«Talento Tech»

Desarrollo de Videojuegos con Unity

CLASE 4

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 04 | Funciones

Temario:

- Funciones(parámetros, Argumentos y retorno),
- Casteo.
- Usando Datos Externos.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Funciones básicas.

En esta clase vamos a conversar acerca de la programación, veremos un concepto fundamental que nos va a servir para hacer nuestros juegos: **las funciones**.
Vamos a crear un script nuevo, lo vamos a llamar “Clase4” y vamos a echar un vistazo a lo primero que aparece cuando creamos un script:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Clase4 : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
    }
    // Update is called once per frame
    void Update()
    {
    }
}
```

Acá podemos ver dos funciones: ***Start()*** y ***Update()***, el primero ejecuta código en el primer frame desde que el objeto aparece y el segundo ejecuta código una vez por frame (cuadro).

¿Qué es una función?

Las funciones **son bloques de código** que podemos colocar en los programas para hacer una tarea específica.

Declarando una función

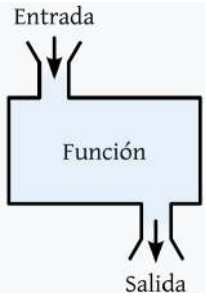
Para usar una función **primero tenemos que declararla**, al igual que con las variables, no podemos usar cosas sin crearlas primero. Declarar en este caso lo podríamos usar como sinónimo de crear.

```
public class Clase2 : MonoBehaviour
{
    int vida = 0;
    void CambiarVida(int a){

        vida += a;
    }
}
```

No te asustes si no entendés, vamos a repasar el código paso por paso para que podamos entender las distintas partes que conforman una función.

Punto 1: Las funciones como dijimos, son bloques de código que cumplen una función, y tienen un output y un input, es decir, pueden recibir información y devolver otra.



El “void” (vacío en inglés), que declaramos al principio, es el tipo de valor de salida que va a tener la función, es decir, que este no va a devolver un valor, pero podría hacerlo si quisiéramos asignando otro tipo de variable.

Punto 2: El nombre de nuestra función. Esta parte es muy importante, ya que el nombre que elijamos para nuestra función tiene que ser una abstracción del código que contenga adentro. Si vamos a hacer una función para saltar, lo ideal es que la función se llame “Saltar” y no “DinosaurioRosa”. Por una cuestión de convención, noten que las funciones tienen la primera letra en mayúscula y no usamos espacios, separamos las palabras con otras mayúsculas.

Punto 3: Las funciones siempre terminan con paréntesis, como Start() y Update(), como dijimos en el **punto 1**, las funciones tienen entrada y salida; los paréntesis representan la entrada, y pueden recibir información o no. La información que recibe se llama **parámetros**, en este caso va a recibir un parámetro, vamos a recibir un “int” que se va a llamar “a” (podría ser cualquier nombre). Cuando utilicemos la función se mostrará cómo ingresamos información a nuestra función.

Punto 4: Nuestra función va a tomar “a” y lo va a sumar a una variable que creamos con anterioridad, que es un número entero (int) y se va a llamar “vida”. Como no le asignamos ningún valor: “vida” es igual a cero.

Utilizando nuestra función

Ya tenemos nuestra función declarada, pero esto **no** significa que **va a ejecutarse sola**, tenemos que llamar la función **dentro de un contexto**. Con contexto nos referimos a en qué momento del juego correríamos esta función; en este caso concreto, lo ideal sería que la vida cambie en función de lo que esté sucediendo en nuestro juego. Si nos atacan o nos curamos podríamos usar esta función para actualizar la vida de nuestro personaje.

Pero no nos adelantemos, vamos a implementar nuestro código al principio de nuestro juego. Para eso existe la función ***Start()*, la cual vamos a utilizar para que la vida tenga un x valor al comienzo**.

Vamos a hacer lo siguiente:

