funciones-y-ciclos

January 28, 2020

1 Funciones y Ciclos

1.1 Que es una funcion?

En los ultimos capítulos hemos utilizados funcioens como `print()` y `len()` para mostrar texto y determinar la longitud de una cadena de caracteres (string). Pero qué es una función, realmente?

En esta sección vamos a observar más de cerca la función `len()` a fin de aprender sobre qué es una función y como es ejecutada.

1.1.1 Las funciones son valores

Una de las características más importantes de una función en Python es que las funciones son valores que pueden ser asignadas a una variable.

[1]:
```
print(len)
```

```
<built-in function len>
```

[2]:
```
type(len)
```

[2]:
```
builtin_function_or_method
```

[3]:
```
# podemos asignar cualquier valor a len:
len = 'no soy el len que buscas'
len
```

[3]:
```
'no soy el len que buscas'
```

[4]:
```
type(len)  # cambió el tipo
```

[4]:
```
str
```

La variable `len` es una palabra reservada en Python, y aunque podemos cambiar su valor, es usualmente una mala idea hacerlo, toda vez que puede ocasionar errores por confusión entre la función incorporada `len()` y la nueva variable `len`.

[5]:
```
# descvinculemos la variable len del valor asignado
del len
```

[6]:
```
len
```

[6]:
```
<function len(obj, /)>
```

[7]:
```
# si hubieramos hecho lo mismo con una palabra no reservada, pasaria lo␣
 ↪siguiente:
a = 1
print(a)
```

```
1
```

[8]:
```
del a
print(a)  # no esta definida, porque la borramos
```

```
      ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-8-c342333e421e> in <module>
         1 del a
    ----> 2 print(a)


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

1.1.2 Ejecucion de Funciones en Python

[9]:
```
# escribiendo solo el nombre no ejecuta la funcion
len
```

[9]:
```
<function len(obj, /)>
```

[10]:
```
# debemos llamar (call) la funcion, con parentesis
len()  # error porque len espera un argumento
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-10-d0b4376e20ed> in <module>
```

```
      1 # debemos llamar (call) la funcion, con parentesis
----> 2 len()  # error porque len espera un argumento


TypeError: len() takes exactly one argument (0 given)
```

[11]:
```
help(len)
```

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

Un argumento es un valor que se ingresa en la función como dato de entrada. Algunas funciones pueden ser llamadas sin ningún argumento, y otras permiten una variedad de argumentos. No obstante, `len()` requiere exactamente un (1) argumento.

Cuando una función termina de ejecutarse, **retorna** un valor como dato de salida. El valor retornado usualmente depende del valor de los argumentos ingresados a la funcion, pero no siempre.

El proceso de ejecución de una función puede ser resumido en tres pasos: 1. La función es llamada, y cualquier argumento(s) es pasada a la función como dato de entrada. 2. La función es ejecutada, y alguna acción es realizada con los argumentos proporcionados. 3. La ejecución retorna, y el llamado original de la función es reemplazado con el valor de retorno.

[12]:
```
num_letras = len('cinco')  # 5 asignado a num_letras debido a retorno
```

[13]:
```
# la funcion len() fue llamada, la longitud es calculada y retorna 5
num_letras
```

[13]: 5

1.1.3 Efectos secundarios de las funciones

Aprendimos como hacer el llamado de una function y que la misma retorna un valor cuando terminan de ejecutar. A veces, sin embargo, funciones hacen más que solo retornar un valor. Como cuando una función cambia o afecta algo externo a la misma, y se crea una especie de **efecto secundario**. Esto lo podemos apreciar con la función `print()`.

[14]:
```
valor_retornado = print('Que valor retorno?')
```

