



# Flujo

## Sesión Conceptual 1





# Inicio

{desafío}  
latam\_



15 minutos

- Identificar las herramientas para construir programas en el Lenguaje Ruby.
- Diferenciar la ejecución de programas en Ruby con la terminal y el editor de código para resolver un problema.
- Crear diagramas de flujo y pseudocódigo.
- Construir aplicaciones en Ruby utilizando funciones matemáticas para realizar cálculos.

## Objetivo



# Desarrollo

{desafío}  
latam\_



150 minutos

**/\* Introducción a la programación \*/**

# ¿Qué es programar?

Programar consiste en escribir estas especificaciones paso a paso. Por ejemplo un programa básico podría ser:

Querido `computador`:

- lee un dato del teclado y guárdalo en el espacio1
- lee un dato del teclado y guárdalo en el espacio2
- suma el dato del espacio1 con el del espacio2 y muestra el resultado al usuario

# ¿Qué es un lenguaje de programación?

Un lenguaje de programación es un lenguaje formal que define un conjunto de reglas para escribir instrucciones para un computador.

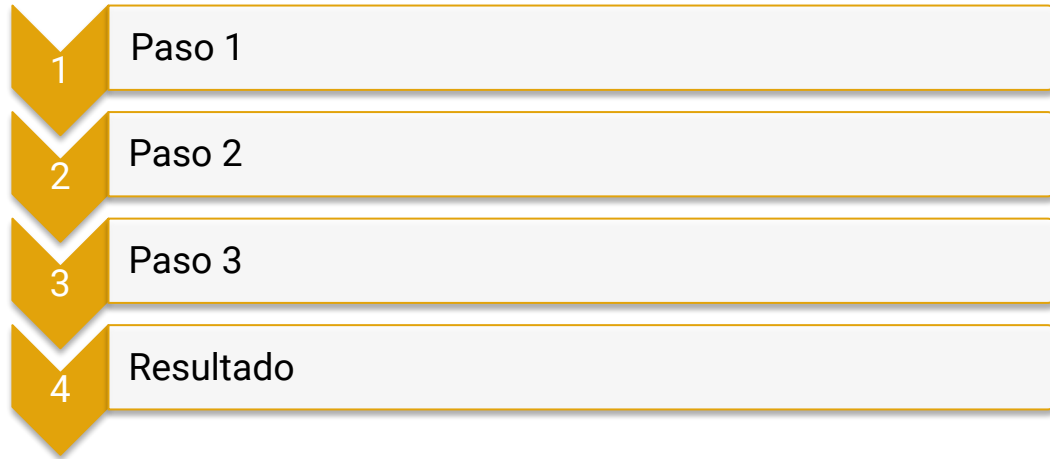
## El trabajo de un programador

El trabajo de un programador no solo consiste en escribir estos códigos, sino también en entender y poder definir cuál es la serie de instrucciones que nos lleva al resultado deseado.



# ¿Qué es un algoritmo?

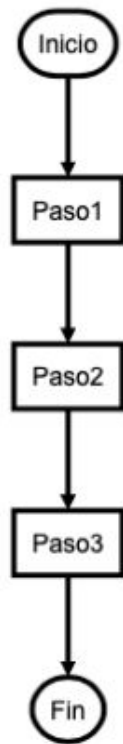
Un algoritmo es una serie finita de pasos para resolver un problema.



## Diagrama de flujo

Un diagrama de flujo es una representación gráfica de un algoritmo, ayuda a visualizarlo en casos complejos.

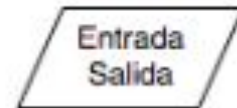
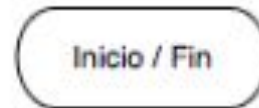
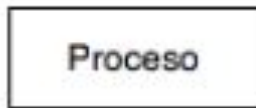
- Los pasos se leen secuencialmente, o sea el paso 3 sucede después del paso 2
- El paso 2 sólo se ejecuta después del paso 1.