```
int vida = 0;

 private void Start() {
    CambiarVida(100);
 }
 void CambiarVida(int a){

    vida += a;
 }
```

Dentro de las llaves escribimos el nombre de la función que acabamos de declarar.
Nótese que para llamar una función, sólo hay que escribir su nombre, y entre paréntesis le tenemos que dar un **valor de entrada**. El que le otorgamos será *"a"* y se le va a sumar a *"vida"*. Si le pasamos un negativo se va a restar (porque si sumamos un número negativo, se resta).

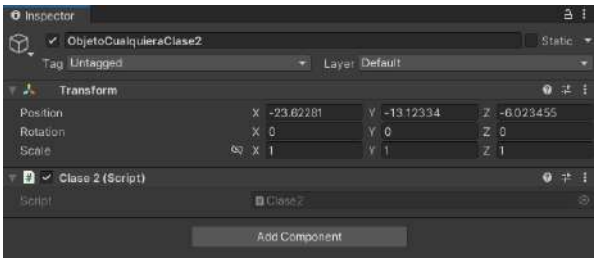
Buenos Aires aprende
Agencia de Habilidades para el Futuro

Pasaremos a agregar un ***Print()*** y vamos a **asignarle nuestro código** a un **objeto** en escena para que se ejecute el código:

```
int vida = 0;

  private void Start() {
      CambiarVida(100);
      print("Vida Actual:" + vida);
  }
  void CambiarVida(int a){

      vida += a;
  }
```



Ahora cuando le demos play a nuestro proyecto. **Vamos a poder ver en la ventana Consola el valor de la variable *"vida"***.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Organizando nuestro código

Ya tenemos lo básico de funciones. Con esto, pasaremos a mover nuestro código de lugar para que no quede amontonado en el Update(), y en este, solo aparezcan los llamados a las funciones que necesitaremos.

Código actual:

```
private int movimientoHorizontal;
    private int movimientoVertical;
    private Vector2 mov;
    [SerializeField] private float speed;
    private float ogSpeed;
    private float multSpeed;
    [SerializeField] private float valMultSpeed = 1.5f;
    private Rigidbody2D rb;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        //Guardamos la velocidad original
        ogSpeed = speed;
            //creamos la velocidad aumentada multiplicando un valor por
        //nuestra velocidad y guardandolo
        multSpeed = speed * valMultSpeed;
    }
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

```
void Update()
  {
     if (Input.GetKeyDown(KeyCode.LeftShift))//Sprint
     { speed = multSpeed; }
     else if (Input.GetKeyUp(KeyCode.LeftShift))
     { speed = ogSpeed; }



     if (Input.GetKey(KeyCode.D))
     { movimientoHorizontal = 1; }
     else if (Input.GetKey(KeyCode.A))
     { movimientoHorizontal = (-1); }
     else { movimientoHorizontal = 0; }//puesto en el else

     if (Input.GetKey(KeyCode.W))
     { movimientoVertical = 1; }
     else if (Input.GetKey(KeyCode.S))
     { movimientoVertical = (-1); }
     else { movimientoVertical = 0; } //puesto en el else

     mov = new Vector2(movimientoHorizontal, movimientoVertical);
     mov = mov.normalized;
  }
 private void FixedUpdate()
  {
     rb.velocity = mov * speed * Time.fixedDeltaTime;
  }
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Explicación de lo nuevo: Veran que si bien este código es MUY parecido al de la clase pasada, tendremos comentarios marcando elementos nuevos.

- se le agregó algo nuevo, un "**Sprint**": La actividad de la clase pasada era generar un código de Sprint, si lo hicieron, tendrán una función similar a esta.

- Se movieron las instrucciones "**MovimientoHorizontal = 0;**" y "**MovimientoVertical = 0;**" a los distintos **else**, en vez de estar al principio. *¿Pueden descubrir el motivo?*

Una vez puestos al día con el código, lo que haremos, será crear funciones para dividir las tareas. Por ejemplo, crearemos 2 funciones para las direcciones llamadas **MovH** y **MovV**:

```
private void MovH(int a) {
    if (Input.GetKey(KeyCode.D))
    { movimientoHorizontal = 1; }
    else if (Input.GetKey(KeyCode.A))
    { movimientoHorizontal = (-1); }
    else { movimientoHorizontal = 0; }//puesto en el else
  }
  private void MovV(int a){
    if (Input.GetKey(KeyCode.W))
    { movimientoVertical = 1; }
    else if (Input.GetKey(KeyCode.S))
    { movimientoVertical = (-1); }
    else { movimientoVertical = 0; } //puesto en el else
  }
```

Como ven, le agregamos un **parámetro** a cada una y utilizamos el mismo en vez de colocar el 1 directamente.

Seguiremos moviendo algunas cosas al **FixedUpdate()**:

