INSTITUTO SUPERIOR TECNOLÓGICO QUITO

Excelencia en Educación Superior

0.1 Tercer Modulo- Python , Funciones

python™

Erick Mejia

1. Funciones

1.1 ¿Qué son?

- Son un mecanismo que te ofrece el lenguaje para agrupar conjuntos de instrucciones de manera que se puedan ejecutar cuando el programador considere oportuno.
- Las funciones van identificadas por un nombre.
- Se debe usar este nombre para invocar (ejecutar) el código de la función.
- Las funciones calculan un resultado (output) en base a unos parámetros de entrada (input).
- En cada invocación de la función, los parámetros de entrada pueden variar.

1.2 Ejemplo

In [1]:
```
def sumar(x, y):
    suma=x+y
    return suma

print(sumar(4,5))
print(sumar(-2,0))
```

```
9
-2
```

In [2]:
```python
#f = c * 1.793 + 32

c = 30
print(c * 1.793 + 32) # celsius to fahrenheit

c = 26
print(c * 1.793 + 32)
```

```
85.78999999999999
78.618
```

In [3]:
```python
def convert_celsius_to_fahrenheit(c):
    return c * 1.791117 + 32
```

In [4]:
```python
degrees = 15
print(convert_celsius_to_fahrenheit(degrees))
```

```
58.866755
```

1.3 ¿Qué ventajas nos proporcionan?

- Eliminación de duplicación de código.
- Incremento de la reutilización de código.
- Manejo de complejidad: permiten descomponer grandes sistemas en piezas más pequeñas y, por tanto, más manejables y comprensibles.

1.4 Programación de funciones en Python

Las funciones se definen por medio de la palabra clave def.

Formato general:

def name(arg1, arg2, ..., argN): statements

Muy comúnmente:

def name(arg1, arg2, ..., argN): ... return value

- Las funciones son instrucciones convencionales, que se ejecutan cuando el flujo de control llega a ellas.
- La definición de la función debe ejecutarse antes de poder invocarse la función.

In [5]:
```python
# Primero, definición.
def crear_diccionario(keys, values):
    return dict(zip(keys, values))
```

In [6]:
```python
# # Después, invocación.
nombres = ['Alfred', 'James', 'Peter', 'Harvey']
edades = [87, 43, 19, 16]
```

```
usuarios = crear_diccionario(nombres, edades)
print(usuarios)
```

```
{'Alfred': 87, 'James': 43, 'Peter': 19, 'Harvey': 16}
```

- Las funciones pueden estar anidadas en sentencias condicionales o bucles.

In [7]:
```
a = -2

if a > 0:
    def operacion(x, y):
        return x + y
else:
    def operacion(x, y):
        return x * y

print(operacion(3, 4))
```

```
12
```

- Argumentos y return son opcionales.

In [8]:
```
def print_hola_mundo():
    print('Hola Mundo')
    return # es opcional pero mejora la legibilidad

print_hola_mundo()
```

```
Hola Mundo
```

- Cuando una función no tiene sentencia return, la función devuelve el objeto None.

In [9]:
```
def print_hola_mundo():
    print('Hola Mundo')
    # return None

valor = print_hola_mundo()
print(valor)
```

```
Hola Mundo
None
```

- La sentencia return puede aparecer en cualquier parte de la función.

In [10]:
```
def div_safe(x,y):
    if y == 0:
        return "hola"
    return x / y

print(div_safe(6,2))
print(div_safe(6,0))
```

```
a = div_safe(6,0)
print(type(a))
```

```
3.0
hola
<class 'str'>
```

In [11]:
```
def div_safe(x,y):
    if y != 0:
        return x / y
    else:
        return 0

print(div_safe(6,0))
```

```
0
```

- Puede haber más de un valor de retorno. Se devuelve una tupla.

