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 definirlas 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 slección y los ciclos, ya que se va a comenzar con un nuevo bloque de código, es necsario dejar una identación para que el interpreta reconozca el código subsecuente como parte del cuerpo de la función. El valor de retorno es opcional, sí la función no devuelve nada, se omite.

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

Edgar De La Rosa
12:06 Hoy
Esta función no recibe parámetros

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

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

Edgar De La Rosa
12:06 Hoy
Se invoca la función

```
saludar()
```

```
    Hola usuario
```

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, predetermnados 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 difinidas anteriormente?

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

```
Hola Edgar
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call
last)
<ipython-input-6-d4ef5d23ab3c> in <module>
      1 saludar("Edgar")
----> 2 saludar()

TypeError: saludar() missing 1 required positional argument: 'nombre'
```

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 covierte 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, diorectamente 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 que recibirá
  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:

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

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

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:

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.

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.

Se puede recorres la lista del final hacía el principio utlizando numeros negativos, donde -1 es el úlitmo elemento, -2, el penúlitmo, etc.:

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

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

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:

También con las listas, utilizando la sintaxis de `[n:m]` se pueden obtener sublistas y modificar multiples 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:

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:

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 utilzar 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 utlilzar las listas creadas aterioremente o bien definir las propias, sé claro y documenta el código que realices:

```
#Añade ejemplos del uso de los métodos de las listas.
```

ANOTA Y COMPARTE TUS REFLEXIONES DE LOS TEMAS VISTOS ANTERIORMENTE:

Haz doble clic (o ingresa) para editar

✓ 0 s se ejecutó 12:51