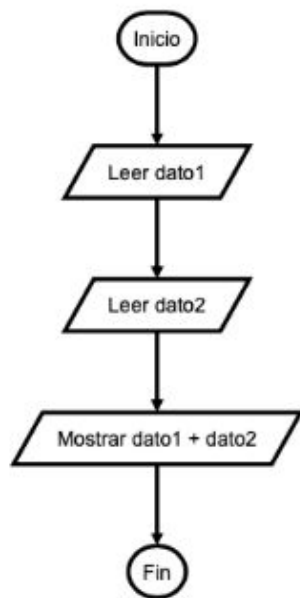
# Símbolos de un diagrama de flujo

Los diagramas de flujo tienen varios símbolos, pero los 4 más utilizados son:

- Inicio y fin del procedimiento.
- Entrada y salida de datos.
- Procesos (instrucciones que ejecuta la máquina).
- Decisiones



# Escribiendo nuestro programa como diagrama de flujo

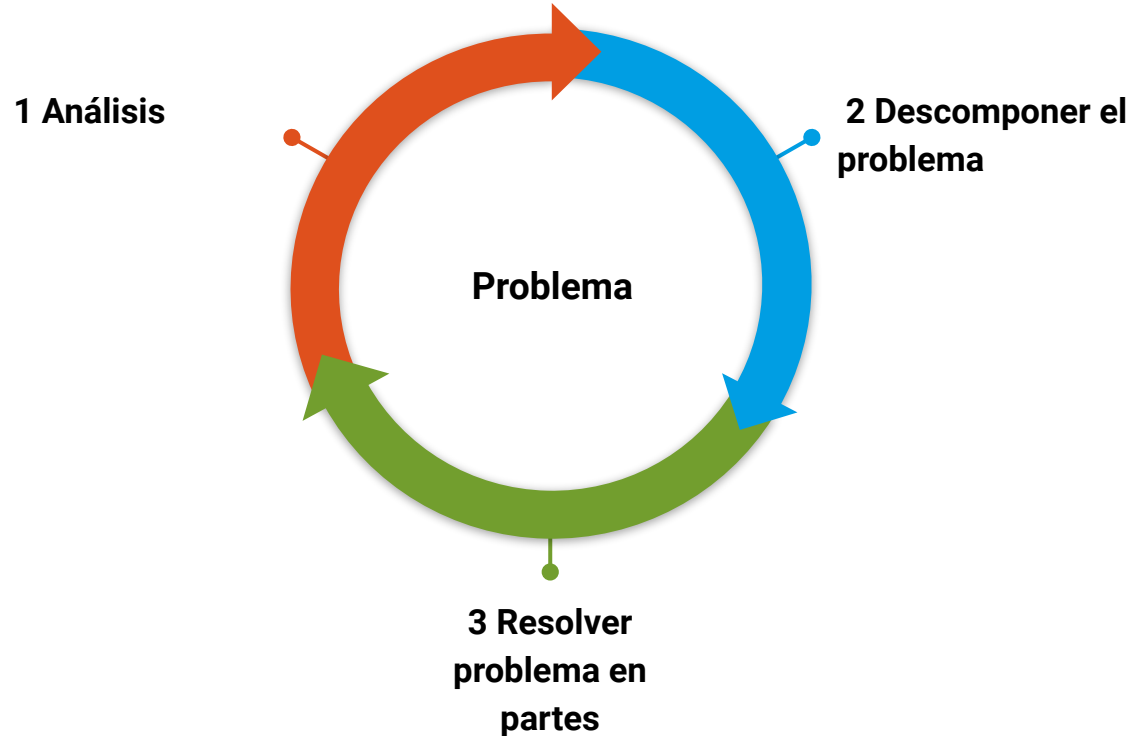


## Pseudocódigo

```
Algoritmo Suma
  Leer dato1
  Leer dato2
  Mostrar dato1 + dato2
FinAlgoritmo
```

- Todo diagrama de flujo puede ser transformado a pseudocódigo y viceversa.
- Todo pseudocódigo o diagrama de flujo puede ser implementado en un lenguaje de programación

# Enfrentándose a un problema



# La importancia de desarrollar el pensamiento lógico

Para desarrollar estas habilidades debemos:

- Dar tiempo necesario para hacer los ejercicios antes de revisar las soluciones.
- Es normal que al intentar resolver los ejercicios por primera vez demore tiempo.
- Tolerancia a la frustración

**/\* Programando con Ruby \*/**

## Formas de trabajar con Ruby

- A través de un programa llamado Ruby Interactivo o IRB.

