<codoacodo/>

Agencia de Aprendizaje a lo largo de la vida

FULL STACK PYTHON
Clase 28

PYTHON 4

BA

Funciones

Les damos la bienvenida

Vamos a comenzar a grabar la clase

Clase 27

Cadenas y Listas

- Cadenas de caracteres.
- Métodos de listas.
- f-strings
- Índices y slicing (rebanadas).
- Tipo de datos compuestos.
- Listas. Métodos.
- Tipos de datos mutables e inmutables.
- Tuplas, diccionarios, conjuntos

Clase 28

Funciones

- Funciones. Concepto.
- Llamada a función.
- Retorno y envío de valores.
- Parámetros, argumentos, valor y referencia.
- Parámetros mutables e inmutables.
- Parámetros por defecto
- Docstring.
- Funciones Lambda/Anónima.

Clase 29

Clases y objetos

- Paradigmas de programación. Programación estructurada vs POO.
- Clases, objetos y atributos.
- Métodos de clase y métodos especiales: init, del y str.

Funciones

En Python, una función es un grupo de instrucciones que constituyen una unidad lógica dentro del programa. Resuelve un problema específico, y permiten la modularidad del código.
Una función puede definir opcionalmente parámetros de entrada, que permiten pasar argumentos a la función en el momento de su llamada. Además, una función también puede devolver un valor como salida.
Las funciones nos permiten dividir el trabajo que hace un programa en tareas más pequeñas, separadas del código principal. Ese es el concepto de función en programación.

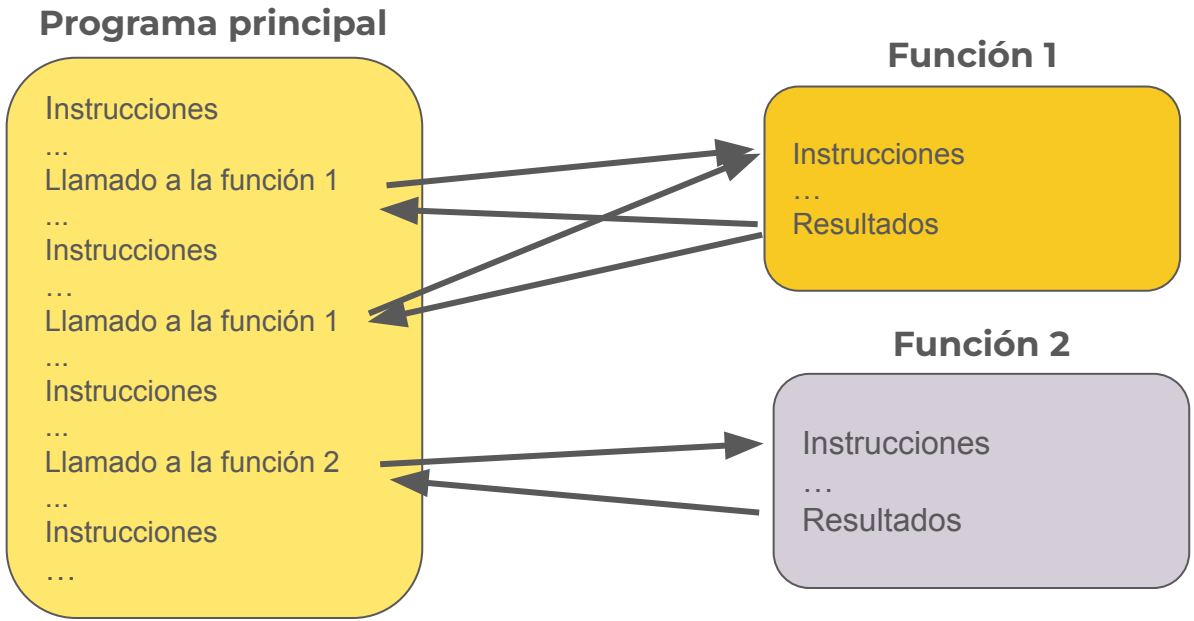
Funciones

Beneficios de la programación funcional:

