


## 9 Funciones y docstring.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra\\_Lineal\\_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

## 10 Definición de funciones

Las funciones son la primera forma de estructurar un programa. Esto nos lleva al paradigma de programación estructurada, junto con las construcciones de control de flujo. Las funciones nos permiten agrupar y reutilizar líneas de código.

La sintaxis es:

```
def nombre_de_la_función(parm1, parm2, ...):  
    bloque_de_código  
    return resultado
```

Una vez definida la función, es posible ejecutarla (hacer una llamada a la función) como sigue:

```
nombre_de_la_función(arg1, arg2, ...)
```

También es posible hacer lo siguiente:

En ambos casos, la función regresa un resultado debido a que existe la declaración `return` dentro de la función. Este resultado puede ser referenciado por una variable haciendo lo siguiente:

```
variable = nombre_de_la_función(arg1, arg2, ...)
```

La `variable` puede ser utilizada posteriormente para otros cálculos.

Observa que: \* Cuando se define la función, se definen los **parámetros** que recibirá, en este caso `param1`, `param2`, ... \* Cuando se ejecuta la función, se pasan los valores los `arg1`, `arg2`, ... , los cuales son los **argumentos** de la ejecución y serán sustituidos en los parámetros de la función.

Veamos un ejemplo simple:

```
# Función que calcula el cuadrado de su argumento.  
def squared(f):  
    return f ** 2
```

```
# Se ejecuta la función con el argumento 2
squared(2)
```

4

```
# Se ejecuta la función con el argumento 3
# el resultado se almacena en f2
f2 = squared(3)
print(f2)
```

9

Veamos ahora un ejemplo más interesante

```
# La siguiente función calcula la secuencia de Fibonacci
def fib(n): # La función se llama fib y tiene el parámetro n
    a, b = 0, 1
    while a < n:
        print(a, end=',')
        a, b = b, a+b
```

Observa que esta función no regresa ningún valor, solo imprime en pantalla un valor conforme lo calcula.

```
fib(50) # ejecutamos la función fib con el argumento 10
```

0,1,1,2,3,5,8,13,21,34,

Le podemos poner otro nombre a la función

```
Fibonacci = fib
```

```
Fibonacci(200)
```

0,1,1,2,3,5,8,13,21,34,55,89,144,

```
print(type(fib))
print(type(Fibonacci))
```

```
<class 'function'>
<class 'function'>
```

```
print(id(fib))
print(id(Fibonacci))
```

140670158000416

140670158000416

Observamos que se puede ejecutar la función `fib()` a través de `Fibonacci()` y que ambos nombres hacen referencia a la misma función.

## 11 Ámbitos

Las funciones (y otros operadores también), crean su propio ámbito, de tal manera que las etiquetas declaradas dentro de funciones son locales.

```
a = 20 # Objeto global etiquetado con a

def f():
    a = 21 # Objeto local etiquetado con a
    return a
```

```
# ¿Que valor tiene 'a' fuera de la función?
print(a)
```

20

```
# ¿Qué valor tiene la 'a' dentro de la función?
print(f())
```

20

Para usar el objeto global dentro de la función debemos usar `global`

```
a = 20

def f():
    global a
    return a
```

```
# ¿Que valor tiene 'a' fuera de la función?
print(a)
```

20

```
# La 'a' dentro de la función hace referencia a la 'a' global
print(f())
```

20

## 12 Retorno de una función

Como se mencionó antes, la declaración `return`, dentro de una función, regresa un objeto que en principio contiene el resultado de las operaciones realizadas por la función.

Veamos un ejemplo.

```
g = 9.81
# Función que calcula la posición y velocidad en el tiro vertical de un objeto.
def verticalThrow(t, v0):
    g = 3.1416 # [m / s**2]
    y = v0 * t - 0.5 * g * t**2
    v = v0 - g * t
    return (y, v) # regresa la posición [m] y la velocidad [m/s] en un objeto de
```

```
t = 2.0 # [s]
v0 = 20 # [m/s]
verticalThrow(t, v0)
```

(33.7168, 13.7168)

```
resultado = verticalThrow(t, v0)
```

```
print(resultado)
```

(33.7168, 13.7168)

## 13 Argumentos por omisión

Los parámetros de una función pueden tener valores (argumentos) por omisión, es decir, si no se da un valor para uno de los parámetros, entonces se toma el valor definido por omisión. Esto crea una función que se puede llamar con menos argumentos de los que está definida inicialmente.

