# Funciones

January 14, 2026

INSTITUTO SUPERIOR TECNOLÓGICO QUITO

Formamos tu PROPÓSITO DE VIDA

## 1 Funciones en Python


### 1.1 Nombre: *Adriel Bedoya*

### 1.2 Funciones

- 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.

[2]:
```
#EJEMPLO
#def variable (vacío o datos)
def sumar (x,y):
    #Creamos una variable que guarde las suma de los parámetros
    suma = x+y
    #Retornamos la suma
    return suma
#Para poder instanciar la función, ponemos nombrefuncion(parámetros)
#2+3 = 5
sumar(2,3)
```

[2]:
```
5
```

[3]:
```
#EJEMPLO 2
"""
Transformar mediante una función de grados centígrados a grados fahrenheit
"""

#Formula: #f = c * 1.793 + 32

def de_centi_a_faren (centigrados):
    return centigrados * 1.793 + 32

#Creamos una variable para los grados
grados_horno_centigrados = 45
#Variable de la función (parámetro)
de_centi_a_faren(grados_horno_centigrados)
```

[3]:
```
112.685
```

### 1.2.1 ¿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.

### 1.2.2 Programación de Funciones en Python

- 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.

**Sintaxis** `def name(arg1, arg2, ..., argN):`

`` `statements` ``

[6]:
```
#PASOS PARA FUNCIONES
#Creamos la función para poder crear un diccionario que tendrá para poner los␣
 ↪parámetros de "clave" y "valor"
def crear_diccionario (keys, values):
    #Retornamos la unión de las dos listas y luego lo transformamos a un␣
 ↪diccionario
    return dict(zip(keys, values))

#OBSERVACIÓN: PRIMERO SE DEBE CREAR EL MÉTODO Y LUEGO LA LÓGICA

#Lista 1
estudiantes = ["Alejandro Bedoya", "David Ruiz", "Martín Rodriguez", "Anderson␣
 ↪Soto"]
#Lista 2
calificaciones_machine = [10,8,4,2]

#Creamos una variable para poder iterar la función con las dos listas antes␣
 ↪creadas
estudiantes_calificaciones = crear_diccionario (estudiantes,␣
 ↪calificaciones_machine)
#print
print(estudiantes_calificaciones)
```

```
{'Alejandro Bedoya': 10, 'David Ruiz': 8, 'Martín Rodriguez': 4, 'Anderson
Soto': 2}
```

[8]:
```
#Podemos poner funciones dentro de condicionales
#valor dado
a = -2
#si "a" es mayor a 0, entonces
if a > 0:
    #Se hará una suma con la función "operación"
    def operacion(x, y):
        return x + y
#sino
else:
    #Se hará una multiplicación con la función "operación"
    def operacion(x, y):
        return x * y
#Llamamos a la función y realiza la operación
print(operacion(3, 4))
```

```
12
```

[10]:
```
#Argumentos y return son opcionales
def print_hola_mundo():
    print('Hola Mundo')
    return # es opcional colocarlo
```

```
print_hola_mundo()
```

```
Hola Mundo
```

[11]:
```
#Cuando se llama a una función y esta no tiene return. Se devuelve None

def print_hola_mundo():

    print('Hola Mundo')
    # return None

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

```
Hola Mundo
None
```

[19]:
```
def division_segura (num1, num2):
    if num2 == 0:
        return "Hola"
    return num1 / num2

print(division_segura(8,2))
print(division_segura(4,0))

resultado = division_segura(4,0)
print(resultado)
print(type(resultado))
```

```
4.0
Hola
Hola
<class 'str'>
```

[26]:
```
#Puede devolver más de un valor de retorno. Se devuelve en forma de tupla
def devolver_coleccion_por_separado (coleccion):
    #Regreso la posición 0, 1 y 2 de la colección
    return coleccion[0],coleccion[1],coleccion[2]

#Creamos una variable para tener la colección separada
lista_alumnos = devolver_coleccion_por_separado(["Alejandro Bedoya", "Adriel␣
 ↪Suárez", "Mateo Santána"])
#print
print(lista_alumnos)
#print lugar 2
print(lista_alumnos[1])
#print tipo de dato (tuple)
print(type(lista_alumnos))
```

```
print("-------------------------------------------------")

"""
También se pueden almacenar en variables por separado
"""
primero, segundo, tercero = devolver_coleccion_por_separado(["Alejandro␣
 ↪Bedoya", "Adriel Suárez", "Mateo Santána"])
print(primero)
print(segundo)
print(tercero)
```