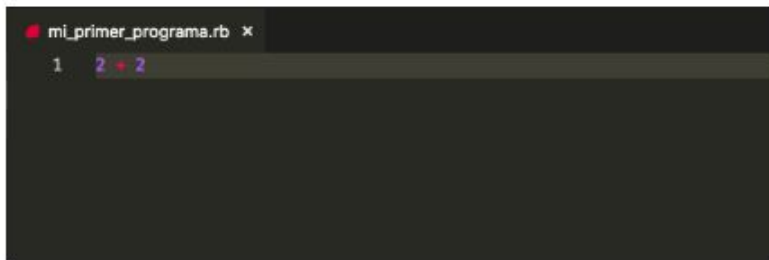
Con IRB podemos escribir código Ruby directamente en nuestro terminal.

```
2.5.1 :001 > 2 + 2
=> 4
2.5.1 :002 > █
```



# Trabajando con Ruby desde el editor de texto

- Con el editor de texto podemos escribir el programa en un archivo .rb y luego ejecutarlo con ruby.



- Ejecutando el programa:

```
ruby nombre_archivo.rb
```

# **`/* Elementos básicos de Ruby */`**

# Comentarios

Los comentarios son líneas ignoradas en la ejecución del código, sirven para dar indicaciones y documentar nuestros programas.

```
# Esta línea es un comentario
# Los comentarios son ignorados
# Pueden ir en una línea sin código.
puts 2 + 2 # 0 puede acompañar una línea de código existente
# puts 4 + 3 Si comentamos al principio todo la línea será
ignorada
```

# Comentarios en múltiples líneas

Existe otra forma de hacer comentarios de largo mayor a una línea, para lograrlo, envolveremos todo el comentario entre un `=begin` y un `=end`.

```
=begin  
Comentario multilínea:  
Ruby  
lo  
ignoraré  
=end  
puts "hola"
```

# Introducción a variables

Podemos entender las variables como contenedores que pueden almacenar valores.

Una variable se compone de:

- Nombre.
- Valor.

## Asignando un valor a una variable

En Ruby podemos asignar un valor a una variable de la siguiente forma:

```
a = 27
```

En este ejemplo estamos asignando el valor numérico `27` a la variable llamada `a`.

# Introducción a tipos de datos

En Ruby y en muchos otros lenguajes de programación- existen diversos tipos de dato.

- Existen los números enteros, los decimales, fechas, horas, arrays, hashes, etc.
- En este capítulo nos enfocaremos en enteros, floats y strings.

## Integers

Los números enteros son números sin decimales que pueden ser positivos o negativos. En inglés se les denomina Integers.

## Floats

Los números de punto flotante pueden tener decimales.

## Strings

A una palabra o frase se le conoce como cadena de caracteres o String.

# Integers

Los números enteros son números sin decimales que pueden ser positivos o negativos. En inglés se les denomina Integers.

```
a = 27
```

```
a = -31
```

# Floats

Los números de punto flotante pueden tener decimales.

```
a = 2.5
```

```
b = 3.1
```



# Strings

A una palabra o frase se le conoce como cadena de caracteres o String. Un String puede estar formado por:

```
a = 'Hola Mundo!' # Una o varias palabras  
b = 'x' # Simplemente un caracter.
```

En ruby las cadenas de caracteres se escriben en comillas dobles o simples, dado que sirve de mecanismo para diferenciar una variable de un string.

# El salto de línea

El salto de línea es un carácter especial que nos permite separar una línea de texto de la siguiente, dentro de un string.