```
Que valor retorno?
```

[15]:
```
valor_retornado  # nada
```

Cuando hacemos el llamado a `print('cadena')` con una cadena de caracteres como argumento, esta cadena es mostrada en la consola o ventana interactiva, pero `print()` retorna nada, como pudimos observar anteriormente.

En realidad, cuando se dice que `print()` no retorna algo, es correcto pero impreciso. Todas las funciones retornan algo, aunque no retornen explicitamente algo, toda vez que la ausencia de valor es representada por un valor especial llamado `None`, que indica la ausencia de datos. `None` es de tipo `NoneType`. Es decir, Nada es un valor especial.

[16]:
```
type(valor_retornado)
```

[16]:
```
NoneType
```

Cuando utilizamos `print()`, el texto que se muestra no es el valor retornado por la función, sino un *efecto secundario* de la misma.

Ahora que sabemos que las funcioens son valores, asi como las cadenas (strings) y los numeros, y que sabemos como las funciones son llamadas y ejecutadas, vamos a ver como podemos crear nuestras propias funciones.

1.2 Escribe tu propia funcion

Mientras escribimos programas mas complejos y mas largos, nos encontramos repitiendo las mismas lineas de codigo. Quizas necesitamos calcular la misma formula con valores distintos varias veces.

La solución a este problema, desde un punto de vista pragmatico, si no sabemos sobre la existencia de funciones, es copiar y pegar el mismo codigo por todo el programa.

El problema con eso es que codigo repetitivo se torna en una pesadilla para la persona que le toca mantener el programa. Si se encuentra un error en algun lado que has copiado y pegado en otras partes, pues tendras que cambiarlo en todos los otros lados tambien. Eso no es una buena idea.

Es esta seccion, vamos aprender como definir nuestras propias funciones para evitar la repetición de codigo cuando necesitamos reutilizar el mismo.

1.2.1 La anatomia de una funcion

Las funciones tienen dos (2) partes: 1. La firma de la función, que define su nombre y los datos de entrada o argumentos que espera. 2. El cuerpo de la función, que contiene el codigo que se ejecuta cada vez que la función es utilizada.

[17]:
```
def mutiplica(x, y):  # firma
    # cuerpo
    producto = x * y
    return producto
```

Por supuesto que esta función no es práctica y es solo de uso demostrativo, toda vez que podemos multiplicar con el operador `*`.

La firma de la función La primera linea de la función se conoce como la firma. Siempre empieza con la palabra reservada `def`, que abrevia la palabra *define* o *definir*: `def multiply(x, y):`

La firma tiene cuatro partes: 1. La palabra `def` 2. El nombre de la funcion `multiplica` 3. Los argumentos / parametros / datos de entrada: `(x, y)` 4. Un colon `(:)`

Los nombres pueden llegar a ser variables, asi que las mismas siguen las mismas reglas de nombres. El nombre de una funcion solo puede contener numeros, letras y guion bajo, y no pueden empezar con un numero.

La lista de parametros es una lista de nombres de parametros rodeados por el parentesis de apertura y clausura. `(x, y)` es la lista de parametros para la función `multiplica`.

Un parametro es como una variable, pero no tiene valor. Es como una plantilla para los valores actuales que son proporcionados cuando la función es llamada con uno o mas paremetros / argumentos.

Código en el cuerpo de la función podrá utilizar los parametros como si fuesen valores. Por ejemplo, el cuerpo de la función puede contener una linea de código con la expresión `x * y`.

Como `x` y `y` no tienen valor, `x * y` no tiene valor. Python guarda la expresion como una plantilla y llena los valores que hacen falta cuando la función es ejecutada.

Un función puede tener cualquier numero de parametros, incluyendo ningún parametro.

El cuerpo El cuerpo de una función es compuesto por el código que se ejecuta cada vez que la función es utilizada en el programa.

[18]:
```
def multiplica(x, y):
    # cuerpo
    producto = x * y
    return producto
```

Es una función sencilla. Su cuerpo solo tiene 2 lineas de código.

La primera linea crea una variable denominada `producto` y le asigna un valor a `x * y`. Toda vez que `x` o `y` no tienen valores todavia, esta linea es realmente una plantilla para el valor `producto`, que se le asigna el resultado de la operación cuando la función es ejecutada.

La segunda linea de codigo se conoce como la **declaración de retorno**. Comienza con la palabra `retorno` y termina con la variable `producto`. Cuando Python llega a la declaración de retorno, detiene la ejecución de la función y retorna el valor de `producto`.

Observemos tambien que ambas lineas de codigo en la funcion estan indentadas a la derecha. Esto es de vital importancia. Cada linea indentada abajo de la firma de la funcion, se entiende que es parte del cuerpo de la funcion.

[19]:
```
def multiplica(x, y):
    # cuerpo
    producto = x * y
    return producto
print('Donde estoy?')  # no es parte del cuerpo de la funcion
```

```
Donde estoy?
```

[20]:
```
def multiplica(x, y):
    # cuerpo
    producto = x * y
```

