Estructuras de datos y funciones

Funciones y variables

{desafío} latam_

Utilizar estructuras de datos apropiadas para la elaboración de un algoritmo que resuelve un problema acorde al lenguaje Python.

Codificar un programa utilizando funciones para la reutilización de código acorde al lenguaje Python.

- Unidad 1:
  Introducción a Python
- Unidad 2:
  Sentencias condicionales e iterativas
- Unidad 3:
  Estructuras de datos y funciones

Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Define funciones que utilizan parámetros de entrada y que producen un retorno para resolver un problema.*
- *Explica el alcance de una variable dentro y fuera de una función distinguiendo el concepto de variable local y global.*

{desafío} latam_

¿Para qué son útiles las funciones?

{desafío} latam_

/* Funciones */

Necesidad de las funciones

Hasta ahora, la idea de utilizar funciones no debiera ser un concepto ajeno. Es más, en las unidades anteriores ya hemos llamado funciones. Algunos ejemplo son:

```
print().
len().
input().
```

Estas funciones son propias de Python (**built-in functions**), es decir, ya vienen creadas en el lenguaje. Y aquellas a las que en este momentos estamos refiriéndonos son funciones definidas por el usuario (**user defined functions**), el programador las crea dependiendo de sus necesidades.

Necesidad de las funciones

La sintaxis más básica para definir una función es la siguiente:

```
def nombre_de_la_funcion():
    pass
```

def es la palabra reservada para definir la función, luego va el nombre de la función. Por convención, Python utiliza **snake_case** para los nombres y es una buena práctica utilizar nombres representativos de la operación que representan.

En este caso particular **pass**, es una palabra reservada en Python que indica que la función no hace nada.

Revisemos el siguiente ejemplo

Creando nuestra primera función

Programa que muestra en varias ocasiones un Menú

`programa.py`

Desde el punto de vista funcional, este código no tiene nada incorrecto, ya que cada una de sus partes funcionan de manera correcta y podríamos dejarlo tal cual está, pero desde el punto de vista práctico ya empezamos a notar algunas cosas.

El código es bastante largo aunque no hemos definido las partes del código interesante, de hacerlo, probablemente el código sería aún más largo.

```
# Se importan muchas Librerías
# Código que hace muchas cosas interesantes

# Menú
print('Opciones: ')
print('1) De acuerdo')
print('2) En desacuerdo')
print('3) No me interesa')

# Más código que hace muchas cosas interesantes

# Nuevamente el Menú
print('Opciones: ')
print('1) De acuerdo')
print('2) En desacuerdo')
print('3) No me interesa')

# Otro código que hace muchas cosas interesantes

# Menú por última vez
print('Opciones: ')
print('1) De acuerdo')
print('2) En desacuerdo')
print('3) No me interesa')

# Código final y fin del Programa
```

Creando nuestra primera función

Programa que muestra en varias ocasiones un Menú

Una manera más efectiva de poder escribir este código es definiendo una función. En este caso, nuestra función se llamará **`imprimir_menu()`**

`imprimir_menu()` es una función que condensará todo el código relacionado al menú. Al hacer esto nuestro programa que era extremadamente largo se reduce a lo siguiente:

```
# Se importan muchas Librerías

# definición de funciones
def imprimir_menu():
    print('Opciones: ')
    print('1) De acuerdo')
    print('2) En desacuerdo')
    print('3) No me interesa')

# Código que hace muchas cosas interesantes

imprimir_menu()

# Más código que hace muchas cosas interesantes

imprimir_menu()

# Otro código que hace muchas cosas interesantes

imprimir_menu()

# Código final y fin del Programa
```

Creando nuestra primera función

Programa que muestra en varias ocasiones un Menú

Cada vez que utilizamos ***`imprimir_menu()`*** *sin la palabra def estamos invocando la función, lo que quiere decir que estamos ejecutando el código al interior de la función.*

Por su parte, la invocación de una función se puede realizar solo después de haberla definido, por eso se hace buena práctica definir todas las funciones del usuario al inicio del código.

Si una función no se invoca, el código en su interior nunca será ejecutado.