```
('Alejandro Bedoya', 'Adriel Suárez', 'Mateo Santána')
Adriel Suárez
<class 'tuple'>
-------------------------------------------------
Alejandro Bedoya
Adriel Suárez
Mateo Santána
```

[27]:
```
#Los argumentos pueden tener valores por defecto, siempre al final
def verificador(a,b,c = None):
    #Si no es nulo, entonces
    if a is not None:
        #mensaje
        print("Puedo aparecer")
    #Si no es nulo, entonces
    if b is not None:
        #mensaje
        print("Puedo aparecer")
    #Si no es nulo, entonces
    if c is not None:
        #mensaje
        print("Puedo aparecer")

#Usamos la función
verificador(4,25,2)
```

```
Puedo aparecer
Puedo aparecer
Puedo aparecer
```

[30]:
```
#Los parámetros se pasan por posición, pero el orden se puede alterar si se␣
 ↪especifica el nombre del parámetro en la llamada, "named values".

#Crearemos una función para calcular la potencia de un número
def potenciacion (num_base, num_exponente):
    #Retornamos la función reservada "pow()"
    return pow(num_base, num_exponente)
```

```
#Podemos hacerlo con varlores default
print(potenciacion(2,3))
#Podemos hacerlo directamente con las variables puestas en la potencia, si se␣
 ↪pone otras variables saldrá el error "potenciacion() got an unexpected␣
 ↪keyword argument 'variable_incorrecta'"
print(potenciacion(num_base = 3, num_exponente = 3))
```

```
8
27
```

- 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.

[33]:
```
def sumar (x,y):
    return x + y

#Podemos hacer que una función se almacene en una variable
f = sumar
#Tendrán el mismo id
print(id(f))
print(id(sumar))

#La variable será de tipo función
print(type(f))

#En ambas se puede poner los parámetros y harán lo mismo
print(sumar(2,2))
print(f(3,3))
```

```
2515217301344
2515217301344
<class 'function'>
4
6
```

[41]:
```
#Creamos una función que obtendrá dos parámetros
def limpiar_texto (strings, operaciones):
    #Creamos una lista vacía para almacenar valores después
    resultado = []
    #Hacemos un for para que recorra una lista asignada
    for valores in strings:
        #Hacemos un for para recorrer las operaciones que tendrá esa lista
        for funciones in operaciones:
            #los valores serán igual a la aplicación de ellos mediante las␣
 ↪operaciones (funciones)
            valores = funciones(valores)
```

```
        #Fuera del for, añadimos los valores ya corregidos
        resultado.append(valores)
    #Retornamos la lista "resultado"
    return resultado

#Creamos otra función llamada "juntar_textos" que recibirá un valor
def juntar_textos(textos):
    #Retornará el texto + "_país"
    return textos + "_país"

#Lista asignada
list_alumnos = ["         ecuador    ", "                              colombia ",␣
 ↪"     perú              ", "         el salvador        " ]
#print para ver el inicio de la lista
print(list_alumnos)
print()

#Operaciones: Quitar espacios, Poner las primeras letras en mayúsculas y␣
 ↪añadimos la función de agregar el "_país"
limpieza = [str.strip, str.title, juntar_textos]

print(limpiar_texto(list_alumnos, limpieza))
```

```
['         ecuador    ', '                              colombia ', '     perú
', '         el salvador        ']

['Ecuador_país', 'Colombia_país', 'Perú_país', 'El Salvador_país']
```

### 1.2.3 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.

[43]:
```
def cambiar_valor (x):
    x+=3 #es como si fuera x= x + 3
    print("Dentro de la función", x)
    return

a= 2
cambiar_valor(a)
#Esta está fuera de la función
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.