Por ejemplo:

```
# Función que calcula la posición y velocidad en el tiro vertical de un objeto.
def verticalThrow(t, v0 = 20): # El valor 20 es un argumento por omisión
    g = 9.81 # [m / s**2]
    y = v0 * t - 0.5 * g * t**2
    v = v0 - g * t
    return (y, v)
```

```
pos, vel = verticalThrow(2.0) # El valor 2.0 corresponde al primer parámetro de 1
                             # En este caso v0 será igual a 20.
print(pos, vel)
```

20.38 0.3799999999999999

```
pos, vel = verticalThrow(2.0, 30) # En este caso v0 = 30
print(pos, vel)
```

40.379999999999995 10.379999999999999

Una función puede tener más de un argumento por omisión. Todos los parámetros que tienen argumentos por omisión deben estar al final de la lista en la declaración de la función.

Por ejemplo:

```
def f(a,b,c,d=10,e=20):
    return a+b+c+d+e
```

```
print(f(1,2,3)) # Se los dos argumentos por omisión 10 y 20
```

36

```
print(f(1,2,3,4)) # Se usa el último argumento por omisión 20
```

30

```
print(f(1,2,3,4,5)) # Se dan todos los argumentos.
```

15

## 14 Argumentos posicionales y keyword

Un **argumento** es el valor que se le pasa a una función cuando se llama. Hay dos tipos de argumentos:

### **Positional argument:**

1. Un argumento que no es precedido por un identificador: `verticalThrow(3, 50)`
2. Un argumento que es pasado en una tupla precedido por `*`: `verticalThrow(*(3, 50))`.

En este caso, el `*` indica a Python que la tupla `(3, 50)` debe desempacarse cuando se reciba en la función, de tal manera que `3` será el primer argumento y `5` el segundo.

La llamada a la función del punto 2 es equivalente a la del punto 1.

### **Keyword argument:**

3. Un argumento precedido por un identificador. `verticalThrow(t=3, v0=50)`
4. Un argumento que se pasa en un diccionario precedido por `**`: `verticalThrow(**{'t': 3, 'v0': 50})`.

En este caso, el `**` indica a Python que el diccionario `{'t': 3, 'v0': 50}` debe desempacarse cuando se reciba en la función. Observa que el diccionario contiene dos pares clave-valor: `'t': 3` y `'v0': 50`.

La llamada a la función del punto 4 es equivalente a la del punto 3.

Veamos los ejemplos en código:

Primer recordemos que la firma de la función es `def verticalThrow(t, v0 = 20):` es decir, el primer parámetro es `t` y el segundo `v0`.

```
# Los argumentos se sustituyen en los parámetros en el orden de acuerdo a su posición
verticalThrow(3,50)
```

```
(105.85499999999999, 20.57)
```

```
# Lo anterior NO es equivalente a:
verticalThrow(50,3)
```

```
(-12112.5, -487.5)
```

```
# Se pueden pasar los argumentos empacados en una tupla
verticalThrow(*(3,50))
```

```
(105.85499999999999, 20.57)
```

```
# Se puede usar el nombre del parámetro para determinar
# como se sustituyen los argumentos:
verticalThrow(t=3,v0=50)
```

```
(105.85499999999999, 20.57)
```

```
# Lo anterior SI es equivalente a:
verticalThrow(v0=50, t=3)
```

```
(105.85499999999999, 20.57)
```

```
# Se pueden pasar los argumentos empacados en un diccionario
verticalThrow(**{'t':3,'v0':50})
```

```
(105.85499999999999, 20.57)
```

```
# También se acepta esta forma:
verticalThrow(**dict(t = 3, v0 = 50))
```

```
(105.85499999999999, 20.57)
```