In [12]:
```
def devolver_primero_segundo(coleccion):
    return coleccion[0], coleccion[1], coleccion[2]

t = devolver_primero_segundo(['Alfred', 'James', 'Peter', 'Harvey'])
print(t)
print(t[1])
print(type(t))
```

```
('Alfred', 'James', 'Peter')
James
<class 'tuple'>
```

In [13]:
```
primero, segundo,tercero = devolver_primero_segundo(['Alfred', 'James','Peter', 'Ha
print(primero)
print(segundo)
print(tercero)
```

```
Alfred
James
Peter
```

- El tipo de los parámetros y el valor de retorno se pueden especificar, pero son simples hints. Esto se llama annotations.

In [14]:
```
def add2(x:int, y:int) -> int:
    return x + y

print(add2(1,2))
print(add2("aa","ee"))
```

```
3
aaee
```

- Los argumentos pueden tener valores por defecto, siempre al final.

In [15]:
```
def f(a, b, c = None):
    if a is not None:
        print('Se ha proporcionado a')
    if b is not None:
        print('Se ha proporcionado b')
    if c is not None:
        print('Se ha proporcionado c')

f(1, 5, 2)
```

```
Se ha proporcionado a
Se ha proporcionado b
Se ha proporcionado c
```

- Los parámetros se pasan por posición, pero el orden se puede alterar si se especifica el nombre del parámetro en la llamada, named values.

In [16]:
```
def power(base, exponente):
    return pow(base, exponente)

print(power(2,3))
print(power(exponente = 3, base = 2))
```

```
8
8
```

- Al ejecutarse una sentencia def, se crea un objeto de tipo función y éste se asocia con el nombre especificado para la función.
- Un objeto de tipo función es como cualquier otro tipo de objeto.
- Variables pueden referenciar objetos de tipo función.

In [17]:
```
def sumar(x, y):
    return x + y

f = sumar

print(id(f))
print(id(sumar))

print(type(f))

print(sumar(5,6))
print(f(5,6))
```

```
2504980379776
2504980379776
<class 'function'>
11
11
```

In [18]:
```
def clean_strings(strings, ops):
    result = []
    for value in strings:
```

```
        for function in ops:
            value = function(value)
        result.append(value)
    return result

def concat5(x):
    return x + '_5'

strings = [' valencia ', ' barcelona', ' bilbao ']
print(strings)
operations = [str.strip, str.title, concat5]
print(clean_strings(strings, operations))
```

```
[' valencia ', ' barcelona', ' bilbao ']
['Valencia_5', 'Barcelona_5', 'Bilbao_5']
```

- Se pueden crear placeholders con la palabra clave pass.
- pass representa una sentencia que no hace nada.
- Útil cuando la sintaxis del lenguaje requiere alguna sentencia, pero no tienes nada que escribir.

In [19]:
```
def mi_funcion():
    pass #TODO implement this

mi_funcion()
```

1.5 Paso de parámetros

- Tipos simples (inmutables) por valor: int, float, string.
  - El efecto es como si se creara una copia dentro de la función, aunque realmente no es esto lo que ocurre.
  - Los cambios dentro de la función no afectan fuera.

In [20]:
```
def cambiar_valor(x):
    x += 3

    print('Dentro de la función:', x) # x ahora referencia a un nuevo objeto(el 3),
    return

a = 2
cambiar_valor(a)
print('Fuera de la función:', a)
```

```
Dentro de la función: 5
Fuera de la función: 2
```

- Tipos complejos (mutables) por referencia: list, set, dictionary.
  - Dentro de la función se maneja el mismo objeto que se ha pasado desde fuera.
  - Los cambios dentro de la función sí afectan fuera.

In [21]:
```
def anyadir_2_a(x):
    x.append(4)

lista = [0, 1,3]
anyadir_2_a(lista)
print(lista)
```

```
[0, 1, 3, 4]
```

In [22]:
```
# with tuples
def anyadir_2_a(x):
    print(id(x))
    return

lista = (0, 1)
print(id(lista))
anyadir_2_a(lista)
print(lista)
```

```
2504980721856
2504980721856
(0, 1)
```

- Si quiero modificar un objeto que es de un tipo básico, los cuales se pasan por valor, puedo simplemente devolver el resultado de la función y hacer una asignación:

In [23]:
```
def multiplicar_por_2(x):
    x = x * 2
    return x

a = 3
a = multiplicar_por_2(a)
print(a)
```

```
6
```