```
private void FixedUpdate(){
    mov = new Vector2(movimientoHorizontal, movimientoVertical);
    mov = mov.normalized;
    rb.velocity = mov * speed * Time.fixedDeltaTime;
 }
```

Recordemos que el **FixedUpdate()** maneja unos tiempos distintos, leyendo una cantidad de veces distinta al **Update()**, por eso debemos de tener cuidado sobre QUÉ ponemos dentro y no simplemente usarlo como reemplazo del **Update()**.

Y terminaremos creando una Función propia para el Sprint que nos pida el valor por el que queremos cambiar nuestra velocidad:

```
private void Sprint(float multSpeed) {
    if (Input.GetKeyDown(KeyCode.LeftShift))//Sprint
    { speed = multSpeed; }
    else if (Input.GetKeyUp(KeyCode.LeftShift))
    { speed = ogSpeed; }
 }
```

Notaran que en este caso se nombró al parámetro de la misma manera que una variable que ya teníamos. *¿Eso* significa que *son la misma variable?*

*Por último, llamaremos a cada función dentro del **Update()**.*

```
void Update() {
    Sprint(multSpeed);
    MovH(1);
    MovV(1);
  }
```

*Dato **IMPORTANTE** a tener en cuenta. El orden de llamado importa y se vuelve más relevante a medida que seguimos agregando instrucciones. Un código llamado de manera desordenada puede causar Bugs difíciles de encontrar.*

Casteo.

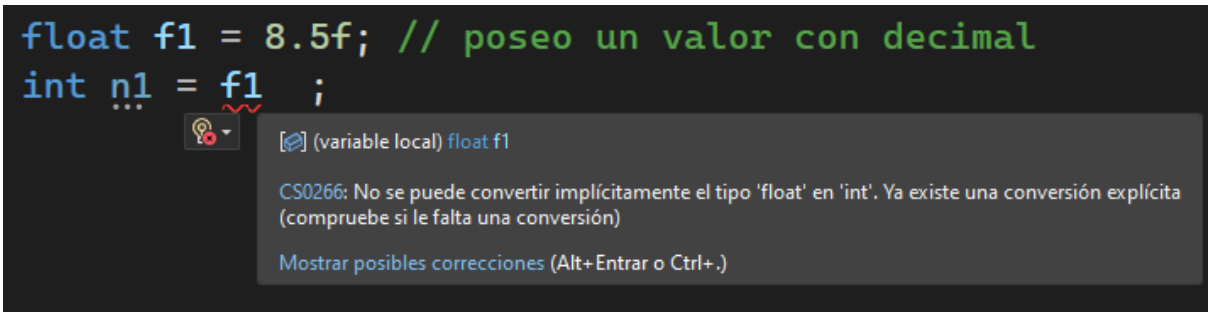
¿Qué es castear y para qué sirve?

¿Qué pasaría si tenemos un dato float, pero necesitamos que sea int?
Para estas situaciones podemos contar con lo que se llama “cast” o “casteo”. Esto nos permite convertir o cambiar el tipo de datos de una variable o expresión a otro tipo de datos. Consiste en colocar un paréntesis con el tipo de dato deseado, antes del dato a castear:

```
(int)f1
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Prestemos atención a que no podemos guardar un **float** dentro de un **int**:

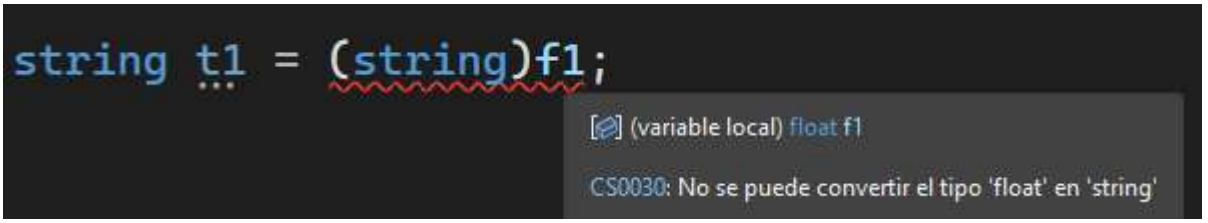


Pero si a mi variable **f1** la casteo como **int** el error desaparece:

```
float f1 = 9.9f; // poseo un valor con decimal
int n1 = (int)f1; // guardo el valor casteandolo como un int
```