```
    return producto
    print('Donde estoy?')  # ahora si es parte del cuerpo
```

[21]:
```
# la indentacion debe mantener el mismo numero de espacios
def multiplica(x, y):
    producto = x * y
     return producto
```

```
  File "<ipython-input-21-c298a17b4d49>", line 4
    return producto
    ^
IndentationError: unexpected indent
```

[23]:
```
# la indentacion debe mantener el mismo numero de espacios
def multiplica(x, y):
     producto = x * y
    return producto
```

```
  File "<tokenize>", line 4
    return producto
    ^
IndentationError: unindent does not match any outer indentation level
```

[24]:
```
# la recomendacion oficial es indentar con 4 espacios
def multiplica(x, y):
    producto = x * y
    return producto
```

[25]:
```
# la ejecución de la función se detiene despues que retorna
def multiplica(x, y):
    producto = x * y
    return producto
    print('No me puedes ver')  # esta linea no es ejecutada
```

[26]:
```
multiplica(2, 2)
```

[26]: 4

1.2.2 Llamando una funcion definida por el usuario

[27]:
```
# el nombre de la funcion con sus datos de entrada / argumentos
multiplica(2, 4)
```

[27]:
```
8
```

Las funciones definidas por el usuario no son disponibles sino hasta hayan sido definidas. Esto es distinto a las funciones incorporadas como `len()` o `print()`, que por eso se le dicen funcion **incorporada**, porque la funcion ya esta incorporada y disponible automaticamente por Python.

[28]:
```
num = suma(2, 4)  # no lo reconoce
print(num)

def suma(x, y):
    resultado = x + y
    return resultado
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-28-ddba928693ca> in <module>
----> 1 num = suma(2, 4)  # no lo reconoce
      2 print(num)
      3
      4 def suma(x, y):
      5     resultado = x + y


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

[29]:
```
def suma(x, y):
    resultado = x + y
    return resultado

num = suma(2, 4)  # ahora si
print(num)
```

```
6
```

1.2.3 Funciones sin declaracion de retorno

Todas las funciones en Python retornan un valor, aunque ese valor sea `None`. Recordemos que `None` es del tipo `NoneType` y representa ausencia de valor.

No todas las funciones necesitan una declaracion de retorno.

[30]:
```
def saludo(nombre):
    print(f'Hola {nombre}!')  # no hay retorno
```

[31]:
```
saludo('Adriaan')
```

```
Hola Adriaan!
```

[32]:
```
# aunque no tiene retorno, todavia retorna un valor
retorno = saludo('Adriaan')  # si no esperabas que saludara, has presenciado un␣
↪efecto secundario
```

```
Hola Adriaan!
```

[33]:
```
print(retorno)
```

```
None
```

1.2.4 Documentando tus funciones

[34]:
```
# podemos obtener ayuda con una opcion utilizando help()
help(len)
```

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

[35]:
```
# que pasa si pedimos ayuda en la funcion suma
help(suma)  # no hay info
```

```
Help on function suma in module __main__:

suma(x, y)
```

Para documentar nuestras funciones, se utiliza un string de triple citas ubicado en la primera linea del cuerpo de la funcion. A esto se le llama **docstring** o cadena de caracteres de documentacion. Docstrings son utilizados para documentar que hace una funcion que parametros espera.

[36]:
```
def suma(a, b):
    '''Retorna el resultado de la suma de a y b'''
    resultado = a + b
    return resultado
```

[37]:
```
help(suma)  # ahora si existe por lo menos una descripcion de la funcion
```

```
Help on function suma in module __main__:

suma(a, b)
    Retorna el resultado de la suma de a y b
```

1.2.5 Ejercicios

1. Esribe una función llamada `cubo()` que toma un parametro numero y retorna el valor de ese numero al poder de tres (3). Prueba que funciona con distintos numeros.
2. Escribe una funcion llamada `greet()` que toma un parametro string llamado `nombre` y muestra el texto `'Hola <nombre>!'`, donde `<nombre>` es reemplazado con el valor del parametro `nombre`, sin < y >.

1.3 Reto: Convierte Temperaturas

Escribe un script llamado `temperatura.py` que define dos (2) funciones: 1. `convierte_cen_a_far()` que toma un parametro `float` que representa Centigrados y retorna un `float` representando la misma temperatura en Fahrenheit, utilizano la foruma `F = C * 9/5 + 32`. 2. `convierte_far_a_cen()`, que define un parametro `float` que representa los grados Fahrenheit y retorna un `float` que representa la misma temperatura en Celsius, utilizando la siguiente formula: `C = (F - 32) * 5 / 9`.

Ambas funciones deben contener una descripcion de la funcion utilizando las triple citas `""" Esta en una descripcion """`. Debemos poder llamar `help(convierte_cen_a_far()` y obtener la descripcion de la funcion.

El script deberia solicitarle al usuario que ingrese: * una temperatura Fahrenheit, para luego mostrar la temperatura en Centigrados. * una temperatura Centigrados, para luego mostrar la temperatura en Fahrenheit.

Todas las temperaturas convertidas deben ser redondeadas a dos (2) puntos decimales.

Este es un ejemplo:

```
Ingrese una temperatura F: 72
72 grados F = 22.22 grados C

Ingrese una temperatura C: 37
37 grados C = 98.60 grados F
```

1.4 Corre en circulos

Una de las mejores cosas sobre las computadoras es que podemos hacer que hagan la misma cosa indefinidamente.

Un **ciclo** o **loop** es un bloque de codigo que se repite indefinidamente o un numero especifico de veces o hasta que una condición se cumpla.

Existen dos (2) tipos de ciclos en Python: * ciclos `while` * ciclos `for`

1.4.1 El ciclo `while`

Este ciclo repite una sección de código mientras una condición sea verdadera.

Existen dos partes en todo ciclo: 1. La decleracion `while` que empieza con la palabra `while`, seguida por una condicion de prueba, y termina con un colon `:`. 2. El cuerpo del ciclo, que contiene el codigo que se repite en cada paso del ciclo. Cada linea esta indentada 4 espacios.

Cuando un ciclo `while` se ejecuta, Python evalua la condicion de prueba y determina si es veridica o no. Si la condicion de prueba es veridica, entonces el codigo del curepo del ciclo se ejecuta. Si la condicion no es veridica, ese pedazo de codigo en el cuerpo del ciclo es ignorada y el resto del programa es ejecutado.

Si la condicion de prueba es veridica y el cuerpo del ciclo es ejecutado, entonces Python arriba al final del cuerpo del ciclo, y regresa nuevamente a la declaracion `while` y re-evalua la condicion de prueba. Si la condicion todavia es veridica, entonces el cuerpo es ejecutado nuevamente; si es falsa, el cuerpo es ignorado. Este proceso se repite indefinidamente hasta que la condicion de prueba no sea veridica.

[38]:
```
# un ejemplo
n = 1
while n < 5:  # n es menor que 5?
    print(n)
    n = n + 1  # n incrementa +1
```

```
1
2
3
4
```

Paso	Valor de `n`	Condicion de Prueba	Que sucede
1	1	$1 < 5$ (True)	`1` se imprime; `n` incrementado a `2`
2	2	$2 < 5$ (True)	`2` se imprime; `n` incrementado a `2`
3	3	$3 < 5$ (True)	`3` se imprime; `n` incrementado a `2`
4	4	$4 < 5$ (True)	`4` se imprime; `n` incrementado a `2`
5	5	$5 < 5$ (False)	Nada se imprime; el ciclo termina

[ ]:
```
# podemos crear un ciclo infinito
n = 1
while n < 5:
    print(n)
```

Porque se produce un ciclo infinito?

Un ciclo infinito no es inherentemente malo. A veces es justo el tipo de ciclo que necesitamos. Por ejemplo, codigo que interactua con hardware puede utilizar un ciclo infinito para constantemente revisar si un boton o switch ha sido activado.

Si entramos a un programa que ejecuta un ciclo infinito, podemos forzar la salida del mismo presionando `Ctrl + C`, lo cual ocasiona un `KeyboardInterrupt` o interrupción de teclado.

Veamos un ejemplo de un `while` loop (ciclo) en practica.

Uno de los usos del `while` loop es revisar si el dato de entrada proporcionado por el usuario cumple con cierta condicion. Si no cumple con la condicion, el programa le sigue solicitando que ingrese dato de entrada hasta que la condicion sea satisfecha.

[1]:
```
prompt = "Ingresa un numero positivo: "
num = float(input(prompt))
while num <= 0:
    print("Eso no es un numero positivo")
    num = float(input(prompt))
```

```
Ingresa un numero positivo: -1
Eso no es un numero positivo
Ingresa un numero positivo: -5
Eso no es un numero positivo
Ingresa un numero positivo: 5
```

Los `while` loops son excelente para repetir una seccion de codigo mientras una condicion se cumpla. Sin embargo, no son buenos para repetir una seccion de codigo un numero especifico de veces. Para eso tenemos el `for` loop.

