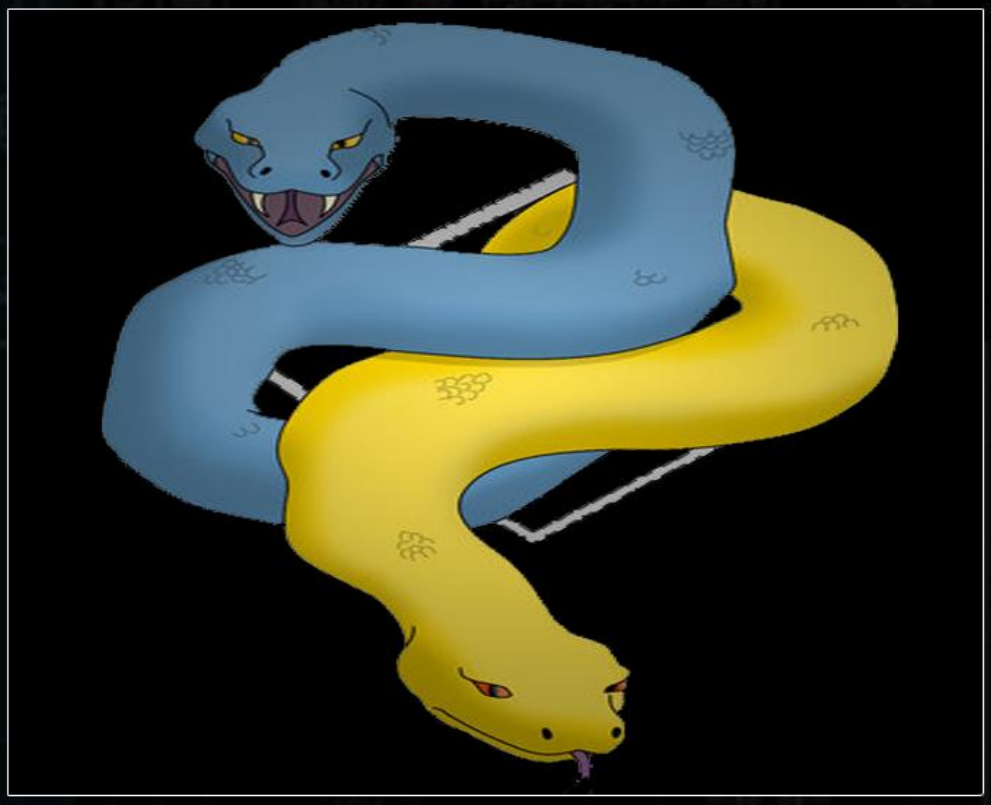
#3

UNDERCODE

TALLER DE PROGRAMACIÓN EN PYTHON



TEMAS

FUNCIONES
CARACTERÍSTICAS
PARTES DE LA FUNCIÓN
RETORNOS DE DATOS
EJERCICIOS
Y MAS...!

TUTOR

WHIZ

Bienvenidos a la tercera entrega de este taller de python.

Luego de haber visto y estudiado muchos de los elementos básicos y fundamentales que cualquier programador debe conocer para poder hacer algo, hoy comenzaremos a tratar con temas más complejos, lo cual significa más herramientas para trabajar. ¡Comencemos!

Funciones

Por fin llegó el momento de hablar de este asunto tan importante. De a poco vamos viendo nuevas formas de escribir nuestro código de manera más ordenada y eficiente. Las funciones, como podrán imaginar, contribuyen a ello.

Pero, ¿qué son, concretamente, las funciones? No son más que segmentos o bloques de código bajo un nombre específico que cumplen ciertas instrucciones predefinidas, retornando siempre un valor determinado. No se asusten si no entienden nada de esta definición. Ahora veremos bien qué significa todo esto.