[46]:
```
#Agregamos un valor
def agregar_valor(x):
    x.append(4)
#Creamos una lista
lista = [0, 1,3]
#Decimos que queremos agregar un valor a la lista (4)
agregar_valor(lista)
#Print
print(lista)
```

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

[48]:
```
# Con tuplas
def agregar_valor(x):
    print(id(x))
    return

lista = (0, 1)

print(id(lista))

anyadir_2_a(lista)

print(lista)
```

```
2515218463040
2515218463040
(0, 1)
```

**En caso de querer modificar un objeto de tipo primitivo, los cuáles pasan por valor, se puede devolver el resultado de la función y hacer una asignación:**

[50]:
```
#Creamos una función que me multiplique el parámetro x 2
def multiplicar_por_2(x):
    #multiplicamos el parámetro * 2
    x = x * 2
    #devolvemos el parámetro
    return x
#Creamos una variable númerica
a = 3
#Pasamos el parámetro a la función
a = multiplicar_por_2(a)
#Print
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:**

[51]:
```
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]
```

[52]:
```
# Alternativa para obtener una copia: slicing.
lista = [0, 1]
anyadir_2_a(lista[:])
print('Fuera de la función: ', lista)
```

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

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

[102]:
```
import copy

#Creamos una lista
lista = [2, 4, 16, 32, [34, 10, [5,5]]]

# copia = lista.copy() --- esta copia es superficial, hace una copia del primer␣
 ↪nivel de la lista
copia = copy.deepcopy(lista) #Copia todos los niveles internos: listas dentro␣
 ↪de listas, diccionarios, objetos, etc.

#Hacemos que el primer puesto de copia sea 454
copia[0] = 454

#Hacemos que dentro de la sublista en el 4, dentro de la sublista en 2, en la␣
 ↪posición 0, sea 64
copia[4][2][0] = 64

#imprimimos lista original
print(lista)

#imprimimos la copia
print(copia)

#Id de ambas
print(f"{id(lista)} - {id(copia)}")
```

```
#Id de ambas
print(f"{id(lista[4])} - {id(copia[4])}")
```

```
[2, 4, 16, 32, [34, 10, [5, 5]]]
[454, 4, 16, 32, [34, 10, [64, 5]]]
2515218798400 - 2515218455808
2515219127872 - 2515218079744
```

#Reasignar un parámetro nunca afecta al objeto de fuera: #Creamos una función def funcion_poco_util(x): #Obtenemos el id del parámetro print(id(x)) #el parámetro se iguala a una lista nueva x = [2, 3] #Añadimos el 4 x.append(4) #mostramos el id print(id(x))

lista = [0, 1] print(id(lista)) #No se podrá reasignar el parámetro dentro de una función funcion_poco_util(lista) print(lista) print("————————————————————————————-")

#Con tuplas def modif_tupla(a): print(id(a))

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

### 1.2.4 Argumentos arbitrarios

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

[62]:
```
#Sirve cuando no se sabe cuantos elementos podrán haber en esta función y se␣
 ↪reciben como una tupla
#sintaxis:
#def nombre (*parámetro):

#Creamos una función que reciba un parametro y se multiplique
def saludar(*names):
    #Se reciben como una tupla
    print(type(names))
    #Recorremos toda la lista y guardamos en una variable
    for name in names:
        #Colocamos un mensaje más el parámetro
        print("Hola " + name)

#Si no hay nada, entonces solo saldrá que es de clase tupla
saludar()
#Si hay parámetros, entonces aplicaremos el bucle
saludar("Manolo", "Pepe", "Luis", "Alex", "Juan")
```

```
<class 'tuple'>
<class 'tuple'>
Hola Manolo
Hola Pepe
Hola Luis
Hola Alex
Hola Juan
```

### 1.2.5 Desempaquetado en Funciones

[65]:
```
#Definimos una función con 3 parámetros
def saludar(quien, mensaje, gesto):
    #Aplicamos el mensaje según los 3 parámetros individuales
    print(f"Hola {quien}, {mensaje}, mira esto: {gesto}")

"""
Hay dos formas de desempaquetar: Manual y Función
"""

#Manualmente
ls_saludo = ['Manolo', 'te quiere saludar', 'añañin']
saludar(ls_saludo[0], ls_saludo[1], ls_saludo[2])

#Mediante función
#El asterisco hace que desempaquete todo lo que está en ls_saludo
saludar(*ls_saludo) #básicamente hace esto saludar(ls_saludo[0], ls_saludo[1],
 ↪ls_saludo[2])
```

```
Hola Manolo, te quiere saludar, mira esto: añañin
Hola Manolo, te quiere saludar, mira esto: añañin
```

[67]:
```
#Es lo mismo para el de arriba, pero con la diferencia de
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'
}

#El doble * sirve para desempaquetar por pares, es decir, por clave - valor
saludar(**dc_saludo)
#Intenta hacer esto
"""
saludar(
    quien=[12, 12, 3, 4, 5, 3],
    gesto='guiño',
    mensaje='te quiere enseñar esto'
)