Acordemonos que `loop` es igual a `ciclo`.

1.4.2 El `for` loop

Un `for` loop ejecuta una seccion de codigo una vez por cada elemento en una coleccion de elementos. El numero de veces que el codigo es ejecutado es determinado por el numero de elementos en la coleccion.

Como su contraparte `while`, el `for` loop tiene dos partes principales: 1. La declaracion `for` empieza con la palabra `for`, seguida por la **expresion de membresia**,y termina en un colon `:`. 2. El cuerpo del ciclo, que contiene el codigo a ser ejecutado en cada paso del ciclo, y es indentado a cuatro espacios.

[39]:
```
# imprime cada letra del string "Python" una vez
for letra in 'Python':
    print(letra)
```

```
P
y
t
h
o
n
```

La declaracion `for` es `for letra in 'Python'`. La expresion de membresia es `letra in 'Python'`.

En cada etapa del loop, la variable `letra` es asignada la proxima letra en el string `'Python'`, y luego el valor de la `letra` es impreso.

El loop se ejecuta una vez por cada letra en el string `'Python'`, por tanto el loop se ejecuta seis (6) veces.

Paso	Valor `letra`	Que sucede
1	'P'	`P` se imprime
2	'y'	`y` se imprime
3	't'	`t` se imprime
4	'h'	`h` se imprime
5	'o'	`o` se imprime
6	'n'	`n` se imprime

Para que veamos porque los `for` loops son mejores para ciclar encima de una coleccion de elementos, vamos a rescribir el `for` loop del ejemplo previo como un `while` loop.

Podemos utilizar una variable para guardar el indice del proximo caracter del string. En cada etapa del loop, vamos a imprimir el caracter ubicado en el indice actual y luego vamos a incrementar el indice.

El ciclo se detendra una vez que el valor de la variable asignada al indice es igual a la longitud del string. Como recorderis, los indices empiezan en `0`, por tanto el ultimo indice de 'Python' es 5.

[40]:
```
palabra = "Python"
indice = 0
while indice < len(palabra):
    letra = palabra[indice]
    print(letra)
    indice = indice + 1
```

```
P
y
t
h
o
n
```

Es un poco mas complejo que el equivalente utilizando el `for` loop, cierto?

A veces es util ciclar sobre un rango de numeros. Python tiene una funcion incorporada para eso, llamada `range()`, que produce un rango de numeros. Por ejemplo, `range(3)` retorna un rango de numeros enteros empezando en `0` hasta, pero no incluyendo, `3`. Es decir, `range(3)` es el rango de numeros `0`, `1` y `2`.

Podemos utilizar `range(n)` donde `n` es cualquier numero positivo, para ejecutar un ciclo / loop exactamente `n` veces.

[41]:
```
for n in range(3):
    print('Python')
```

```
Python
Python
```

Python

[42]:
```
# tambien podemos indicar el numero de partida
for n in range(10, 20):
    print(n)
```

```
10
11
12
13
14
15
16
17
18
19
```

Veamos un ejemplo un poco más práctico.

Todos hemos pasado por ese momento con los amigos o familia en el restaurante donde toca pagar la cuenta y hay que sacar la cuenta.

[43]:
```
monto = float(input('Ingresa un monto: '))
for num_de_personas in range(2, 6):
    print(f'{num_de_personas} personas: ${monto / num_de_personas:.2f} cada␣
 ↪una')
```

```
Ingresa un monto: 57.65
2 personas: $28.82 cada una
3 personas: $19.22 cada una
4 personas: $14.41 cada una
5 personas: $11.53 cada una
```

1.4.3 Ciclos anidados (nested loops)

Siempre que hemos indentado el codigo correctamente, podemos poner ciclos adentro de otros ciclos.

[9]:
```
for n in range(1, 4):
    for j in range(4, 7):
        print(f"n = {n} y j = {j}")
```

```
n = 1 y j = 4
n = 1 y j = 5
n = 1 y j = 6
n = 2 y j = 4
n = 2 y j = 5
n = 2 y j = 6
n = 3 y j = 4
n = 3 y j = 5
n = 3 y j = 6
```

Cuando Python entra al cuerpo del primer ciclo, la variable `n` es asignada el valor `1`. Luego el cuerpo del segundo ciclo es ejecutado y `j` es asignada el valor `4`. Lo primera que se imprime es `n = 1 y j = 4`.