- El carácter es `\n`, indica que el salto de línea.
- `\n` cuando está entre comillas dobles se lee como si fuera un solo carácter.

```
a = "hola\n a\n todos"
print a
# hola
# a
# todos
```

```
puts "\n".length
puts '\n'.length
```

# Operando con variables

- Podemos operar con variables de tipo entero tal como si estuviéramos trabajando en una calculadora.
- Para indicar que queremos el resultado como float expresamos uno de los operandos como tal.

```
a = 5
```

```
b = 2
```

```
puts a + 2
```

```
# 7
```

```
puts b - a
```

```
# 3
```

```
puts a * b
```

```
#20
```

```
puts a / b
```

```
# 2
```

```
# Corrigiendo el resultado pasando a float
```

```
a = 5.0
```

```
b = 2
```

```
puts a / b
```

```
# 2.5
```

# Modificando una variable

```
a = 'HOLA! '  
a = 100  
  
puts a  
# 100  
  
a = 2  
a = a + 1 # => 3  
puts a # 3
```

Una variable puede cambiar de valor por medio de una nueva asignación. También podemos modificar el valor de una variable operando sobre ella misma.

## Operando con variables de tipo String

```
a = 'HOLA'  
b = ' MUNDO'  
puts a + b  
# "HOLA MUNDO"
```

# Concatenando números

Al asignar los valores utilizando comillas simples, Ruby los interpreta como String.

- La acción de 'sumar', o sea unir dos o más Strings se conoce como concatenación, porque enlaza dos cadenas de caracteres

```
a = '7'  
b = '3'  
  
puts a + b  
#"73"
```

# Interpolación

La interpolación es un mecanismo que nos permite introducir una variable (o un dato) dentro un String sin necesidad de concatenarlo.

- La interpolación sólo funciona sobre comillas dobles

```
edad = 30
texto = "tienes #{edad} años"
puts texto
# "tienes 30 años"
```

# La interpolación es más fácil de usar que la concatenación

La interpolación debido a que es más rápida y presenta una sintaxis más amigable para el desarrollador.

```
nombre = 'Carlos'  
apellido = 'Santana'
```

```
# Concatenación
```

```
puts "Mi nombre es " + nombre + " " + apellido  
# "Mi nombre es Carlos Santana"
```

```
# Interpolación
```

```
puts "Mi nombre es #{nombre} #{apellido}"  
# "Mi nombre es Carlos Santana"
```



# Constantes

Existe un tipo especial de variable llamado constantes. Una constante sirve para almacenar un valor que no cambiará a lo largo de nuestro programa.

- La regla para definir una constante es que su nombre debe comenzar con una letra mayúscula.

```
Foo = 2
```

```
# => 2
```

```
Foo = 3
```

```
# (irb):2: warning: already initialized  
constant Foo
```

```
# (irb):1: warning: previous definition of Foo  
was here
```

```
# => 3
```

```
Foo
```

```
# => 3
```

## Las constantes pueden modificarse, pero bajo alerta

Es convención de que las constantes se escriben completamente con mayúsculas, de esta forma si vemos algo como:

```
NO_ME_CAMBIES = 1
```

# Salida de datos

Hay varias formas de mostrar datos en pantalla, las dos más utilizadas son puts y print. Ambas son muy similares:

- puts agrega un salto de línea.
- print no lo agrega

```
puts "con puts:"  
puts "hola"  
puts "a"  
puts "todos"
```

```
#con puts:  
#hola  
#a  
#todos
```

```
print "con print:"  
print "hola"  
print "a"  
print "todos"
```

```
#con print:holaatodos
```

# Entrada de datos

## Ingresando datos con gets

Antes de mostrar el valor, la consola quedará bloqueada hasta que ingresemos una secuencia de caracteres y presionemos la tecla enter.

```
2.5.1 :001 > nombre = gets  
Carlos Santana  
=> "Carlos Santana\n"  
2.5.1 :002 > █
```

## Removiendo el salto de línea con chomp

```
nombre = gets.chomp  
puts nombre # "Carlos Santana"
```

Para solucionar este problema podemos utilizar `.chomp`:

## Ingresando datos por consola

```
primero = ARGV[0].to_i  
segundo = ARGV[1].to_i  
tercero = ARGV[2].to_i  
puts primero + segundo + tercero
```

Esto nos permitirá llamar al programa y entregarle todos los datos al mismo tiempo.