Tengan en cuenta que al usar el valor o mostrarlos, este siempre se **guardará redondeado para abajo**. Es decir, no importa que mi variable **f1** valga **8.9**, el resultado guardado en **n1** será **8**.

¿Y si tuviéramos que convertir un número a un string?
Notarán que si probamos lo mismo, utilizando el tipo de dato **string**, esto nos tirara un error:



Buenos Aires aprende
Agencia de Habilidades para el Futuro

Dentro de la programación existen varias formas de convertir tipos de datos. En este caso, si utilizamos la **conversión Explícita** que es lo que denominamos como **casteo / casting / cast**, no nos lo permite.
Para este caso en particular nos ayudaremos con una función propia de **C#** llamada **ToString()**, que se utiliza de la siguiente manera:

```
// Uso la funcion ToString() para convertirlo en string
string t1 = f1.ToString();
```

Ejemplo práctico de casteo: Timer

Para poner en práctica lo que aprendimos, crearemos un **Timer**, siendo siempre útil en nuestros juegos.

Paso 1: Crearemos nuestra variable que almacenará el dato del tiempo y un bool que servirá para chequear nuestra situación.

```
public float tiempoRestante;//variable de tiempo
private bool timerOn; //Un bool que usaremos para chequear
void Start() {
    timerOn = true;
}
```

Paso 2:

Crearemos un **if** que de condicion pregunte por si el estado de **bool**

```
private void TimerCheck()
  {
     if (timerOn)
     {

     }
  }
```

¿Por qué usaremos el bool de esta manera? En su gran mayoría al ser una de las variables más ligeras al contener sólo **"true" o "false"**, el uso que se le da es de **"revisar" o "chequear"** situaciones para que el programa sepa si debe realizar un número de instrucciones o no.

Paso 3:

Seguiremos colocando una instrucción que vaya restando a nuestra variable tiempo, si es que nuestra variable es mayor a 0.

```
if (tiempoRestante > 0)
        {
           tiempoRestante -= Time.deltaTime;
        }
```

Paso 4:

Pondremos un else, para que cuando nuestro tiempo se acabe, nos avise, asigne el valor 0 a mi variable (por si por algún motivo termina teniendo algún valor negativo o con decimales) y le asignaremos **false** a nuestro **TimerOn**

```
else
        {
            Debug.Log("Se termino el tiempo!");
            tiempoRestante = 0;
            timerOn = false;
        }
```

Paso 5:

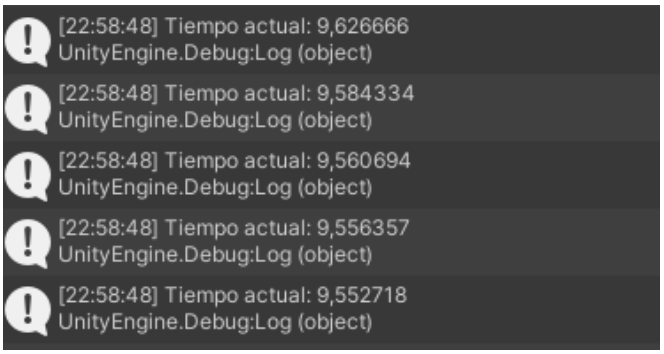
Por último, le agregamos un **Debug.Log** para mostrar los segundos que van pasando. Dejando al código de esta manera:

```
 if (timerOn)
     {

         if (tiempoRestante > 0)
         {
             tiempoRestante -= Time.deltaTime;
         }
         else
         {
             Debug.Log("Se termino el tiempo!");
             tiempoRestante = 0;
             timerOn = false;
         }
```

```
        Debug.Log("Tiempo actual: " + tiempoMostrado);
    }
```

Notaran que al ejecutarlo, la consola arrojará cosas como estas:



Acá es donde entra nuestro **casteo**, usándolo en la variable del Debug.Log o claro, si lo desean podemos crear una variable que directamente guarde el valor convertido.

```
Debug.Log("Tiempo actual: " + (int)tiempoMostrado);
```

O

```
private int tiempoMostrado;
```

```
tiempoMostrado = (int)tiempoRestante;
Debug.Log("Tiempo actual: " + tiempoMostrado);
```

Dependiendo del uso que le daremos al timer, será que nos convenga más una forma u otra.