Despues de ejecutar la funcion `print()`, Python regresa el ciclo anidado o interior (inner loop), hace la asignacion de `j` a `5`, se imprime `n = 1 y j = 5`. Python no sale del ciclo exterior porque el ciclo interior, que esta adentro del cuerpo del ciclo exterior, no ha terminado de ejecutar.

Luego, `j` es asignado el valor `6` y Python imprime `n = 1 y j = 6`. En este punto, el ciclo interior / anidado termina de ejecutar y el control regresa al ciclo exterior. La variable `n` es asignada el valor `2`, y el ciclo interno se ejecuta una segunda vez. Es decir, `j` se le asigna el valor `4` y se imprime `n = 2 y j = 4`.

Los dos ciclos se siguen ejecutando de la misma manera, es decir, entramos al ciclo exterior, luego al ciclo interior, nos quedamos ciclando ahi hasta que termine de ejecutar, luego regresamos al ciclo externo, y repetimos el proceso hasta que ambos ciclos (interno y externo) terminen de ejecutar.

Es importante mencionar que el uso de ciclos anidados es bastante comun, sin embargo, la anidacion de ciclos incrementa la complejidad del codigo, toda vez que el numero de pasos requeridos para que se termine de ejecutar incrementa dramaticamente. Para ser mas precisos incrementa `n**2` donde `n` representa el numero de elementos en el ciclo externo, por lo cual se dice que el tiempo de complejidad es quadratico relativo al numero de elementos en una coleccion.

Por tanto, la anidacion de ciclos es algo que debemos utilizar solo si es estrictamente necesario o si el numero de elementos en la coleccion sobre el cual se construye el ciclo externo o interno no es considerablemente grande.

Imaginense tener 10 elementos en el ciclo externo y 10 elementos en el ciclo interno, eso es ejecutar 100 pasos. Ya se pueden imaginar si incrementamos el numero a 20, aunque es solo el doble de 10, el numero pasos requeridos se quadriplica (`20**2 = 400`). Esta es la razon que la complejidad de tiempo es quadratica.

Los ciclos son herramientas poderosas. Hacen uso de una de las grandes ventajas que nos proporcionan las computadoras: la habilidad de repetir la misma operacion varias veces.

1.4.4 Ejercicios

1. Escribe un `for` loop que imprime los numeros enteros de `2` hasta `10` en una nueva linea, utilizando la funcion incorporadad `range()`.
2. Utiliza el `while` loop para imprimir los numeros de `2` hasta `10`. Consejo: debemos crear un numero primero.
3. Escribe una funcion llamada `dobles()` que toma un numero como argumento / parametro y dobla este numero. Luego utiliza esta funcion `dobles()` en un ciclo para doblar el numero `2` tres (3) veces, mostrando el resultado en cada linea separada. Aqui hay un ejemplo:

```
4
8
16
```

1.5 Reto: Calcula tus inversiones

En este reto, escribiremos un programa llamado `inversion.py` que le da seguimiento al monto invertido en un tiempo determinado.

Se hace un deposito inicial, denominado el monto principal. Cada año ese monto incrementa un determinado porcentaje, que se conoce como el retorno anual del la inversion.

Por ejemplo, hacemos un deposito inicial de `$100` con un retorno anual de `5%` incrementa `$5` todos los años. El segundo año, el retorno anual sobre el nuevo monto de `$105` es `$5.25`.

Escribe una funcion llamada inversion con tres parametros: * El monto principal * El retorno anual * El numero de años para calcularlo

La firma de la función podrá verse algo asi: `def inversion(monto, porcentaje, años):`

La función imprime el monto del inversión, redondeado a 2 puntos decimales, al final de cada año por el monto especificado de años.

Por ejemplo, `inversion(100, 0.05, 4)` debería imprimir:

```
año 1: $105.00
año 2: $110.00
año 3: $115.76
año 4: $121.55
```

Para terminar el programa, hay que solicitarle al usuario que ingrese un monto inicial, un porcentaje anual, y un numero de años. Luego hacemos el llamado a `invest()` para que muestre las calculaciones de los valores ingresados por el usuario.