Creando nuestra primera función

Programa que muestra en varias ocasiones un Menú

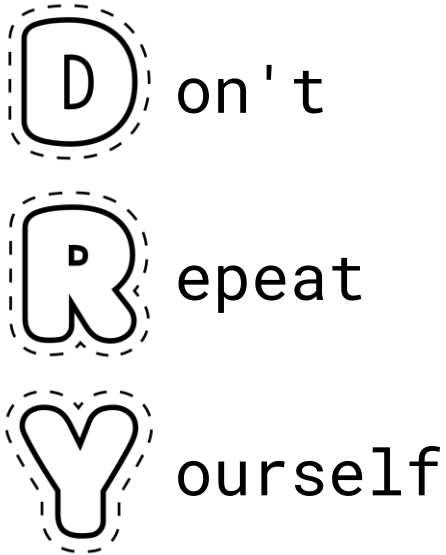
Es muy importante recalcar que para invocar una función es imperativo utilizar paréntesis de la siguiente forma: **`imprimir_menu()`**.

Ahora el código de **programa.py** no solo es más corto, sino que el uso de funciones entrega otras ventajas, por ejemplo, en el caso de querer cambiar el formato del menú, solo tenemos que modificar la función **`imprimir_menu()`**.

Si queremos modificar el menú solo agregamos el punto a 4 líneas y no a todas las veces que aparece el menú.

```
# Se importan muchas Librerías

# definición de funciones
def imprimir_menu():
    print('Opciones: ')
    print('1). De acuerdo')
    print('2). En desacuerdo')
    print('3). No me interesa')

# Código que hace muchas cosas interesantes

imprimir_menu()

# Más código que hace muchas cosas interesantes

imprimir_menu()

# Otro código que hace muchas cosas interesantes

imprimir_menu()

# Código final y fin del Programa
```

Principio DRY



Este es el principio básico en el diseño de Software busca evitar la redundancia de código. Si bien no hay parámetros grabados en piedra, una buena regla podría ser la siguiente:

“Si tengo que copiar y pegar un trozo de código 2 o más veces es muy probable que necesite crear una función.”

Ejercicio guiado

"Mini encuesta"

Mini encuesta

- Una empresa de encuestas se contacta con nosotros para generar una pequeña encuesta de 3 preguntas.

- Las respuestas dadas deben ser:
  - almacenadas en una lista
  - mostradas al final del programa para conocimiento del usuario

Mini encuesta

Solución

Paso 1

Crearemos el programa **`mini_encuesta.py`**, luego 3 preguntas ficticias y las almacenaremos en una lista.

```
preguntas = ['Enunciado Pregunta 1','Enunciado Pregunta 2','Enunciado Pregunta 3']
```

Mini encuesta

Solución

Paso 2

Creamos un mecanismo que permita mostrar cada pregunta con un menú de Opciones. Las opciones para cada pregunta deben ser:

- Opción 1: De Acuerdo
- Opción 2: En desacuerdo
- Opción 3: No me interesa

Deberíamos repetir esto para cada una de nuestras preguntas.

```
# Código para la primera pregunta
print(preguntas[0])
print('Opciones: ')
print('1). De acuerdo')
print('2). En desacuerdo')
print('3). No me interesa')
respuestas.append(input('> '))
```

Mini encuesta

Solución

Paso 3

Finalmente, creamos un mecanismo para mostrar las respuestas entregadas y terminar el programa.

```
print(f'La respuesta a la pregunta 1 fue {respuestas[0]}')
print(f'La respuesta a la pregunta 1 fue {respuestas[0]}')
print(f'La respuesta a la pregunta 1 fue {respuestas[0]}')

print('Muchas gracias por responder la encuesta')
```

Mini encuesta

Solución

```
preguntas = ['Enunciado Pregunta 1', 'Enunciado Pregunta 2','Enunciado Pregunta 3']

respuestas = []

# Hacer preguntas

print(preguntas[0])
print('Opciones: ')
print('1). De acuerdo')
print('2). En desacuerdo')
print('3). No me interesa')
respuestas.append(input('> '))

print(preguntas[1])
print('Opciones: ')
print('1). De acuerdo')
print('2). En desacuerdo')
print('3). No me interesa')
respuestas.append(input('> '))

print(preguntas[2])
print('Opciones: ')
print('1). De acuerdo')
print('2). En desacuerdo')
print('3). No me interesa')
respuestas.append(input('> '))


print(f'La respuesta a la pregunta 1 fue {respuestas[0]}')
print(f'La respuesta a la pregunta 2 fue {respuestas[1]}')
print(f'La respuesta a la pregunta 3 fue {respuestas[2]}')

print('Muchas gracias por responder la encuesta')
```

Mini encuesta

Solución

Si aplicamos el principio DRY para condensar el menú en la función imprimir_menu() y de esa manera evitar y pegar código de manera innecesaria, nuestro código podría entonces reescribirse de la siguiente manera:

```
def imprimir_menu():
    print('Opciones: ')
    print('1). De acuerdo')
    print('2). En desacuerdo')
    print('3). No me interesa')
preguntas = ['Enunciado Pregunta 1', 'Enunciado Pregunta 2','Enunciado Pregunta 3']
respuestas = []
# Hacer preguntas
for p in preguntas:
    print(p)
    imprimir_menu()
    respuestas.append(input('> '))
print(f'La respuesta a la pregunta 1 fue {respuestas[0]}')
print(f'La respuesta a la pregunta 2 fue {respuestas[1]}')
print(f'La respuesta a la pregunta 3 fue {respuestas[2]}')
print('Muchas gracias por responder la encuesta')
```

{desafío} latam_

/* Anatomía de una función */

Parámetros y argumentos

Parámetro: es un elemento que podrá ser utilizado dentro de la función para realizar sus cálculos. Permite crear funciones que son reutilizables en muchos casos.
Argumento: corresponde a los valores que tomará el parámetro para ser utilizado dentro de la función.

```
def 2_elevado_2():
    print(2**2)
def 3_elevado_2():
    print(3**2)
def 4_elevado_2():
    print(4**2)
2_elevado_2()
3_elevado_2()
4_elevado_2()
```

```
4
9
16
```

```
def elevado_2(x):
    print(x**2)

elevado_2(2)
elevado_2(3)
elevado_2(4)
```

```
4
9
16
```

```
def elevar(x,y):
    print(x**y)

elevar(2,2)
elevar(3,3)
elevar(4,2)
```

```
4
27
16
```

```
def elevar(base,
exponente):

print(base**exponente)

elevar(2,2)
elevar(3,3)
elevar(4,2)
```

```
4
27
16
```

Los parámetros pueden ser cualquiera de los tipos de datos vistos anteriormente.

Incluso estos podrían ser estructuras de datos.

Retorno

- En ocasiones, las funciones necesitan devolver un valor que pueda ser utilizado posteriormente, y a este valor se le conoce como retorno.
- **print** muestra en pantalla, pero no es un valor que se pueda utilizar por sí mismo.
- El retorno de una función implica el **término de la función**. Cualquier código que se encuentre después del **return** no será ejecutado.

```
def prueba_return():
    a = "Esta línea se va a imprimir"
    b = "Esta línea no se va a imprimir"
    return a # Punto de salida
    print(b)

print(prueba_return())
```

```
'Esta línea se va a imprimir'
```

Al utilizar return no se debe utilizar print() ya que en este caso el retorno sería un NoneType. Para evitar ese problema el print() se aplica fuera del resultado de la operación.

Múltiples retornos

En algunos casos especiales, podríamos requerir que nuestra función retorne más de un valor de salida.

Supongamos que queremos calcular el cuadrado de un número y al mismo tiempo el cubo de un número.

Una práctica común para este tipo de salidas es desempaquetarlo, es decir, asignar cada uno de los componentes de la tupla a un elemento.

```
def cuadrado_cubo(base):
    cuadrado = base**2
    cubo = base**3
    return cuadrado, cubo

print(cuadrado_cubo(2))
```

```
(4, 8)
```

```
valor_cuadrado, valor_cubo = cuadrado_cubo(2)
print(valor_cuadrado)
print(valor_cubo)
```

```
4
8
```

Ejercicio guiado

"Estandarización"

{desafío} latam_

Estandarización

Calcular la versión estandarizada para el vector [1,2,3,4,5,6]
La función tiene que retornar la media, la desviación estándar y la versión estandarizada

La estandarización es un proceso en el cual un vector (una lista de números) de datos es transformado y llevado a un rango de datos más acotado. Para ello es necesario el cálculo de dos estadísticos que permitirán llevarlo a este estado:

Media: $Media\ (\bar{x}) = \frac{\sum_{i=1}^{N} v_i}{N}$

Desviación Estándar (s): $s=\sqrt{\frac{\sum (x_i-\bar{x})^2}{n-1}}$

Vector V normalizado: $V_{estandarizado} = \frac{v-\bar{x}}{\sigma}$

Estandarización

Solución

Paso 1

Creemos entonces un script llamado **`est.py`**. Dentro de este script crearemos los estadísticos media y desviación estándar como dos funciones.

Paso 2

Para definir la media crearemos la función media de la siguiente manera:

```
def media(lista):
            return sum(lista)/len(lista)
```

Esta función simplemente suma todos los valores al interior de una lista y los divide por el número de elementos que contiene.

Estandarización

Solución

Paso 3

Definir la Desviación Estándar crearemos la función **sdd** de la siguiente manera:

```
import math
def sdd(lista, media):
          diff = [(elemento-media)**2 for elemento in lista]
          return math.sqrt(sum(diff)/(len(lista)-1))
```

- El primero calcula las diferencias entre cada elemento de la lista con la media y eleva dicho resultado al cuadrado. Dado que esto se debe hacer elemento por elemento es que se implementa mediante un List Comprehension.
- Luego se calcula la raíz cuadrada de la división de la suma de las diferencias calculadas en el paso anterior, que a su vez se dividen por el número de elementos menos 1.

Estandarización

Solución

Paso 4

Calcular la media y la desviación estándar de la lista en cuestión.
Calcular la diferencia de cada elemento con su media, resultado que posteriormente se divide por la desviación estándar; y nuevamente debido a que estos cálculos se deben implementar elemento a elemento es que se utiliza una List Comprehension. Nuestra función **resultado()** devolverá los 3 resultados

```
def resultado(lista):
            m = media(lista)
            sd = sdd(lista, m)
            lista_estandarizada = [(valor-m)/sd for valor in lista]
            return m, sd, lista_estandarizada
```

Estandarización

Solución

```python
import math
def media(lista):
    return sum(lista)/len(lista)

def sdd(lista, media):
    diff = [(elemento-media)**2 for elemento in lista]
    return math.sqrt(sum(diff)/(len(lista)-1))

def resultado(lista):
    m = media(lista)
    sd = sdd(lista, m)
    lista_estandarizada = [(valor-m)/sd for valor in lista]
    return m, sd, lista_estandarizada


lista = [1,2,3,4,5,6]

m, desv_st, l_e = resultado(lista)
print('La media es:', m)
print('La Desviación Estándar es:', desv_st)
print('La lista estandarizada es:', l_e)
```

/* Tipos de argumentos y parámetros */

Tipos de argumentos

Como revisamos anteriormente, los **argumentos** son aquellos valores que toman los **parámetros** de una función para poder ejecutar el código al interior de ellas, y al interior, pueden ser de cualquier tipo de dato, pero también pueden corresponder a **estructuras de datos**.

```
precios = {'Notebook': 700000,
           'Teclado': 25000,
           'Mouse': 12000,
           'Monitor': 250000,
           'Escritorio': 135000,
           'Tarjeta de Video': 1500000}

def filtrar(diccionario, umbral):
    filtro = {k:v for k,v in diccionario.items() if v > umbral}
    return filtro

filtrar(precios, 12000)
```

```
{'Notebook': 700000,
 'Teclado': 25000,
 'Monitor': 250000,
 'Escritorio': 135000,
 'Tarjeta de Video': 1500000}
```

Funciones como argumentos

En Python **no estamos restringidos** a utilizar solo valores o estructuras de datos como argumentos, de hecho, podríamos utilizar incluso **funciones**.

En el caso de pasar una función como parámetro SÓLO se debe utilizar el nombre, sin paréntesis. Por ejemplo, generemos una función que entregue el tipo de dato de cada uno de los elementos en una lista.

```
lista_numeros = [1,2,3,4,5]
lista_string = ['a','b','c','d','e']

def sumar_contar_tipos(lista,funcion):
            tipos = [type(elemento) for elemento in
lista]
            opcion = funcion(lista)
            return tipos, opcion

tipo, conteo = sumar_contar_tipos(lista_string, len)
print(tipo)
print(conteo)

tipo, suma = sumar_contar_tipos(lista_numeros, sum)
print(tipo)
print(suma)
```

Parámetros obligatorios

El carácter obligatorio se da porque en caso de no ingresar el argumento respectivo la función fallará.

En particular en funciones que tienen muchos parámetros (estas pueden ser definidas por el usuario o predefinidas en alguna librería de Python) el referenciar el parámetro es de gran utilidad para que otros usuarios puedan entender qué argumentos necesita la función.

```
def extremo_multiplicado(lista,factor):
    minimo = min(lista)
    maximo = max(lista)
    return factor*minimo, factor*maximo

print(extremo_multiplicado(4,[1,2,3,4]))
```

```
print(extremo_multiplicado([1,2,3,4], 4))
# se entregan en orden
print(extremo_multiplicado(factor = 4, lista = [1,2,3,4]))
```

```
(4,16)
```

Parámetros opcionales o por defecto

Existen casos en los cuales queremos utilizar un parámetro solo en ciertas ocasiones o para diferenciar el comportamiento de la función, es decir, el parámetro será opcional, y no siempre habrá que definirlo. Esto quiere decir que cuando no se defina, este tomará un **valor por defecto**.

```
def elevar(base, exponente, redondear = False):
    if redondear:
        valor = round(base**exponente,2)
    else:
        valor = base**exponente

    return valor
```

```
print(elevar(2, 3))
```

```
19.241905543136184
```

```
print(elevar(2, 3, redondear = True))
```

```
19.24
```

{desafío} latam_

*Args y **Kwargs

Los ***args** permiten utilizar tantos parámetros como sean necesarios sin la necesidad de que sean definidos a priori. Por ejemplo, podríamos crear la función suma() que sea la solución genérica.

```
def f(*args):
    return args

output = f(1,[2,3],'hola',{'clave':[4]})
print(type(output))

<class 'tuple'>
```

Los ****kwargs** permitirán un número indeterminado de argumentos, pero estos argumentos deben incluir un nombre ya que el nombre del argumento también pasará a la función.

```
def f(**kwargs):
            return kwargs

output = f(valor = 1, texto = 'hola', lista_nombres
= [4,5,6,7])
print(type(output))
```

{desafío} latam_

Ejercicio guiado

"Expandiendo los diccionarios"

Expandiendo los diccionarios

Crear una función `get_multiple()` que permita extraer distintos valores añadiendo un número cualquiera de claves.

Paso 1

Creemos el script **get_multiple.py**

Paso 2

Definamos una función que permita tomar muchas claves.

Lo más apropiado para esto será utilizar un *arg.

```
def get_multiple(diccionario, *claves):
    pass
```

Expandiendo los diccionarios

Crear una función `get_multiple()` que permita extraer distintos valores añadiendo un número cualquiera de claves.

Paso 3

Iterar por el diccionario chequeando si existen todas esas claves.

```
def get_multiple(diccionario, *claves):
    return {clave: diccionario[clave] for clave in claves}
```

El diccionario resultante sólo llamara las claves que están ingresadas dentro del *arg *claves y contendrán la clave y el valor asociado a esa clave.

Expandiendo los diccionarios

Crear una función `get_multiple()` que permita extraer distintos valores añadiendo un número cualquiera de claves.

Paso 4

Finalmente, tomemos un diccionario cualquiera y probemos si es posible devolver varias de sus claves.

```
diccionario_prueba = {'manzana': 'verde',
                      'platano': 'amarillo',
                      'frutilla': 'roja'}
resultado = get_multiple(diccionario_prueba, 'manzana', 'platano')
print(resultado)
```

```
{'manzana': 'verde', 'platano': 'amarillo'}
```

/* Variables */

Variables

Locales

Son aquellas definidas dentro de un contexto específico, en este caso en una función. Para la función, serán variables locales aquellas que se definan dentro de su bloque, este ambiente local se conoce como scope.

Globales

Cualquier variable que no se defina dentro dentro de un scope local será entonces una variable global, estas pueden ser accedidas desde cualquier parte en Python.

Alcance de variables

La disponibilidad de las variables tienen una cierta jerarquía. Al momento de llamar una variable, Python buscará si existe dicha variable en su propio **scope**; y en caso de existir utilizará dicho valor, en caso contrario escalará a un ambiente superior.

En este caso tenemos dos ambientes, un **ambiente local** dentro de la función **`get_continent()`** y el **ambiente global**, la variable continent ha sido definida en el ambiente global. Por otra parte, la función **`get_continent()`** imprime la variable continent, pero esta variable no está definida en el ambiente local. Por lo tanto, Python escalará al ambiente global donde sí está definida y la utilizará.

```
continent = 'South America'
def get_continent():
    print(continent)
print_continent()
```

```
South America
```

Una variable local no puede salir al ambiente global por sí sola.

