la posición final es el final de la secuencia.

Slicing - Ejemplo

```
>>> lista =
 [1,5,7,9,11,13,14]

>>> sublista = lista[1:4]
[5,7,9]

>>> sublista1 = lista[3:]
[9,11,13,14]

>>> sublista2 = lista[:4]
[1,5,7,9]
```

`s[1:]` -> Quita el primer elemento de la lista. Similar a `x // 10` de números.

`s[0]` -> Obtiene la primera posición de la lista. Similar a `x % 10` de números.

Slicing - Importantes

L = [1, 2, 3, 4, 5]	Listas se crean con[ ]
T = (10, 20, 30, 40, 50)	Tuplas se crean con ( )
L[0]	Retorna 1er item de L (1)
T[0]	Retorna 1er item de T (10)
L[1:4]	**Slicing**: Retorna 2do al 4to item de L ([2, 3, 4]). Los slice son listas. El rango del Slice es abierto a la derecha, no incluye ese elemento
T[1:4]	**Slicing**: Retorna 2do al 4to item de T ((20, 30, 40))
L[-1]	Retorna el último elemento de L (5). Los índices negativos, iniciando en -1, recorren la lista de derecha a izquierda.
T[-1]	Retorna el último elemento de T (50).
L[-3:-1]	Retorna [3,4]. Slicing con negativos funciona de igual manera, abierto a la derecha.
T[-3:-1]	Retorna [30,40]
L[1] = 22	Asigna 22 al 2do elemento de L (L == [1, 22, 3, 4, 5])
T[1] = 22	ERROR: no puede modificar las tuplas
L[0:2] = [11, 22]	Asigna 11 y 22 al 1ro y 2do elementos de L (L == [11, 22, 3, 4, 5])

l1 = [10, 20, 30, 40, 50]

l1 [0 :]	Retorna toda la lista, desde 0 hasta el final
l1 [: 5]	Retorna toda la lista, desde el inicio hasta el índice 4, que para l1 es el final (50)
l1 [: len(l1)]	Retorna toda la lista, desde el inicio hasta len(l1) == 5 abierto
l1 [1 :]	Retorna la lista sin el primer elemento. **Quita el primer elemento de la lista**.
l1 [: -1]	Retorna la lista sin el último elemento. **Quita el último elemento**
l1 [: len(l1)-1]	Retorna la lista sin el último elemento. **Quita el último elemento**

Iteración con listas

Las listas se pueden recorrer, en forma completa o parcial, de dos formas:

- Utilizando **slicing**.
- Por medio de un **índice**: el cual es un entero que sirve para indicar la posición que se desea accesar.

```
# Haga un programa que retorne la cantidad de números pares de una lista de números enteros
# Solución con while descomponiendo la lista
```

```
def contar_pares(lista):          # versión con slicing
    cont = 0                      # contador de pares
    while lista != []:
        if lista[0] % 2 == 0:
            cont += 1
        lista = lista[1:]
    return cont
```

Iteración con listas

```
# Haga un programa que retorne la cantidad de números pares de una
lista de números enteros
#Utilizando while con índices
```

```
def contar_pares(lista):        # versión con índices
    i = 0
    n = len(lista)
    cont = 0                    # contador de pares
    while i < n:
        if lista[i] % 2 == 0:
            cont += 1
        i += 1
    return cont
```

Estatuto for

Sintaxis:

```
for iterador in iterable:
        bloque-1
```

- Un iterable es un objeto que puede retornar sus elementos uno a la vez como por ejemplo:
  - Las secuencias (strings, listas, tuplas)
  - Funciones especiales como range
  - Archivos
  - clases definidas por el usuario que tengan la opción de obtener un ítem del objeto.
- El bloque-1 es ejecutado una vez para cada ítem provisto por el iterador. Cuando el iterable se recorre en su totalidad el ciclo termina.
- El cambio de cada elemento del iterable es **automático**.

For each

```
# Haga un programa que retorne la cantidad de números pares de una
lista de números enteros
#Utilizando for each
```

```
def contar_pares(lista):
    cont = 0                        # contador de pares
    for item in lista:
        if item % 2 == 0:
            cont += 1
    return cont
```

Estatuto range

Función range: retorna un iterable numérico desde inicio hasta fin – 1.