"""
print()
```

```
Hola [12, 12, 3, 4, 5, 3], te quiere enseñar esto, mira esto: guiño
```

### 1.2.6 Recursividad

Son funciones que se llaman así mismas

[70]:
```python
#Primera forma de hacerlo
#Definimos la función de factorial que obtendrá un parámetro
def factorial(n):
    #Si el número llega a 0, entonces retorna 1
    if n == 0:
        return 1
    #si aún no pasa
    else:
        #retorna n * el número (se reduce de uno en uno mientras sigue el bucle)
        return n * factorial(n-1)

print(factorial(3)) #3x2x1

#Otra forma de hacerlo
#Definimos una función que obtenga un parámetro
def fact(n):
    #creamos una variable que sería el acumulador
    a = 1
    #Creamos un bucle for que vaya por índice en el rango de 1 al parámetro + 1
    #el +1 es porque necesitamos tener el rango total y no excluir ningún número
    for i in range(1, n+1):
        #Mientras recorre, se hará a = a * i
        a *= i
    #retorna a
    return a

print(fact(3)) #1x2x3
```

```
6
6
```

## 1.3 Funciones Lambda

Es posible definir funciones **anonimas**, son expresiones; por lo tanto, pueden **aparecer** en lugares donde una secuencia def no puede (dentro de una lista o como parámetro de una función). Son útiles cuando se **quiere pasar una función como argumento a otra**

### 1.3.1 Formato

`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.

[71]:
```
#Ejemplo de una suma
def suma(x,y,z):
    return x+y+z
print(suma(1,2,3)) #6
```

```
6
```

[72]:
```
#Ahora con lambda
suma_variable = lambda x,y,z : x+y+z
print(suma_variable(1,2,3)) #6
```

```
6
```

[77]:
```
# La funcion 'sort' puede recibir una función optional como parámetro.
# Esta función 'key' se invoca para cada elemento de la lista antes de realizar␣
 ↪las comparaciones.
# Ejemplo: ordenar strings por número de carácteres.

ciudades = ["Quito", "Cuenca", "Guayaquil", "Manabí"]

#Ordenamos por orden alfabético, recuerda que sort es para strings y sorted es␣
 ↪para números
ciudades.sort()
#print
print(ciudades)

#Ordena el string por número de caracteres
ciudades.sort(key = lambda x : len(x))
print(ciudades)

#Misma que la de arriba, pero de otra forma
ciudades.sort(key = len)
print(ciudades)

#Misma que la primera de sort key:len pero de otra forma
def por_numero_ch (x):
    return len(x)

ciudades.sort(key = por_numero_ch)
print(ciudades)
```

```
['Cuenca', 'Guayaquil', 'Manabí', 'Quito']
['Quito', 'Cuenca', 'Manabí', 'Guayaquil']
['Quito', 'Cuenca', 'Manabí', 'Guayaquil']
['Quito', 'Cuenca', 'Manabí', 'Guayaquil']
```

[79]:
```
#Mismo ejemplo pero sin lambda
```

```
#Creamos una lista que tenga las ciudades
cities = ['Valencia', 'Lugo', 'Barcelona', 'Madrid']

#Creamos una función que recibe un parámetro
def num_caracteres(x):
    #Retornamos el número de carácteres
    return len(x)
#Ordenamos en base al número de caracteres
cities.sort(key = num_caracteres)
#print
print(cities)
```

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

[81]:
```
#Nos podemos guardar una referencia para utilización repetida
#Creamos una lista
cities = ['Valencia', 'Lugo', 'Barcelona', 'Madrid']
#Dentro de una variable, usaremos la función lambda para poder tener el número␣
 ↪de carácteres
f = lambda x : len(x)

#Bucle de repetición
for s in cities:
    #Aplicamos el lambda
    print(f(s))
```

```
8
4
9
6
```

[82]:
```
#Las expresiones lambda pueden formar parte de una lista

# Lista de lambdas para mostrar las potencias de 2.
#Lista donde cada función lambda potencia desde el 0 al 4
pows = [lambda x: x ** 0,
        lambda x: x ** 1,
        lambda x: x ** 2,
        lambda x: x ** 3,
        lambda x: x ** 4]

#Creamos un bucle donde recorra todas las funciones
for f in pows:
    #Damos un parámetro para que haga las 5 potencias
    print(f(2))
```

