ENGIN 604 Introducción a Python para las Finanzas
Funciones - Pauta

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

1. def(x)

Una función se declara usando la keyword def y solo funciona cuando es llamada. Su objetivo principal es evadir la redundancia de código por parte del programador. Por ejemplo, para crear una función que multiplique por dos cualquier número (int o float) y luego calcule la raiz:

```
def mi_funcion(x): # la creación de una función debe partir con def,
                   # luego el nombre de la función y entre parentesis
                   # los argumentos que tendrá
  '''
  Documentación (docstring):
  Multiplica por dos el argumento númerico y luego calcula la raiz.
  '''
  a = x * 2 # multiplica por dos x (body)
  b = a ** 0.5 # calcula la raiz de x multiplicado por dos (body)

  return(b) # la función debe terminar con return para que "retorne"
            # el resultado final de la función

# se verifica la función
mi_funcion(4) # (x * 2) ** 0.5 = (4 * 2) ** 0.5
```

```
## 2.8284271247461903
```

Las funciones también pueden tener más de un parámetro (argumento):

```
def mi_funcion2(x, y): # def, nombre función, parámetros (argumentos)
  '''
  Documentación (docstring):
  Suma los dos argumentos numéricos (x + y).
  '''
  a = x + y # suma x e y (body)

  return(a) # return

# se verifica la función
mi_funcion2(4, 5) # x + y = 4 + 5
```

```
## 9
```

Las funciones también pueden retornar más de un output:

```
def mi_funcion3(x, y): # def, nombre función, parámetros (argumentos)
  '''
  Documentación (docstring):
  Multiplica y suma ambos argumentos (x e y). Retorna por separado la multiplicación
  y la suma.
  '''
  a = x * y # multiplicación entre x e y (body)
  c = x + y # suma x e y (body)

  return(a, c) # return

# se verifica la función
mult_x_y, sum_x_y = mi_funcion3(4, 5) # x * y = 4 * 5 ^ x + y = 4 + 5

mult_x_y
```

```
## 20
```

```
sum_x_y
```

```
## 9
```

1.1. Aplicaciones

1. Construya una función que para un número n entregue como *output* la suma de los cuadrados de $1^2 + 2^2 + 3^2 + 4^3 + ... + n^2$.

```
def elevado(n):
  total = 0
  for i in range(1,n+1):
    total += i ** 2
  return(total)

elevado(5)
```

```
## 55
```

2. Construya una función que entregue como output si un número es o no divisible por 4.

```
def divisible(n):
  if (n % 4) == 0:
    return(print('El número '+str(n)+' es divisible por 4.'))
  else:
    return(print('El número '+str(n)+' no es divisible por 4.'))

divisible(16)
```

```
## El número 16 es divisible por 4.
```

3. Construya una función que entregue la media aritmética de un conjunto de datos.

```
def media_aritmetica(n):
  output = sum(n) / len(n)
  return(output)

media_aritmetica([1,2,3,4,5])
```

```
## 3.0
```

4. Construya una función que permita calcular el valor presente neto (VPN) de un flujo efectivo.

$$VPN = \sum_{t=0}^{T} \frac{F_t}{(1+r)^t}$$

Donde F_t es el flujo en el periodo t, r es la tasa de descuento y T es el número total de periodos. Considere los siguientes flujos:

Cuadro 1: Peridodos & Flujos

Periodo	Flujo ($)
0	-500000
1	100000
2	150000
3	180000
4	200000
5	300000

Si la tasa de descuento es 12 % el valor presente neto es:

$$VPN = -500000 + \frac{100000}{(1+0,12)^1} + \frac{150000}{(1+0,12)^2} + \frac{180000}{(1+0,12)^3} + \frac{200000}{(1+0,12)^4} + \frac{300000}{(1+0,12)^5} = 134316{,}91$$

```
def valor_neto_presente(flujo, tasa):
   total = 0
   for i, num in enumerate(flujo):
     if i == 0:
       total += num
     else:
       total += num / (1 + tasa) ** (i)
   return(total)

flujos = [-500000,100000,150000,180000,200000,300000]

valor_neto_presente(flujos, 0.12)
```

```
## 134316.91292132728
```

2. Funciones Anónimas (`Lambda`)

Las funciones anónimas o lambda consisten en escribir una función en una sola sentencia. Se utilizan mediante la *keyword* `lambda`, que le "dice" a Python "se está declarando una función anónima". Son útiles para utilizar dentro de una función definida (`def`), por ejemplo:

```
seq = [1, 2, 4]

def apply_a_una_lista(lista, f):
    return [f(x) for x in lista]

apply_a_una_lista(seq, lambda x: x ** 2)
```

```
## [1, 4, 16]
```

La función `apply_a_una_lista` tiene como argumento una lista de números (`seq`) y una función (`lambda`) que se aplica a una *list comprehension* retornando cada número de esa lista al cuadrado. Se denominan anónimas debido que no se definen utilizando `def`.

3. Importar Funciones

1. Guarde la funciones creadas en 1.3 y 1.4 en un script con el nombre `funciones_auxiliares.py`. Luego importelas a su espacio de trabajo y compruebe que pueden ser utilizadas de igual forma, es decir, sin tener que programarlas desde cero cada vez que se quieran utilizar.

   Para cargar las funciones desde el archivo `funciones_auxiliares.py`:

```
# tiene que estár en el mismo directorio, si se usa * y no el nombre de cada función
# se cargan todas las funciones disponibles
from funciones_auxiliares import media_aritmetica, valor_neto_presente
```

```
# función de 1.3
media_aritmetica([1,2,3,4,5])
```

```
## 3.0
```

```
# función de 1.4
flujos = [-500000,100000,150000,180000,200000,300000]

valor_neto_presente(flujos, 0.12)
```

```
## 134316.91292132728
```