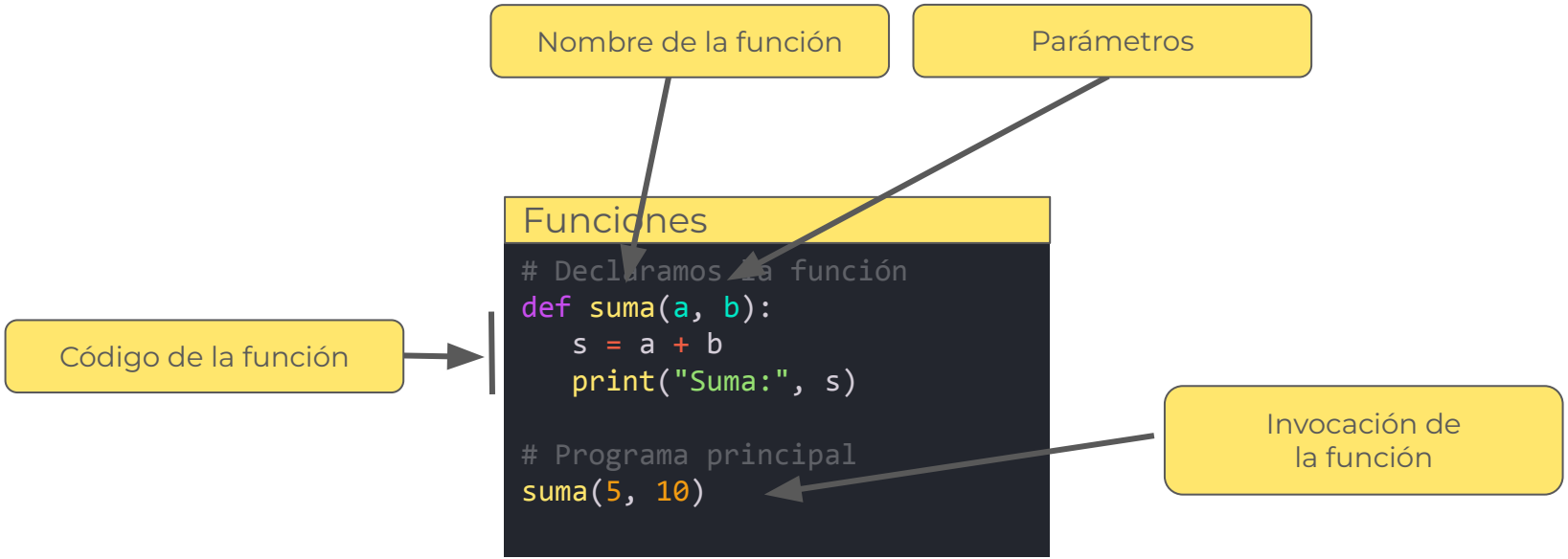
- **Facilita el trabajo en equipo.** Al modularizar el código permite trabajar en unidades lógicas separadas.
- **Encapsulamiento.** Dividir y organizar el código en partes más sencillas que se pueden encapsular en funciones y ser reutilizado a lo largo del proyecto.
- **Simplifica la lectura.** El código estructurado en funciones tiene un cuerpo principal reducido y funciones bien delimitadas.
- **Reutilización.** El código encapsulado en una función puede utilizarse en diferentes proyectos.
- **Mantenimiento:** El software que utiliza funciones es más fácil de mantener

Funciones

Las funciones permiten ejecutar código fuera del flujo normal del programa, recibiendo y devolviendo datos.



Funciones | Definición



Funciones | Definición

Elementos de una función:

- Primera línea: cabecera o definición de la función:
  - **def**: palabra reservada que define una función.
  - nombre o identificador: se utiliza para invocar la función.
  - Parámetros: encerrados entre paréntesis, son opcionales.
  - Dos puntos: indican el cierre de la cabecera.
- Cuerpo: se delimita con la indentación, y constituye el código que se encapsula en la función.
- **Return**: Es opcional, permite a la función devolver valores al cuerpo principal del programa.

Funciones | Definición

Los nombres de las funciones siguen las mismas pautas vistas para nombrar variables, aunque utilizando verbos en infinitivo.

Ejemplo de uso de una función

```
# Definición de la función
def imprimir_mensaje_cinco_veces():
    for i in range(5):
        print("Este es el mensaje " + str(i))

# Invocación de la función
imprimir_mensaje_cinco_veces()
```

Terminal

```
Este es el mensaje 0
Este es el mensaje 1
Este es el mensaje 2
Este es el mensaje 3
Este es el mensaje 4
```

La función *def nombre_funcion()* se debe definir antes de ser invocada por primera vez. Desde el programa principal se invoca a la función escribiendo su nombre.Una función sin parámetros, como la del ejemplo, realiza un objetivo sin recibir información desde el programa principal.

Funciones | Parámetros y Argumentos

Los **parámetros** son las variables que ponemos cuando se define una función. En la siguiente función tenemos un parámetro: *numero*:

```
def multiplicar_por_5(numero):
    return numero * 5
```

Los **argumentos** son los valores que se pasan a la función cuando ésta es invocada, “7” en el ejemplo:

```
# Programa principal
num = 7
producto = multiplicar_por_5(num) # Invocamos la función
print(f'El resultado de multiplicar {num} por 5 es {producto}')
```

Dentro de la función, los **argumentos** se copian en los parámetros y son usados por ésta para realizar la tarea.

Funciones | Parámetros y Argumentos

Esta función tiene un sólo **parámetro** que indica hasta qué valor calculará:

```python
# Declaración
def tabla_multiplicar(hasta):
    for i in range(hasta):
        print(f'1 x {i} = {1 * i}')
```

```python
# Ejecución
tabla_multiplicar(4)
```

En este ejemplo la función muestra un texto concatenado a un **argumento** pasado por **parámetro**:

```python
# Declaración
def saludar_dos(nombre):
    print(f'Hola {nombre}')
```

```python
# Ejecución
saludar_dos("Codo a Codo") # Argumento fijo
nombre = input("Ingrese su nombre: ")
saludar_dos(nombre) # Argumento variable
```

Funciones | Parámetros

La siguiente función tiene dos parámetros, cuyos argumentos se obtienen mediante la función **input()**:

Función con dos parámetros

```
def imprimir_mensaje_N_veces(n, m):
    for i in range(n):
        print(m)

mensaje = input("Mensaje: ")
veces = int(input("Nro. de veces que desea imprimir: "))
imprimir_mensaje_N_veces(veces, mensaje)
```

Terminal

```
Mensaje: ¡Hola Codo!
Nro. de veces que desea
imprimir: 4
¡Hola Codo!
¡Hola Codo!
¡Hola Codo!
¡Hola Codo!
```

Los argumentos son **veces** y **mensaje**, y los parámetros son **n** y **m**. Al llamar a la función, **veces** se copia en **n**, y **mensaje** se copia en **m.** Se mantiene el orden y la correspondencia entre argumentos y parámetros.

Funciones | Paso por valor y por referencia

Dependiendo del tipo de dato que enviemos a la función, se tienen dos comportamientos diferentes:

- **Paso por valor:** se crea una copia del valor de los argumentos en los respectivos parámetros. Las modificaciones en los valores de los parámetros no afectan a las variables externas.
- **Paso por referencia:** se pasa un puntero a la posición de memoria donde se aloja el dato, por lo que cualquier cambio que se haga en su valor dentro de la función afecta el contenido de la variable en el resto del programa.

En Python, los argumentos de tipos de datos simples (enteros, flotantes, cadenas, lógicos) se pasan por valor, y los tipos de datos compuestos (listas, diccionarios, conjuntos...)se pasan por referencia. +info

Funciones | Parámetros opcionales

En una función Python se pueden indicar una serie de **parámetros opcionales** con el operador **=**. Son parámetros que, si no se incluyen al invocar a la función, toman ese valor por defecto.

Parámetros opcionales

```
def sumar(a = 0, b = 0):
    return a + b

print(sumar(2,6))
print(sumar(5))
print(sumar())
```

Terminal

```
8
5
0
```

En una función se pueden especificar tantos parámetros opcionales como se quiera. Sin embargo, una vez que se indica uno, todos los parámetros a su derecha también deben ser opcionales.

Funciones | Parámetros opcionales

En este ejemplo se utilizan argumentos opcionales para calcular la raíz de un número:

Raíz de un número

```
def fn_raiz(num, raiz=2):
    return num**(1/raiz)

# Programa principal
print(fn_raiz(4))
print(fn_raiz(8))
print(fn_raiz(8,3))
```

Terminal

```
2.0
2.8284271247461903
2.0
```

La función posee dos argumentos. El segundo es opcional. Si no se incluye el parámetro correspondiente en la llamada, se asume que es 2, y se calcula la raíz cuadrada.

Funciones | Parámetros posicionales

Al invocar una función con diferentes argumentos, los valores se asignan a los parámetros en el mismo orden en que se indican. Sin embargo, el orden se puede cambiar si llamamos a la función indicando el nombre de los parámetros:

Ejemplo de uso de una función

```
def potencia(base, exponente = 2):
    return base**exponente

print(potencia(6,1))
print(potencia(exponente = 5, base = 2))
print(potencia(8))
print(potencia(base = 2))
```

Terminal

```
6
32
64
4
```

Funciones | Devolución de valores

Una función puede devolver información para ser utilizada o almacenada en una variable. Se utiliza la palabra clave **return**, que regresa un valor y finaliza la ejecución de la función. Si existe código después del **return**, nunca será ejecutado. Puede haber más de un **return** por función.

Ejemplo de uso de una función con return

```
# La función resta dos valores numéricos.
def restar(num1,num2):
    resta= num1-num2
    return resta # Retorna un valor
# Programa principal
resultado= restar(10,3)
print("El 1er resultado es:", resultado)
print("El 2do resultado es:", restar(10,4))
```

Terminal

```
El 1er resultado es: 7
El 2do resultado es: 6
```

Funciones | Devolución de valores

La sentencia **return** es opcional, y puede devolver, o no, un valor. Es posible que aparezca más de una vez dentro de una misma función.

No se puede utilizar dentro de una función una variable que tenga el mismo nombre que la función.

Función con return que no devuelve valores

```
#Muestra el cuadrado de un número sólo si este es par
def cuadrado_de_par(numero):
    if not numero % 2 == 0:
        return
    else:
        print(numero ** 2)
cuadrado_de_par(8) # 64
cuadrado_de_par(3) # nada, porque no es par
```

Funciones | Devolución de valores

La siguiente función muestra por pantalla si el número es par o no, utilizando dos instrucciones **return**.

Función con dos return

```
def es_par(numero):
    if numero % 2 == 0:
        return True
    else:
        return False
print(es_par(2)) #True
print(es_par(5)) #False
```

Dependiendo de la naturaleza del argumento, se ejecuta uno u otro **return**. La función finaliza luego de devolver el valor. Es decir, aunque haya más de un **return** solo se ejecutará uno de ellos.

Funciones | Devolución de varios valores

Es posible devolver **más de un valor** con una sentencia **return**. La siguiente función *cuadrado_y_cubo()* devuelve el cuadrado y el cubo de un número:

Función que devuelve dos valores

```
# La función devuelve dos valores
def cuadrado_y_cubo(numero):
    return numero ** 2, numero ** 3
# Programa principal
cuadrado, cubo = cuadrado_y_cubo(2)
print("Cuadrado:", cuadrado)
print("Cubo:", cubo)
```

Terminal

```
Cuadrado: 4
Cubo: 8
```

Al invocar la función utilizando el operador de asignación se debe proveer una variable para cada valor que retorne. El orden en que se almacenan es el mismo en el que aparecen en el **return**.

Funciones | Devolución de varios valores

Otra manera de escribir una función que devuelva varios valores es utilizar en el **return** un tipo de dato compuesto, como una **lista** o **tupla**.

Función que devuelve varios valores

```
def tabla_del(numero):
    resultados = [] #creamos la lista
    for i in range(11):
        resultados.append(numero * i)
    return resultados
# Programa principal
res = tabla_del(3)
print(res)
```

Terminal

```
[0, 3, 6, 9, 12, 15,
18, 21, 24, 27, 30]
```

Funciones | Funciones que usan funciones

Es posible combinar todo lo visto hasta ahora, incluso escribir funciones que llamen a otras funciones.

En el ejemplo, *calcular()* devuelve una lista con la tabla solicitada (o la tabla del 1 por defecto). La función *calcular_todas()* utiliza la primera función para mostrar en la terminal las tablas del 0 al 10:

Función que devuelve varios valores

```python
# Genera la tabla del "n"
def calcular(n = 1):
    tabla = []
    for i in range(0,11):
        tabla.append(f"{n}x{i}={n*i}")
    return tabla
# Muestra en terminal todas las tablas
def calcular_todas():
    for i in range(0,11):
        print(f"Tabla del {i}:")
        tabla = calcular(i)
        for j in tabla:
            print(j)
        print("-"*10)
#Programa principal
calcular_todas()
```

Ámbito y ciclo de vida de las variables

En **Python** las variables están definidas dentro de un **ámbito** que determina dónde la variable puede ser utilizada. El **ciclo de vida** de una variable se refiere al tiempo en que una variable permanece en memoria.

Los parámetros y variables definidos dentro de una función tienen cómo ámbito la propia función, y no pueden ser utilizados fuera de ella. Su ciclo de vida dura el tiempo en que está ejecutándose la función. Una vez que termina su ejecución, desaparecen de la memoria.

Es posible definir, dentro de una función, una variable local con el mismo nombre que tiene una declarada en el programa principal. Pero se trata de **otra variable** con el mismo nombre, no comparten su contenido.

Ámbito y ciclo de vida de las variables

Algunos ejemplos sobre el alcance de las variables:

Variables

```
def funcion1():
    a = 3
    print(a)  # 3

funcion1()
print(a)  # ERROR!
```

Variables

```
a = 5
def funcion1():
    a = 3
    print(a)  # 3

funcion1()
print(a)  # 5
```

Variables

```
a = 5
def funcion1():
    global a
    a = 3
    print(a)  # 3

funcion1()
print(a)  # 3
```

global permite utilizar una variable de un ámbito superior dentro de otro.
Importante: Las variables que tienen un tipo de dato compuesto siempre se comportan como variables globales (las listas, por ejemplo).

Ámbito y ciclo de vida de las variables

Alcance y visibilidad de las variables:

- **Variables globales:** Se pueden acceder en cualquier parte del programa.
- **Variables locales:** Sólo son visibles en el ámbito donde fueron declaradas.
- **Variables libres:** Son visibles en una sub-función, pero no desde el programa principal.

Buenas prácticas de programación respecto del uso de variables:

- No utilizar variables globales desde dentro de una función
- No anidar definiciones de funciones
- Si se desea utilizar los valores de una variable del programa principal en una función, se debe pasar como un parámetro.

Funciones anónimas o lambda

A diferencia de las funciones que se definen con la palabra reservada **def**, las **funciones anónimas** se definen con la palabra reservada **lambda**. Su sintaxis es:

Funciones anónimas o lambda

```
lambda parámetros: expresión
```

Dada su sintaxis, estas funciones se suelen llamar “funciones lambda”.

- Pueden tener varios parámetros pero una única expresión.
- Se suelen usar en combinación con otras funciones, generalmente como argumentos de otra función.

Funciones anónimas o lambda

Ejemplo de función lambda

```
cuadrado = lambda x: x ** 2
print(cuadrado(4))
```

Terminal

```
16
```

En este ejemplo, **x** es el parámetro y **x ** 2** la expresión que se evalúa y se devuelve. Esta función no tiene nombre y toda la definición devuelve una función que se asigna al identificador cuadrado. Luego podemos utilizarla como una función normal.

La función **map()** en Python aplica una función a cada uno de los elementos de una lista:

Función map()

```
map(una_funcion, una_lista)
```

Funciones anónimas o lambda con map()

Una función lambda combinada con otras funciones, como **map()**, adquiere más potencia.

Función lambda con map()

```
enteros = [1, 2, 4, 7]
cuadrados = list(map(lambda x : x ** 2, enteros))
print(cuadrados) # [1, 4, 16, 49]
```

En el código anterior se define una lista, y luego se utiliza **map()** para aplicar una **función lambda** a cada uno de sus elementos. Esto da como resultado una nueva lista, que se almacena en *cuadrados*.
La ventaja de usar map() es la simplicidad con la que se puede hacer esto, sin usar bucles ni otra estructura accesoria.

Funciones anónimas o lambda con map()

Un ejemplo más interesante es, en lugar de pasar una lista de valores, pasamos como segundo parámetro una lista de funciones:

Función lambda con map()

```
def cuadrado(x):
    return x ** 2

def cubo(x):
    return x ** 3

enteros = [1, 2, 4, 7]
funciones = [cuadrado, cubo]
for e in enteros:
    valores = list(map(lambda x : x(e), funciones))
    print(valores)
```

Terminal

```
[1, 1]
[4, 8]
[16, 64]
[49, 343]
```

Docstrings | Documentar funciones

Los comentarios delimitados por triples comillas dobles sirven para **documentar funciones** o bloques de código. Son la primera sentencia de cada uno de ellos. +info

Todos los módulos deberían tener **docstrings**, y todas las funciones y clases exportadas por un módulo también deberían tenerlos. En los objetos, veremos que el **docstring** se convierte en el atributo especial __doc__.

Función documentada con docstring

```
def suma(a, b):
    """Esta función devuelve la suma de los parámetros a y b"""
    return a + b
```

Docstrings | Documentar funciones

El siguiente es un ejemplo real de documentación mediante **docstrings**:

Función documentada con docstring

```
def cuad(x):
    """Dado un número x, calcula x²"""
    return x * x # También podríamos haber hecho x ** 2

def modulo_vector(x, y):
    """Calcula el módulo de un vector en 2D.
    Argumentos:
        x: (float|int) coordenada de las abscisas
        y: (float|int) coordenada de las ordenadas
    Devuelve: (float) el módulo del vector
    """
return (x ** 2 + y ** 2) ** 0.5
```

Material extra

Artículos de interés

Material extra:

- Variables y funciones: Paso por valor y referencia, en Hektor Docs
- Guía de funciones de Python con ejemplos, en freeCodeCamp
- Expresiones Lambda en Python, en freeCodeCamp
- La función map(), en Hektor Docs

Videos:

- Uso de parámetros y return, en Code Hive
- Funciones Lambda, en Píldoras Informáticas
- https://www.youtube.com/watch?v=4dkjpHI6vpA, en Píldoras Informáticas

No te olvides de dar el presente

Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.
- Realizar los Ejercicios de repaso.

Todo en el Aula Virtual.

Muchas gracias por tu atención.

Nos vemos pronto