## 15 Número variable de parámetros

Dada la funcionalidad descrita en la sección anterior, es posible que una función reciba un número variable de argumentos.

```
# *args: número variable de Positional arguments empacados en una tupla
# *kwargs: número variable de Keyword arguments empacados en un diccionario
def parametrosVariables(*args, **kwargs):
    print('args es una tupla : ', args)
    print('kwargs es un diccionario: ', kwargs)
    print(set(kwargs))
```

```
parametrosVariables('one', 'two', 'three', 'four', a = 4, x=1, y=2, z=3, w=[1,2,2])
```

```
args es una tupla : ('one', 'two', 'three', 'four')
kwargs es un diccionario: {'a': 4, 'x': 1, 'y': 2, 'z': 3, 'w': [1, 2, 2]}
{'a', 'y', 'w', 'z', 'x'}
```

```
parametrosVariables(1,2,3, w=8, y='cadena')
```

```
args es una tupla : (1, 2, 3)
kwargs es un diccionario: {'w': 8, 'y': 'cadena'}
{'w', 'y'}
```

```
# Los argumentos que vienen en un diccionario se desempacan
# y se pueden usar dentro de la función:
def funcion_kargs(**argumentos):
    for key, val in argumentos.items():
        print(f" {key} : {val}")
```

```
funcion_kargs(nombre = 'Luis', apellido='de la Cruz', edad=15, peso=80.5 )
```

```
nombre : Luis
apellido : de la Cruz
edad : 15
peso : 80.5
```

```
funcion_kargs(nombre = 'Luis', apellido='de la Cruz', estudios='primaria', edad=1
```

```
nombre : Luis  
apellido : de la Cruz  
estudios : primaria  
edad : 15  
peso : 80.5  
num_cuenta : 12334457
```

```
# Se puede definir un diccionario  
mi_dicc = {'nombre':'Luis', 'apellido':'de la Cruz', 'edad':15, 'peso':80.5}
```

```
# Se usa el diccionario para llamar a la función  
funcion_kargs(**mi_dicc)
```

```
nombre : Luis  
apellido : de la Cruz  
edad : 15  
peso : 80.5
```

## 16 Funciones como parámetros de otras funciones.

Las funciones pueden recibir como argumentos objetos muy complejos, incluso otras funciones. Veamos un ejemplo simple:

```
# Un función simple  
def g():  
    print("Iniciando la función 'g()'")  
  
# Una función que reibirá otra función:  
def func(f):  
    print("Iniciando la función 'func()'")  
    print("Ejecución de la función 'f()', nombre real '" + f.__name__ + "()'" )  
    f() # Se ejecuta la función que se recibió en el parámetro f
```

```
func(g)
```

```
Iniciando la función 'func()'  
Ejecución de la función 'f()', nombre real 'g()'  
Iniciando la función 'g()'
```

### 16.0.1 Ejemplo 1. Integración numérica.



En este ejemplo el objetivo es crear un función que recibirá como argumentos la función matemática a integrar, los límites de integración y el número de puntos para realizar la integración. Regresará como resultado un número que es la aproximación de la integral.

```
import math

def integra(func,a,b,N):
    # Se utiliza el método de Simpson para la integración.
    # El parámetro 'func' es la función a integrar
    print(f"Integral de la función {func.__name__}() en el intervalo ({a},{b}) us
    h = (b - a) / N
    resultado = 0
    x = [a + h*i for i in range(N+1)]
    for xi in x:
        resultado += func(xi) * h
    return resultado
```

```
# Integral de la función sin() de la biblioteca math.
print(integra(math.sin, 0, math.pi, 100))

# Integral de la función cos() de la biblioteca math.
print(integra(math.cos, -0.5 * math.pi, 0.5 * math.pi, 50))
```

Integral de la función sin() en el intervalo (0,3.141592653589793) usando 100 puntos  
1.9998355038874436

Integral de la función cos() en el intervalo (-1.5707963267948966,1.5707963267948966)  
usando 50 puntos  
1.9993419830762613

## 17 Funciones que regresan otra función.

Como vimos antes, una función puede regresar un objeto de cualquier tipo, incluyendo una función. Veamos un ejemplo:

```
# La funcionPadre() regresará como resultado una de dos funciones
# definidas dentro de ella.
def funcionPadre(n):

    # Se define la función 1
    def funcionHijo1():
        return "funcionHijo1(): n = {}".format(n)

    # Se define la función 2
    def funcionHijo2():
        return "funcionHijo2(): n = {}".format(n)
```

```
# Se determina la función que se va a regresar
if n == 10:
    return funcionHijo1
else:
    return funcionHijo2
```

```
# La funcionPadre() regresa una función
funcionPadre(36)
```

```
<function __main__.funcionPadre.<locals>.funcionHijo2(>
```

```
# Asignamos el resultado de la funcionPadre() a un nombre
f1 = funcionPadre(10)
f2 = funcionPadre(20)
```

```
print(f1()) # Resultado de la funcionf1(), generada con la funcionPadre()
print(f2()) # Resultado de la funcionf2(), generada con la funcionPadre()
```

```
funcionHijo2(): n = 20
funcionHijo1(): n = 10
```