```
1
2
4
```

```
8
16
```

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

[84]:
```
#Creamos una función que tendrá 3 parámetros: operador, primer número y segundo␣
 ↪número
def switch_dict(operator, x, y):
    # crea un diccionario de opciones y selecciona 'operator'
    return {
        'add': lambda: x + y,#suma
        'sub': lambda: x - y,#resta
        'mul': lambda: x * y,#multiplicación
        'div': lambda: x / y,#división
    }.get(operator, lambda: None)() # retorna None si no encuentra 'operator'

print(switch_dict('add',2,2)) #2+2
print(switch_dict('div',10,2)) #10/2
print(switch_dict('mod',17,3)) #No existe mod, así que se va None
```

```
4
5.0
None
```

## 1.4 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.

[85]:
```
X = 10
def func():
    X = 20 # Local a la función. Es una variable diferente, no visible fuera de␣
 ↪'func'
    def func_int():
        X = 30
        print(X)
    func_int()
func()
```

```
30
```

### 1.4.1 Regla LEGB

1. **L**ocal: variables locales a F
2. **E**nclosing: variables localizadas en funciones que contienen a (o por encima de) F.

3. **G**lobal: variables definidas en el módulo (fichero) que no están contenidas en ninguna función. Este scope no abarca más de un fichero
4. **B**uilt-ins: proporcionadas por el lenguaje.

**Las resoluciones ocurren de abajo hacia arriba**

[86]:
```
#Global (X y func)
X = 99

#Creamos una función
def func(Y):
    #Todo lo que esté dentro, son variables locales, es decir, no se podrán␣
 ↪usar en ningún otro lado
    Z = X + Y
    return Z

print(func(1))
```

```
100
```

[89]:
```
X = 1 #se mantiene

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

#Recordemos que no podemos asignar un valor dentro de la función, no lo toma
func()
print(X)
```

```
1
```

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

[91]:
```
#Es parecido al anterior PERO
X = 1

def func():
    global X #Aquí la transformamos de variable local a GLOBAL
    X = 2002 #Eso permitirá reemplazar la variable de arriba tranquilamente

func()
print(X)
```

```
2002
```

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

[92]:
```
#Tenemos 2 variables
y, z = 1, 2
```

```
#Creamos una función que no tendrá parámetros
def todas_globales():
    #Hacemos que x sea una variable global
    global x #x almacena las variables de arriba, no se referencian, solo se␣
 ↪modifican
    #Se pueden sumar
    x = y + z

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

```
3
1
2
```

[95]:
```
X = 77

def f1():
    X = 88
    def f2():
        print(X)
f2() #F2 no está definido porque está dentro de una función, es una variable␣
 ↪encapsulada
f1()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[95], line 7
      5     def f2():
      6         print(X)
----> 7 f2() #F2 no está definido porque está dentro de una función
      8 f1()

NameError: name 'f2' is not defined
```

### 1.4.2 Closures (Cierres)

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

[96]:
```
def f1():
    X = 88 # Enclosing scope (para f2)
    def f2():
        print(X)
    return f2 # Devuelve f2, sin invocarla
```

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

```
88
```

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

[97]:
```
def f1():
    contador = 10
    def f2():
        nonlocal contador
        contador += 1
        print(contador)
    return f2

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

```
11
12
13
```

## 1.5 Ejercicios

[100]:
```
"""
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.
"""

#LÓGICA:
#1) Creamos una función que tendrá un parámetro para la lista
#2) Creamos una lista vacía llamada "resultado"
#3) Hacemos un for que recorrerá el parámetro de la lista que nos manden de␣
 ↪afuera
#4) Creamos una variable que almacene la multiplicación entre números (para el␣
 ↪cuadrado)
#Podría ser numeros * numeros o pow (numeros, 2)
#5) Añadimos los números resultantes a la lista resultado
#6) Retornamos fuera del for la lista resultado
#7) Creamos una lista con números
#8) hacemos un print haciendo referencia a la función y la lista para dar

#Paso 1:
def lista_cuadratica (lista):
    #Paso 2:
```

```
    resultado = []
    #Paso 3:
    for numeros in lista:
        #Paso 4:
        cuadratica = pow(numeros,2)
        #Paso 5:
        resultado.append(cuadratica)
    #Paso 6:
    return resultado
#Paso 7:
dinero = [2,3,4]
#Paso 8:
print(lista_cuadratica(dinero))
```

