

III Campus Infantil de Software Libre.

En busca de
la Piedra
Filosofal



Organiza:



Patrocina:



Gabinete de Acción Social
Universidad de Granada

¿Qué brújula utilizaré?

Para encontrar esta piedra mágica tenemos dos herramientas. La principal es el Terminal, que podemos encontrar en cualquier distribución de GNU/Linux, en nuestra lista de programas, con un icono con forma de pantalla en color negro. El terminal será nuestra brújula particular, escribimos órdenes para que nos indique el camino correcto.

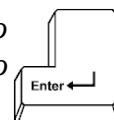


La segunda herramienta, es el editor de texto. En GNU/Linux tenemos varias alternativas: Gedit, Kate o Geany. Puedes usar el que más te guste. En este editor de texto escribiremos los programas y los guardaremos en ficheros con nombres acabados en .rb, por ejemplo: piedra.rb, calamaro.rb o mi_programa.rb

El editor de texto nos da la ventaja de guardar nuestro trabajo: todas las órdenes que le indiquemos a nuestra brújula quedarán a salvo y podrás volver a usarlas sin necesidad de escribirlas de nuevo.

Para lanzar el programa que contiene el fichero de texto tendrás que escribir la orden **ruby piedra.rb** en el terminal.

Abrir un terminal, escribir la palabra `irb` y pulsar la tecla intro. Cuando aparezca el símbolo `irb(main):001:0>` escribe la orden `puts "Hola Ruby!"`. Después tienes que conseguir el mismo resultado usando la orden en un fichero de texto.



Larry, el Alquimista



He de presentaros a mi ayudante, a lo largo de este manual te propondrá ejercicios y te ayudará con las órdenes de Ruby.

Vamos a seguir conociendo órdenes para encontrar la piedra mágica, nuestro amigo el alquimista nos ayudará a resolver operaciones matemáticas.

Si la piedra filosofal quieres encontrar, con Ruby tienes que programar

Nuestro amigo el alquimista necesita tu ayuda, quiere saber cuántos minutos tiene un día. Ten en cuenta que una hora tiene 60 minutos y que un día tiene 24 horas.

Escribe en el Terminal una orden que resuelva este problema.

Cuando quieras realizar varias operaciones (suma,multiplicación...) te será de gran ayuda el uso de paréntesis. Es necesario que lo utilices para establecer qué operación se ejecutará en primer lugar. Por ejemplo, comprueba el resultado de la siguiente operación, con y sin paréntesis.

$$1 + 3 / 3$$

$$(1 + 3) / 3$$

¿Es esta la piedra mágica?

Ahora aprenderás los símbolos que utiliza Ruby para comparar números, palabras, expresiones e incluso todo tipo de piedras. Por ejemplo, si queremos saber si dos piedras preciosas son distintas, escribiremos en el intérprete de Ruby la siguiente orden:

```
rb(main):001:0> "Ruby" != "Diamante"
```

```
25==15
4 > 0
6 > 10
7.7 != 7.6
```



Nuestro amigo el alquimista todavía no domina muy bien el español y nos responderá en Inglés. En el caso de que la cadena "Ruby" sea distinta a "Diamante" responderá **TRUE** y si "Ruby" es igual a "Diamante" responderá **FALSE**.

Observa los símbolos que ha utilizado mi nuevo ayudante. Prueba varios ejemplos con cada símbolo. Recuerda que puedes utilizar tanto números como letras o palabras completas. Al igual que con las operaciones matemáticas, puedes usar paréntesis para ayudarte cuando utilices varios símbolos en una expresión.

¿Qué piedra pesa más?

En los ejemplos anteriores, hemos descubierto que podemos hacerle preguntas a Ruby y nos dice si es verdadera o falsa. En este apartado aprenderás a ordenarle a Ruby lo que quieres que haga, dependiendo de la respuesta que dé a tu pregunta.

Imagina que tienes dos piedras preciosas y te quieres quedar con la que pesa más, ya que es más valiosa. Utilizaremos dos variables: `pedra1` y `pedra2`. En cada una de ellas guardaremos el peso correspondiente a cada piedra. Le preguntaremos a nuestra piedra filosofal cual de ellas pesa más y nos responderá con un mensaje. A continuación te mostraré como resolver el ejercicio en Ruby.

```
pedra1 = 40
pedra2 = 12

if piedra1 > piedra2
  puts "pedra1 es la de mayor peso"
elsif piedra1 == piedra2
  puts "Las piedras tienen el mismo peso."
else
  puts "pedra2 es la de mayor peso"
end
```

Una Cajita Diferente

Habitualmente cuando escribimos programas, necesitamos guardar el resultado de expresiones con operadores, como los que hemos visto en ejemplos anteriores. Para esto necesitamos lo que en programación llamamos **variables**. Tendremos que ayudarnos de un nuevo símbolo: “ = ”, que te servirá para asignar un valor a una variable.

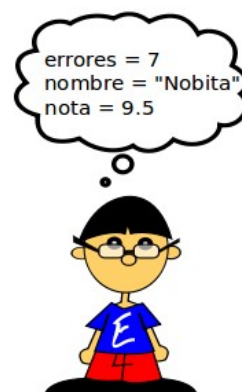
A partir de ahora podrás usar variables y operadores para crear expresiones, de esta forma escribirás y entenderás tu código con más facilidad.

Un uso muy frecuente en programación para las variables es usarlas como un contador. Podemos sumar o restar una cantidad determinada a la variable. Si haces un juego con 10 adivinanzas y quieres que cuente los errores y puntos de cada jugador, para saber quién ha ganado, puedes llevar la cuenta según este ejemplo:

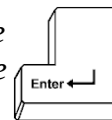
Si el jugador falla una adivinanza: **`errores = errores + 1`**

Si el jugador acierta una adivinanza: **`puntos += 5`**

`total = puntos - errores`



Crea una variable llamada *joya* y guarda en ella el nombre de la piedra preciosa que mas te guste, recuerda utilizar comillas dobles para almacenar una cadena de caracteres dentro de una variable.



Santo y Señá...

Nuestro siguiente objetivo es aprender como repetir órdenes de Ruby, el número de veces que necesitemos para nuestro programa. Existen varias formas de hacerlo, y siguiendo el consejo del joven alquimista usaremos la orden *“While (expresión) do”* que significa repetir mientras la expresión sea cierta (True). Veamos un ejemplo.

```
clave = "_"  
while (clave != "piedrafilosofal") do  
  puts "Introduce la clave..."  
  clave = gets.chomp # teclado  
end  
  
puts "puedes PASAR !"
```

En este ejemplo, aparece por primera vez la almohadilla (#). Este signo se usa cuando el desarrollador quiere hacer un comentario para aclarar el significado de lo que ha escrito en su programa.

Si te has fijado bien, habrás comprobado que hay un orden que se ha querido “colar”. ¿Sabes cuál es? ¡ Exacto ! La orden **gets.chomp** , su función es recoger los datos que el programa pide al usuario, es decir, el número o la palabra que escribimos desde el teclado.

Un Juego Aleatorio

Ya conoces varias órdenes y has practicado varios ejemplos escritos en Ruby. Es el momento de crear tu primer juego. Veamos cómo hacerlo. El juego consiste en adivinar el número secreto. El ordenador va a elegir un número al azar entre 1 y 20.

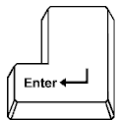
El jugador tendrá cinco oportunidades para adivinar el número. El ordenador te dará pistas. Si no aciertas, el ordenador te dirá si el número que has elegido es mayor o menor que el secreto. Aquí tienes un ejemplo del funcionamiento del juego.

Si la piedra filosofal quieres encontrar, con Ruby tienes que programar

```
Soy el temible pirata Roberts, y tengo un secreto!  
Es un numero entre 1 y 20. Te dare 5 oportunidades  
10  
Tu numero es mayor que el secreto.  
5  
Tu numero es mayor que el secreto.  
3  
Tu numero es menor que el secreto.  
4  
Adivinaste mi numero en 4 intentos!
```

```
-----  
(program exited with code: 0)  
Press return to continue  
■
```

Mi querido amigo, el alquimista, dice que para escribir el código de tu programa necesitarás varias variables para guardar datos. El nombre del jugador, el número secreto, el número de intentos... Además tendrás que usar el operador == para comparar si el número secreto es igual al número que escribe el jugador. Una variable para contar intentos. ¡ A Programar !



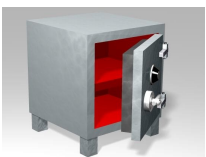
Presta atención, te voy a mostrar como guardar un número al azar en una variable con Ruby:

```
secreto = rand(20)
```

Mostrar en pantalla un mensaje que contiene una variable (Lo necesitarás para el mensaje final de tu juego):

```
puts "Lo has adivinado en #{intentos} intentos !!"
```

Mi Caja Fuerte



Ruby te da la oportunidad de coleccionar nombres, números e incluso piedras preciosas. Existen un tipo de “variable especial” llamada vector. Es una variable que almacena un conjunto de datos. El alquimista nos ha sugerido un ejemplo, para ver como almacenar varios datos en un vector, ya sean números, letras o palabras:

```
mis_piedras = ["Ruby", "Diamante", "Oro", "Plata"]  
mezcla = ("casa", 'M', 75, 'p', 166.186)
```

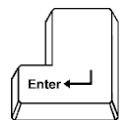
Abre un terminal. Ejecuta la orden irb. Escribe el ejemplo anterior u otro que se te ocurra. Prueba estas órdenes, ¿Para qué sirve cada una de ellas?

```
mis_piedras
```

```
mis_piedras[0]
```

```
mis_piedras[1,3]
```

```
mis_piedras[-1]
```



Si la piedra filosofal quieres encontrar, con Ruby tienes que programar

Una vez que tenemos nuestra caja fuerte definida, podemos realizar varias operaciones sobre ella. Por ejemplo, añadir una piedras más: `mis_piedras.insert(1,'piedra_nueva')` e incluso podemos ordenarla por orden alfabético con la orden: `mis_piedras.sort()`

Por ultimo, para conocer el número de piedras preciosas que contiene nuestra caja fuerte escribiremos lo siguiente:

```
mis_piedras.size
```

De Roca en Roca

La utilidad que tienen las colecciones a parte de guardar múltiples datos, es poder usar esos datos de uno en uno. En el apartado Santo y Señá aprendiste a repetir ordenes en Ruby mediante la sentencia **while**. Existe otra orden en Ruby que se aplica a los elementos de un conjunto de objetos, como el que aparece en el apartado anterior. La sintaxis que se utiliza es la siguiente : `variable.each {ordenes}`

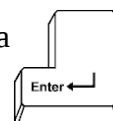
Presta atención a estas instrucciones de ejemplo:

```
variable = [2,4,13,8]
variable.each {|i| puts i}
```

En lugar de las llaves {} del each, se pueden usar las palabras **do** y **end**, y expresar el bloque entre varias líneas. Esto es conveniente cuando lo que va dentro del bloque puede tener una lógica un poco más elaborada que no cabe en una sola línea.

```
variable = [1,2,3,4]
suma = 0
variable.each do |i|
  suma += i
  puts suma
end
```

El alquimista quiere que le ayudes a enseñar las tablas de multiplicar a sus amigos. Usa la orden **each** para crear un programa que imprima en pantalla la tabla de multiplicar del 6. ¿Te atreves a imprimir todas las tablas? Del 1 al 9 ¡Ánimo!



Agrupando Ordenes

Una función es un conjunto de órdenes que actúan conjuntamente para conseguir un objetivo. Puedes usar las funciones para organizar tus programas. Una función tiene un conjunto de órdenes de Ruby. Puedes entender el significado de una función como un pequeño programa que realiza una sola tarea. Las variables almacena datos y las funciones almacenan ordenes de Ruby.

Esta es la sintaxis para crear una función:

```
def mayorDeEdad(anios)
  if anios >= 18
    puts "Eres mayor de edad"
  else
    puts "Eres un pezqueñin"
  end
end
```

Una vez que ya has creado la función, cada vez que necesites usarla solo tendrás que escribir el nombre de la función seguido de paréntesis.

```
#Llamada a la función
mayorDeEdad(12)
```

Paso de argumentos; puede que tu función realice una serie de cálculos, pero para hacer esos cálculos la función necesita uno o varios datos. Puedes crear una función para que calcule la tabla de multiplicar y que esa función reciba un argumento que sea el número de la tabla que quieres calcular.

Crearás muchas funciones que realizaran euros. El resultado de los cálculos puedes cálculos, para obtener el resultado de esos asignar a una variable.

cálculos mediante la función, necesitaras la orden **return**. Imagina una función que le pases como argumento una cantidad en pesetas y te devuelva la misma cantidad en

```
def peseta_a_euro(pesetas):
  return pesetas / 166.386
end
```

Un Objeto Filosofal

En capítulos anteriores has practicado como guardar datos en variables y como guardar órdenes en funciones. El siguiente paso es guardar datos/variables y ordenes/funciones dentro de objetos. Una pelota es un objeto, que tiene unas características determinadas, por ejemplo, color, peso o tamaño. Estas características las podemos almacenar en variables dentro de un objeto pelota en Ruby.

Por otro lado, con una pelota se pueden hacer muchas cosas: lanzarla, inflarla o golpearla. En Ruby reflejaremos estas acciones con una función para cada una de ellas, que esta dentro de nuestro objeto. Las funciones también son conocidas como métodos del objeto o clase.

```
class Pelota
```

```
  def initialize ()
```

```
    color = "rojo"
```

```
    tamaño = 10
```

```
    tamaño = 5
```

```
  end
```

```
  def lanzar(metros)
```

```
  def inflar (aire)
```

```
  def golpear()
```

```
end
```

```
miPelota = Pelota.new
```

```
miPelota.lanzar(15)
```

```
miPelota.inflar(9)
```

La orden para crear un objeto en Ruby es **class**. Además aparece un método especial, se llama **initialize**, a este método se le llama constructor de la clase. Se utiliza para dar un valor inicial a las propiedades de nuestro objeto.

En la columna de la derecha puedes ver las ordenes necesarias para usar nuestro objeto después de haberlo creado. Tendrás que utilizar el nombre de tu objeto seguido de un punto y la propiedad o función que quieras usar. Si te has fijado, los métodos acaban con paréntesis y las propiedades no. Para crear un nuevo objeto de tipo Pelota usaremos el nombre de la clase seguido de `.new` tal como aparece en el ejemplo.

Joyería

En una joyería podemos encontrar una gran variedad de piedras preciosas. En Ruby podemos hacer algo parecido con nuestros objetos. Podemos usar un fichero para escribir todas nuestras clases, a estos ficheros les llamaremos módulos. Cuando necesitemos usar uno de nuestros objetos en un programa usaremos la orden **require** seguida del nombre de nuestro archivo. Por ejemplo para usar funciones relacionadas con las matemáticas se usa el modulo siguiente:

```
require 'Math'
```

Ruby, tiene una gran variedad de módulos que podemos usar en nuestros programas. En próximos capítulos usaremos los módulos Time y Gosu para que puedas crear tus propios videojuegos. Si quieres obtener ayuda sobre un modulo determinado en el terminal escribe la orden `help()`. Después escribe el nombre del modulo. Prueba con el módulo time y la función sleep.

Escribe un programa donde se use esta función.

Si la piedra filosofal quieres encontrar, con Ruby tienes que programar

Este tutorial tiene licencia GFDL, cuyo texto puedes obtener en
<http://gugs.sindominio.net/licencias/gfdl-1.2-es.html>

Agradecimiento a Daniel García Moreno, por su proyecto TBO, utilizado en la edición de este tutorial.

Copyright (c) 2011 Fco Javier Lucena Lucena. Se otorga permiso para copiar, distribuir y/o modificar este documento bajo los términos de la Licencia de Documentación Libre de GNU, Versión 1.2 o cualquier otra versión posterior publicada por la Free Software Foundation; sin Secciones Invariantes ni Textos de Cubierta Delantera ni Textos de Cubierta .