### 17.0.1 Ejemplo 2. Polinomios de segundo grado.

Implementar una fábrica de polinomios de segundo grado:

$$p(x) = ax^2 + bx + c$$

```
# Esta función recibe los coeficientes del polinomio
# y regresa una función que calcula el polinomio de
# segundo grado.
def polinomio(a, b, c):

    def polSegundoGrado(x):
        return a * x**2 + b * x + c

    return polSegundoGrado
```

```
# Dos polinomios de segundo grado
p1 = polinomio(2, 3, -1) # 2x^2 + 3x - 1
p2 = polinomio(-1, 2, 1) # -x^2 + 2x + 1

# Evaluación de los polinomios en el intervalo
# (-2,2) con pasos de 1
```

```
for x in range(-2, 2, 1):
    print(f'x = {x:3d} \t p1(x) = {p1(x):3d} \t p2(x) = {p2(x):3d}')
```

|        |            |            |
|--------|------------|------------|
| x = -2 | p1(x) = 1  | p2(x) = -7 |
| x = -1 | p1(x) = -2 | p2(x) = -2 |
| x = 0  | p1(x) = -1 | p2(x) = 1  |
| x = 1  | p1(x) = 4  | p2(x) = 2  |

## 17.0.2 Ejemplo 2. Polinomios de cualquier grado.

Implementar una fábrica de polinomios de cualquier grado:

$$\sum_{k=0}^n a_k x^k = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

```
# Esta función recibe un conjunto de argumentos variable
# para construir un polinomio de cualquier grado.
# Regresa la función que implementa el polinomio.
def polinomioFactory(*coeficientes):
```

```
    def polinomio(x):
        res = 0
        for i, coef in enumerate(coeficientes):
            res += coef * x ** i
        return res
```

```
    return polinomio
```

```
# Se generan 4 polinomios de diferente grado
p1 = polinomioFactory(5)           # a_0 = 5
p2 = polinomioFactory(2, 4)        # 4 x + 2
p3 = polinomioFactory(-1, 2, 1)    # x^2 + 2x - 1
p4 = polinomioFactory(0, 3, -1, 1) # x^3 - x^2 + 3x + 0
```

```
# Evaluación de los polinomios en el intervalo
# (-2,2) con pasos de 1
for x in range(-2, 2, 1):
    print(f'x = {x:3d} \t p1(x) = {p1(x):3d} \t p2(x) = {p2(x):3d} \t p3(x) = {p3(x):3d} \t p4(x) = {p4(x):3d}')
```

|        |           |            |            |             |
|--------|-----------|------------|------------|-------------|
| x = -2 | p1(x) = 5 | p2(x) = -6 | p3(x) = -1 | p4(x) = -18 |
| x = -1 | p1(x) = 5 | p2(x) = -2 | p3(x) = -2 | p4(x) = -5  |
| x = 0  | p1(x) = 5 | p2(x) = 2  | p3(x) = -1 | p4(x) = 0   |
| x = 1  | p1(x) = 5 | p2(x) = 6  | p3(x) = 2  | p4(x) = 3   |

## 18 Documentación con *docstring*

Python ofrece dos tipos básicos de comentarios para documentar el código:

### 1. Lineal.

Este tipo de comentarios se llevan a cabo utilizando el símbolo especial `#`. El intérprete de Python sabrá que todo lo que sigue delante de este símbolo es un comentario y por lo tanto no se toma en cuenta en la ejecución:

```
a = 10 # Este es un comentario
```

### 2. Docstrings

En programación, un *docstring* es una cadena de caracteres embebidas en el código fuente, similares a un comentario, para documentar un segmento de código específico. A diferencia de los comentarios tradicionales, las docstrings no se quitan del código cuando es analizado, sino que son retenidas a través de la ejecución del programa. Esto permite al programador inspeccionar esos comentarios en tiempo de ejecución, por ejemplo como un sistema de ayuda interactivo o como metadatos. En Python se utilizan las triples comillas para definir un *docstring*.

```
def funcion(x):  
    '''  
    Esta es una descripción de la función ...  
    '''  
  
def foo(y):  
    '''  
    También de esta manera se puede definir una docstring  
    '''
```

```
def suma(a,b):  
    '''  
    Esta función calcula la suma de los parámetros a y b.  
    Regresa el resultado de la suma  
    '''  
    return a + b
```

```
suma
```

```
<function __main__.suma(a, b)>
```

```
# En numpy se usa la siguiente definición de docstrings  
def suma(a,b):  
    '''  
    Calcula la suma de los dos parámetros a y b.  
  
    Args:  
        a: int Numero a sumar
```

```
        b: int Numero a sumar
Return:
        c: int Suma del numero a y b
...
c = a + b
return c
```

suma

<function \_\_main\_\_.suma(a, b)>

Existen diferentes estilos de documentación tipo *docstring* vease por ejemplo: [Numpy](#), [Matplotlib](#).

Para más información véase [PEP 257 – Docstring Conventions](#) y [PEP 8 – Style Guide for Python Code](#).