```
[4, 9, 16]
```

[108]:
```
"""
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.).
"""
#Esta función basa en argumentos arbitrarios, es decir, que no se sabe cuantos␣
 ↪números recibirá
#LÓGICA:
#1) Creamos una función con el parámetro arbitrario (con el *)
#2) Creamos un acumulador dentro de la función para que se vayan sumando entre␣
 ↪ellos
#3) Creamos un for para recorrer los números que nos den
#4) Hacemos que acumulador se vaya sumando por cada número que esté
#5) Retornamos el acumulador fuera del for para que se cuenten la cantidad␣
 ↪todal de la suma
#6) Hacemos referencia a la función y ponemos los datos

#Paso 1:
def sumar_num (*numeros):
    #Paso 2:
    acumulador = 0
    #Paso 3:
    for recorrer_num in numeros:
        #Paso 4:
        acumulador += recorrer_num
    #Paso 5:
    return acumulador
#Paso 6:
sumar_num(1,2,3)
```

[108]: 6

[112]:
```
"""
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'.
"""
#LÓGICA
#1) Creamos una función que almacene el string
#2) Retornamos un string_slice con -1 para que se lea al revés
#3) Llamamos a la función con la palabra

#Paso 1:
def convertir_reves(string):
    #Paso 2:
    return string[::-1]
#Paso 3:
convertir_reves("Hola")
```

[112]: 'aloH'

[113]:
```
"""
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'.
"""
#Formula:
#variable = lambda parámetro : operación

#LÓGICA:
#1) Crear una variable donde se almacene el lambda, luego dar el parámetro y al␣
 ↪final realizar la operación
#2) Imprimir a la variable con su función lambda

#Paso 1:
invertir_string = lambda string : string[::-1]
#Paso 2:
print(invertir_string("hola"))
```

aloh

[118]:
```
"""
Escribe una función que compruebe si un número se encuentra dentro de un rango␣
 ↪específico.
"""
```

```
#LÓGICA:
#1) Crear una función que reciba 3 parámetros: el rango mínimo, el rango máximo␣
 ↪y el número a encontrar
#2) Hacemos un if donde diga que si encontrar es mayor o igual al mínimo y␣
 ↪encontrar es menor o igual al máximo retorne "dentro del rango"
#3) Caso contrario, saldrá que está fuera de rango
#4) Hacemos referencia a la función y ponemos los 3 valores

#Paso 1:
def encontrar_numero(minimo, maximo, encontrar):
    #Paso 2:
    if minimo <= encontrar <= maximo:
        return "Está dentro del rango"
    #Paso 3:
    else:
        return "Está fuera del rango permitido"
#Paso 4:
encontrar_numero(2,20,14)
```

[118]: 'Está dentro del rango'

```
"""
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.
"""

#LÓGICA
#1) Crear una función que reciba un número
#2) Creamos una lista vacía
#3) Hacemos un if para solo tener números positivos recibiendo
#4) Hacemos un for para recorrer los números del 1 al 5
#5) Añadimos los números a la lista_m, los números que saldrán de la operación␣
 ↪de multiplos
#6) Retornamos lista_m cuando finalice
#7) Hacemos referencia a la función y ponemos un número para probar

#Paso 1:
def multiplos(numero):
    #Paso 2:
    lista_m = []
```

```
    #Paso 3:
    if numero < 0:
        return "Debe ser un número positivo"
    #Paso 4:
    for numeros in range(1,6):
        #Paso 5:
        lista_m.append(numero * numeros)
    #Paso 6:
    return lista_m

#Paso 7:
multiplos(3)
#multiplos(-2)
```

[122]:
```
[3, 6, 9, 12, 15]
```

[128]:
```
"""
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
"""
#LÓGICA
#1) Crear una función que obtenga la lista
#2) Crear un for que recorra cada número
#3) Crear un for que recorra los números y compara con los siguientes
#4) Crear un if, si la lista recorrida en total es igual a la lista de␣
 ↪comparación de repetidos, entonces dará verdadero
#5) Si no hay repetidos, dará falso
#6) Crear las funciones tanto para True como para False
#7) Verificación

#Paso 1:
def validacion_repetidos(lista):
    #Paso 2:
    for numeros in range(len(lista)):
        #Paso 3:
        for comparar in range(numeros + 1, len(lista)):
            #Paso 4:
            if lista[numeros] == lista[comparar]:
                return True
    #Paso 5:
    return False