- Crear un programa que solicite tu nombre y luego te salude.
- Concatenar números por error.
- Transformar los datos.
- Ingresando un dato al azar

A vertical line separates the white left side from the orange right side. It features a series of white symbols: a closing curly brace '}', an '@' symbol, another closing curly brace '}', a question mark '?', and a stylized 'i' or 'l' character.

**Creando un  
programa con  
lo aprendido**

# Errores sintácticos vs errores semánticos

- Los errores sintácticos son muy similares a los errores gramaticales, donde escribimos algo que contraviene las reglas de cómo debe ser escrito.
- Los errores semánticos son muy distintos, el programa funcionará de forma normal pero el resultado será distinto al esperado. Los errores semánticos corresponden a errores de implementación o de diseño.

Por ejemplo: Asignar una variable es de izquierda a derecha,  $a = 5$  mientras lo contrario sería un error sintáctico  $5 = a$ .



**/\* Introducción a objetos \*/**

# Objetos y métodos

El método + nos permite sumar dos enteros, o concatenar dos strings, pero no nos permite sumar un string con un entero, porque no tiene sentido la idea de 'sumar un texto'.

- Las operaciones que realizamos sobre ellos reciben el nombre de métodos.
- Los métodos nos permiten operar con los objetos bajo ciertas restricciones.

```
2 + 2 # => 4
'hola' + ' a todos' # 'hola a todos'
2 + 'hola' # TypeError: String can't be coerced into Integer
```

# Revisando la documentación

Consultar la documentación es un hábito que debemos adoptar. Todos los programadores la ocupan, incluyendo los expertos.

## ¿Cómo se lee la documentación?

- Utilizaremos la documentación de ruby-doc.
- En esta página podemos ver la documentación con todos los objetos incluidos en Ruby.
- Comencemos revisando uno de los objetos que más hemos utilizado: Los Integers.

# Leyendo la documentación

En la documentación veremos 2 columnas:

- La columna derecha nos muestra toda la documentación de los métodos.
- La columna izquierda nos muestra 3 secciones:
  - **In files:** Muestra los archivos donde está definido nuestro objeto.
  - **Parent:** Muestra de dónde heredó algunos de los comportamientos, en algunas ocasiones la consultaremos.
  - **Methods:** Corresponde a los métodos. Esta será la sección que consultaremos con mayor frecuencia.

## ¿Cómo ocupamos estos métodos?

Usando métodos de clase	Usando métodos de instancia	Utilizando el método .size
<p>Para ocupar un método de clase utilizamos la sintaxis nombre del tipo de dato (nombre de la clase) + .metodo</p> <pre>Integer.sqrt(4) # 2 Time.now</pre>	<p>Para utilizar un método utilizaremos la sintaxis: objeto.método por ejemplo:</p> <pre>'paralelepipedo'.size</pre>	<p>Podemos ver en la documentación que el método .size del objeto String devuelve un entero con la longitud del String.</p> <pre>'Paralelepipedo'.size # =&gt; 14</pre>

## Métodos con opciones

También existen métodos que requieren recibir información para operar. Por ejemplo el método `.count`, del objeto `String`, recibe un `substring` (conjunto de caracteres más corto) para poder contar cuántas veces está contenido ese `substring` en el `String` principal.

## El retorno de un método

Lo que devuelve un método se conoce como el retorno del método, y la información que recibe se conoce como parámetro.

De tal manera podemos decir que el método `.count`, del objeto `String`:

- Recibe como parámetro, al menos, un `substring`. Este es obligatorio.
- Retorna un entero correspondiente a la cantidad de veces que está contenido el `substring` en el objeto `String`.

# Ejercitando lo aprendido

## Contexto:

Lo aprendido lo podemos utilizar para crear un pequeño programa donde el usuario introduce un valor y mostramos la cantidad de letras:

¿Qué método tenemos que ocupar `.size` o `.count`?

Algoritmo

Código



**/\* Objetos y sus tipos \*/**



# Clases y objetos

En Ruby, existen distintos tipos de datos. Ya sabemos que estos tipos de datos son clases y los elementos de un tipo en específico reciben el nombre de objetos.