No sé si lo habían notado antes pero programar no es muy diferente a escribir un papel con instrucciones. Esos archivos con la extensión .py que estuvimos guardando en las entregas anteriores se asemejan mucho a los papeles que le dejamos a la abuela para saber cómo encender la pc o entrar a una página web. Vamos con un ejemplo para que quede claro.

Supongamos que queremos explicarle a la abuela cómo entrar a su cuenta de correo, a su cuenta del banco y a la página de noticias. Por suerte, nuestra abuela es bastante despierta y ya aprendió cómo encender la pc y abrir el navegador. Para lograr que la nona cumpla con su objetivo debemos ser claros y ordenados al momento de explicar los pasos a seguir.

Primero, analicemos algunos aspectos. Vamos a tener que enseñarle, como mínimo, los siguientes puntos:

- Abrir pestañas nuevas.
- Ingresar a una página por medio de la barra de direcciones.
- Ingresar sus datos para acceder a las páginas que requieran credenciales.

A esto sigue la navegación propia de cada página que no viene al caso.

Como vemos, todos los pasos recién mencionados deberán ser repetidos en cada una de las páginas que mencionamos al principio (y en otras, si decidiéramos añadir más páginas).

Bien, ahora que ya tenemos una idea general de cuáles son los pasos necesarios para que la abuela ingrese a sus páginas sin inconvenientes, hagamos una lista con las instrucciones:

Ingresar a cuenta de correo:

1.- Identificar y clickear la barra de direcciones.
2.- Escribir "correo.com" y presionar la tecla "Enter".
3.- Identificar formulario de usuario y escribir "usuario".
4.- Identificar formulario de contraseña y escribir "contraseña".
5.- Identificar botón con la etiqueta "Iniciar sesión" y clickearlo.

Ingresar a cuenta del banco:

1.- Identificar sector de pestañas.
2.- Clickear última pestaña (la que se encuentra vacía o con el signo +).
3.- Identificar y clickear la barra de direcciones.
4.- Escribir "banco.com" y presionar la tecla "Enter".
5.- Identificar formulario de usuario y escribir "usuario".
6.- Identificar formulario de contraseña y escribir "contraseña".
7.- Identificar botón con la etiqueta "Iniciar sesión" y clickearlo.

Ingresar a página de noticias:

1.- Identificar sector de pestañas.
2.- Clickear última pestaña (la que se encuentra vacía o con el signo +).
3.- Identificar y clickear la barra de direcciones.
4.- Escribir "noticias.com" y presionar la tecla "Enter".

¡Muy bien! ¡Hemos hecho un excelente trabajo! Ahora nuestra abuela podrá ingresar a estas tres páginas sin ningún tipo de inconveniente. De todas formas, hay algo que no cierra…

Seguro que alguno de ustedes se ha dado cuenta que hemos repetido unas cuantas instrucciones. No importa ahora, sólo lo hemos hecho unas dos o tres veces. No es mucho. Pero, ¿qué sucedería si tuviéramos que hacer lo mismo con otras páginas? Imagínense el tiempo que perderíamos sólo escribiendo lo mismo una y otra vez. Realmente sería mucho trabajo de más.

Entonces, ¿cómo podemos solucionar esta gran problemática? Probemos reemplazando las notas anteriores por las que siguen:

Abrir nueva pestaña:

1.- Identificar sector de pestañas.

2.- Clickear última pestaña (se encuentra vacía o con el signo +).

Ingresar a página (nombre de página):

1.- Identificar y clickear la barra de direcciones.

2.- Escribir nombre de página y presionar la tecla "Enter".

Ingresar credenciales (usuario, contraseña):

1.- Identificar formulario de usuario y escribir usuario.

2.- Identificar formulario de contraseña y escribir contraseña.

3.- Identificar botón con la etiqueta "Iniciar sesión" y clickearlo.