1.6 El Concepto de Alcance

Cualquier conversación sobre funciones y ciclos en Python estaría incompleta sin hacer mención del concepto de **alcance**, que puede ser uno de los conceptos mas dificiles de entender en programacion.

En esta sección vamos a introducir el concepto y al final de la sección entenderán lo que es **alcance** y porque es importante. Además, aprenderemos lo que es la regla *LEGB* (por sus siglas en ingles), utilizada para la resolución de alcance.

1.6.1 Que es Alcance?

Cuando le asignamos un valor a una variable, le estamos dando a ese valor un nombre. Los nombres son únicos. Por ejemplo, no puedes asignarle el mismo nombre a dos numeros distintos.

[44]:
```
x = 2
x
```

[44]: 2

[45]:
```
x = 3
x
```

[45]: 3

No obstante, veamos como podemos asignarle el mismo nombre a 2 distintos valores

[17]:
```
x = 'Hola Mundo'

def func():
    x = 'Hola Panama'
    print(f'Adentro de la funcion, x tiene un valor de: {x}')

func()
print(f'Afuera de la funcion, x tiene un valor de: {x}')
```

```
Adentro de la funcion, x tiene un valor de: Hola Panama
Afuera de la funcion, x tiene un valor de: Hola Mundo
```

Lo que sucede es que la funcion tiene un **alcance** sobre el codigo que existe dentro de su cuerpo, y el codigo afuera de la funcion tiene otro **alcance**. Es decir, podemos nombrar un objeto adentro de `func()` con el mismo nombre de algo afuera de `func()` y Python mantiene las dos separadas, toda vez que el alcance es distinto.

El cuerpo de la funcion tiene algo que se conoce como **alcance local**, con su propia lista de nombres disponibles a la misma. Codigo escrito fuera del cuerpo de la funcion esta en el **alcance global**.

Podemos pensar en alcance como una lista de nombres enlazadas a objetos. Cuando utilizamos en nombre en particular en nuestro codigo, como una variable o el nombre de una funcion, Python revisa el alcance actual y determina si ese nombre existe o no.

1.6.2 Resolucion de alcance

Los alcances tienen jerarquía.

[46]:
```
x = 5  # alcance global
```

[47]:
```
def exterior():  # esto esta en el alcance de cierre de interior()
    y = 3  # esto esta en alcance local de exterior()

    def interior():
        z = x + y  # z esta en alcance local de interior()
        return z

    return interior
```

Cuando ejeutamos un pedazo de codigo, Python primero busca por la variable en el alcance local, si no la encuentra la busca en el alcance de cierre, si no la encuentra tampoco, la busca en el alcance global, y por ultimo si no la encuentra ahi tampoco, la busca en el alcance incorporado, como vimos con las funciones `len()` o `print()` que siempre estan disponibles.

Esta es la jerarquia: 1. Local 2. Cierre 3. Global 4. Incorporada

En ingles sería: Local, Enclosing, Global, Built-in (LEGB), conocido como la regla LEGB. Nosotros las podemos llamar asi, o LCGI.

Un resumen: 1. Local. El alcance actual. Puede ser en el cuerpo de una funcion o afuera de la misma. Siempre representa el alcance en el cual el interpretador Python esta actualmente en. 2. Cierre. El alcance de cierre. Este es el alcance un nivel arriba del alcance local. Si el alcance local es una funcion interna, entonces el alcance de cierre es la funcion externa. Si el alcance esta afuera de una funcion, el alcance de cierre seria el global. 3. Global. El alcance global es el alcance mas alto a nivel de script. Contiene todos los nombres deinidos en el script que no forman parte del cuerpo de una funcion. 4. Incorporado. El alcance incorporado contiene todos los nombres, como los nombres clave (`def`), que estan incorporadas a Python. Funciones como `round()` y `abs()` estan en el alcance incorporado. Cualquier cosa que utilizarias sin haberla definido previamente esta en el alcance incorporado.

