ENGIN 604 Introducción a Python para las Finanzas
Secuencias & Estructuras de Datos - Pauta

Profesor: *Gabriel E. Cabrera*
Ayudante: *Alex Den Braber*

Open in Colab

1. Estructuras de Datos Básicas

Como regla general, las estructuras de datos son objetos que contiene otros objetos (e.g escalares) como secuencias. En Python se tiene las siguientes estructuras de datos *built-in*:

Cuadro 1: Tipos de Estructura de Datos

Estructura	Ejemplo	Brackets	índice	Elementos Repetidos	Mutable	Operaciones de Conjuntos
Lista	A = [1,2,'hola']	[ ]	Sí	Sí	Sí	No
Tupla	B = (3,3,5)	()	Sí	Sí	No	No
Diccionario	D = {'Día':01,'Mes':3}	{}	Sí	Sí	Sí	No
Conjunto	C = {6,7,8}	{}	No	No	No	Sí

1.1. Lista (`list`)

1. Construya una lista que contenga el nombre de las siguientes empresas tecnológicas: 'Facebook', 'Apple', 'Amazon' y 'Netflix'.

   a. Muestre el primer y último elemento de la lista creada.

```
faan = ['Facebook', 'Apple', 'Amazon', 'Netflix']

# n° de elementos en la lista
len(faan)

# solo el primer elemento
faan[0]

# solo el segundo elemento
faan[1]

# solo el tercer elemento
faan[2]

# solo el último elemento
faan[3]
```

```
# todos los elementos
faan[:]

# primer elemento más todos los demás
faan[0:]

# segundo elemento más todos los demás
faan[1:]

# primer elemento hasta el tercer elemento (inclusive), el intervalo será [,)
faan[0:3]

# seleccionar todo usando un índice inexistente en la lista pero mayor al existente
faan[0:1000]
```

b. Muestre el primer y último elemento de la lista creada usando índices negativos.

```
# primer elemento
faan[-4]

# último elemento
faan[-1]
```

c. Agregue "Nvidia" al ínicio de la lista. ¿Como lo agrego en la cuarta posición?

```
faan.insert(0, 'Nvidia') # in-place
faan.insert(3, 'Nvidia') # cuarta posición (in-place)
```

d. Agregue "Google" al final de la lista.

```
# importante (in-place) por default es la última posición
faan.append('Google')
```

e. ¿Cuantos elementos tiene la lista hasta ahora?

```
len(faan)
```

f. En que posición se ubica la empresa 'Apple'.

```
# mostrará la primera posición que contenga el elemento,
# pero no todos los que existan en la lista
faan.index('Apple')
```

g. Elimine "Nvidia" y "Netflix" de la lista.

```
# forma 1
del faan[faan.index('Nvidia')] # mostrará el primero
del faan[faan.index('Netflix')]

# forma 2
faan.pop(faan.index('Nvidia')) # .pop() por default remueve el último
faan.pop(faan.index('Netflix'))

# forma 3
faan.remove('Nvidia')
faan.remove('Netflix')
```

h. Ordene de mayor a menor la lista según el número de caracteres de cada elemento en la lista.

```
faan.sort(key=len)
```

2. Construya las siguientes listas:
   - parte_a = `[[0, 'a'], [1, 'b']]`
   - parte_b = `[[2, 'c'], [3, 'd']]`

   a. Concatene o combine ambas listas. ¿En que se diferencia concatenar con `+` y `extend`?

```
parte_a = [[0, 'a'], [1, 'b']]
parte_b = [[2, 'c'], [3, 'd']]

lista_combinada1 = parte_a + parte_b

# usando extend
parte_a.extend(parte_b) # in-place
```

   b. Muestre el segundo elemento de la lista generada en (a).

```
# elementos en la lista anidada
lista_combinada1[0]
lista_combinada1[0][0]
lista_combinada1[0][1]
```

3. Genere lista que contenga desde el 1 hasta el 10.

   a. Ordene la lista de mayor a menor. ¿En que se diferencia ordenar con `sort` y `sorted`?

```
num_list = list(range(1, 11))

# sort
num_list.sort(reverse=True) # in-place

# sorted (bult-in)
sorted(num_list)
```

   b. Seleccione los valores cuyo índice sea par.

```
num_list[::2]
```

1.2. Tupla (`tuple`)

Se creará una tupla (`tuple`) si se utiliza coma (,) para separar valores cuando se asigna a una única variable:

```
tupla = 1, 2, 'Facebook', 'Amazon'
tupla
```

Que incluso pueden almacenar listas (`list`):

```
tupla_anidada2 = (1,2), ('Facebook', 'Amazon'), ['Apple', 'Netflix']
tupla_anidada2
```

Solo se puede modificar un elemento de la tupla (`tuple`) que permita modificación, por ejemplo la lista (`list`):

```
tupla_anidada2[2].append('Google')
tupla_anidada2
```

Permite concatenación mediante el signo `+`:

```
('Facebook', 'Amazon') + ('Apple', 'Netflix')
```

E incluso multiplicación de elementos usando `*`:

```
('Facebook', 'Amazon') * 3
```

Para forzar una lista (`list`) a que sea tupla (`tuple`) basta con:

```
tuple([1, 2, 3, 4])
```

1.3. Diccionario (`dict`)

Los diccionarios (`dict`) son estructuras de datos mutables, se caracterizan con el uso de llave-valor (*key-value*).

```
dict1 = {'Name' : 'Janet Yellen',
         'Country' : 'United States',
         'Profession' : ' United States secretary of the treasury',
         'Age' : 74}

type(dict1)
```

Para acceder a los valores asociado a la llave, basta con usar `[ ]` y escribir la el nombre de la llave:

```
print(dict1['Name'], dict1['Age'])
```

Hereda los métodos `.keys()` para accader al nombre de las llaves:

```
dict1.keys() # keys (llaves)
```

Hereda los métodos `.values()` para accader a los valores asociados a las llaves:

```
dict1.values() # valores
```

Mediante `.items()` se accede a la llave (*key*) y los valores (*values*):

```
dict1.items()  # items = keys + values
```

1.4. Conjunto (`set`)

La estructura de datos de conjunto (`set`) permite solo valores únicos. Para crear un conjunto (`set`), se crea una lista y luego `set()`:

```
set1 = set([1, 2, 3, 4, 5, 5])
set1
```

```
set2 = set([3, 4, 5, 6, 9, 9, 7])
set2
```

Podemos aplicar operaciones de conjuntos:

- Todo los items que están en `set1` y `set2`:

  ```
  # union
  set1.union(set2)
  ```

- Items que estén en `set1` y `set2`

  ```
  # intersección
  set1.intersection(set2)
  ```

- Items que estén en `set1` pero no en `set2`:

  ```
  set1.difference(set2)
  ```

- Items que estén en `set2` pero no en `set1`:

  ```
  set2.difference(set1)
  ```

- Items que estén en `set1` o `set2` pero no en ambos:

  ```
  set1.symmetric_difference(set2)
  ```