Al ejecutarlo verán que se envían cientos de mensajes por segundo, pero al menos, ya no contamos con números tan rebuscados. En futuras clases veremos cómo crear una UI para colocar el timer quedando más **“estético”**.

Usando datos externos.

Para usar datos externos a nuestra clase o Script hay MUCHAS formas. En este caso usaremos una manera sencilla, no ideal, pero práctica y sin mucha complejidad. Más adelante nos adentraremos en otros métodos.

Supongamos que, tenemos un enemigo que al tardar en derrotarlo, este se vuelve en un estado “Berserk”, aumentando su daño. Algo muy común en juegos claro, pero ¿Cómo lo hago? ¿Cómo accedo a la información del timer? No puedo crear el timer dentro del Script de mi enemigo si, justamente, maneja el tiempo del juego.

Lo que haremos es crear una variable de referencia al dato que queremos y le asignaremos el valor necesario.
Acá armamos un código sencillo con variables del enemigo para utilizar de ejemplo:

```
public class Clase4Berserk : MonoBehaviour
{
    private int vida = 100;
    private int daño = 10;
    private bool BerserkOn;
```

Pasaremos a crear una variable del **TIPO** de dato de mi timer.

¿Qué significa esto?

Nosotros/as podemos crear variables que, de tipo de dato, tengan una clase que hayamos hecho o propia de Unity/c#, esto nos permitirá acceder a todas las variables, funciones o características que tenga.

```
[SerializeField]
private Clase4Timer timer;
```

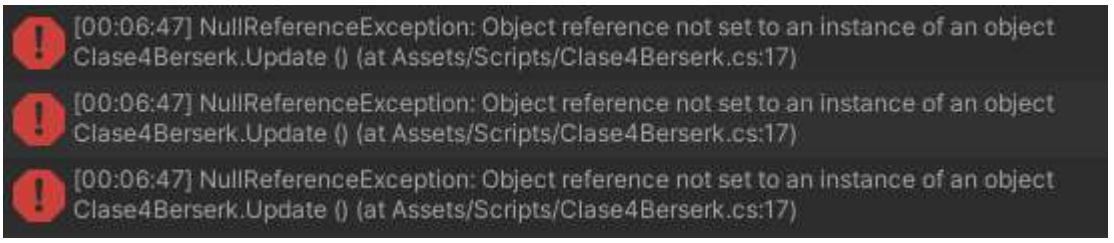
Procederemos a utilizar esto accediendo a la variable "**tiempoRestante**". Armaremos un **if** dentro del **Update()** para decir que, si el tiempo es 0, nuestro personaje entra en modo berserk, aumentando su daño.

```
if (!BerserkOn && timer.tiempoRestante <= 0)
 {
    BerserkOn = true;
    Debug.Log("Modo Salvaje");
    daño *= 2;

 }
```

Fijense que utilizamos el "BerserkOn" para evitar que esto se siga haciendo continuamente una vez que el tiempo llegue a 0;

ATENCIÓN: ¿Que pasó al querer ejecutarlo?

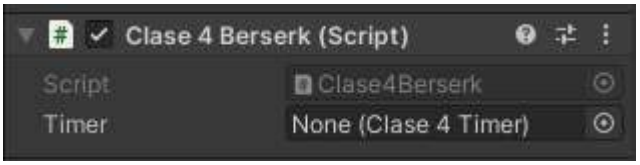


De seguro les aparecieron cientos de errores diciendo lo mismo

Si leemos el error nos dice que una referencia del objeto no está seteada. ¿Qué significa? Que nuestra variable **Timer no fue asignada**, es decir no **tiene ningún valor**, por lo tanto, no sabe a qué objeto nos referimos y no puede acceder al dato que buscamos.