Clases más frecuentes

Integer	String	Float	Time	Boolean	Nil
Corresponde a un número entero.	Corresponde a un carácter o una cadena de caracteres.	Corresponde a un número que puede tener decimales.	Corresponde a una fecha y hora.	Corresponde a verdadero (true) o falso (false).	Corresponde al objeto nulo, la ausencia de un valor.

## ¿Cómo saber de qué clase es un objeto?

Podemos saber el tipo de dato utilizando el método `.class`

Por ejemplo si dentro de IRB escribimos `2.class` obtendremos como resultado Integer, o si probamos con `'hola'.class` obtendremos como resultado String.

## ¿Por qué es importante el tipo de objeto?

Existen distintas reglas para operar entre estos distintos tipos de objetos. Estas reglas las conoceremos consultando la documentación oficial.

**Por ejemplo:** Al sumar dos números obtenemos el resultado de la suma, pero al 'sumar' dos palabras obtenemos la concatenación de estas.

En algunas situaciones, cuando faltemos a estas reglas, las operaciones no serán válidas.

# Concatenando strings

Observemos el siguiente ejemplo: el método `+` del objeto `String` recibe como parámetro otro `String` a concatenar.

 **`str + other_str → new_str`**

Concatenation—Returns a new `String` containing *other\_str* concatenated to *str*.

```
"Hello from " + self.to_s  #=> "Hello from main"
```

# Transformando tipos de objetos

Existen distintos métodos que nos permiten transformar un objeto de un tipo a otro. Dentro de estos métodos podemos destacar:

El método to_i (To Integer)	El método to_s (To String)
Nos permite convertir un String en un Integer.	Nos permite convertir un Integer en un String.

**/\* Operaciones aritméticas \*/**

# Operadores aritméticos

Los operadores aritméticos nos permiten realizar operaciones matemáticas sobre números.

Operador	Nombre	Ejemplo => Resultado
+	Suma	$2 + 3 \Rightarrow 5$
-	Resta	$2 - 3 \Rightarrow -1$
*	Multiplicación	$3 * 4 \Rightarrow 12$
/	División	$12 / 4 \Rightarrow 3$
**	Potencia	$2 ** 4 \Rightarrow 16$

## Creando un calculadora

Esto nos permite que el usuario ingrese los valores, transformarlos a números y luego operar.

```
a = gets.to_i  
b = gets.to_i  
puts "a + b es: #{a + b}"  
puts "a * b es: #{a * b}"
```

Este código podemos guardarlo en el archivo `calculadora1.rb` y ejecutarlo con `ruby calculadora1.rb`.

# Precedencia de operadores

Un concepto muy importante que debemos conocer es el de precedencia. Dicho de otro modo, saber en qué orden se realiza un grupo de operaciones .

## Ejemplo de precedencia

Por ejemplo, en la siguiente instrucción ¿cuál es el resultado?.

```
10 - 5 * 2
```

```
10 - 5 * 2 # 0
```

# Orden de las operaciones

Veamos una tabla simplificada de precedencia. Esta tabla está ordenada de mayor a menor prioridad, esto quiere decir que la operación de exponenciación precede a (se realiza antes que) la suma.

Operador	Nombre
**	Exponenciación (potencia)
*, %	Multiplicación, división y módulo
+, -	Suma y resta



# Operaciones y paréntesis

Al igual que en matemáticas, los paréntesis cambian el orden en que preceden las operaciones. Dando prioridad a las operaciones que estén dentro de los paréntesis.

$$(10 - 5) * 2$$



$$5 * 2$$

$$10$$

# Operaciones con números enteros y decimales

**Float:** Tipo de dato asociado a los números decimales, llamado float.

```
(3.1).class # float
```

**Enteros y floats:** La división entre entero y float, o float y entero da como resultado un float.

```
5.0 / 3.0 # 1.6666666666666667
```

```
5 / 3.0 # 1.6666666666666667
```

# Ejercicio de Pitágoras

## Contexto:

La fórmula de Pitágoras nos permite calcular el largo de la hipotenusa de un triángulo rectángulo a partir de los largos de los catetos. Crearemos un programa donde el usuario introduzca los valores de ambos catetos y entreguemos como resultado el largo de la hipotenusa.