- Si quiero que no se modifique un objeto de un tipo complejo, los cuales se pasan por referencia, puedo pasar una copia a la función:

In [24]:
```
def anyadir_2_a(x):
    x.append(2)
    print('Dentro de la función: ', x)

lista = [0, 1]
anyadir_2_a(lista.copy())
print('Fuera de la función: ', lista)
```

```
Dentro de la función:  [0, 1, 2]
Fuera de la función:  [0, 1]
```

In [25]:
```
# Alternativa para obtener una copia: slicing.
# lista = [0, 1]
# anyadir_2_a(lista[:])
# print('Fuera de la función: ', lista)
```

- Si la lista tiene sublistas anidadas hay que hacer un deep copy

In [26]:
```
import copy
lista = [2, 4, 16, 32, [34, 10, [5,5]]]
# copia = lista.copy()
copia = copy.deepcopy(lista)
copia[0] = 454
copia[4][2][0] = 64
print(lista)
print(copia)

print(f"{id(lista)} - {id(copia)}")
print(f"{id(lista[4])} - {id(copia[4])}")
```

```
[2, 4, 16, 32, [34, 10, [5, 5]]]
[454, 4, 16, 32, [34, 10, [64, 5]]]
2504980708672 - 2504980685760
2504980819648 - 2504980698176
```

- Reasignar un parámetro nunca afecta al objeto de fuera:

In [27]:
```
# def funcion_poco_util(x):
#       print(id(x))
#       x = [2, 3]
#       x.append(4)
#       print(id(x))

# lista = [0, 1]
# print(id(lista))
# funcion_poco_util(lista)
# print(lista)
```

In [28]:
```
# def modif_tupla(a):
#       print(id(a))

# t = (1,2)
# print(id(t))
# modif_tupla(t)
# print(t)
```

1.6 Argumentos arbitrarios

- No se sabe a priori cuantos elementos se reciben.
- Se reciben como una tupla.
- Los parámetros se especifican con el símbolo '*'.

In [29]:
```
# def saludar(*names):
#     print(type(names))
#     for name in names:
#         print("Hola " + name)

# saludar()
# saludar("Manolo", "Pepe", "Luis", "Alex", "Juan")
```

In [30]:
```
# def consulta_sql(**kwargs):
#     print(kwargs)
#     sql = "SELECT * FROM bicis WHERE "
#     args = [f"{key}={value}" for key, value in kwargs.items()]
#     sql += " AND ".join((args))
#     return sql

# print(consulta_sql(station='alameda', free=10))
```

1.6.1 Desempaquetado en funciones

In [31]:
```
# def saludar(quien, mensaje, gesto):
#     print(f"Hola {quien}, {mensaje}, mira esto: {gesto}")

# ls_saludo = ['Manolo', 'te quiere saludar', 'args']

# saludar(ls_saludo[0], ls_saludo[1], ls_saludo[2])
# saludar(*ls_saludo)
```

In [32]:
```
# def saludar(quien, mensaje, gesto):
#     print(f"Hola {quien}, {mensaje}, mira esto: {gesto}")

# dc_saludo = {
#     'quien': [12, 12, 3,4 , 5,3],
#     'gesto': 'guiño',
#     'mensaje': 'te quiere enseñar esto'
# }

# saludar(**dc_saludo)
```

1.7 Recursividad

- Funciones que se llaman a sí mismas

In [33]:
```
# def factorial(n):
#     if n == 0:
#         return 1
#     else:
#         return n * factorial(n-1)

# print(factorial(10))

# def fact(n):
#     a = 1
```

```
#     for i in range(1, n+1):
#         a *= i
#     return a

# print(fact(10))
```

In [34]:
```
# fact = lambda x: 1 if x == 0 else x * fact(x-1)

# fact(10)
```

1.8 Funciones Lambda

- Es posible definir funciones anónimas (sin nombre).
- Lambdas son expresiones; por lo tanto, puede aparecer en lugares donde una sentencia def no puede (por ejemplo, dentro de una lista o como parámetro de una función).
- Útiles cuando se quiere pasar una función como argumento a otra.

Formato general:

`lambda <lista de argumentos> : <valor a retornar>`

- Importante: `<valor a retornar>` no es un conjunto de instrucciones. Es simplemente una expresión `return` que omite esta palabra.
- Las funciones definidas con def son más generales:
  - Cualquier cosa que implementes en un lambda, lo puedes implementar como una función convencional con def, pero no viceversa.

In [35]:
```
# Ejemplo: función suma.

def suma(x, y, z):
    return x + y + z
print(suma(1, 2, 3))

# Ejemplo: función suma como expresión lambda.

suma2 = lambda x, y, z : x + y + z

print(suma2(1, 2, 3))
```

```
6
6
```

In [36]:
```
# La funcion 'sort' puede recibir una función optional como parámetro.
# Esta función 'key' se invoca para cada elemento de la lista antes de realizar␣↪la
# Ejemplo: ordenar strings por número de carácteres.

cities = ['Valencia', 'Lugo', 'Barcelona', 'Madrid']

cities.sort()
print(cities)

cities.sort(key = lambda x : len(x))
```

```
print(cities)

cities.sort(key = len)
print(cities)

def longitud(x):
    return len(x)

cities.sort(key = longitud)
print(cities)
```

```
['Barcelona', 'Lugo', 'Madrid', 'Valencia']
['Lugo', 'Madrid', 'Valencia', 'Barcelona']
['Lugo', 'Madrid', 'Valencia', 'Barcelona']
['Lugo', 'Madrid', 'Valencia', 'Barcelona']
```

In [37]:
```
# Mismo ejemplo sin lambda

cities = ['Valencia', 'Lugo', 'Barcelona', 'Madrid']

def num_caracteres(x):
    return len(x)

cities.sort(key = num_caracteres)
print(cities)
```

```
['Lugo', 'Madrid', 'Valencia', 'Barcelona']
```

- Nos podemos guardar una referencia para utilización repetida.

In [38]:
```
# cities = ['Valencia', 'Lugo', 'Barcelona', 'Madrid']

# f = lambda x : len(x)

# for s in cities:
#     print(f(s))
```

- Las expresiones lambda pueden formar parte de una lista.

In [39]:
```
# Lista de lambdas para mostrar las potencias de 2.
# pows = [lambda x: x ** 0,
#         lambda x: x ** 1,
#         lambda x: x ** 2,
#         lambda x: x ** 3,
#         lambda x: x ** 4]

# for f in pows:
#     print(f(2))
```

- Diccionarios se pueden utilizar como ‘switch’ en otros lenguajes
- Ideal para escoger entre varias opciones

In [40]:
```
# def switch_dict(operator, x, y):
#     # crea un diccionario de opciones y selecciona 'operator'
#     return {
#         'add': lambda: x + y,
#         'sub': lambda: x - y,
#         'mul': lambda: x * y,
#         'div': lambda: x / y,
# }.get(operator, lambda: None)() # retorna None si no encuentra 'operator'

# print(switch_dict('add',2,2))
# print(switch_dict('div',10,2))
# print(switch_dict('mod',17,3))
```

1.9 Scopes

- La localización de una asignación a una variable en el código determina desde donde puedes acceder a esa variable
- Por ejemplo, una variable asignada dentro de una función sólo es visible dentro de esa función.
- El alcance de la visibilidad de una variable determina su scope.
- El término scope hace referencia a un espacio de nombres (namespace). Por ejemplo, una función establece su propio namespace.

In [41]:
```
# X = 10

# def func():
#     X = 20 # Local a la función. Es una variable diferente, no visible␣↪fuera de

#     def func_int():
#         X = 30
#         print(X)

# func_int()

# func()
```

Regla LEGB Dada una función F, tenemos los siguientes scopes:

- **L**ocal: variables locales a F.
- **E**nclosing: variables localizadas en funciones que contienen a (o por encima de) F.
- **G**lobal: variables definidas en el módulo (fichero) que no están contenidas en ninguna función. Este scope no abarca más de un fichero.
- **B**uilt-ins: proporcionadas por el lenguaje.

La resolución de nombres ocurre de abajo a arriba.

In [42]:
```
# Global (X y func)
# X = 99

# def func(Y): # Local (Y y Z)
```

```
#     Z = X + Y
#     return Z

# print(func(1))
```

In [43]:
```
# X = 1

# def func():
#     X = 2 # X es una variable diferente

# func()
# print(X)
```

- La sentencia global nos permite modificar una variable global desde dentro de una función.

In [44]:
```
# X = 1

# def func():
#     global X
#     X = 2

# func()
# print(X)
```

- No hay necesidad de usar global para referenciar variables. Sólo es necesario para modificarlas.

In [45]:
```
# y, z = 1, 2

# def todas_globales():
#     global x
#     x = y + z # Las tres variables son globales.

# todas_globales()
# print(x)
# print(y)
# print(z)
```

In [46]:
```
# X = 77

# def f1():
#     X = 88     # Enclosing scope (para f2)
#     def f2():
#         print(X)
#     f2()

# f1()
```

Closures

- Las funciones pueden recordar su enclosing scope, independientemente de si éstos continuan existiendo o no.

In [47]:
```
# def f1():
#     X = 88 # Enclosing scope (para f2)
#     def f2():
#         print(X)
#     return f2 # Devuelve f2, sin invocarla

# accion = f1()
# accion()
```

- Sentencia nonlocal para modificar variables que no son locales, pero tampoco globales.

In [48]:
```
# def f1():
#     contador = 10
#     def f2():
#         nonlocal contador
#         contador += 1
#         print(contador)
#     return f2

# accion = f1()
# accion()
# accion()
# accion()
```

1.10 Ejercicios

1. Escribe una función que reciba como entrada una lista con números y devuelva como resultado una lista con los cuadrados de los números contenidos en la lista de entrada.
2. Escribe una función que reciba números como entrada y devuelva la suma de los mismos. La función debe ser capaz de recibir una cantidad indeterminada de números. La función no debe recibir directamente ningún objeto complejo (lista, conjunto, etc.).
3. Escribe una función que reciba un string como entrada y devuelva el string al revés. Ejemplo: si el string de entrada es 'hola', el resultado será 'aloh'.
4. Escribe una función lambda que, al igual que la función desarrollada en el ejercicio anterior, invierta el string recibido como parámetro. Ejemplo: si el string de entrada es 'hola', el resultado será 'aloh'.
5. Escribe una función que compruebe si un número se encuentra dentro de un rango específico.
6. Escribe una función que reciba un número entero positivo como parámetro y devuelva una lista que contenga los 5 primeros múltiplos de dicho número. Por ejemplo, si la función recibe el número 3, devolverá la lista [3, 6, 9, 12, 15]. Si la función recibe un parámetro incorrecto (por ejemplo, un múmero menor o igual a cero), mostrará un mensaje de error por pantalla y devolverá una lista vacía.

7. Escribe una función que reciba una lista como parámetro y compruebe si la lista tiene duplicados. La función devolverá True si la lista tiene duplicados y False si no los tiene.
8. Escribe una función lambda que, al igual que la función desarrollada en el ejercicio anterior, reciba una lista como parámetro y compruebe si la lista tiene duplicados. La función devolverá True si la lista tiene duplicados y False si no los tiene.
9. Escribe una función que compruebe si un string dado es un palíndromo. Un palíndromo es una secuencia de caracteres que se lee igual de izquierda a derecha que de derecha a izquierda. Por ejemplo, la función devolverá True si recibe el string "reconocer" y False si recibe el string "python".

1.11 Soluciones

In [49]:
```python
# Función que calcula los cuadrados de los números en una lista
def calcular_cuadrados(numeros):
    # Cada número se multiplica por sí mismo (número * 2)
    lista_cuadrados = [numero ** 2 for numero in numeros]
    return lista_cuadrados

# Ejercicio 1
lista = [1, 2, 3, 4, 5]
resultado = calcular_cuadrados(lista)
print(f"Lista original: {lista}")
print(f"Lista con cuadrados: {resultado}")
```

```
Lista original: [1, 2, 3, 4, 5]
Lista con cuadrados: [1, 4, 9, 16, 25]
```

In [50]:
```python
# EJERCICIO 2: Suma de cantidad indeterminada de números
def sumar_numeros(*numeros):
    # El asterisco permite recibir cualquier cantidad de argumentos
    suma_total = 0
    for num in numeros:
        suma_total += num
    return suma_total
```

In [51]:
```python
# Función que suma una cantidad indeterminada de números
def sumar_numeros(*numeros):
    # *numeros recibe los argumentos como una tupla
    # sum() suma todos los números
    total = sum(numeros)
    return total

# Ejercicio 2
resultado = sumar_numeros(10, 20, 30, 40)
print(f"Suma: {resultado}")
```

```
Suma: 100
```

In [52]:
```python
# Función que invierte un string
def invertir_string(texto):
    # [::-1] invierte el string
    texto_invertido = texto[::-1]
```

```
    return texto_invertido

# Ejercicio 3
palabra = "hola"
resultado = invertir_string(palabra)
print(f"Palabra original: '{palabra}'")
print(f"Palabra invertida: '{resultado}'")
```

```
Palabra original: 'hola'
Palabra invertida: 'aloh'
```

In [53]:
```
# Función lambda que invierte un string
invertir_lambda = lambda texto: texto[::-1]

# Ejercicio 4
resultado = invertir_lambda("hola")
print(f"invertir_lambda('hola') = '{resultado}'")
```

```
invertir_lambda('hola') = 'aloh'
```

In [54]:
```
# Función que verifica si un número está dentro de un rango
def numero_en_rango(numero, minimo, maximo):
    # Comparamos si el número está entre el mínimo y máximo
    if numero >= minimo and numero <= maximo:
        return True
    else:
        return False

# Ejercicio 5
resultado = numero_en_rango(5, 1, 10)
print(f"¿5 está entre 1 y 10? {resultado}")
```

```
¿5 está entre 1 y 10? True
```

In [55]:
```
# Función que devuelve los 5 primeros múltiplos de un número
def obtener_multiplos(numero):
    # Validamos que el número sea positivo
    if numero <= 0:
        print(f"Error: El número debe ser positivo. Recibiste: {numero}")
        return []

    # Creamos una lista con los 5 primeros múltiplos
    multiplos = [numero * i for i in range(1, 6)]
    return multiplos

# Ejercicio 6
resultado = obtener_multiplos(3)
print(f"Los 5 primeros múltiplos de 3 son: {resultado}")
```

```
Los 5 primeros múltiplos de 3 son: [3, 6, 9, 12, 15]
```

In [56]:
```
# Función que detecta si una lista tiene duplicados
def tiene_duplicados(lista):
    # Comparamos la longitud de la lista original con un conjunto
    # Un set elimina elementos duplicados
    if len(lista) != len(set(lista)):
        return True
    else:
```

```
        return False

# Ejercicio 7
lista = [1, 2, 3, 2, 5]
resultado = tiene_duplicados(lista)
print(f"Lista: {lista}")
print(f"¿Tiene duplicados? {resultado}")
```

```
Lista: [1, 2, 3, 2, 5]
¿Tiene duplicados? True
```

In [57]:
```
# Función lambda que detecta duplicados
tiene_duplicados_lambda = lambda lista: len(lista) != len(set(lista))

# Ejercicio 8
lista = [1, 2, 3, 2, 3, 1, 5]
resultado = tiene_duplicados_lambda(lista)
print(f"Lista: {lista}")
print(f"¿Tiene duplicados? {resultado}")
```

```
Lista: [1, 2, 3, 2, 3, 1, 5]
¿Tiene duplicados? True
```

In [58]:
```
# Función que verifica si un string es un palíndromo
def es_palindromo(texto):
    # Convertimos a minúsculas y comparamos con su inverso
    texto_limpio = texto.lower()

    if texto_limpio == texto_limpio[::-1]:
        return True
    else:
        return False

# Ejercicio 9
palabra = "reconocer"
resultado = es_palindromo(palabra)
print(f"¿'{palabra}' es palíndromo? {resultado}")
```

```
¿'reconocer' es palíndromo? True
```

2. Github

https://github.com/Erick-305/machine-learning.git