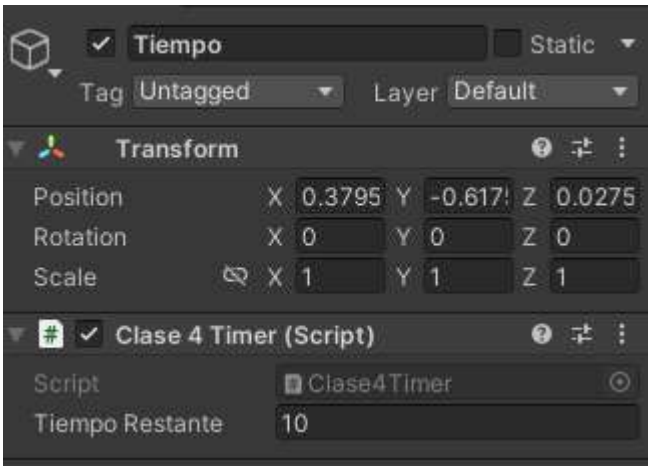
*Pensemoslo asi, nosotros podemos hacer que CADA **GameObject** de la escena tenga el Script **Clase4Timer**. ¿Cómo puede saber el juego a cual de todos nos referimos?*

Si revisamos los componentes de nuestro berserk notaremos lo siguiente:

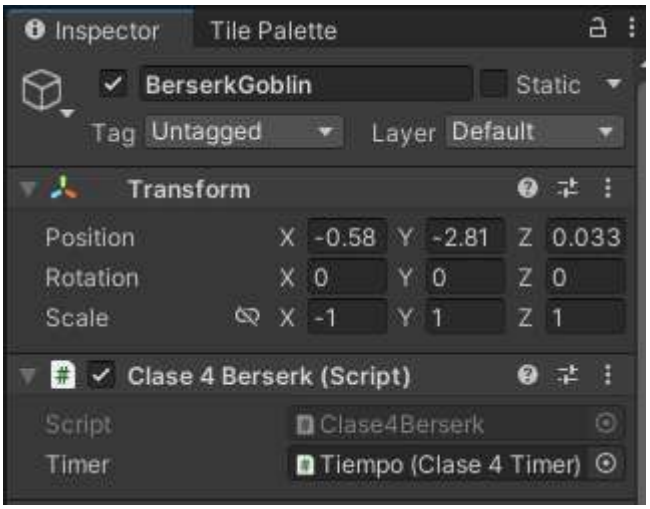


Veremos que la variable que creamos con el **SerializedField** indica que está vacía (“**None**”) y entre paréntesis, nos dirá qué tipo de dato necesita (“**Clase 4 Timer**”).
Así deberemos de buscar al objeto que contenga mi Script “**Clase4Timer**”, es decir, el código del timer que creamos recién.

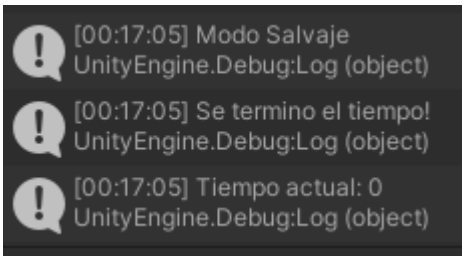
En este caso, creamos un **Empty** llamado **Tiempo** y le asignamos ese Script.



Y para asignar la referencia de la variable, arrastraremos el GameObject desde la **Hierarchy(Jerarquía)** al Script de nuestro Berserk. Quedando de esta manera:



Finalmente, al ejecutarlo, debería de quedar similar a lo siguiente:



TIP: Si les cuesta encontrar el mensaje, reduzcan el tiempo inicial del Timer y/o borren la instrucción de mostrar el tiempo por la consola.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejercicios prácticos:

1. Notarán que tanto el **Timer** como el **Berserk** no están en **funciones**. Organicen los códigos según consideren, creando las respectivas funciones para las instrucciones escritas.

2. Dado que esperamos que hayan empezado con la idea de su proyecto, organicen todo sus Scripts de manera que estén correctamente divididos por las funciones, dependiendo los objetivos que cumplan sus líneas de código.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad