~ Made By M4N14CK ~

RUBY

Aprende Ruby, uno de los lenguajes de programación más bellos, artísticos y útiles. ¡Práctica escritura de código en Ruby, obtén puntos y presume de tus habilidades ya!

¡Bienvenido a Ruby!

Ruby en un lenguaje de programación de propósito general, dinámico y orientado a objetos Está considerado dentro de los 10 mejores lenguajes de programación a nivel mundial. Mucho de su crecimiento es atribuido a la popularidad del software escrito en Ruby, particularmente el framework para desarrollo web Ruby on Rails.

Citando a su creador, Yukihiro "Matz" Matsumoto: "*Ruby es simple en apariencia, pero es muy complejo internamente, tal como nuestro cuerpo humano.*"

Matsumoto ha dicho que Ruby está diseñado para la productividad del programador y para que sea divertido, siguiendo los principios de diseño correcto de interfaz de usuario.

En Ruby, todo (incluso un simple número) es un objeto. Aprenderemos más acerca de objetos en las siguientes lecciones.

Sabiendo esto sabemos que en ruby todo es un **objeto.**

Ruby también es completamente gratuito. No sólo libre de cargos, pero también libre para ser usado, copiado, modificado y distribuido.

Instalacion de Ruby en Linux

Instalar Ruby en Linux es un proceso sencillo, pero hay varias formas de hacerlo dependiendo de tu distribución y preferencias.

En Ubuntu/Debian y derivados:

sudo apt update
sudo apt install ruby-full

```
sudo apt install ruby-full
```

Una vez puesto sudo apt install ruby-full en la terminal nos aparecerá este mensaje:

```
Installing:
  ruby-full

Installing dependencies:
  libruby     rake  ruby      ruby-did-you-mean  ruby-power-assert  ruby-sdbm       ruby3.3
  libruby3.3  ri    ruby-dev  ruby-minitest      ruby-rubygems      ruby-test-unit  ruby3.3-dev

Summary:
  Upgrading: 0, Installing: 15, Removing: 0, Not Upgrading: 8
  Download size: 8,833 kB
  Space needed: 36.9 MB / 1,017 GB available

Continue? [Y/n]
```

Le daremos a la tecla enter para continuar y se instalarán las dependencias.

Verifica la versión:

ruby --version **tambien** ruby -v

```
└─$ ruby --version
ruby 3.3.7 (2025-01-15 revision be31f993d7) [x86_64-linux-gnu]
```

==========

En Arch Linux/Manjaro:

sudo pacman -S ruby

¡Listo! Ahora tienes Ruby instalado en tu Linux.

¡Hola, Ruby!

Vamos a crear nuestro primer programa en Ruby, el clásico programa "Hola Mundo". Para esto, utilizamos el método incorporado puts. Pero antes crearemos una carpeta y un archivo con el nombre holaMundo.rb en nuestro SO de linux y usaremos ya sea nano, vim o VScode pero eso ya depende de ustedes en este caso usaremos vim.

Para nosotros crear la carpeta en nuestro sistema linux usaremos el comando **mkdir** con el siguiente nombre **conceptosBasicos**

ejemplo:

```
mkdir conceptosBasicos
```

una vez creado la carpeta accedemos a ella con el comando **cd conceptosBasicos** y le daremos a enter.

```
cd conceptosBasicos/
```

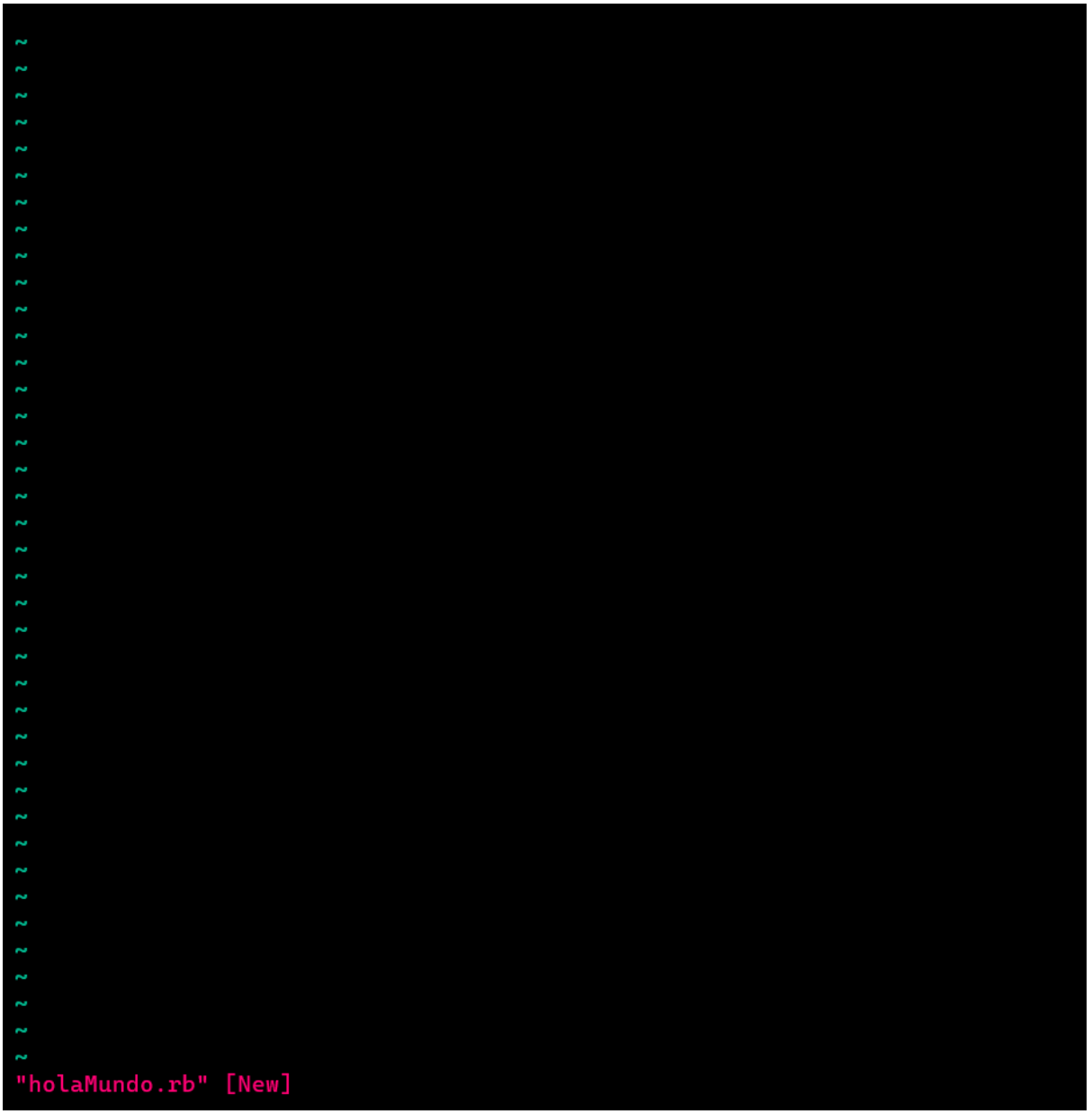
y en nuestro prompt se tendrá que ver ahora la ruta en la que estamos que en este caso es **conceptosBasicos**.

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
```

y allí mismo creamos el archivo usando **vim holaMundo.rb**

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ vim holaMundo.rb
```

Una vez hecho todo esto nos aparecerá algo como esto:



ahora para nosotros poder hacer nuestro hola mundo usando **vim** primero presionaremos la tecla **i** para **insertar** lo que queremos poner y en la parte de hasta abajo podremos ver que se cambio su modo a **- - INSERT - -** :

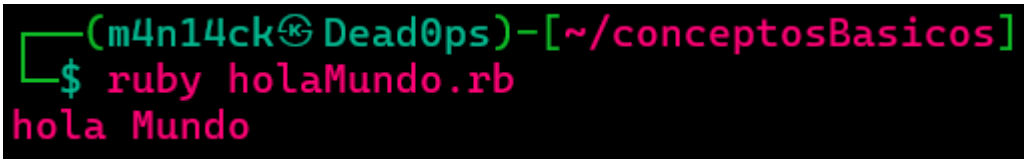
ahora sí pondremos nuestro primer hola mundo que en este caso se hace con el método **puts “ “**

Este código desplegará el texto "Hello World" en la pantalla. **Todos los valores de texto (strings) deben ser encerrados entre comillas sencillas o dobles.**

ahora para salir del editor daremos primero a la tecla **esc** y luego pondremos **:wq** y le daremos **enter**

ahora para ver nuestro **hola Mundo** en la pantalla simplemente pondremos **ruby holaMundo.rb** y luego le damos a **enter.**

y podremos ver este resultado en la consola:



!Felicidades acabas de crear tu primer programa en ruby¡

Recuerda que esto es solo el comienzo a si que no te rindas, además saber ruby te será util para crear **payloads** en la famosa herramienta **MetaSploit** a si que si quieres crear **herramientas** con este lenguaje y usarlas será mejor **leer el pdf** y **hacer algunos ejercicios** que se estarán dejando a lo largo de este documento así que adelante **NO TE RINDAS!!!!**

Comentarios

Los comentarios son líneas de anotaciones dentro del código Ruby que son ignoradas en tiempo de ejecución del programa.

En Ruby, el símbolo de numeral es utilizado para crear un comentario de una sola línea.

para esto creamos otro **archivo** llamado **vim comentarios.rb**

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ vim comentarios.rb
```

de la misma forma como lo hicimos anteriormente se abre el editor y presionando la tecla **i** se podrá empezar a editar sin problemas.

ahora para poder poner un comentario en ruby usaremos la **#** y esto no afectará a nuestro programa ya que el intérprete de ruby no lo va a leer como una instrucción de código pues será ignorada durante su ejecución y esto es útil para especificar que hace cada línea de código que pongamos.

ejemplo:

```
# Esto es un comentario
# Este es otro comentario xD

# Imprimiendo un hola como estas usando el metodo puts

puts "hola como estan?"

# Imprimiendo lo mismo pero usando print

print "hola como estan?"

# La diferencia en usar puts y print es que
# puts da un salto de linea
#
# print no da salto de linea
```

Ahora guardamos el con presionando la tecla **esc** y luego **:wq** para guardar y salir

Una vez guardado simplemente ponemos **ruby comentarios.rb**

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ ruby comentarios.rb
```

Este será el resultado al darle enter:

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ ruby comentarios.rb
hola como estan?
hola como estan?
```

y como se puede observar no salen los comentarios que se hicieron con **#**, sabiendo esto nosotros ahora podremos poner comentarios para saber que hace cada linea de nuestro programa.

Tambien podemos crear cometarios de varias lineas. Todo lo que esten entre las palabras reservadas **=begin** y **=end** es considerado un comentario.

Esto lo podemos hacer ya sea en un nuevo archivo o en el mismo archivo en mi caso creare un nuevo archivo llamado **comentariosMultiLinea.rb**

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ vim comentariosMultiLinea.rb
```

Ejemplo:

```
=begin

  Esto es un comentario multilinea, a diferencia de los comentarios que utilizan #
Aqui se puede escribir de forma continua sin tener que poner varias # para cada comentario
y esto es perfecto para dar una explicacion de lo que hace el codigo.

  Tambien es perfecto para crear un problema de programacion sin tanto #
y esto es perfecto pues ahorramos tiempo la verdad

=end
~
~
~
```

y de la misma forma al ejecutarlo no saldrá nada en consola pues solo es un comentario pero multilínea y no se está utilizando algún método o función para que imprima o muestre algo por consola.

Variables

Una **variable** es el nombre de una ubicación de almacenamiento para un valor. Es llamada **variable** porque la información almacenada en esa ubicación puede ser modificada cuando el programa es ejecutado. Para asignar un valor a una variable, utiliza el signo **igual(=).**

Ejemplo:

```
# A si es como se declara una variable en ruby

x = 8
~
```

Esta declaración de asignación declara una variable llamada x y le otorga el valor 8.
El signo “igual” es llamado el operador de asignación.

Posteriormente podemos utilizar el nombre de la variable para acceder a su valor.
Por ejemplo, para desplegar el valor almacenado en la variable, podemos utilizar **puts** o **print** y referirnos al nombre de la variable:

```
# A si es como

x = 8
puts x
~
```

Los nombres de variables pueden consistir de caracteres alfanumericos y el carácter de guión bajo(_), pero **no pueden** comenzar con una letra mayúscula.

Constantes

Las variables que comienzan con una letra mayúscula son llamadas **constantes.**
El valor de una variable constante no puede ser modificado una vez que ha sido asignado.

Recuerden siempre crear el archivo por cada nueva lección:

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ vim constantes.rb
```

Constantes Ejemplo:

```
MiNum = 24

# Intentando cambiar el valor pero dara error xd
MiNum = 2
~
```

Salida:

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ ruby constantes.rb
constantes.rb:4: warning: already initialized constant MiNum
constantes.rb:1: warning: previous definition of MiNum was here
```

Tipos de Datos

Todas las variables en ruby pueden ser de todos los **tipos de datos.** Ruby determina automáticamente el tipo de dato por el valor asignado a la variable.

Ejemplo:

```
# Tipos de datos

x = 42 # tipo entero
y = 1.58 # tipo flotante
c = "hola" # tipo cadena
```

Puedes reasignar un valor diferente a una variable en cualquier momento.
Para insertar el valor de una variable dentro de una string de comillas dobles(una string es una secuencia de caracteres, como el “hola mundo”), que utiliza el símbolo # y llaves con el nombre de una variable.

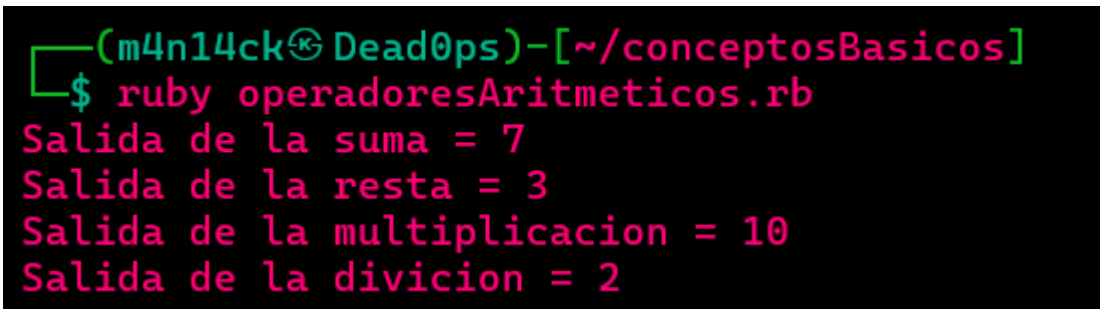
Ejemplo:

```
# Tipos de datos

x = 42 # tipo entero
y = 1.58 # tipo flotante
c = "hola" # tipo cadena

# insertando un tipo entero en una string

edad = 24
puts "El tiene #{edad} años"
```

Haciendo matemáticas

Las matemáticas son una parte fundamental de la programación. Ruby soporta los siguientes operadores aritméticos:

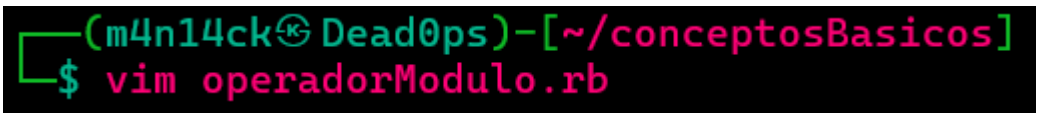
```ruby
# Operadoes aritmeticos en ruby

x = 5
y = 2

# Suma

suma = x + y

puts "Salida de la suma = #{suma}"

# Resta

resta = x - y

puts "Salida de la resta = #{resta}"

# Multiplicacion

multi = x * y

puts "Salida de la multiplicacion = #{multi}"

# Divicion

div = x/y

puts "Salida de la divicion = #{div}"
~
~
```

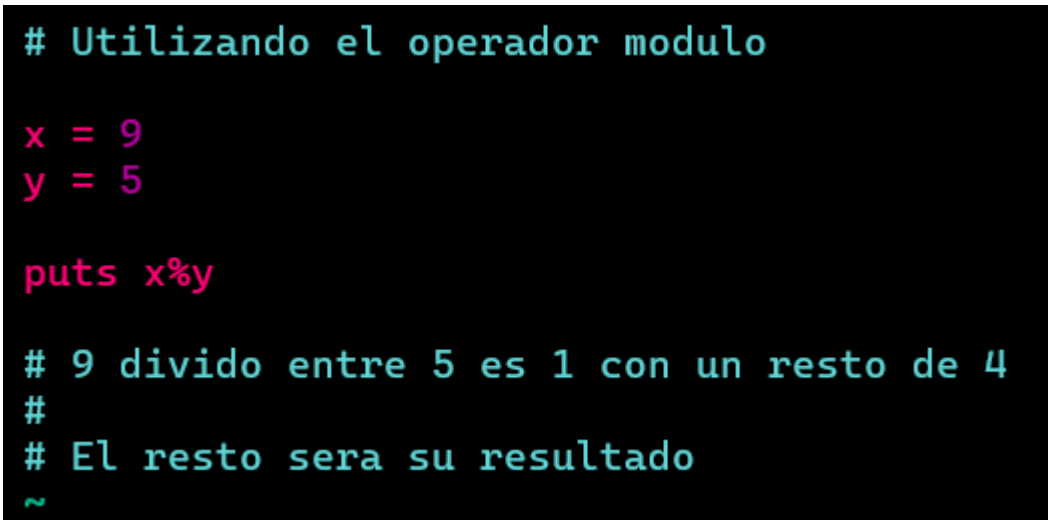


Cuando se dividen dos valores enteros, el resultado será un entero como se muestra en el ejemplo anterior. Si deseas tener un resultado de punto flotante, un operando debe ser un valor de punto flotante**.**

Operador Módulo

El operador **módulo**, representado por el símbolo de porcentaje (%), representa el resto de una operación de división.





Operador Exponente

El operador **exponente** está representado por ** para elevar un número a una potencia y realizar la exponenciación.

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ vim operadorExponente.rb
```

```ruby
# Utilizando el operador exponente (**)

a = 2
b = 5

puts a**b

# El resultado es 32
#
# Basicamente es multiplicar 5 veces 2
#
# 2 * 2 * 2 * 2 * 2 = 32
```

Todos los operadores también pueden ser utilizados con valores de punto flotante.

Operadores abreviados de asignación

Todos los operadores aritméticos tienen formas abreviadas correspondientes para la asignación. Por ejemplo, **a = a + 8** puede ser escrito como **a += 8.** Lo mismo aplica a otros operadores:

```
=begin

  Simplificaremos la manera en como se puede sumar un resultado
  en una variable por ejemplo:

  Supongamos que a = 5 y ahora queremos sumarle 5 pues se hace de
  la siguiente forma a = a + 5 esto quiere decir que la variable
  la cual es "a" va a almacenar a + 5 lo cual es 5 + 5 el cual
  nos dara como resultado 10

=end

a = 5

a = a + 5
puts a

# pero nosotros queremos acortar esa parte
# y para hacerlo ponemos a += 5 lo cual
# hace lo mismo pero de una forma mas corta

a += 5
puts a

# El resultado sera 15 por que se hizo
# la operacion antes osea el a = a + 5
# lo cual da 10 y ahora se puso a += 5
# lo cual da como resultado 15
~
```

Si lo ejecutamos este será el resultado:

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ ruby operadoresAbreviadosDeAsignacion.rb
10
15
```

y de la misma forma se puede hacer usando también variable con variable

Ejemplo:

x += y # x=x+y
x -= y # x=x-y
x *= y # x=x*y
x /= y # x=x/y
x %= y # x=x%y
x **= y # x=x**y

Estos son llamados operadores de auto-asignación, ya que ejecutan una asignación y una operación aritmética al mismo tiempo.

Asignación paralela

Ruby también soporta asignación paralela de variables. Esto permite que varias variables sean inicializadas con una sola línea de código.

```
# A si es como inicializamos las varibales de forma normal

x = 1
y = 2
z = 3

puts "inicializamos de forma normal: \n #{x}\n #{y}\n #{z}\n"


# A si es como se inicializa las variables de forma paralela

x, y ,z = 10,20,30

puts "inicializamos de forma paralela #{x} #{y} #{z}"
```

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ ruby asignacionParalela.rb
inicializamos de forma normal:
 1
 2
 3
inicializamos de forma paralela 10 20 30
```

La asignación paralela es también útil para intercambiar los valores almacenados en dos variables: **a**, **b** = **b**, **a**

Precedencia de operadores

Ruby evalúa una expresión matemática utilizando un orden en las operaciones que está basado en la precedencia de los operadores. La exponenciación tiene la mayor procedencia seguida de la multiplicación, división y el módulo de izquierda a derecha, y la adición y sustracción de izquierda a derecha.

Puedes intercambiar el orden de las operaciones utilizando paréntesis.

```
=begin

  Se puede observar que por la jerarquia de operaciones
  primero se hara la operacion de la suma de "3 + 2" el cual
  da como resultado "5" y luego se multiplica el resultado
  osea el "5 * 4" el cual nos dara como resultado "20" y ese
  resultado el cual es "20" se guardara en la variable "x"

=end

x = (3 + 2) * 4
puts x


~
```


```
=begin

  Se puede observar que por la jerarquia de operaciones
  primero se hara la operacion de la suma de "3 + 2" el cual
  da como resultado "5" y luego se multiplica el resultado
  osea el "5 * 4" el cual nos dara como resultado "20" y ese
  resultado el cual es "20" se guardara en la variable "x"

=end

x = (3 + 2) * 4
puts x


~
```


¿Cuál es la salida de esta operación?

5-2*3+3

Utilizando la jerarquía primero se hará la multiplicación que en este caso es **2 * 3** el cual da como resultado **6** para luego hacer la operación de la **resta** y por último la **suma** por lo que la resta da **5 - 6 = -1** y la la suma da **-1 + 3 = 2** por lo que **2** es el resultado final

Strings

Tal y como se mencionó en las lecciones anteriores, una **string(**cadena de texto**)** es un texto entre comillas simples o dobles.

Sin embargo, algunos caracteres no pueden ser incluidos directamente en una string. Por ejemplo, las comillas simples no pueden ser directamente incluidas en una string de comillas simples, porque esto designa el final de la string. Caracteres como estos pueden ser incluidos es una string utilizando una **secuencia de escape**, la cual se indica con una **barra invertida (\):**

```
# Cadena de texto con barra invertida \


text = 'Ruby\'s syntax is fun'
puts text
```

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ ruby strings.rb
Ruby's syntax is fun
```

Una string formada con comillas dobles también puede incluir la secuencia de escape **\n**, la cual representa una nueva línea.

```
# Haciendo saltos de linea \n

text = "Hola Mundo \n xD"
puts text
~
```

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ ruby saltoDeLinea.rb
Hola Mundo
 xD
```

Solo las secuencias de escape \ y \\ pueden ser utilizadas con strings de comillas simples.

Interpolación de String

Puedes incrustar cualquier expresión de ruby dentro de strings de comillas dobles utilizando **#{}**, tal cual como haces con los nombres de variables. Ruby evalúa los marcadores y los reemplaza con valores.

```
# Ejemplo de una interpolacion utilizando #{}

a = 5
b = 2

puts "La suma de a + b es: #{a + b}"
~
```

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ ruby interpolacionDeStrings.rb
La suma de a + b es: 7
```

Fijate que no hay espacio entre la almohadilla (**#**) y la llave abierta **{}**. En caso de haber un espacio, este será interpretado como texto literal.

Concatenación

Las strings pueden ser unidas utilizando el signo **+** en un proceso llamado **concatenación**. Cuando concatenamos strings, no importa si han sido creadas con comillas simples o dobles.

```
# A si es como se hace una concatenacion en ruby usando
# el simbolo de +

a = "hola "
b = "como estas?"

puts a + b

# Recuerda poner un espacio al final dentro de las comillas
# para que cuando se haga la concatenacion y se muestre por
# consola estas no esten juntas
~
```

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ ruby concatenacion.rb
hola como estas?
```

Aunque tus strings contengan números, ellas serán unidas como strings en lugar de sumadas como enteros. Añadir una string a un número produce un error, porque aunque pueden verse similares, son dos entidades completamente diferentes: “1” es una string, mientras que 1 es un entero.

Repitiendo una string

Las strings pueden ser repetidas utilizando el símbolo * y un valor entero. El orden entre el string y el entero importa: el **string** tiene que aparecer **primero**.

```ruby
# Repitiendo una String

# Saldra a 5 veces a si "aaaaa"
a = "a"
puts a * 5

# Saldra 5 cuatro veces a si 5555
puts '5' * 4
~
~
```

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ ruby repitiendoUnaString.rb
aaaaa
5555
```

Las **strings** no pueden ser multiplicadas por otras **strings** pues como vas a multiplicar texto con texto xd.

Input

Para obtener entrada del usuario en ruby, se utilizara el método **gets**, el cual retorna lo que el usuario escribe como una string. Para almacenar la entrada para su uso posterior, puedes asignar el valor de retorno a una variable.

```
=begin

  Ejemplo del uso del metodo gets

Se le pide al usuario que ingrese su nombre y este se almacena
en la variable "x" y se mostrara en la consola

la variable "x" tiene almacenado el metodo "gets" por lo que
podemos deducir que el metodo gets es un "input"una "entrada"

lo que hace es esperar a que el usuario ingrese algo y se
guarda en la variable "x"

=end

print "Digite su nombre: "
x = gets

puts x
```

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ ruby input.rb
Digite su nombre: m4n14ck
m4n14ck
```

El método **gets** obtiene una línea de texto, incluyendo el salto de línea al final. Si no quieres incluir el salto de línea, puedes utilizar el método **gets.chomp.**

```
print "Digite su nombre: "
x = gets.chomp

puts "Su nombre es: #{x}"
```

```
Digite su nombre: m4n14ck
Su nombre es: m4n14ck
```

El valor de la entrada es un **string**. Para convertirla a un entero, puedes utilizar el método **gets.to_i**

Examen de Módulo 1

¿Cuál es la salida de este código?

temp = “7”
temp += “2”

puts temp

Completa los espacios en blanco para **asignar 42** a la **variable**, **sumarle 7** y desplegar como salida su valor:

x = ?
x ?= 7
puts x

Completa los espacios en blanco para hacer del texto un comentario:

?begin

Esto es algo de texto
pero con varias
lineas xd

?end

Completa los espacios en blanco para incrustar el valor de la variable en la **string** y despliega como salida en la pantalla.

nombre = “Amy”
msg = “Hola, mi nombre es ?{ }”
puts ?

¿Cuál es la salida de este código?

a = 4
puts a**2

Antes de continuar

Antes de comenzar crearemos dos carpetas llamada **Leccion1** y **Leccion2** para hacer las cosas más limpias.

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ mkdir Leccion1 Leccion2
```

y veremos las carpetas creadas

```
Leccion1
Leccion2
```

Ahora usaremos el comando **mv** seguido de esta instrucción ***.rb** y el lugar donde queremos mover todo los archivos que terminen en **.rb** el cual es la carpeta que creamos llamada **Leccion1** entonces quedaria asi **mv *.rb Leccion1**

```
┌──(m4n14ck㉿Dead0ps)-[~/conceptosBasicos]
└─$ mv *.rb Leccion1
```

Una vez hecho todo esto si hacemos un **ls** podremos notar que los archivos ya no están pues los hemos movido a la carpeta **Leccion1.** Una vez terminado esto nos movemos a la carpeta llamada **leccion2** y desde allí seguiremos con las nuevas lecciones eso seria todo.

Booleanos

Accedemos a la carpeta y creamos el archivo le pondremos **trueFalse.rb** y accedemos con el editor de código **vim** o **VScode** o el que ustedes prefieran.

En Ruby, hay dos valores Booleanos: **true(verdadero)** y **false(falso).**

```
# Los booleanos utilizan true o false

# Ejemplo

isOnline = true
userIsAdmin = false
```

Otro valor que te encontrarás usualmente es **nil**. El cual representa la ausencia de valor.

Si intentas evaluar un valor distinto que **true** o **false** como Booleano, Ruby automáticamente lo tratara como un Booleano.

Cuando esto es realizado, un valor no-Booleano que es evaluado como verdadero es llamado **truthy** y un no-Booleano que evalúa a falsos es llamado **falsey**

En ruby solo **false** y **nil** son falsey. Todo lo demás es **truthy(**incluso 0 es **truthy)**

Solo **true** y **false** son Booleanos. **nil** no es un Booleano. 0 no es un Booleano. La string “Hello” no es un Booleano. Sin embargo, en un contexto donde se espera un Booleano, Ruby los evalúa como Booleano (**truthy** y **falsey**)

Comparación

Una comparación Booleana utilizando el operador **==** retorna **true** (verdadero) cuando dos operando son iguales, y **false**(falso) cuando no lo son:

```
# Vamos a comprar 2 numero con el operador ==
#
# y esto nos tendra que regresar ya sea true o false

# Declaramos las variables

a = 5
b = 8

puts a == b # Esto mostrara false

puts a == 5 # Esto demostrara true
~
~
```

Ten cuidado de no confundir **asignación**(un signo igual(**=**)) con **comparación**(dos signos iguales(**==**))

Otro operador de comparación, el **operador** “**no igual que**” (**!=**)**,** evalúa a verdadero si los operandos comparados no son iguales, y falso sí lo son.

```
=begin

 Aqui hacemos lo mismo que una comparacion como ==
 pero a qui estamos diciendo si no es igual
 por lo que si estamos comparando pero estamos dicendo
 que si es "diferente"

=end

a = 8
b = 7

puts a != b # Esto muestra true
~
```

Como la comparación que se hizo si es **diferente** entonces muestra **true** por que 8 **no es igual a 7** por lo cual es cierto(**true**) pero si fueran **iguales** entonces no daría un **false** pues porque lógicamente **no es diferente** por lo tanto si es diferente da **true** pero si es igual es **false**.

Ruby también tiene operadores que determinan si un valor es mayor que o menor que otro. Estos operadores son **mayor**(**>**) y **menor**(**<**) respectivamente. De forma similar, los operadores **mayor** o **igual que**(**>=**) y **menor** o **igual que**(**<=**)

```
# Ejemplos de los operadores > y <
# tambien de >= y <=

puts 12 > 8 # Esto es true por que 12 si es mayor que 8

puts 5 < 2  # Esto es false por que 5 no es menor que 2

puts 5 >= 5.0
# Esto es true por que 5 aun que no es mayor que 5 es igual a 5
# por lo que se cumple la condicion y por ende da true

puts 3 <= 6
# Esto da true por que aun que 3 no sea igual que 6 es menor que 6 por lo tanto
se cumple y da como resultado true
~
```

También está el método **.eql?**, el cual resulta en verdadero solo si ambos argumentos tienen el **mismo tipo e igual valores.**

```
=begin

  Usando .eql? el cual como sabemos solo dara VERDADERO si
  ambos argumentos tienen el mismo tipo e igual valores.

=end

puts 3 == 3.0 # Esto dara true por que son iguales

puts 3.eql?(3.0) # En este caso dara false por que aun que sean iguales
                 # No son del mismo tipo por ende da false
~
```

El 3 es un tipo entero y 3.0 es un tipo flotante por ende usando **.eql?** nos dará falso por que no son del mismo t**ipo**

Los operadores mayor que y menor que también pueden ser utilizados para comparar strings **lexicográficamente** (el orden alfabético de las palabras basado en el orden alfabético de las letras que las componen).

Declaraciones if

Puedes utilizar una expresión **if** para ejecutar código cuando se cumple una cierta condición. Si una expresión condicional evalúa a **verdadero**, el código **if** es ejecutado. En otro caso, el código es ignorado.

```
=begin

  La condicional if se utiliza para saber si una expresion
  es verdadero o falso pero tambien para hacer una accion o
  mostrar algo en pantalla si la expresion es verdadero o
  falso

=end

a = 42

  if a > 7
    puts "#{a} es mayor que 7"
  end

=begin

  Como se puede observar se cumple la condicion y
  una vez que se cumple muestra por pantalla un mensaje
=end
```

La condición **a > 7** es evaluada. Cuando es **verdadera,** las declaraciones dentro del **if** son ejecutadas y el programa despliega "**47 es mayor que 7**"**.** Puedes tener varias declaraciones dentro de una sola expresión **if**.

También la palabra clave **end** es requerida para indicar el final del **if,** básicamente el **end** es para cerrar el bloque de código del **if** a si se sabrá cuando se habrá terminado de declarar las **declaraciones** que están dentro del o todo lo que esté dentro de **if**.

Las expresiones **if** pueden ser anidadas, una dentro de otra. Esto significa que el **if** más interno es el código del externo. Esta es una forma de ver si varias condiciones son satisfechas. Ten presente que una vez que una condición **if** falle, se saldrá del bloque completo.

```
num = 16

if num > 7
  puts "Mayor que 7"

  if num < 42
    puts "Entre 7 y 42"
  end

end
~
```

Cada **if** tiene que tener un **end** correspondiente.

Declaraciones else

Un bloque **else** en una expresión **if** contiene código que es invocado cuando la condición **if** evalúa a **falso.**

```
=begin

  Aqui solo se explicara el bloque else pues
  ya se explico lo de la condicional if y bien
  ahora el bloque else funciona como un tipo de
  excepcion pues si la condicion no se cumple
  saltara al else y se le puede poner un mensaje
  por si algo no salio bien o si no cumplio con
  la condicion

=end
edad = 15

if edad > 18
  puts "bienvenido"
else
  puts "demasiado Joven"
end
~
```

La palabra clave **end** solo es necesaria para la declaración **if**, ya que el bloque **else** es parte de la expresión **if.**

Declaraciones elsif

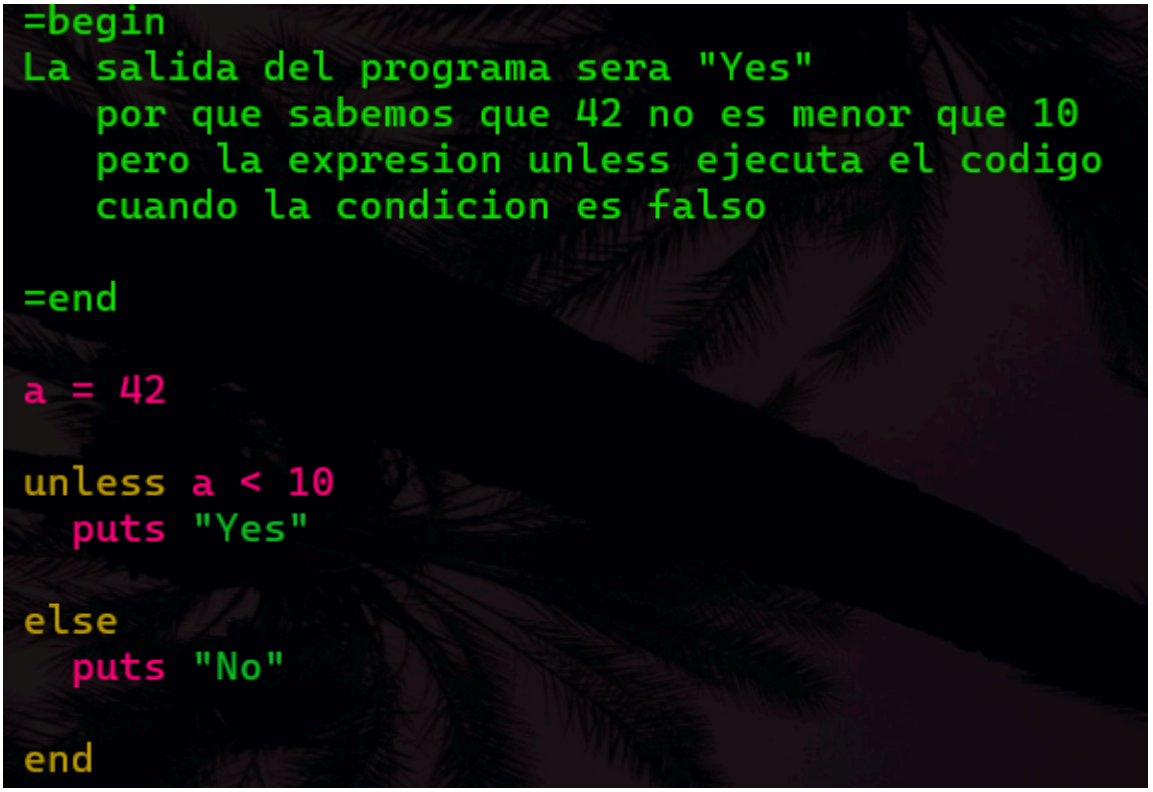
El bloque **elsif**(abreviado para else if) es útil cuando quieres evaluar varias condiciones. Una serie de expresiones **if elsif** pueden tener un bloque else final, el cual es invocado si ninguno de las expresiones **if** o **elsif** son verdaderas.

```
=begin

  El elsif basicamente funciona como la condicional if pero
  este sirve mas como opcion

=end

num = 8

if num == 3
  puts "El numero es 3"

elsif num == 7
  puts "El numero es 7"

elsif num == 8
  puts "El numero es 8"

else
  puts "Numero no encontrado"
end
~
~
-- INSERT --
```

Cuando un bloque **elsif** es ejecutado, se sale de la expresión **if** completa

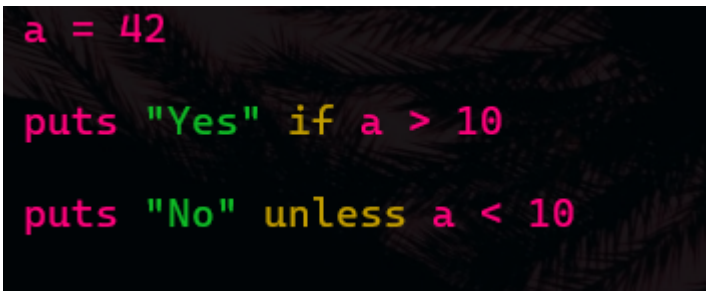
unless

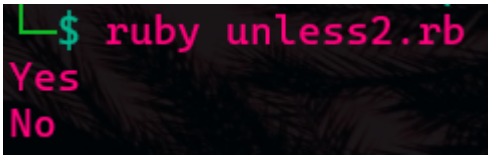
La expresión **unless** es lo opuesto de una expresión **if**. Ejecutar código cuando una condicional es **falso.**

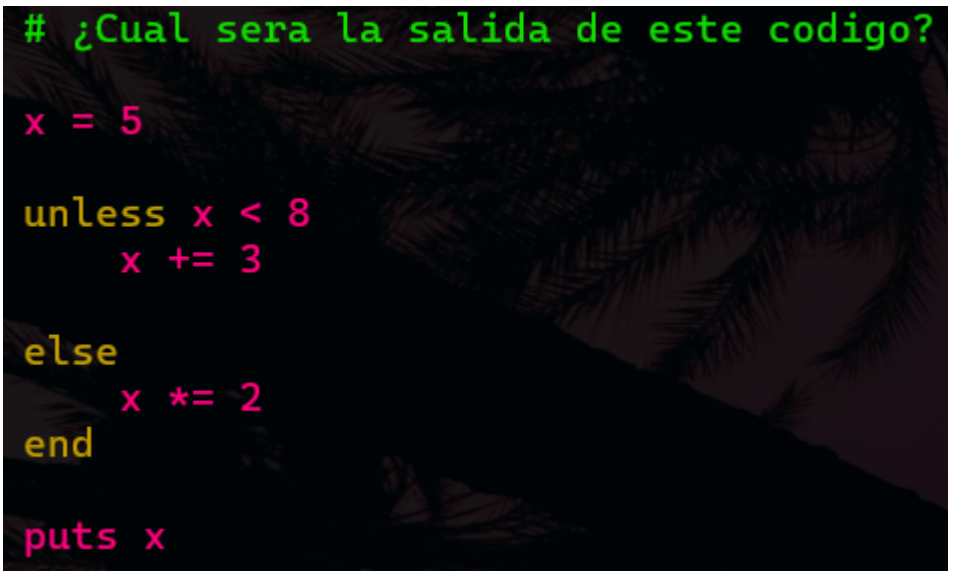


Puedes utilizar un bloque else con el unless, tal y como hiciste con la expresión if. La palabra clave **end** también es requerida para cerrar el bloque.

Los **modificadores if y unless** también pueden ser utilizados para ejecutar código.







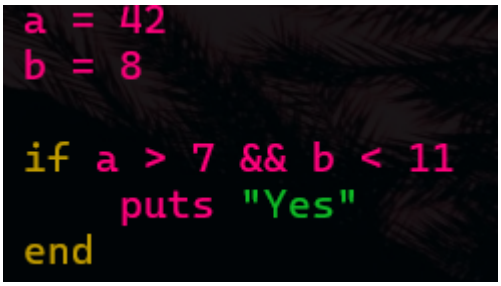
```ruby
# ¿Cual sera la salida de este codigo?

x = 5

unless x < 8
    x += 3

else
    x *= 2
end

puts x
```

```
└─$ ruby unless3.rb
10
```

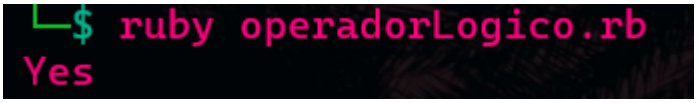
Operadores Lógicos

Los operadores lógicos son utilizados para formar criterios más complejos que evalúan más de una condición para una expresión **if**.

Ruby tiene tres operadores lógicos: **y (&&)**, **o (||)**, y **no (!)**.
Un condicional utilizando **"y"** evalúa como verdadero sí, y sólo si, ambos operandos son verdadero. En otros casos, la condición completa es evaluada como falso.



```ruby
a = 42
b = 8

if a > 7 && b < 11
    puts "Yes"
end
```



```
└─$ ruby operadorLogico.rb
Yes
```

Ruby puede utilizar palabras en lugar de los símbolos de operadores lógicos (**and, or, not**), pero estos tienen menor precedencia y generalmente son evitados.

```ruby
num = 8

if num >= 5 && num <= 10
    puts "Esta en el Rango"
end
```

OR

El operador or **(||)** evalúa a verdadero si alguno (**o ambos**) de sus operandos es verdadero, y falso ambos operandos son falsos.

```ruby
a = 3
b = 5

if a == 3 || b > 10
    puts "Bienvenido"
end
```

```
└─$ ruby operadorLogicoOr.rb
Bienvenido
```

Fíjate que **(||)** y **(or)** no son lo mismo y tienen una prioridad diferente en las operaciones.

```ruby
# ¿Cual es la salida de este codigo

a = 5

if a > 6 || a < 8

    puts a/2

else

    puts a

end
```

```
└─$ ruby operadorLogicoOr2.rb
2
```

NOT

El operador **not (!)** revierte el estado de un sólo operando. El resultado de "**not true**" es "**false**", y "**not false**" es "**true**".

```
a = 7

puts !(a > 5)
```

```
└─$ ruby Not.rb
false
```

Es este código, **a > 5** evalúa a **verdadero** y luego el **operador not** lo revierte a **false**.

Puedes enlazar varias condiciones con los operadores lógicos para validar en conjunto. Los paréntesis pueden ser utilizados para agrupar condiciones por separado para mayor claridad y poder controlar el orden de las operaciones.

Por ejemplo:
(a>b && b < 100) || (a<b && b > 100)

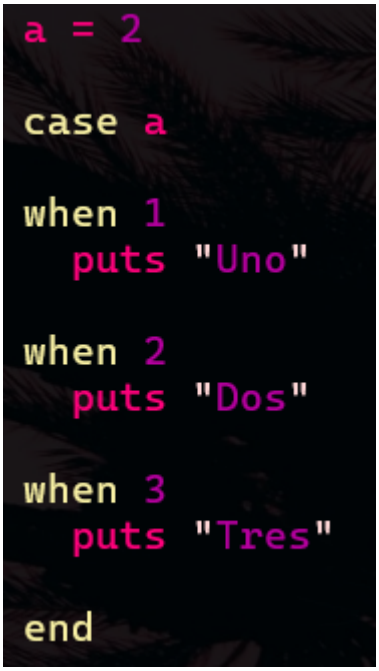
```
# ¿Cuál es la salida de este código?

a = 5

if !(a > 2)
    print("2")

elsif !(1 + 2 == a)
    print("3")

else
    print("4")
end
```

```
└─$ ruby Not2.rb
3
```

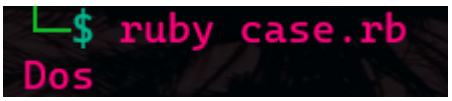
Declaraciones case

Tal y como hemos visto en las lecciones anteriores, podemos validar varias condiciones utilizando la expresión **if/elsif/else**.

Una opción más simple y flexible es la expresión case, la cual valida un valor en declaraciones **when**.

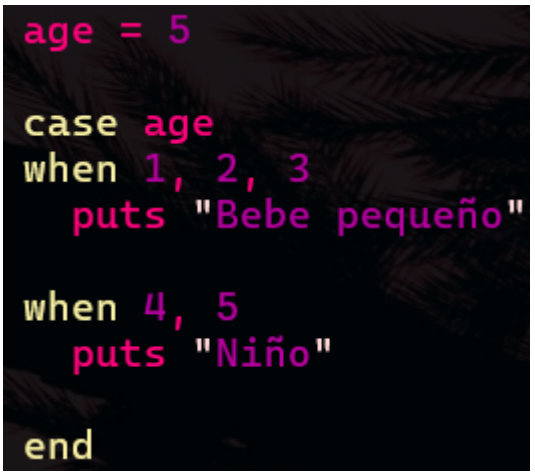


```
a = 2

case a

when 1
  puts "Uno"

when 2
  puts "Dos"

when 3
  puts "Tres"

end
```



```
└─$ ruby case.rb
Dos
```

Puedes tener tantas declaraciones **when** como necesites para un sólo case. Fíjate que la expresión **case** debe ser finalizada con la palabra clave **end**.



```
age = 5

case age
when 1, 2, 3
  puts "Bebe pequeño"

when 4, 5
  puts "Niño"

end
```

```
└─$ ruby case2.rb
Niño
```

Si olvidas poner una coma entre dos valores, Ruby devolverá un error.

```ruby
# ¿Cual sera la salida de este codigo?

x = 8

case x

when 7, 8
  puts x + 1

when 5, 6
  puts x - 1

end
```

```
└─$ ruby case3.rb
9
```

Declaraciones case

Una declaración **else** puede ser proveída para ejecutar código si ninguna condición **when** coincide:

```ruby
age = 18

case age

when 1, 2, 3
    puts "Little baby"

when 4, 5
    puts "Child"

else
    puts "Not a baby"
end
```

```
└─$ ruby case4.rb
Not a baby
```

Las declaraciones case nos permiten controlar más fácilmente el flujo del programa. Las declaraciones if siempre deberían ser utilizadas para determinar si una condición es verdadera, mientras que las declaraciones case son para cuando necesitas tomar diferentes decisiones basado en un valor.

Bucles

Los bucles son utilizados para ejecutar el mismo bloque de código un número específico de veces.

El bucle **while** ejecuta un bloque de código mientras que su condición es **verdadera**.

```
x = 0

while x < 10
  puts x
  x += 1
end
```

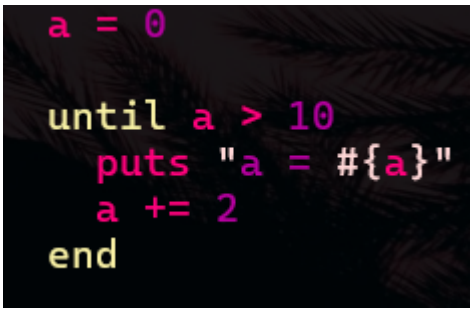
```
└─$ ruby bucle.rb
0
1
2
3
4
5
6
7
8
9
```

Esto desplegará los números del **0 al 9**. Después que la variable de control del bucle se vuelve 10, la condición **x < 10** es evaluada como **falso** y el **bucle finaliza su ejecución**.

Si **omitimos** la **declaración x+=1**, el bucle será ejecutado por siempre, porque la condición se mantendrá verdadera. Esta situación es llamada un **bucle infinito**.

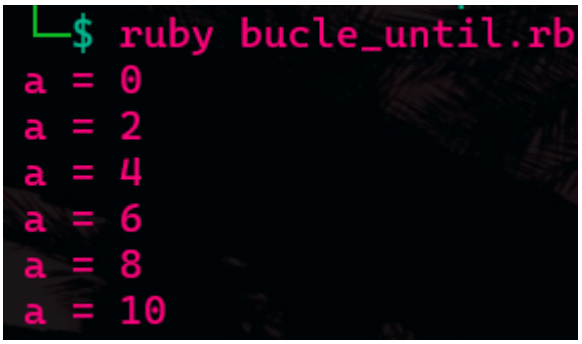
Bucles until

El bucle **until** es lo opuesto a un bucle while: será ejecutado mientras su condición sea **falsa**.

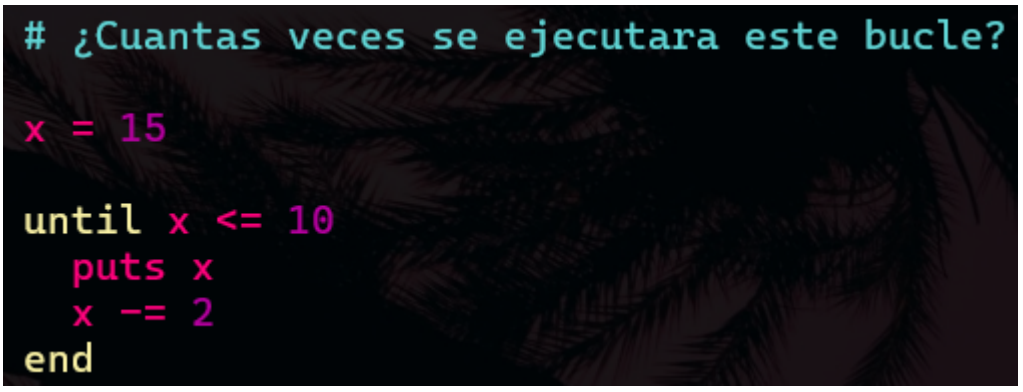


```ruby
a = 0

until a > 10
  puts "a = #{a}"
  a += 2
end
```

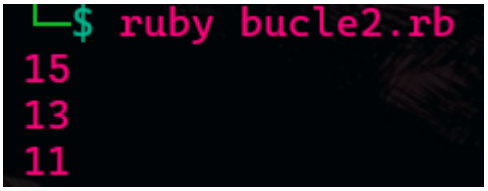
Esto imprimirá todos los números pares del 0 al 10.



```
└─$ ruby bucle_until.rb
a = 0
a = 2
a = 4
a = 6
a = 8
a = 10
```



```ruby
# ¿Cuantas veces se ejecutara este bucle?

x = 15

until x <= 10
  puts x
  x -= 2
end
```



```
└─$ ruby bucle2.rb
15
13
11
```

Rangos

Un **rango** representa una secuencia. **0 a 10, 56 a 82, y "a" a "z"** son todos ejemplos de rangos.
Ruby tiene operadores especiales para crear rangos con facilidad.

Estos son los operadores de rango **".." y "..."**. La forma de dos puntos crea un rango inclusivo, mientras que la forma de tres puntos crea un rango que excluye el mayor valor especificado.

```
a = (1..7).to_a
puts a
puts "\n"

b = (79...82).to_a
puts b
puts "\n"

c = ("a".."d").to_a
puts c
puts "\n"
```

```
└─$ ruby rangos.rb
1
2
3
4
5
6
7

79
80
81

a
b
c
d
```

El método **to_a** es utilizado para convertir un rango en un arreglo, con el fin de que lo podamos desplegar.

Los rangos pueden ser utilizados en declaraciones **case** para valores de **when.**

```ruby
age = 42

case age

when 0..14
    puts "Child"

when 15..24
    puts "Youth"

when 25..64
    puts "Adult"

else
    puts "Senior"
end
```

```
└─$ ruby rangos2.rb
Adult
```

```ruby
# ¿Cuál es la salida de este código?

x = 8

case x

when 0...8
  puts "1"

else
  puts "2"

end
```

```
└─$ ruby rangos3.rb
2
```

Bucle for

El bucle for es una declaración útil cuando necesitas iterar sobre un conjunto específico de valores, por ejemplo, un rango.

El bucle for consiste en una variable vacía y un rango. En cada iteración del bucle, a la variable vacía le será asignado el elemento correspondiente del rango.

```
for i in (1..10)
    puts i
end
```

```
└─$ ruby bucle_for.rb
1
2
3
4
5
6
7
8
9
10
```

Esto desplegará los números del **1 al 10.**
Durante la primera iteración, a la variable **i** le será asignado el primer **valor del rango, 1**.
En la segunda iteración, le será asignado el valor 2, y así sucesivamente, hasta el final del rango.

El bucle for ejecuta una vez un bloque de código para cada elemento en el rango.

break

La declaración break puede ser utilizada para detener un bucle.

```
for i in 1..5
    break if i > 3
    puts i
end
```

```
└─$ ruby break.rb
1
2
3
```

El bucle detiene su ejecución cuando la condición i > 3 es alcanzada.

next

De forma similar, la declaración next puede ser utilizada para saltar una iteración del bucle y continuar con la siguiente.

```
for i in 0..10
    next if i %2 == 0
    puts i
end
```

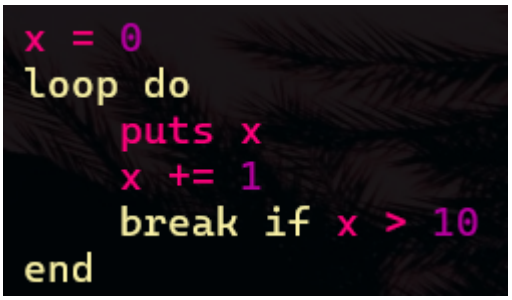
```
└─$ ruby next.rb
1
3
5
7
9
```

Esto desplegará sólo los números impares **desde 0 hasta 10** porque los números pares serán **omitidos** en la iteración del bucle.

Ruby también tiene la declaración **redo**, la cual causa que la iteración del bucle actual sea repetida. La declaración retry produce que el bucle completa comience nuevamente desde el inicio.

loop do

Otra declaración de bucle en Ruby es la declaración **loop do**. Permite que el código se ejecute hasta que se alcance una condición **break**.



Esto imprimirá los números del **0 al 10**. Cuando **x > 10** es evaluado como **verdadero, se detendrá el bucle**.

Si no incluimos una **condición break**, el bucle será ejecutado por siempre.