Para implementar el código sólo necesitamos conocer la fórmula.

$$h = \sqrt{c_1^2 + c_2^2}$$

En Ruby se puede hacer el exponente como \*\*, o sea se puede programar como a\*\*2, también es lo mismo que a\*a las dos opciones son válidas para resolver el problema

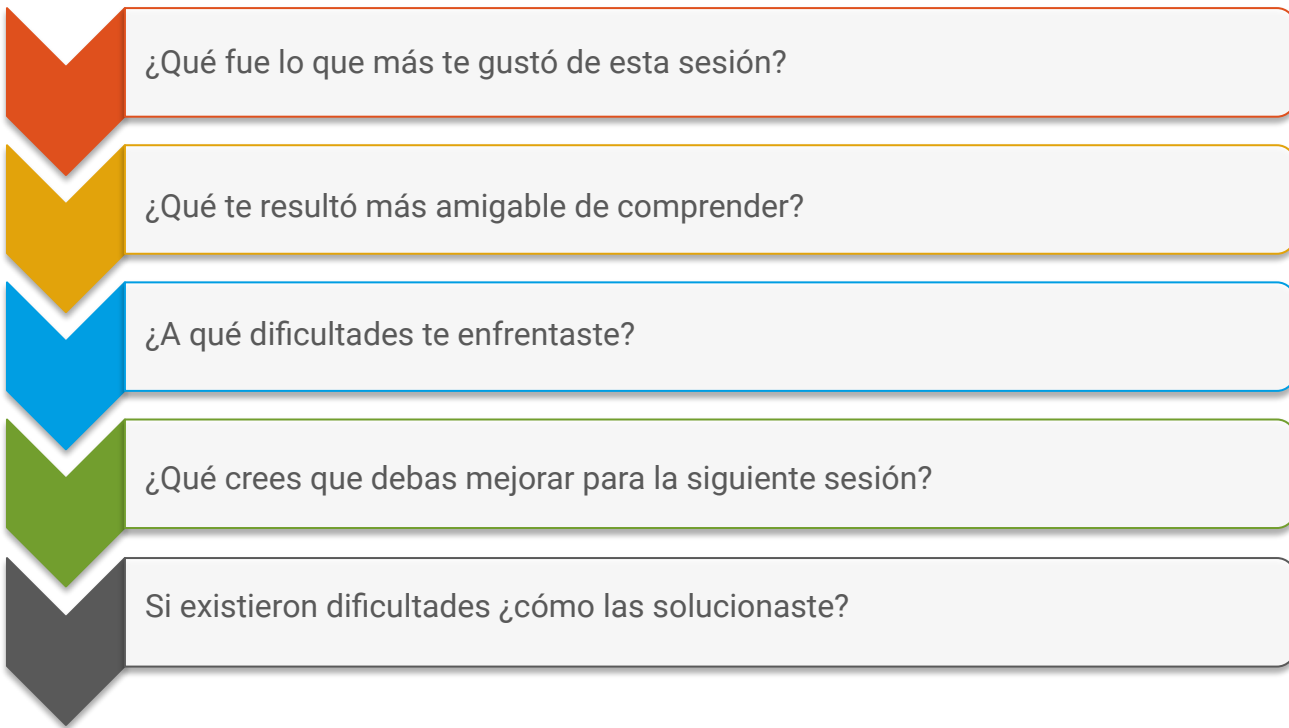


# Cierre

{desafío}  
latam\_



15 minutos



¿Qué fue lo que más te gustó de esta sesión?

¿Qué te resultó más amigable de comprender?

¿A qué dificultades te enfrentaste?

¿Qué crees que debas mejorar para la siguiente sesión?

Si existieron dificultades ¿cómo las solucionaste?



*Academia de  
talentos digitales*

[www.desafiolatam.com](http://www.desafiolatam.com)



/DesafioLatam



/DesafioLatam



/DesafioLatam



/DesafioLatam