Perfecto! Ahora sí nuestras notas son 100% funcionales. Lo que hicimos fue crear una acción o *función* por cada nota. Entonces, finalmente nos quedan tres notas: una para explicar cómo abrir una pestaña, otra para ingresar a una página (sirve para cualquier página, siempre y cuando se conozca su dirección) y, finalmente, una última nota para ingresar credenciales (sirve para cualquier página, siempre y cuando se conozcan el usuario y la contraseña). ¿Se entiende lo que hicimos?

Resumo un poco para los que todavía están en el aire. Al principio hicimos una nota para cada página a la que la abuela quería ingresar.

1 Nota = 1 Página

Por cada nota que hicimos tuvimos que desarrollar al menos 2 de los 3 puntos mencionados al comienzo (abrir pestañas, ingresar a la página e ingresar credenciales). Eso nos llevó a repetir instrucciones y, lo que es peor, a tener que volver a repetirlas en caso de querer hacer otra nota.

Para resolver ese problema, lo que hicimos fue crear notas que expliquen los 3 posibles pasos que debemos seguir en cualquier página, es decir, cada nota termina representando una función.

1 Nota = 1 Función

Entonces, nos quedaron 3 notas: una para abrir pestañas, otra para ingresar a una página y una última para ingresar credenciales. De esto surgen varias ventajas:

- No repetimos instrucciones
- Podemos aplicarlo a un número indefinido de páginas
- Es más ordenado
- Si tuviéramos que modificar alguna instrucción por cualquier razón, sólo lo haríamos en la nota correspondiente.

Este último punto es uno de los más importantes. Imagínense, por ejemplo, si repentinamente se cambiara el sistema de ingreso de credenciales por uno más seguro, que requiera acceso por reconocimiento de voz. En el primer sistema de notas (una nota = una página), estaríamos obligados a modificar esas instrucciones en cada una de las notas que tengamos hechas; si tuviéramos 100 páginas (100 notas) deberíamos modificarlas a todas. Qué trabajo, ¿no?

Gracias a nuestro segundo sistema, en cambio, podemos resolverlo fácilmente: buscamos la nota correspondiente al ingreso de credenciales y modificamos las instrucciones necesarias. Es decir, sólo debemos modificar una nota, y listo. Con esto ya podemos ingresar nuestras credenciales de forma correcta en cualquier página que lo solicite.

Características de las funciones

Cuando hablamos de funciones debemos distinguir dos aspectos fundamentales que pueden confundir un poco al principio: la *definición* y la *llamada o invocación* de la función.

Definición de una función

De forma genérica, podemos decir que las definiciones son fundamentales para establecer con claridad y exactitud la naturaleza de una entidad cualquiera.

La programación no es la excepción. Cada elemento integrante de un lenguaje debe ser definido de forma precisa de manera tal que podamos identificarlo y distinguirlo de las demás entidades del lenguaje.

Al definir una función, lo que hacemos es determinar con exactitud cada uno de los siguientes aspectos:

- Nombre de la función.
- Información que podremos brindarle a la función (son los llamados parámetros).
- Conjunto de instrucciones que se ejecutarán al llamar a la función.
- Valor a retornar (ya lo explicaremos).

La palabra clave para indicar que estamos por definir una función es *def*.

Veamos, de forma abstracta, cómo debemos definir una función:

```
1  def nombre(parametro1, parametro2):
2      instrucción1
3      instrucción2
4      ...
```

Analicemos la imagen. En la primera línea encontramos cuatro cosas:

- La palabra clave *def*.
- El *nombre* de la función (esta se llama “nombre”).
- Entre paréntesis indicamos los *parámetros*.
- Como con toda estructura de control, finalizamos con *dos puntos* (:).

Las siguientes líneas corresponden al *bloque de código* que se ejecutará cada vez que llamemos a la función. El bloque de código se encuentra *indentado* y se compone de una o más *instrucciones*.

La indentación, a diferencia de muchos lenguajes, representa en python no sólo un tipo de notación para mejorar la legibilidad del código sino que, además, sirve al intérprete para saber dónde comienza y dónde termina cada uno de los bloques de código. Es decir, permite establecer y entender la estructura del programa.

La indentación puede realizarse con espacios o tabuladores pero nunca con ambos. De combinarse los dos anteriores, nuestro intérprete nos informará que existe un error de indentación:

```
IndentationError: unindent does not match any outer
indentation level
>>>
```

De todas formas, si bien cada programador tiene su propio estilo, por convención, la indentación debería realizarse con *cuatro espacios*, tal y como lo indica uno de los doce mandamientos de python:

“Indenta con cuatro espacios por nivel.

Sin tabuladores.

Si rompes este mandamiento serás lapidado en la plaza del pueblo.”

Llamada a una función

Las llamadas a función no son más que las funciones en acción, es decir, si yo tengo una función determinada, ésta no se ejecutará hasta que yo la llame.

Si prestamos atención, esto nos da una enorme ventaja al momento de programar. Si tenemos un bloque de código de varias líneas que se necesitamos ejecutar en varias ocasiones, lo que

hacemos es convertirlo en función y sólo empleamos una línea por cada ejecución: la correspondiente a la llamada. Como ven, el resultado es un código muchísimo más pequeño y ordenado.

Hagamos un sencillo ejemplo. Vamos a definir dos funciones: “limpiar” y “ensuciar”. Ambas van a trabajar sobre una misma variable llamada “pisoLimpio” la cual puede almacenar uno de los dos valores booleanos True y False.

```
1   # DEFINIMOS
2   def limpiar(pisoLimpio):
3       print " [*] Limpiando el piso"
4       pisoLimpio = True
5       print " [+] OK"
6
7   def ensuciar(pisoLimpio):
8       print " [*] Ensuciando el piso"
9       pisoLimpio = False
10      print " [+] OK"
11
12  pisoLimpio = None
13
14  # LLAMAMOS
15  if not pisoLimpio:
16      limpiar(pisoLimpio)
17  else:
18      ensuciar(pisoLimpio)
```

Si analizamos el código, vemos que definimos dos funciones e hicimos dos llamadas, ¿no? Sin embargo, sólo una de ellas se va a ejecutar. Miren el código, analícenlo e intenten responder cuál de ellas será. En el próximo párrafo explicamos la solución.

Muy bien, primero aclaremos algunos puntos y luego veremos que la respuesta viene sola:

- Las funciones “limpiar” y “ensuciar” modifican el valor de la variable pisoLimpio.

- La variable pisoLimpio es inicializada con el valor None. A esta altura, ya sabemos que None es, en términos booleanos, falso. Por lo tanto, podemos afirmar que, inicialmente, el piso se encuentra sucio.

- Si analizamos el bloque if – else y lo traducimos al español, nos quedaría algo así: “Si el piso no está limpio, entonces llamemos a la función limpiar. De lo contrario (si está limpio), llamemos a la función ensuciar”.

Ahora sí. Sabiendo lo antedicho, estamos en condiciones de practicar el recorrido mismo realizado por el intérprete: ya que el piso no está limpio, entonces llamemos a la función “limpiar”. Fácil, ¿no?

Ahora que ya vimos, en términos generales, cómo definir una función y cómo llamarlas, profundicemos un poco en cada uno de sus aspectos.

Desentrañando las funciones

Volvamos un poco atrás y veamos más a fondo el tema de las definiciones.

Ya vimos que al definir una función nos encontramos con varios elementos. Ya los nombramos y sabemos cómo utilizarlos pero eso no es todo. Veamos cuánto jugo podemos sacarle a cada uno de ellos y avancemos a otro nivel.

¡Comencemos!

1. Nombre. Podemos nombrar o rotular a una función como se nos ocurra, siempre y cuando no sea una palabra clave de python (para más información: https://docs.python.org/2/reference/lexical_analysis.html#keywords) y utilicen los caracteres válidos para los identificadores: las mayúsculas y minúsculas de la A a la Z, el guion bajo y, a excepción del primer carácter, los dígitos del 0 al9.

2. Parámetros. Un parámetro es una *variable que puede ser recibidapor una subrutina.*

Se denomina *subrutina o procedimiento* a un segmento de código separado del bloque principal y que puede ser invocado en cualquier momento desde éste o desde otra subrutina.

La subrutina (en nuestro caso la función) utiliza los argumentos para alterar su comportamiento en tiempo en ejecución.

Habrán notado que primero dije parámetros y, luego, argumentos. Si bien se utilizan muchas veces como sinónimos, debemos saber que no son lo mismo. Un *parámetro* representa un valor que el procedimiento espera que se transfiera cuando es llamado. Un *argumento* representa el valor que se transfiere a un parámetro del procedimiento cuando se llama al procedimiento.

```
1   # Esto es un comentario.
2
3   # Definimos la función miFuncion.
4   def miFuncion(parametro):
5       """
6       Esta función espera recibir un valor
7       (parámetro) para imprimirlo en pantalla.
8       """
9       print parametro
10
11  # Definimos la variable argumento que
12  # contiene un dato de tipo str.
13  argumento = "argumento"
14
15  # Llamamos a la función miFuncion y
16  # le pasamos como argumento la variable
17  # argumento.
18  miFuncion(argumento)
```

Las comillas triples se utilizan en este caso como *cadena de documentación*, cuya función es explicar qué hace la función. Su uso no es obligatorio pero sí se considera buena práctica. En

python, además, la cadena de documentación se encuentra disponible en tiempo de ejecución como atributo de la función. En caso de emplearse, debe ser lo primero que se define en la función.

Volviendo a los parámetros, y como dijimos en la definición, éstos no son obligatorios, quedando su uso determinado por nuestras necesidades al crear una función.

```
# Definimos la función saludar
def saludar():
    """
    Este es el clásico "Hola mundo!" pero
    en una función. Como ven, no requiere
    ningún parámetro para su ejecución.
    """
    print "Hola mundo!"
```

Al ser python un lenguaje de *tipado dinámico*, no debemos preocuparnos por indicar en la definición el tipo de dato que almacenarán los parámetros. Es el mismo intérprete quien se encarga de verificar esto durante la llamada a la función.

```
# Definimos la función imprimirTipo.
def imprimirTipo(dato):
    """
    No hace falta indicar qué tipo de dato
    recibirá el parámetro 'dato'.
    """
    print type(dato)

numero = 1
cadena = "cadena"

# Llamamos a la función imprimirTipo y le
# pasamos como argumento, primero la
# variable de tipo int y luego la de tipo
# str.
imprimirTipo(numero)
imprimirTipo(cadena)
```

En caso de que debamos emplear más de un parámetro en nuestra función, debemos utilizar las *comas*(,) para separarlos.

```
# Definimos la función imprimirDatosPersonales
def imprimirDatosPersonales(nombre, apellido, edad):
    """
    Esta función recibe varios parámetros, separados
    por comas.
    """
    print " [+] Nombre:", nombre
    print " [+] Apellido:", apellido
    print " [+] Edad:", edad
```

Ahora, ¿qué sucedería si al momento de pasar los argumentos alteramos el orden? Veamos:

```
1  # Definimos las variables...
2  nombre = "Marcelo"
3  apellido = "Perez"
4  edad = 32
5
6  # Y las pasamos como argumentos pero sin
7  # respetar el orden establecido en la
8  # definición de la función.
9  imprimirDatosPersonales(edad, nombre, apellido)
```

La salida sería la siguiente:

```
>>> imprimirDatosPersonales(edad, nombre, apellido)
 [+] Nombre: 32
 [+] Apellido: Marcelo
 [+] Edad: Perez
>>>
```

Con esto comprobamos que esta forma de recibir múltiples parámetros está bien, siempre y cuando sepamos que el orden de los argumentos sea el correcto. De lo contrario, deberemos buscar una forma de imprimir bien la información, a pesar de que se altere el orden de los argumentos.

Para eso podemos utilizar las llamadas *keywords* y pasar argumentos como pares de *clave=valor*.

Veamos un ejemplo para que se entienda lo antedicho.

```
>>> imprimirDatosPersonales(edad=32,
                            nombre="Marcelo",
                            apellido="Perez")
 [+] Nombre: Marcelo
 [+] Apellido: Perez
 [+] edad: 32
>>>
```

Otra de las características de los parámetros en python es que podemos asignarles un valor por defecto (*parámetros por omisión*). Esto nos sirve mucho para manejar correctamente aquellos casos en los que no se nos pasen los argumentos requeridos por la función.

```
1  # Función saludar modificada
2  def saludar(nombre="mundo"):
3      """
4      Utilizamos el parámetro por omisión "mundo"
5      en caso de no recibir un argumento en
6      ejecución.
7      """
8      print "Hola", nombre, "!"
```

Ahora, ¿qué sucedería si, al momento de definir la función, no supiéramos cuántos argumentos recibirá nuestra función en tiempo de ejecución?

Esto se resuelve con los denominados *parámetros arbitrarios*. Gracias a esta característica, todos los argumentos recibidos llegarán a la función en forma de *tupla*.

```
1   def imprimeArbitrarios(fijo, *arbitrarios):
2       print " [+] Parámetro fijo(",\
3           type(fijo),\
4           "):",\
5           fijo
6
7       if arbitrarios:
8           print " [+] Parámetros arbitrarios(",\
9               type(arbitrarios),\
10              "):"
11
12          for parametro in arbitrario:
13              print "\t-", parametro
```

Los parámetros arbitrarios *suceden siempre a los fijos* y para indicar su uso se utiliza *un asterisco*(*) inmediatamente antes del nombre que le demos al parámetro.

Veamos a nuestra función en acción:

```
>>> imprimeArbitrarios("arg1",
                       "arg2",
                       "arg3",
                       "arg4")
 [+] Parámetro fijo( <type 'str'> ): arg1
 [+] Parámetros arbitrarios( <type 'tuple'> ):
        - arg2
        - arg3
        - arg4
>>>
```

Pero esto no es todo. Los argumentos arbitrarios, al igual que los fijos, también se pueden pasar en forma de *keywords*. Para ello, debemos utilizar *dos asteriscos* (**) delante del nombre del parámetro. Modifiquemos la función del ejemplo anterior para ejemplificar.

```python
def imprimeArbitrarios(fijo, *arbitrarios, **claves):
    print " [+] Parámetro fijo(",\
        type(fijo),\
        "):",\
        fijo

    if arbitrarios:
        print " [+] Parámetros arbitrarios(",\
            type(arbitrarios),\
            "):"

        for parametro in arbitrarios:
            print "\t-", parametro

    if claves:
        print " [+] Claves arbitrarias(",\
            type(claves),\
            "):"

        for clave in claves:
            valor = claves.get(clave)
            print "\t-", clave, ":", valor
```

Ahora veamos los resultados:

```
>>> imprimeArbitrarios("fijo",
                       "arbitrario 1",
                       "arbitrario 2",
                       nombre="Marcelo",
                       apellido="Perez",
                       edad=32)
 [+] Parámetro fijo( <type 'str'> ): fijo
 [+] Parámetros arbitrarios( <type 'tuple'> ):
        - arbitrario 1
        - arbitrario 2
 [+] Claves arbitrarias( <type 'dict'> ):
        - edad : 32
        - nombre : Marcelo
        - apellido : Perez
>>>
```

Se podría decir que lo que hemos estado haciendo hasta ahora es empaquetar varios argumentos en uno solo: varios argumentos comunes en una tupla y varios de tipo clave en un diccionario.

Python nos permite realizar el paso inverso, es decir, una descomposición de un argumento en varios, en caso de ser necesario (*desempaquetado de datos*).

Al igual que en los casos anteriores, esto se aplica tanto para los argumentos comunes como para los de tipo clave y la metodología es similar: agregando uno o dos asteriscos, respectivamente, al argumento que vayamos a desempaquetar.

Volvamos a la función “imprimirDatosPersonales” para entenderlo.

```python
def imprimirDatosPersonales(nombre, apellido, edad):
    print " [+] Nombre:", nombre
    print " [+] Apellido:", apellido
    print " [+] Edad:", edad

# Almacenamos los datos en una lista
# pero también se puede hacer con tupla:
# datos = ("Marcelo", "Perez", 32)
datos = ["Marcelo", "Perez", 32]
```

Ahora veamos qué sucede al pasar como argumento la variable datos:

```
>>> imprimirDatosPersonales(*datos)
 [+] Nombre: Marcelo
 [+] Apellido: Perez
 [+] Edad: 32
>>>
```

¡Perfecto! Como vemos, funcionó a la perfección. Lo único que tuvimos que hacer fue agregar *un asterisco* al argumento datos.

Para seguir con el ejemplo, intentemos ahora pasar los datos en forma de clave. Para eso nos armaremos un diccionario con los pares clave: valor.

```
1  def imprimirDatosPersonales(nombre, apellido, edad):
2      print " [+] Nombre:", nombre
3      print " [+] Apellido:", apellido
4      print " [+] Edad:", edad
5
6  # Almacenamos los datos en un diccionario.
7  datos = {
8      "edad":32,
9      "nombre":"Marcelo",
10     "apellido":"Perez"
11     }
```

Ahora pasemos la variable datos con *dos asteriscos* y veamos qué sucede.

```
>>> imprimirDatosPersonales(**datos)
 [+] Nombre: Marcelo
 [+] Apellido: Perez
 [+] Edad: 32
>>>
```

Tal como esperábamos, pudimos desempaquetar los argumentos sin ningún tipo de inconveniente. Una vez más, python nos demuestra su inmenso poder.

Retorno de datos

Como dijimos anteriormente, las funciones devuelven siempre, y como mínimo, un valor determinado.

En caso de no especificarse nada, el valor de retorno es *None*.

```
1  def retornarValor():
2      """
3      Aunque no esté explícito,
4      esta funciónva a retornar
5      un None.
6      """
7      print " [+] Retornando valor"
```

Lo expuesto en la cadena de documentación podemos comprobarlo de la siguiente manera:

```
>>> valorRetornado = retornarValor()
 [+] Retornando valor
>>> print valorRetornado
None
>>>
```

Para los que no entendieron nada de lo que pasó, aquí va la explicación: la función “retornarValor” devuelve, con cada ejecución, un valor None. Aprovechando esto, lo que hicimos fue asignar ese valor a la variable “valorRetornado”. Es como decir: “Al valor retornado por la función retornarValor guardémoslo en la variable valorRetornado para luego imprimirlo en pantalla”.

Podríamos conformarnos con lo anterior pero, siendo que estamos en un taller de python, sabemos que estamos lejos de decir basta. Python nos permite retornar cualquier tipo de dato, y no sólo eso, sino que, de hecho, pueden ser múltiples si queremos.

La manera explícita para hacer que una función retorne un dato específico es por medio de la sentencia *return*.

Al emplearla, la función terminará su ejecución, devolviendo el valor que le indiquemos.

```
1  def retornarValorModificado(num):
2      """
3      Utilizamos la sentecia return
4      para devolver o retornar un
5      valor específico
6      """
7      print " [+] Modificando número"
8
9      numModificado = num + 1
10     return numModificado
```

```
>>> num = 10
>>> numModificado = retornarValorModificado(num)
 [+] Modificando número
>>> print numModificado
11
>>>
```

Para retornar varios valores debemos hacerlo separándolos por medio de *comas* (,). A dichos valores los podemos recibir de dos maneras: por medio de una única variable o con tantas variables como valores a retornar existan.

Volvamos al ejemplo de los datos personales para ver cómo funciona todo esto.

```
1  def retornarVariosDatos():
2      return "Marcelo", "Perez", 32
```

Entonces, podemos recibir los datos de dos formas. Veamos ambas:

```
>>> datos = retornarVariosDatos()
>>> print datos
('Marcelo', 'Perez', 32)
>>>
```

```
>>> nombre, apellido, edad = retornarVariosDatos()
>>> print nombre, apellido, edad
Marcelo Perez 32
>>>
```

Es importante recordar que cada vez que utilizamos la sentencia return, estamos dando fin a la función. En ciertas ocasiones, vamos a necesitar devolver varios valores sin detener la función. Para eso podemos utilizar la sentencia *yield*, la cual retorna valores pero no interfiere en la ejecución de la función.

```
1 def retornarSinParar():
2     for i in range(10):
3         yield i
```

El resultado es, independientemente del número de valores que termine devolviendo la función, un objeto de tipo *generator*.

```
>>> datos = retornarSinParar()
>>> print type(datos)
<type 'generator'>
>>>
```

En los próximos talleres, cuando hablemos acerca de programación orientada a objetos(POO), veremos qué es un objeto. Por ahora, quedémonos con la idea de que es un tipo de dato más, tal como lo son las listas, los diccionarios, etcétera.

En cuanto al objeto generator, si analizamos su estructura, veremos que almacena cada uno de los datos en forma de nodo. Es decir, por cada dato se crea un nodo. Los nodos presentan un valor de carga (el que acaba de retornar la función) y un método next que apunta al próximo nodo (ya veremos en POO a qué nos referimos al hablar de métodos).

Sigamos con el ejemplo para entender un poco mejor el párrafo anterior.

```
>>> datos = retornarSinParar()
>>> for dato in datos:
        print dato


0
1
2
3
4
5
6
7
8
9
>>>
```

Esa es la forma más simple de imprimir las cargas de cada nodo.

Vamos a hacerlo ahora de forma “manual” para entender mejor esto de los nodos.

```
>>> datos = retornarSinParar()
>>> datos.next()
0
>>> datos.next()
1
>>> datos.next()
2
>>> datos.next()
3
>>> datos.next()
4
>>> datos.next()
5
>>> datos.next()
6
>>>
```

Como vemos, la instancia del objeto “retornarSinParar” siempre es la misma (“datos”) pero el valor obtenido al ejecutar el método next va cambiando.

No se preocupen si no entienden nada de lo relacionado a POO. Son conceptos sencillos que los trataremos como corresponde en el taller correspondiente.

Por ahora, lo importante es que ya sabemos cómo recibir múltiples datos por medio de la sentencia yield (para no interrumpir la función) e imprimirlos por medio de un sencillo bucle for.

Codeando las notas de la abuela

Muy bien, existen muchos otros conceptos que debemos tratar en algún momento pero ya fue suficiente por hoy.

Dejemos algo de teoría para la próxima entrega y volvamos con la abuelita para poner en práctica todo lo que vimos a lo largo de este taller.

La tarea es transcribir las notas que hicimos para la abuela a puro código python.

Hasta ahora, lo único que ya sabemos es que para cumplir con este objetivo deberemos crear tres funciones: una para abrir pestañas, otra para ingresar a una página y otra para ingresar credenciales.

De más está decir que la creación de dichas funciones no intenta ser un límite sino que, por el contrario, espero que dejen volar su imaginación y codeen todo lo que se les venga a la mente.

Nos vemos en la próxima entrega con las soluciones y mucho más python para disfrutar.