#Paso 6:
lista_no_repetida = [1,2,3,4,5,6,7,8,9]
```

```
lista_repetida = [1,2,3,4,5,5,5,6,7,8,9]
#Paso 7:
validacion_repetidos(lista_no_repetida) #False porque no está repetida
```

[128]: False

[127]:
```
#Paso 7:
validacion_repetidos(lista_repetida) #True porque si hay valores que se repiten
```

[127]: True

[130]:
```
"""
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.
"""

#Formula:
#variable = lambda parámetro : operación

#LÓGICA
#1) Crear el lambda de la siguiente manera:
#Primero haremos referencia que queremos una lista, la operación será el total␣
 ↪de números que tiene la lista y si esta es diferente a un set de la misma,␣
 ↪dará verdadero
validacion = lambda lista : len(lista) != len(set(lista))

#2) Print
lista_sapos = [1,2,3,3,4]
lista_no_sapos = [1,2,3,4]
print(validacion(lista_sapos))
print(validacion(lista_no_sapos))
```

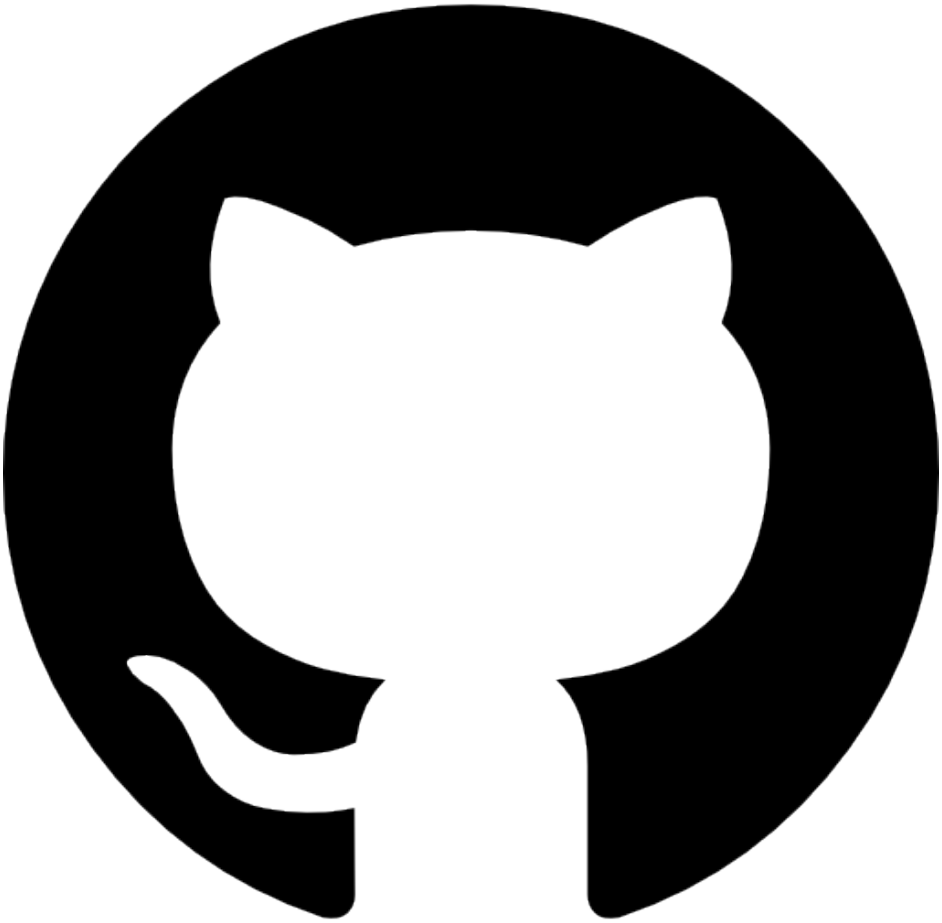
```
True
False
```

[132]:
```
"""
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".
```

```
"""
#LÓGICA
#1) Crear una función que reciba un string
def palindromo(string):
    #Retornamos que si la palabra string es == a la palabra en reverso
    return string == string[::-1]
#Saldrá true si son iguales
print(palindromo("oso"))
#Saldrá false si no lo son
print(palindromo("casa"))
```

```
True
False
```

## 2 Github



### 2.0.1 Click aquí para ver el repositorio