1.6.3 Rompamos las reglas

[48]:
```
# cual sera el dato de salida de esta funcion?
total = 0
def agrega_al_total(n):
    total = total + n  # se crea una variable local "total", luego Python no␣
 ↪encuentra "total" despues
agrega_al_total(5)
print(total)
```

```
      ␣
↪---------------------------------------------------------------------------

       UnboundLocalError                         Traceback (most recent call␣
↪last)

       <ipython-input-48-cf2da4340420> in <module>
         3 def agrega_al_total(n):
         4     total = total + n  # se crea una variable local "total", luego␣
↪Python no encuentra "total" despues
   ----> 5 agrega_al_total(5)
         6 print(total)


       <ipython-input-48-cf2da4340420> in agrega_al_total(n)
         2 total = 0
         3 def agrega_al_total(n):
   ----> 4     total = total + n  # se crea una variable local "total", luego␣
↪Python no encuentra "total" despues
         5 agrega_al_total(5)
         6 print(total)


       UnboundLocalError: local variable 'total' referenced before assignment
```

Pensariamos que conforme a la regla LCGI, Python debió haber reconocido que `total` no existe en el alcance local de la funcion y por tanto debió utilizar la variable `total` ubicada en el alcance global, a fin de resolver el nombre. Sin embargo, el script intenta crear una asignacion a la variable `total`, que crea una nueva variable en el alcance local, y luego cuando Python ejecuta el lado derecho de la asignacion encuentra que el nombre `total` en el alcance local no se le ha asignado algo todavia.

Hay que tener cuidado con este tipo de errores, por lo que es mejor utilizar nombres unicos cuando se pueda, sin consideracion del alcance en el cual nos encontramos.

Podemos resolver este problema utilizando la palabra `global`.

[49]:
```
# cual sera el dato de salida de esta funcion?
total = 0
def agrega_al_total(n):
    global total
    total = total + n
agrega_al_total(5)
print(total)
```

```
5
```

Esta vez si logramos lo deseado. Pero porque?

La linea `global` le dice a Python que mire en el alcance global por un nombre `total`. De esta manera, la linea `total = total + n` no crea una nueva variable local.

Aunque esto arregla el script, el uso de la palabra `global` es considerado mala practica en general, salvo que tengas un buen motivo para hacer uso de la misma. Si te encuentras utilizando `global` para arreglar problemas como el anterior, es mejor considerar si existe una mejor manera para arreglar el codigo.

1.7 Resumen

Aprendimos sobre los dos conceptos esenciales en programacion: funciones y ciclos (functions and loops).

Primero aprendidmos como definir nuestras propias funciones, y vimos que las funciones estan hechas de dos partes: 1. La firma de la función, que empieza con la palabra `def` e incluye el nombre de la función con los parametros de la función. 2. El cuerpo de la función, que contiene el código que se ejecuta cuando la función es llamada.

Las funciones nos ayudan a evitar la repetición de código similar en un programar, creando componentes re-utilizables. Esto ayuda a que el código sea mas facil de leer y mantener.

Luego aprendimos sobre los dos tipos de ciclos (loops) en Python: 1. El **while** loop, que repite un bloque de codigo mientras una condición sea verídica 2. El **for** loop que repite un bloque de codigo por cada elemento en una coleccion de elementos.

Finalmente, aprendimos sobre **alcance** y como Python resuelve el alcance utilizando la regla LCGI / LEGB: 1. Local 2. Cierre 3. Global 4. Incorporado

1.8 Prueba de conocimiento

Cuales son las dos piezas fundamentales de toda función en Python? 1. El encabezado de una función y el cuerpo de una función. 2. La cabeza de una función y la cola de una función. 3. La firma de una función y el corpus de una función. 4. La firma de una función y el cuerpo de una función.

Considera la siguiente funcion:

```
def cuadrado(num):
    num_cuadrado = num ** 2
    return num_cuadrado
```

Cual de las siguientes lineas de codigos es la firma de la función? 1. `num_squared = num ** 2` 2. `def cuadrado(num)` 3. `return num_cuadrado` 4. Ninguna

Cual de las siguientes palabras de utilizan para crear un loop en Python? 1. loop 2. for 3. foreach 4. do 5. while

Que numeros produce el siguiente ciclo?

```
for num in range(1, 5):
    print(num)
```

1. 1 2 3 4
2. 1 2 3 4 5
3. 4 3 2 1
4. 0 1 2 3 4

Que representan las siglas LEGB?

1. Life Expectancy Grizzly Bears
2. Local Enclosed Global Built-in
3. Local Extended Global Basic

Cual es el valor de `x` despues de la ejecucion del siguiente codigo:

```
x = 0
for i in range(3):
    x = x + i
```

Cual es el valor de `x` despues de la ejecucion del siguiente codigo:

```
x = 0
while x < 10:
    x = x + 1
```