Sintaxis: `range(inicio, fin[, incremento])`

Ejemplos:

```
>>> range(1, 10)      # iterable de 1 a 9

>>> range(10)         # iterable de 0 a 9

>>> range(1, 10, 2)   # iterable de 1 a 9 de 2 en 2

>>> range(10, 1, -1)  # iterable de 10 a 0
```

Estatuto for i (index in range)

```
# Haga un programa que retorne la cantidad de números pares de una lista de números enteros
#Utilizando for i (index in range)
```

```
def contar_pares(lista):
    cont = 0
    n = len(lista)
    for i in range(n):
        if lista[i] % 2 == 0:
            cont += 1
    return cont
```

Concatenación de Secuencias

S1+S2

retorna una secuencia formado por S1, seguida de S2.

Tanto S1 como S2 deben ser del mismo tipo.

```
>>>[1,2,3]+[4,5,6]
[1,2,3,4,5,6]
>>> [1,2]+(3,4)
TypeError: can only concatenate list (not
 "tuple") to list
>>>[1,2,3]+[]
[1,2,3]
```

Ejercicios: 2 versiones con FOR each y FOR i

```
# recorrido de lista contando
```

1- Hacer una función que cuente los números negativos de una lista

```
# recorrido de lista buscando un caso
```

2- Hacer una función que determine si una lista tiene al menos un número negativo.

```
# recorrido de lista y composición de nueva lista
```

3- Hacer una función que obtenga los números negativos de una lista.

El método append

Método **append**: sirve para agregar elementos al final de una lista. Más eficiente que la concatenación.

Sintaxis: `secuencia.append (elemento)`

```
def pares(lista): # version con append
    result = []
    for ele in lista:
        if ele % 2 == 0:
            result.append(ele)
    return result
```

Funciones iterativas con listas

```
def pares(lista): # version con append
    result = []
    for ele in lista:   # con for each
        if ele % 2 == 0:
            result.append(ele)
    return result
```

```
def pares(lista): # version con append
    result = []
    for i in range(len(lista)):   # con for each
        if ele % 2 == 0:
            result.append(lista[i])
    return result
```

Salidas abruptas en ciclos

Función que retorna si una lista tiene al menos un número par.

```
def hay_par(lista):    # solución 1
    for ele in lista:
        if ele % 2 == 0:
            return True
    return False
```

Esta solución tiene dos salidas (dos `return`) en el ciclo, lo cual rompe un principio de programación estructurada y **no** se va a considerar una buena práctica.

Salidas abruptas en ciclos

Retorna si una lista tiene al menos un número par.

```
def hay_par(lista):    # solución 2
    result = False
    while lista != []:
        if lista[0] % 2 == 0:
            result = True
            lista = []
        else:
            lista = lista[1:]
    return result
```

Esta solución al cumplirse la condición de encontrar un número par, realiza la salida del ciclo asignando a la lista una lista nula.

Salidas abruptas en ciclos

Retorna si una lista tiene al menos un número par.

```
def hay_par(lista):    # solución 3
    result = False
    while lista != [] and not result:
        if lista[0] % 2 == 0:
            result = True
        else:
            lista = lista[1:]
    return result
```

Esta solución tiene se conoce como ciclo controlado por una bandera o centinela, que en este caso es la variable result que también se usa para controlar el ciclo. Cuando se encuentra un par, la variable result se convierte en True y el ciclo no se va a ejecutar más.

Salidas abruptas en ciclos

Retorna si una lista tiene al menos un número par.

```
def hay_par(lista):    # solución 4
    result = False
    for ele in lista:
        if ele % 2 == 0:
            result = True
            break
    return result
```

Esta solución utiliza el estatuto break provisto por el lenguaje y que realiza la salida abrupta del ciclo y se va a considerar una buena práctica.

Ejercicios nivel medio

Desarrolle un programa que elimine las apariciones de un elemento en una lista.

> eliminar('a',[5,'a',10.2,True,'a','c'])
[5,10.2,True,'c']

Desarrolle un programa que retorne el número menor de la lista sin usar min

> menor ([76,8,2,19,18])
2

Desarrolle un programa que una una lista de números en un solo número, los elementos deben ser números enteros positivos o negativos o cero.

> to_number([-10,-256,32,10,0,1])
10256321001

Desarrolle un programa que descomponga un lista en una lista de positivos, una de negativos y una de ceros, y las retorne en una sola lista.

> descomponer([0,-9,0,11,0,4,98,-25])
[[-9, -25], [0, 0, 0], [11, 4, 98]

Desarrolle un programa que sustituya un elemento por otro, en una lista. No use replace

> substitute('a', 65, [1, 'a', 'a', True, 2, 56])
[1, 65, 65, True, 2, 56]