La manera de poder utilizar una variable local en el ambiente global es como ya hemos mencionado anteriormente mediante return, la que viene a ser la conexión entre el scope local de una función y el ambiente local.

Modificando variables globales desde un entorno local

Existen ocasiones pero muy específicas en las que podríamos querer modificar una variable global desde un entorno local. Esta práctica es **poco común y no recomendada**, pero aún así es importante entender que es posible.

```
continent = 'South America'

def get_continent():
    global continent
    continent = 'Africa'

get_continent()
print(continent)
```

```
Africa
```

Precauciones con variables globales

Trabajar con variables globales puede resultar un poco confuso, ya que son consideradas una mala práctica, ya que podrían resultar en errores difíciles de entender en nuestro código.

¿Cómo se rompe un programa con variables globales?

Un programa puede tener miles de líneas de código e incorporar varias librerías (programas de terceros). En este aspecto, es sencillo que alguien llame **por error** a una variable global de su código de la misma forma que otra persona la llamó en su librería, y cualquier cambio en ella podría alterar el funcionamiento del código.

Ejercicio guiado

"Loto"

{desafío} latam_

Loto

Crear un programa que permita simular un sorteo del Loto.

Reglas: Se tiene un pool de números del 1 al 41, y uno a uno se tomarán 6 números al azar, y un séptimo que representará el comodín.

Paso 1

Creemos el archivo **`loto.py`**

Paso 2

Creemos el pool de números a escoger. Para evitar escribir los 41 números podemos utilizar un list comprehension de la siguiente manera:

```
pool = [n for n in range(1,42)]
```

Loto

Crear un programa que permita simular un sorteo del Loto.

Reglas: Se tiene un pool de números del 1 al 41, y uno a uno se tomarán 6 números al azar, y un séptimo que representará el comodín.

Paso 3

Escojamos un número al azar. Para ello podemos utilizar random.choice de la librería random:

```
elegido = random.choice(pool)
print("El primer número es", elegido)
```

Loto

Crear un programa que permita simular un sorteo del Loto.

Reglas: Se tiene un pool de números del 1 al 41, y uno a uno se tomarán 6 números al azar, y un séptimo que representará el comodín.

Paso 4

El mismo número que ha sido escogido podría ser removido del pool para la siguiente elección:

```
# sacamos el 2do, pero debemos evitar que se vuelva
# a sacar el número anterior
pool.remove(elegido)
elegido = random.choice(pool)
print("El segundo número es", elegido)
```

Loto

Crear un programa que permita simular un sorteo del Loto.

Reglas: Se tiene un pool de números del 1 al 41, y uno a uno se tomarán 6 números al azar, y un séptimo que representará el comodín.

Paso 5

Finalmente, el séptimo número es el comodín. Por lo tanto debe acompañarse de un mensaje distinto.

```
# El séptimo número es el comodín
pool.remove(elegido)
print(pool)
elegido = random.choice(pool)
print("El Comodín número es", elegido)
```

Loto

Crear un programa que permita simular un sorteo del Loto.

Reglas: Se tiene un pool de números del 1 al 41, y uno a uno se tomarán 6 números al azar, y un séptimo que representará el comodín.

Paso 6

Condensar todo en una función.

```
def sacar_numero(posicion):
    global pool
    elegido = random.choice(pool)
    pool.remove(elegido)
    print(f'El {posicion} es {elegido}')
```

En este caso posicion corresponderá al número que se está extrayendo en el sorteo.

Loto

Crear un programa que permita simular un sorteo del Loto.

Reglas: Se tiene un pool de números del 1 al 41, y uno a uno se tomarán 6 números al azar, y un séptimo que representará el comodín.

Paso 7

Al ejecutar el programa podemos notar cada uno de los números extraídos y además que el pool de número disminuye en cada iteración. Esto no es necesario y en este caso lo hacemos sólo para asegurarnos que el programa efectivamente haga lo que nos interesa.

```
python loto.py
El primer número es 28
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41]
El segundo número es 38
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 39, 40, 41]
El tercer número es 25
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 39, 40, 41]
El cuarto número es 37
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 39, 40, 41]
El quinto número es 24
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 39, 40, 41]
El sexto número es 16
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 39, 40, 41]
El comodín es 35
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 26, 27, 29, 30, 31, 32, 33, 34, 36, 39, 40, 41]
Sorteo Terminado
```

Próxima sesión...

- *Desafío evaluado.*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales