

PROGRAMACIÓN PYTHON

▼ UNIDAD 3 - Funciones, cadenas y listas.

Funciones.

Como todos los lenguajes de programación, en Python es posible definir funciones propias para ejecutar código personalizado. Para definir las es necesario utilizar la palabra reservada `def`. La sintaxis de cualquier función en Python es la siguiente:

```
def nombre_funcion(parámetros):
    código
    return retorno
```

Toda función contiene un **nombre**, **parámetros** y **valor de retorno** (opcional). Al igual que las variables, no es necesario definir el tipo de dato que retornará la función. Al igual que sucede con las estructuras de selección y los ciclos, ya que se va a comenzar con un nuevo bloque de código, es necesario dejar una indentación para que el interprete reconozca el código subsecuente como parte del cuerpo de la función. El valor de retorno es opcional, si la función no devuelve nada, se omite.

Como ejemplo se va a desarrollar una función sin argumentos que nada más salude al usuario:

```
def saludar():
    print("Hola usuario")
```

Una llamada a la función, es igual que en otros lenguajes de programación, escribiendo su nombre y los argumentos correspondientes:

```
saludar()

    Hola usuario
```



Edgar De La Rosa
26 ene 2023



Esta función no recibe parámetros



Edgar De La Rosa
26 ene 2023



Se invoca la función

▼ Pase de argumentos.

Todas las funciones en Python tienen la capacidad de recibir parámetros, que son las variables locales con las que operará la función.

En Python, a las funciones se les puede pasar un argumento, argumentos posicionados, por nombre, predeterminados y de longitud variable.

Parámetro sencillo.

Modificar la función anterior pero agregando como argumento el nombre de a quien saludará:

```
def saludar(nombre):
    print("Hola", nombre)

saludar("Edgar")

    Hola Edgar
```

En otros lenguajes de programación existe la sobrecarga de funciones ¿Recuerdas qué significa ese término? Consiste en tener varias funciones con el mismo nombre cuya única diferencia es la cantidad y opcionalmente el tipo de argumentos que recibe. Por ejemplo en C++ es válido tener:

```
# Código C++
int suma(int n1, int n2);
int suma(float n1, float n2, float n3);
```

En el ejemplo anterior, el compilador se encargará de llamar a la función correspondiente cuando sea llamada en base a los valores pasados.

¿Qué pasa en Python si intentas llamar las dos funciones definidas anteriormente?

```
saludar("Edgar")
saludar()
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-66-d4ef5d23ab3c> in <module>
----> 1 saludar("Edgar")
      2 saludar()

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

SEARCH STACK OVERFLOW

Eso es debido a que Python no soporta la sobrecarga de funciones, sin embargo proporciona los otros medios de pase de argumentos que servirán para asemejar la funcionalidad descrita, con la ventaja de que el código será más simple y sencillo de leer.

▼ **Parámetros por posición.**

Los parámetros por posición son aquellos que en la firma de la función tiene un orden y que al ser invocada, los datos que se le pasen como argumentos se almacenarán en las variables correspondientes de acuerdo a su posición. Además se exige que la función sea llamada exactamente con la misma cantidad de parámetros con la que fue declarada.

```
def resta(n1, n2):
    return n1 - n2

res = resta(2,3)
print(res)
```

-1

▼ **Parámetros por nombre.**

En Python es posible utilizar el nombre del parámetro para asignarle un valor, usando este modo, es válido ignorar la posición de los argumentos:

```
print(resta(n2 = 3, n1 = 2))
```

-1

▼ **Parámetros con valor predeterminado.**

En la firma de una función, es posible declarar un parámetro que tenga un valor de manera predeterminada, esto permitirá que dicho argumento sea opcional al momento de invocar la función, el interprete detectará que no le fue asignado algún valor y por tanto utilizará el valor predeterminado:

```
def resta(m1, n2, n3 = 0):
    return m1 - n2 - n3

print(resta(10, 5))
print(resta(10, 5, 8))
print(resta(n3 = 8, m1 = 4, n2 = 6))
```

5
-3
-10

▼ **Parámetros de longitud variable.**

Python permite que el número de argumentos que se le pase a una función pueda ser variable, es decir que la función pueda recibir n cantidad de datos como argumentos. Para indicar al interprete que el argumento será de longitud variable, es necesario anteponer un asterisco (*) antes del nombre del parámetro. Al hacerlo, automáticamente el interprete convierte el parámetro en una tupla:

```
def resta(*nums):
    print(type(nums))
    res=0
    resta = 0
    for n in nums:
        res -= n

    return res

print(resta())
print(resta(1))
print(resta(2,3))
print(resta(4, 5, 6, 7, 8,91))

<class 'tuple'>
0
<class 'tuple'>
-1
<class 'tuple'>
-5
<class 'tuple'>
-121
```

Si agregamos dos asteriscos (*), es posible pasarle como parámetro los datos en forma de **clave = valor*, o bien, directamente un diccionario. Para obtener los datos, se utilizará `items()`:

```
def resta(**nums):    #esta función con la indicación de ** se esta especificando que es un diccionario
    print(type(nums))

    res = 0
    for clave, valor in nums.items():
        res -= valor

    return

#pasando diccionario desde print
print(resta(entero = 10, flotante = 5.5, imaginario = 3 + 8j))

#declarar el diccionario por fuera que venga por ejemplo de: datagramas, o datos de sensores

calc={"ent":.6, "flot":.56.3, "img":.2.7}
print(resta(**calc)) #Al igual que la firma con ** se le especifica que es un diccionario

<class 'dict'>
None
<class 'dict'>
None
```

▼ Operaciones con cadenas y formateo.

Como se mencionó en lecciones anteriores, las cadenas son secuencias de caracteres y son inmutables. no tiene un tamaño en memoria predeterminado y para declararlas, es necesario colocar el texto entre comillas dobles o sencillas.

Dentro de ellas es posible usar secuencias de escape pero también existen las conocidas como *raw strings* que ignoran dichas secuencias:

```
print("\135 \135")

print(r"\135 \135")

] ]
\135 \135
```



Edgar De La Rosa
26 ene 2023



con * se avisa que serán parámetros variables

Las cadens se pueden concatenar usando el operador "+", sin importar la cantidad:

```
cad1 = "Concatenación de"
cad2 = "cadenas."

print(cad1 + " " + cad2)

Concatenación de cadenas.
```

Con la función `format()` es posible agregar variables a una cadena, acomodándolas por posición o mediante su nombre:

```
cad3 = "El {} es un entero y el {} es de tipo flotante.".format(56, 32.5)
cad4 = "Número complejo: {comp} y una cadena {str}".format(str="de caracteres", comp=34 + 6j)

print(cad3)
print(cad4)

El 56 es un entero y el 32.5 es de tipo flotante.
Número complejo: (34+6j) y una cadena de caracteres
```

Las `f-strings()` permiten utilizar variables dentro de las cadenas como los ejemplos anteriores pero además agrega la posibilidad de realizar operaciones dentro de ellas e incluso invocar a una función:

```
calc = {"ent": 6, "flot": 56.3, "img": 2.7}
print(f'La resta de {calc["ent"]} menos {calc["flot"]} es: {numA - numB}')
print(f'La resta de {calc["ent"]} - {calc["flot"]} - {calc["img"]} es: {resta(**calc)}')

La resta de 6 menos 56.3 es: 50.3
<class 'dict'>
La resta de 6 - 56.3 - 2.7 es: None
```

Con las cadenas es posible realizar:

- Multiplicación de una cadena por un número entero.
- Buscar si una cadena está contenida en otra.
- Convertir una cadena a su valor numérico y viceversa (este último caso sólo funciona con caráctres).
- Obtener el tamaño de una cadena.
- Conversión de clases.
- Indizarlas.
- Segmentar cadenas.

```
print("Hola " * 3)
print("entero" in cad3)
print(chr(234))
print(ord("$"))
print(len(cad4))
```

```
Hola Hola Hola
True
ê
36
51
```

Las cadenas en Python son una clase muy poderosa en Python y la cual dispone de numerosos métodos que incrementan la flexibilidad, uso y operaciones con este tipo de objeto. Para conocer los métodos e incluso más formas de formateo de cadenas, consulta la [documentación oficial](#).

▼ Operaciones con listas.

Algunas propiedades de las listas:

- Son ordenadas.
- Pueden contener distintos tipos de datos.

- Son iterables.
- Son mutables.
- Son dinámicas.

Para acceder a los elementos de una lista, se utilizan los corchetes `[n]`, donde `n` es el índice del elemento al que queremos acceder. El tamaño de la lista va desde 0 hasta `n - 1`.

```
lista = [38, "Hola", 3.1416, 45 + 8j, [1, 2, 3, 4]]

print(lista)
print(lista[2])
print(lista[4])
```

```
[38, 'Hola', 3.1416, (45+8j), [1, 2, 3, 4]]
3.1416
[1, 2, 3, 4]
```

Se puede recorrer la lista del final hacia el principio utilizando números negativos, donde `-1` es el último elemento, `-2`, el penúltimo, etc.:

```
print(lista[-1])
print(lista[-2])
print(lista[-3])

[1, 2, 3, 4]
(45+8j)
3.1416
```

Para modificar un elemento de la lista, tan sólo es necesario asignar el nuevo valor al elemento correspondiente:

```
lista[0] = 69
print(lista)

[69, 'Hola', 3.1416, (45+8j), [1, 2, 3, 4]]
```

Para eliminar un elemento de la lista mediante su índice, se utiliza `del`:

```
del lista[1]

print(lista)

[69, 3.1416, (45+8j), [1, 2, 3, 4]]
```

Si se desea acceder a los elementos que forman parte de una lista que está dentro de una lista, se agregan tantos corchetes con su respectivo índice como listas internas se tengan:

```
print(lista[3][2])

3
```

También con las listas, utilizando la sintaxis de `[n:m]` se pueden obtener sublistas y modificar múltiples valores. Además las listas soportan el operador `+` que añade una lista a la otra y por otro lado, con una lista es posible asignar sus `n` elementos a `n` * cantidad de variables independientes:

```

lista2 = ["Hola", "Hello", "Ohayo"]
lista += lista2

print(lista)
print(lista[1:5])

esp, ing, jap = lista[4:7]

print(esp, ing, jap)

[69, 3.1416, (45+8j), [1, 2, 3, 4], 'Hola', 'Hello', 'Ohayo']
[3.1416, (45+8j), [1, 2, 3, 4], 'Hola']
Hola Hello Ohayo

```

También es posible iterar sobre las listas utilizando ciclos para acceder a sus elementos, acompañarlos con su índice e incluso iterar varias listas al mismo tiempo:

```

for e in lista:
    print(e)

print("-----")

for indice, e in enumerate(lista):
    print(indice, e)

print("-----")

lista3 = ["Español:", "Ingles:", "Japonés:"]
for(e1, e2) in zip(lista2, lista3):
    print(e2, e1)

69
3.1416
(45+8j)
[1, 2, 3, 4]
Hola
Hello
Ohayo
-----
0 69
1 3.1416
2 (45+8j)
3 [1, 2, 3, 4]
4 Hola
5 Hello
6 Ohayo
-----
Español: Hola
Ingles: Hello
Japonés: Ohayo

```

NOTA: La función `zip()` crea un objeto *Zip* el cual es un iterador de tuplas donde cada elemento que se le pase como argumento, es emparejado con el del otro parámetro respetando posición en la que se encuentran.

Además de las operaciones básicas anteriores, las listas tiene sus propios métodos que también realizan operaciones sobre ellas:

- `list.append(x)`
- `list.extend(iterable)`
- `list.insert(i, x)`
- `list.remove(x)`
- `list.pop([i])`
- `list.clear()`
- `list.index(x[, start[, end]])`
- `list.count(x)`
- `list.sort(*, key=None, reverse=False)`
- `list.reverse()`
- `list.copy()`

Para conocer la manera correcta de utilizar los métodos anteriores, revisa la [documentación oficial de Python sobre listas](#).

▼ Ejercicio:

Después de haber revisado la documentación oficial, en el siguiente apartado añade ejemplos (al menos uno por cada método), del uso de dichas funciones, puedes utilizar las listas creadas anteriormente o bien definir las propias, sé claro y documenta el código que realices:

```
#Añade ejemplos del uso de los métodos de las listas.
#Ejemplo list.append(x)
cadena1 = "Edgar"
lista1 = ["Hola: "]
fechaNacimiento = ["Naciste en:", "1978", "Noviembre"]
#agrega una cadena de caracteres a una lista
lista1.append(cadena1)
print(lista1)
#Concatena dos listas
lista1.extend(fechaNacimiento)
print(lista1)
#Inserta en una posición indicada por medio del índice(primer parámetro) un elemento(segundo parámetro)
lista1.insert(3, "15")
print(lista1)
lista2 += lista1
#Elimina un elemento de la lista. Pero el elemento tiene que ser igual.
lista1.remove("Naciste en:")
print(lista1)
#Remueve o elimina el elemento por medio de índice. Este caso se eliminó el primer elemento
lista1.pop(0)
print(lista1)
#Eliminará todos los elementos de la lista. Es decir la lista queda vacía
lista1.clear()
print(lista1)

print("Ahora realizaremos prueba con lista2, por lo tanto lista2 tiene guardado: ", lista2)
#index regresa un valor entero donde esta la cadena a buscar. La cadena a buscar es el primer parámetro. Mientras
#que el segundo parámetro corresponde desde que posición de la lista y el segundo parámetro hasta que posición de
#la lista se busca la cadena a buscar
#Por ejemplo: Se busca el nombre Edgar desde el primer elemento hasta el último elemento de la lista por lo
#tanto regresa el valor de 1 porque Edgar esta en la posición 1 de la lista
lista2.index("Edgar", 0, 5)
#La siguiente instrucción regresa un especie de excepción porque la cadena a buscar esta en la última posición
#Entonces, la última posición en lugar de 4 debe ser 5
#lista2.index("1978", 0, 4)
lista2.index("1978", 0, 5)

lista3 = [1, 2, 5, 6, 1, 1, 8]
#Con el método count: cuenta el número de veces que esta el elemento en la lista. Ese número de veces es mostrado.
#Es decir el método count lo regresa
print("El número de veces que aparece el número 1 en la lista3 es:", lista3.count(1))
lista4 = ["El", "perro", "esta", "contento.", "porque", "el", "perro", "mueve", "la", "cola"]
print("El número de veces que aparece perro en la lista4 es:", lista4.count("perro"))

#Ordena de manera ascendente los elementos de la lista
lista3.sort()
print(lista3)
lista4.sort()
print(lista4)
##Los elementos de la lista los ordena de reversa: es decir descendente
lista3.sort(reverse=True)
print(lista3)
#Modifica la lista en reversa
lista3.reverse()
print(lista3)
lista4.sort(reverse=True)
lista4.reverse
print(lista4)
#EL método copy prepara los elementos de una lista para copiarlos a otra lista. En el ejemplo de la siguiente
#instrucción, copia los elementos de lista3 a lista4
lista4 = lista3.copy()
print(lista4)
```

```
['Hola: ', 'Edgar']
['Hola: ', 'Edgar', 'Naciste en:', '1978', 'Noviembre']
['Hola: ', 'Edgar', 'Naciste en:', '15', '1978', 'Noviembre']
['Hola: ', 'Edgar', '15', '1978', 'Noviembre']
['Edgar', '15', '1978', 'Noviembre']
[]
Ahora realizaremos prueba con lista2, por lo tanto lista2 tiene guardado: ['Hola: ',
El número de veces que aparece el número 1 en la lista3 es: 3
El número de veces que aparece perro en la lista4 es: 2
[1, 1, 1, 2, 5, 6, 8]
['El', 'cola', 'contento.', 'el', 'esta', 'la', 'mueve', 'perro', 'perro', 'porque']
[8, 6, 5, 2, 1, 1, 1]
[1, 1, 1, 2, 5, 6, 8]
['porque', 'perro', 'perro', 'mueve', 'la', 'esta', 'el', 'contento.', 'cola', 'El']
[1, 1, 1, 2, 5, 6, 8]
```

ANOTA Y COMPARTE TUS REFLEXIONES DE LOS TEMAS VISTOS ANTERIORMENTE: Los métodos relacionados con listas son muy similares a los métodos de Haskell. Considero que los métodos para las listas son muy potenciales, por lo tanto, manejar los métodos permiten hacer procesamiento de datos mas rapidamente

Haz doble clic (o ingresa) para editar

✓ 0 s se ejecutó 23:12

