



Ruby Fácil

versión Creative Commons

Diego F. Guillén Nakamura



Reconocimiento-No comercial-Compartir bajo la misma licencia 2.5 Colombia

Usted es libre de:



copiar, distribuir y comunicar públicamente la obra



hacer obras derivadas

Bajo las condiciones siguientes:



Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).



No comercial. No puede utilizar esta obra para fines comerciales.



Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- ♦ Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- ♦ Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- ♦ Nada en esta licencia menoscaba o restringe los derechos morales del autor.

Advertencia

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.
Esto es un resumen fácilmente legible del texto legal (la licencia completa).

Esta obra se hace disponible bajo la licencia cc-by-nc-sa de Creative Commons:
<http://creativecommons.org/licenses/by-nc-sa/2.5/co/>

La versión impresa (completa) del libro, incluye más capítulos, está disponible para la venta, y se consigue en: <http://www.lulu.com/content/1759456>

Logo de Ruby

Derechos Reservados © 2006 de Yukihiro Matsumoto

Se usa bajo la licencia Creative Commons Attribution-ShareAlike 2.5:

<http://creativecommons.org/licenses/by-sa/2.5/>

Contenido

Prefacio.....	v
Organización del Libro.....	v
Agradecimientos.....	vi
Parte I.....	1
Breve Historia de Ruby.....	1
Audiencia	1
Motivación.....	2
Cómo Instalar Ruby en Windows.....	2
Cómo Instalar Ruby en Linux.....	3
Parte II Las Bases.....	5
Lección 1. Ruby Interactivo.....	6
Lección 2. Números.....	8
Lección 3. Funciones Matemáticas Pre-definidas.....	11
Lección 4. Cadenas de Caracteres.....	12
Lección 5. Arreglos (Arrays).....	14
Lección 6. Fechas y Horas.....	16
Lección 7. Hash.....	17
Lección 8. Each y Bloques.....	18
Lección 9. Contadores y Acumuladores.....	20
Lección 10. Expresiones Condicionales.....	22
Parte III El Lenguaje.....	23
Cómo usar SciTE.....	24
Lección 11. Lógica Condicional.....	25
Lección 12. Ciclos Repetitivos.....	27
Lección 13. Condiciones Múltiples (Case).....	31
Lección 14. Funciones.....	33
Lección 15. Clases.....	36
Lección 16. Atributos de Clases.....	39
Lección 17. Control de Acceso a la Clase.....	42
Lección 18. Herencia y Taxonomías.....	44
Lección 19. Expresiones Regulares.....	45
Lección 20. Archivos.....	47
Lección 21. Directorios.....	50
Lección 22. Entrada/Salida, Corrientes y Tubos.....	51
Lección 23. Formatos de salida.....	53
Lección 24. Yield.....	55
Lección 25. Módulos y Mixins.....	56
Parte IV Usos Avanzados (omitido).....	59
Soluciones a Ejercicios Selectos.....	60
Bibliografía.....	73
Libros sobre Ruby.....	73
Otros Enlaces sobre Ruby en Internet.....	74

Curiosidades Matemáticas.....75

Apéndices.....76

Apéndice A. El Código ASCII.....76

Apéndice B. Cómo usar SciTE.....77

Sobre el Autor.....81

Prefacio

Este libro se originó hace varios meses, a través de un intercambio de correspondencia con mi padre, un ingeniero electricista ya retirado, de 77 años, que vive en Colombia, quien me manifestó su interés en aprender algún lenguaje de programación. Yo le sugerí Ruby, pues es un lenguaje popular que considero sencillo, bonito, y elegante. Empezamos así un intercambio de correspondencia por correo electrónico, que poco a poco fue tomando la forma de un libro.

Queremos que llegue a las manos de profesionales y estudiantes, y especialmente, de principiantes, a quienes queremos motivar para que aprendan a programar con Ruby.

Organización del Libro

El libro está organizado en cuatro partes:

- La Parte 1 cubre la instalación y selección de herramientas.
- La Parte 2 introduce los elementos básicos del lenguaje, que en términos técnicos se conoce como “primitivas”; esto incluye los tipos de datos principales, y su uso.
- La Parte 3 describe las estructuras de control, programación por objeto, manejo de archivos y directorios.
- La Parte 4 explica temas que tienen aplicación comercial en la empresa: programación web, interfaces gráficas, reporte y manejo de errores, acceso a bases de datos.

Esta sección se omite en esta versión del libro; los lectores interesados pueden comprar el libro que se encuentra disponible en:
<http://www.lulu.com/content/1759456>

Cada capítulo sugiere algunos ejercicios para poner en práctica lo aprendido. El rango de dificultad oscila desde lo bien básico, como verificar todo el código descrito en la lección, hasta problemas un poquito más sofisticados de aplicación en matemáticas y otros campos. Al final del libro, el lector encontrará las soluciones a casi todos los ejercicios.

Consideramos que la mejor manera de aprender a programar es “aprender haciendo”. Asumimos que el lector tiene un computador en casa (o escuela u oficina), conectado a internet, así que instalará y seguirá las lecciones al lado de la máquina.

El libro tiene una pequeña inclinación hacia las matemáticas, pues consideramos que la programación es “matemática viva”, y esta es la base para Ingeniería de Sistemas. El nivel de matemáticas mínimo requerido es quizás álgebra de colegio, para animar a los estudiantes jóvenes a aprender este tema. Algunos de los ejercicios suponen conocimiento básico de cálculo. De todas maneras, al final del libro, para estudiantes curiosos, hemos puesto una sección de referencias con enlaces a páginas de internet que explican estos temas.

Agradecimientos

Agradezco a mi esposa, Mikiko (en Gold Coast, Australia), por el tiempo que me ha dado para estar sentado a altas horas de la noche, o muy tempranas horas de la mañana, escribiendo este libro.

También agradezco especialmente a mi padre, Jairo (en Cali, Colombia), por servir de “conejillo de indias”, por su paciencia, feedback, y dedicación a este proyecto. A mi madre, Orfa, también hay agradecerle el tiempo que da a mi padre para que “haga las tareas” de Ruby.

A Fritz Schenk (Los Angeles, EEUU), a Esteban Cardona (Bogotá, Colombia), y a mi cuñado Javier Medrano (Buenos Aires, Argentina), gracias a todos por su inspiración, feedback, y ayuda en proyectos de este tipo. A mis hermanas Dra. Margarita (Bogotá, Colombia), y Claudia (Buenos Aires, Argentina), gracias por estar siempre dispuestas a ayudar.

Diego Guillén
Gold Coast, Australia
Diciembre 2007
ruby.facil@yahoo.com

Parte I

Breve Historia de Ruby

El lenguaje Ruby fue inventado por Yukihiro “Matz” Matsumoto, en Japón, en 1995. Siguiendo la tradición de los lenguajes de programación que han sido desarrollados recientemente, y que gozan de popularidad, Ruby es un lenguaje interpretado, gratuito (Open Source), y orientado por objeto. “Matz” admite que se inspiró en los lenguajes Perl y Python, pero Ruby es mucho más orientado por objeto; de hecho, todo en Ruby son objetos.

En Japón, este lenguaje goza de una popularidad mayor que la del lenguaje Python, que también es muy popular en el mundo entero.

Ruby es un lenguaje genérico que se puede utilizar en muchos campos: desde procesamiento de texto y programación web, hasta ingeniería, genética, y programación comercial a gran escala.

La comunidad Ruby sigue creciendo mundialmente, y ahora que también ha salido la plataforma “Ruby on Rails” para programar aplicaciones Web, este lenguaje cobra cada día más seguidores.

Audiencia

Este libro ha sido escrito principalmente para novatos, pero esperamos que los profesionales en sistemas también lo encuentren útil.

En los 1980s, cuando pasamos por la universidad, saber programar un computador era una cosa esotérica, y la gente lo consideraba cosa de “brujos”. Hoy en día esta tecnología se encuentra en todas partes, y hasta los estudiantes de las escuelas primarias están aprendiendo a programar.

Así que este libro es para la gente curiosa, que tiene acceso a Internet, y que tiene interés en saber cómo funcionan las cosas. El único pre-requisito es tener un conocimiento de matemáticas básicas (álgebra, y trigonometría), para poder hacer los ejercicios.

Motivación

Siguiendo la evolución de este lenguaje desde el principio, nos damos cuenta de que su acogida ha sido tremenda en los últimos dos años. Nos bastaría con ver la cantidad de libros que han salido sobre el tema. Como van las cosas, muy pronto el conocimiento de Ruby será necesario para conseguir un empleo en el área de sistemas y computadores.

El objetivo al escribir este libro no es reemplazar ni superar a los otros textos existentes sobre Ruby, sino más bien ayudar a demistificar el tema de la programación y hacerlo accesible a un gran número de personas. Y qué mejor que hacerlo con un lenguaje moderno y bonito, como lo es Ruby.

En una época en que los computadores se hacen cada vez más poderosos, los lenguajes de programación ya no tienen por qué ser orientados solamente hacia la máquina; deben ser orientados hacia los seres humanos que lo van a usar. Ruby ha sido diseñado con el programador en cuenta; como dice Matz: Ruby ha sido diseñado bajo el principio de “menor sorpresa”, es decir, entre menos nos sorprenda, mejor. Ha sido diseñado para seres humanos. Ruby es un lenguaje fácil, elegante y entretenido. Aprendamos a programar en Ruby!

Cómo Instalar Ruby en Windows

Para Windows, bajar a Ruby desde el siguiente sitio:

<http://www.ruby-lang.org/en/downloads/>

El instalador de Ruby se llama:

`Ruby_x.y.z_One-Click_Installer`

(donde x.y.z son números de versión;

al escribir este documento, estos números eran 1.8.6)

El 25 de Diciembre de 2007 salió la versión 1.9 para desarrollo (“development”) e incluye algunas adiciones experimentales al lenguaje, preparándolo para la versión grande futura 2.0. Puesto que este libro es introductorio, vamos a concentrarnos en la versión 1.8.6, que es la versión oficial, y la más estable.

Después de que haya bajado, ejecutar el instalador, y aceptar todos los defaults, hasta ver el botón de [Finish]. Asumimos que queda instalado en el directorio: [c:\ruby](#)

Cómo Instalar Ruby en Linux

Para instalar Ruby con Ubuntu (o Debian), la distribución de Linux más popular, la forma más fácil es escogerlo e instalarlo con el instalador Synaptic:

System > Administration > Synaptic Package Manager.

Alternativamente, ejecutar el siguiente comando desde una consola:

```
sudo apt-get install ruby irb rdoc
```

Esta página se ha dejado intencionalmente en blanco.

Parte II Las Bases

Para las lecciones de esta sección, se asume lo siguiente:

1. Que Ruby ya ha sido instalado propiamente, y,
2. que vamos a usar el comando de Ruby Interactivo, `irb`.

En este conjunto de lecciones vamos a examinar el uso de variables y estructuras de datos primitivas (es decir, sencillas) en Ruby, a saber:

- números (enteros, reales, racionales y complejos),
- las funciones matemáticas del módulo `Math`,
- arreglos (arrays),
- hashes,
- fechas y horas,
- cadenas de caracteres (strings),
- bloques,
- contadores y acumuladores, y,
- expresiones condicionales.

Lección 1. Ruby Interactivo

Ruby interactivo es un medio ambiente de ejecución en el que se pueden observar los valores que van tomando las variables, y se pueden ejecutar instrucciones del lenguaje línea por línea. Es ideal para aprender a programar, y para entender lo que pasa dentro de la máquina.

Para empezar a Ruby interactivo, hacer lo siguiente:

En Windows, abrir una ventana de comando:

```
[Inicio] → Ejecutar... → cmd → [Enter]
```

El comando para correr Ruby interactivo es: **irb** [Enter]

En Linux, abrir una consola y escribir: **irb**

Se abre una pantalla, como en la Figura 1.

```
$ irb
irb(main):001:0> a = 5      # comentario para explicar el codigo;
=> 5                       # lo que sale despues de => es el resultado
irb(main):002:0> b = 7      # de la evaluacion hecha por Ruby
=> 7
irb(main):003:0> a+b
=> 12
irb(main):004:0> a-b
=> -2
irb(main):005:0> a*b
=> 35
irb(main):006:0> a**b
=> 78125
irb(main):007:0> quit
$
```

Figura 1. Una sesión con Ruby interactivo (irb)

El signo # se usa para indicar que lo que sigue es un comentario que no debe ser interpretado por Ruby. Se puede poner al principio de la línea, o después de la instrucción, seguido de una explicación de lo que estamos haciendo.

Para terminar la ejecución de irb, escribir: quit o exit.

Ruby es un lenguaje de programación moderno. En ésta, y en las lecciones que siguen, veremos cómo cobran vida las matemáticas con Ruby.

Empezamos con una asignación a una variable.

```
a=5
```

Así se guarda el valor 5 en una variable 'a'. El signo "=" se usa en las asignaciones, pero no tiene el mismo significado que "igualdad" en matemáticas, sino más bien es como decir: "guarde el resultado de la expresión de la derecha en la variable de la izquierda". Es decir, se escribe "a=5", pero se interpreta como "a ← 5", como meter un número dentro de una cajita. Esto es importante recordarlo, para más adelante cuando estudiemos igualdad y hagamos comparaciones.

Por convención, en Ruby usamos palabras que empiezen con letras minúsculas, para designar variables. Los siguientes son algunos nombres aceptables para variables:

```
laCaja, unBanco, voltaje.
```

También, por convención, usaremos palabras en mayúsculas para designar constantes. Por ejemplo:

```
PI = 3.14159
```

La expresión compleja debe quedar a la derecha y la variable debe quedar sola a la izquierda (del signo =).

Tomar en cuenta que la aritmética de números enteros es diferente que la aritmética de números reales. Es decir, las siguientes dos asignaciones dan resultados diferentes:

```
a = 3 / 2          #→ 1
b = 3.0 / 2.0      #→ 1.5
```

También se pueden hacer asignaciones con palabras, o “cadenas de caracteres”, como se les conoce en el lenguaje técnico:

```
nombre = "Augustus Caesar"
saludos = "Salut, " + nombre
```

En este caso, el operador "+" funciona como concatenador (pega una cadena de letras con otra), produciendo:

```
saludos = "Salut, Augustus Caesar"
```

El comando "puts" (put string) imprime un mensaje, o el contenido de una variable:

```
puts saludos          #→ "Salut, Augustus Caesar"
```

Con esto básico, ya podemos empezar a explorar el lenguaje Ruby.

Ejercicio 1.1: Ensayar las operaciones básicas: + (suma), - (resta), * (multiplicación), / (división), ** (potencia). Entrar una asignación por línea. Las asignaciones pueden ser complejas y se pueden usar paréntesis; tales como en este ejemplo: (8 * 7) ** 2 + (48 / 3).

Lección 2. Números

Ruby maneja varios tipos de números: enteros, reales, racionales, y complejos.

De la lección anterior, recordemos que la aritmética de números enteros es diferente a la de números reales. Es decir, las siguientes dos asignaciones dan resultados diferentes:

```
a = 3 / 2      #→ 1      # division entera ignora el sobrante
b = 3.0 / 2.0  #→ 1.5
```

Los números enteros se expresan como una cadena de dígitos numéricos, precedidos, o no, por un signo. Ejemplo: -15, 0, 3000.

Otras formas de expresar números enteros son las siguientes:

```
123_456      # equivale a 123456;
              # el "underscore" _ facilita la lectura
0377         # octales (base 8) empiezan con cero
0xff         # hexadecimales (base 16) empiezan con 0x
0b1011       # binarios (base 2) empiezan con 0b
```

Los números reales, también llamados “números de punto flotante”, incluyen un punto decimal entre varios dígitos numéricos.
Ejemplos: -32.68, 4500.651

Otras formas de expresar números de punto flotante son las siguientes:

```
1.0e6        # equivale a 1.0 * 10**6; notacion cientifica
4e20         # equivale a 4 * 10**20; omite el punto decimal y cero
```

La verdad es que Ruby maneja cada número como si fuera un objeto, de manera que se le puede aplicar una función que actúa sobre ese tipo de número.

Algunas de las funciones que actúan sobre los números enteros son las siguientes:

```
i = 5          #→ 5      # definamos un entero
i.to_s         #→ "5"    # convierte a cadena (string)
i.to_f         #→ 5.0    # convierte a real (floating)
i.next         #→ 6      # sucesor (n+1)
j = 7          #→ 7      # definamos otro entero
i == j         #→ false  # comparacion por igualdad (doble signo =)
                # compara el contenido de i con j
j % 5          #→ 2      # division por modulo; retorna el sobrante
```

Algunas de las funciones que actúan sobre los números reales son las siguientes:

```

r = -3.14          #→ -3.14
r.to_i             #→ -3    # convierte a numero entero (integer)
r.to_s             #→ "-3.14" # convierte a cadena (string)
r.abs              #→ 3.14   # retorna el valor absoluto
s = 3.14           #→ 3.14
r == -s            #→ true   # comparacion por igualdad
r.ceil             #→ -3     # el entero inmediato superior a r (ceiling)
r.floor            #→ -4     # el entero inmediato inferior a r (floor)
r.round            #→ -3     # redondeo al mas proximo entero
(r/0.0).finite?    #→ false  # prueba si la operacion es finita
(r/0.0).infinite?  #→ -1    # el resultado tiende a -infinito
(0/r).zero?        #→ true   # prueba si la operacion da cero o no
-r % 3.0           #→ 0.14   # division por modulo; retorna el sobrante
0 == 0.0           #→ true   # como era de esperarse

```

Los números enteros se pueden usar para generar letras del código ASCII:

```
65.chr      → "A"
```

Los números racionales son los que se pueden expresar como cociente de dos enteros. Para este efecto, Ruby tiene un módulo 'rational', que debemos incluir antes de poder usar sus definiciones:

```

require 'rational'
r = Rational(2, 3) #→ Rational(2,3)
s = Rational(3, 4) #→ Rational(3,4)
r + s              #→ Rational(17,12)
r * 3              #→ Rational(2, 1)
r * 2.0            #→ 1.3333333333333333    # racional por real da real
r <=> s            #→ -1                     # r es menor que s
Rational(391,493) #→ Rational(23, 29)       # los reduce
r.to_f             #→ 0.6666666666666667    # convierte a real
s == 0.75          #→ true                  # como es de esperarse

```

Ruby también tiene un módulo especial para manipular números complejos, 'complex'. Los números complejos son pares de la forma $a + ib$, donde a y b son reales, e i es el número imaginario $\sqrt{-1}$:

```

require 'complex'
x = Complex(1,0) #→ Complex(1,0)
y = Complex(0,1) #→ Complex(0,1)
z = x+y          #→ Complex(1,1)
z.conjugate      #→ Complex(1, -1)
x*y              #→ Complex(0,1)
i = Complex::I   #→ Complex(0,1)          # la constante i esta definida
i*i == -1        #→ true
z.abs            #→ 1.4142135623731         # radio del vector (1,1)
z.arg            #→ 0.785398163397448       # angulo del vector (1,1)
z.arg * 180 / Math::PI #→ 45.0            # el mismo angulo en grados
z.polar          #→ [1.4142135623731, 0.785398163397448]
# lo anterior retorna un arreglo con el numero complejo expresado en
# coordenadas polares: [ valor absoluto (z.abs), argumento (z.arg)]

```

Ejercicio 2.1: Verificar todas las operaciones anteriores.

Ejercicio 2.2: Calculemos π (π) por varios métodos numéricos:

Para este ejercicio, que involucra números reales, debemos escribir por lo menos uno de los números en forma real (decimal).

Ejercicio 2.2.1: Hiparco calculó π en el siglo II AC como $377 / 120$.

Ensayarlo con irb.

Ejercicio 2.2.2: Arquímedes de Siracusa predijo que π estaba entre $3 \frac{10}{71}$ y $3 \frac{1}{7}$, considerando un polígono de 96 lados. Expresar estos números con irb.

Ejercicio 2.2.3: En China, Liu Hui (263 DC) predijo que π estaba cerca de $355 / 113$. Expresar estos números con irb.

Ejercicio 2.2.4: Fibonacci (Leonardo de Pisa) calculó π como $864 / 275$.

Expresar estos números con irb.

Ejercicio 2.3: Escribir una ecuación que convierta temperatura de Fahrenheit a Centígrados.

Ejercicio 2.4: Escribir una ecuación que convierta temperatura de Centígrados a Fahrenheit.

Ejercicio 2.5: Escribir una ecuación que convierta de grados a radianes.

Ayuda: 2π radianes equivale a 360 grados.

Ejercicio 2.6: Escribir una ecuación que convierta de radianes a grados.

[Algunas soluciones a los ejercicios se encuentran al final del texto]

Lección 3. Funciones Matemáticas Pre-definidas

Las funciones matemáticas se encuentran agrupadas en un "módulo" separado llamado "Math". Por eso, para invocarlas, hay que preceder el nombre de la función con el nombre Math, y poner un punto entre ambas:

```
Math.sqrt(25)           #→ 5           #resultado
```

Ejercicio 3.1: Ensayar las siguientes funciones elementales:

sin(), cos(), tan(), exp(), log(), log10().

Ejercicio 3.2: Calcular la hipotenusa de un triángulo rectángulo usando la función Math.hypot(x,y).

Ejercicio 3.3: La ecuación algebraica $ax^2 + bx + c = 0$ tiene soluciones

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad . \text{ Calcular x con irb, cuando } a=3, b=6, y, c=2.$$

Ejercicio 3.4: Ensayar las funciones trigonométricas inversas:

acos(), asin(), atan().

Ejercicio 3.5: Ensayar las funciones hiperbólicas:

sinh(), cosh(), tanh().

Ejercicio 3.6: Euler usó estas fórmulas para calcular π (expresarlas y evaluarlas en irb):

$$\pi / 4 = 5 \arctan(1/7) + 2 \arctan(3/79)$$

$$\pi / 4 = 2 \arctan(1/3) + \arctan(1/7)$$

Ejercicio 3.7: En 1961, Wrench y Shanks de IBM calcularon π a 100,200 dígitos usando la siguiente fórmula (les tomó 8 horas en un computador IBM 7090 de la época):

$$\pi = 24 \arctan(1/8) + 8 \arctan(1/57) + 4 \arctan(1/239)$$

Ejercicio 3.8: Confirmar que la constante π de Ruby, expresada como Math::PI, tiene un valor que corresponde a varias de las fórmulas anteriores. A cuáles?

Otro valor también definido en el módulo Math es el de la constante exponencial e (tiene muchos usos en ingeniería y en matemáticas, pues es la base de los logaritmos naturales) $e = \text{Math}::E \quad \# \rightarrow 2.71828182845905$

Lección 4. Cadenas de Caracteres

Otra estructura de datos importante en programación son las cadenas de caracteres, o “Strings”, en inglés. Con frecuencia se usa la palabra “cadenas” simplemente. Las cadenas se usan para representar palabras y grupos de palabras.

Las cadenas se escriben entre comillas, simples o dobles:

```
cadenal = "esta es una cadena de caracteres\n"
cadena2 = 'y esta es otra'
```

La diferencia es que las cadenas que usan comillas dobles pueden incluir caracteres especiales tales como `\t` (tab), `\n` (carriage return), y números en diferentes representaciones (octales, `\061`, hexadecimal, etc).

Como se vio en la lección anterior, los enteros y los reales también se pueden convertir a cadenas:

```
i = 5           #→      5           # valor entero
i.to_s         #→      "5"         # cadena
r = 3.14        #→      3.14        # valor real
r.to_s         #→      "3.14"      # cadena
```

Alternativamente, las cadenas se pueden convertir a números:

```
"123".oct      #→      83          # cadena octal a entero
"0x0a".hex     #→      10          # cadena hexadecimal a entero
"123.45e1".to_f #→      1234.5      # convierte a real
"1234.5".to_i  #→      1234        # convierte a entero
"0a".to_i(16)  #→      10          # convierte desde hexadecimal
"1100101".to_i(2) #→      101      # convierte desde binario
```

La notación especial `#{}` reemplaza el valor de una variable dentro de una cadena:

```
edad = 25
"La edad es #{edad}"           #→ "La edad es 25"
"La edad es " + edad.to_s      # expresión equivalente
```

Algunas de las funciones que aplican a cadenas son las siguientes:

```
s = "ja"       #→ "ja"
s * 3          #→ "jajaja"    # concatena tres copias
s + "ji"       #→ "jajji"     # el signo + concatena cadenas

z = "za"       #→ "za"
z == s         #→ false       # el contenido es distinto

y = "ab"       #→ "ab"
y << "cde"     #→ "abcde"     # append: concatena al final
y.length       #→ 5           # el número de caracteres
```

Funciones relacionadas con letras minúsculas y mayúsculas:

```
y.upcase      #→ "ABCDE"
"MN".downcase #→ "mn"
"Az".swapcase #→ "aZ"
"sam".capitalize #→ "Sam"
```

Las letras de la cadena se pueden acceder individualmente poniendo un índice entre paréntesis rectangulares [idx]: (el primer elemento tiene índice cero)

```
y[0]    #→ 97    #retorna el código del caracter en la posición 0, "a"
y[1]    #→ 98    #retorna el código del caracter en la posición 1, "b"
```

Al asignar un valor a una cadena con índice, se cambia el caracter en esa posición:

```
y[0] = "z"      #→      "z"
y      #→      "zbcde"
```

Para insertar y borrar letras de la cadena:

```
y.delete("z")    #→      "bcde"
y.insert(2,"x")  #→      "zbxcd"
y.reverse        #→      "edcbz"
```

Para seleccionar parte de la cadena, o una subcadena, se puede especificar un rango:

```
n = "0123456789"
n[5...n.length] # subcadena desde el quinto elemento hasta el final
#→ "56789"
```

Alternativamente, se puede especificar el primer elemento de la subcadena, y la cantidad de elementos deseados (un número mayor devuelve toda la subcadena hasta el final):

```
n[5, 3] # subcadena desde el quinto elemento, y 3 caracteres de longitud
#→ "567"
n[5, n.length] # subcadena desde el quinto elemento, hasta el final
#→ "567890"
```

Ejercicio 4.1: Verificar todas las operaciones anteriores con Ruby interactivo.

Lección 5. Arreglos (Arrays)

El concepto de “arrays” en inglés, o “vectores”, o “arreglos”, consiste un conjunto de datos discretos (números o palabras), todos bajo el nombre de una variable, de manera que puedan después accesarse por índice.

```
a = ["Juan", "da", "a", "Maria"]
a.size  #→ 4      # el número de elementos en la lista
```

`a[i]` nos retorna el (i-1)-ésimo elemento de la lista. Por eso decimos que tiene base 0 (el primer elemento tiene índice 0, el segundo 1, etc).

```
a[3]          #→ "Maria"
```

Para alterar una lista, podemos añadir elementos al final:

```
b= "un libro azul".split #parte la cadena en un arreglo
a.concat(b) #→ ["Juan","da","a","Maria","un","libro","azul"]
```

Otras funciones prácticas con arreglos:

```
a.first      #→ "Juan"      # accesa el primero
a.last       #→ "azul"      # accesa el ultimo
a.empty?     #→ false      # pregunta si esta vacio
```

Un arreglo complejo puede tener otros arreglos adentro:

```
c = [1, 2, 3, [4, 5, [6, 7, 8], 9], 10]
```

Para “aplanarlo” usar la función `flatten`:

```
c.flatten    #→ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Para insertar elementos se usa la función `insert`(índice, valor):

```
c.insert(3, 99) #→ [1, 2, 3, 99, 4, 5, 6, 7, 8, 9, 10]
```

Para invertir el orden de un arreglo, se usa la función `reverse`:

```
["a", "b", "c"].reverse #→ ["c", "b", "a"]
```

Para ordenar un arreglo, usar la función `sort`:

```
["p", "z", "b", "m"].sort      #→ ["b", "m", "p", "z"]
```

Pop y push, remueven y añaden (respectivamente) elementos hacia el final del arreglo:

```
d=["Mozart", "Monk"]
d.pop                #→ "Monk"
d                    #→ ["Mozart"]
d.push("Bach")       #→ ["Mozart", "Bach"]
```

Para insertar y remover elementos hacia el comienzo del arreglo, usar unshift y shift, respectivamente:

```
d.unshift("Beethoven") #→ ["Beethoven", "Mozart", "Bach"]
d.shift                #→ "Beethoven"
d                      #→ ["Mozart", "Bach"]
```

También se pueden insertar elementos al final del arreglo con <<:

```
d << "Listz"          #→ ["Mozart", "Bach", "Listz"]
```

Para concatenar los elementos de un arreglo produciendo una cadena:

```
["p", "z", "b"].join(", ") #→ "p, z, b"
```

Ejercicio 5.1: Verificar todo lo anterior con Ruby interactivo.

Lección 6. Fechas y Horas

Todos los lenguajes de programación tienen una forma especial de manipular fechas, pues esta es una de las tareas cotidianas en las oficinas.

Ruby tiene un módulo llamado “Time” para manipular fechas y horas.

La fecha y hora corriente se obtienen con la función now:

```
t = Time.now      #→ "Mon Sep 10 23:11:06 +1000 2007"
```

El resultado es una cadena de letras y números, que se pueden después acceder individualmente con otras funciones adicionales:

```
t.day           #→ 10
t.month         #→ 9
t.year          #→ 2007
t.hour          #→ 23
t.min           #→ 11
t.sec           #→ 6
t.wday          #→ 1           # primer día de la semana
t.yday          #→ 253        # día del año
t.strftime("%B") #→ "September" # nombre del mes completo
t.strftime("%b") #→ "Sep"      # idem abreviado
t.strftime("%A") #→ "Monday"   # día de la semana
t.strftime("%a") #→ "Mon"      # idem, abreviado
t.strftime("%p") #→ "PM"       # AM o PM
```

Una aplicación práctica, comúnmente usada, es crear dos variables de tiempo a intervalos diferentes, y luego sacar la diferencia, para ver cuánto tiempo se ha demorado la ejecución de un programa.

Para crear una fecha nueva, se puede usar la función mktime():

```
vacacion = Time.mktime( 2007, "dec", 31, 1, 15, 20 )
#→ "Mon Dec 31 01:15:20 +1000 2007 "
```

La clase Date sirve para especificar fechas, que no requieren el uso del tiempo.

```
require 'date'
fecha = Date.new(2007, 06, 15)
fecha.to_s           #→ "2007-06-15"
hoy = Date.today
puts "#{hoy.day}/#{hoy.month}/#{hoy.year}" #→ "17/12/2007"
```

Ejercicio 6.1: Verificar todas las operaciones anteriores con Ruby interactivo.

Lección 7. Hash

Existe otra estructura de datos bastante conveniente llamado el “Hash”.

Consiste en un arreglo que permite almacenar valores indexados por palabras (a diferencia del arreglo, que indexa los valores por números naturales).

Así se usaría un hash que empieza vacío, y poco a poco se le van añadiendo elementos:

```
genios = Hash.new          # declara un hash nuevo, la variable genios
genios["matematicas"] = "Gauss" # "matematicas" indice, "Gauss" valor
genios["violin"] = "Paganini"
```

Normalmente se usa un hash para almacenar pares de cosas relacionadas entre sí, tales como las palabras de un diccionario y sus definiciones, o las veces que ocurre cada palabra en un texto, por ejemplo.

Un hash se puede inicializar con todos los valores desde el principio:

```
lang= {                                # declara un hash, la variable lang
  'Perl' => 'Larry',                  # relaciona lenguajes con sus autores
  'Python' => 'Guido',                # el signo "=>" separa llave de valor
  'Ruby' => 'Matsumoto'
}

lang['Ruby']                           #→ 'Matsumoto' #accesa un valor
lang.has_key?('Java')                  #→ false #pregunta si existe llave
lang.has_value?('McCarthy')            #→ false #pregunta si existe valor
lang.sort                              # ordena por orden de llave (key)
lang.values_at('Python', 'Ruby')       #→ ['Guido', 'Matsumoto']
lang.values                            #→ ['Larry', 'Guido', 'Matsumoto']
lang.keys                             #→ ['Perl', 'Python', 'Ruby']
lang.length                           #→ 3 #numero de pares en el hash
```

Para reemplazar pares existentes, con valores nuevos, simplemente hacer asignaciones con índice y valor:

```
lang['Ruby'] = 'Matz'                  # reemplaza 'Matsumoto' con 'Matz'
lang['COBOL'] = 'CODASYL'              # agrega un nuevo par al hash
```

Para eliminar pares que ya no se necesitan, usar la función delete:

```
lang.delete('COBOL')                  # elimina el par 'COBOL'=>'CODASYL'
```

Ejercicio 7.1: Verificar todas las operaciones anteriores con Ruby interactivo.

Lección 8. Each y Bloques

Antes de seguir con cosas más complicadas, aprendamos cómo “iterar” o visitar los elementos de un Array o un Hash, uno por uno, para hacer algo con ellos.

Hay una función llamada “each” (“cada uno”), que se puede aplicar a un Array o a un Hash, o a un string de varias líneas, para iterar sobre sus elementos. La sintaxis es (notar el uso de llaves {}):

```
variable.each {bloque}
```

El resultado es el siguiente: Las instrucciones que se encuentran dentro del bloque, se van a aplicar a cada uno de los elementos de la variable. Es decir, se produce una enumeración, dentro de un ciclo repetitivo donde en cada pasada, o iteración del ciclo, se evalúa uno de los elementos del array (o hash), en forma sucesiva, empezando con el primero, luego el segundo, etc, hasta llegar al último.

Veamos un ejemplo con un Array numérico:

```
a = [2,4,13,8]          # empecemos con este Array
a.each {|i| puts i}     # por cada elemento |i| de 'a', imprimirlo:
#→2                    # dentro del bloque, i+2 y luego puts 2
#→4                    # sigue con i+4 y luego puts 4
#→13                   # etc
#→8
```

Ahora veamos cómo se aplica el mismo concepto a un Hash. Recordemos que el Hash almacena pares de elementos: {llave => valor}.

```
b = {'sol'=>'dia', 'luna'=>'noche'} # empecemos con un hash
b.each {|k,v| puts k + " : " + v} # para cada par |k, v| de b
#→sol : dia                      # k←'sol', v←'dia', e imprime el 1er par
#→luna : noche                   # k←'luna', v←'noche', e imprime el 2do par
```

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.

Veamos un ejemplo aplicado a una cadena de varias líneas. El siguiente programa procesa y cuenta las líneas de un cuento, una línea a la vez:


```

lineas = "Erase una vez\nen un lugar lejano...\nFin\n"
num = 0
lineas.each do |linea|      #usa do al principio del bloque
  num += 1                 # el contenido del bloque va de por medio
  print "Line #{num}: #{linea}" # se extiende sobre varias líneas
end                        #usa end al final del bloque

```

El significado de la expresión `num += 1` lo estudiaremos en la lección siguiente.

Ejercicio 8.1. Verificar lo que ocurre con las expresiones listadas en esta lección. Cambiar los valores del arreglo `a`, hash `b` o cadena `linea`, hasta asegurarse cómo funciona esa lógica.

Ejercicio 8.2. Usar Ruby interactivo para calcular los cuadrados y la raíz cuadrada de los números de un Array.

Ejercicio 8.3. Usar Ruby interactivo para calcular la longitud de las palabras almacenadas en un Array.

Lección 9. Contadores y Acumuladores

Otro tema útil es el de "contadores" y "acumuladores". Se trata de variables que se utilizan para llevar cuentas, tales como sumas, o para contar el número de veces que ocurren las iteraciones. Este tema no es específico a Ruby, pues en los otros lenguajes de programación se hace lo mismo.

El primer tipo de variable es el "contador". La expresión contadora siguiente es bastante común:

```
cuenta += 1
```

Esto es equivalente a decir:

```
cuenta = cuenta + 1
```

Recordemos se trata de una asignación, no de una igualdad. Así que quiere decir lo siguiente, leyendo de derecha a izquierda:

1. primero, se computa: `cuenta + 1`,
2. y luego se asigna el resultado a: `cuenta`.
3. el efecto final es que su contenido se ha incrementado en 1.

[Recordemos un concepto fundamental: que en una asignación, siempre la expresión de la izquierda del signo "=" esta sola, es decir, es el nombre de la variable que recibe el resultado de la expresión que se calcula a la derecha del signo "="]

El contador se usa para llevar la cuenta dentro de un ciclo.

Por ejemplo, si queremos saber cuántos elementos tiene un arreglo 'a':

```
cuenta = 0 # hay que iniciar el contador en 0
a.each { cuenta += 1 } # cuenta dentro del ciclo
puts "tiene #{cuenta} elementos" # e imprimimos el resultado
```

El ejemplo anterior es solamente para ilustrar el uso del contador, pues, como ya vimos en una lección anterior, en Ruby bastaría con hacer `a.size` para saber el número de elementos de un arreglo.

De manera similar se podría pensar en un contador que contara hacia abajo:

```
count -= 1
```

Otra expresión relacionada, parecida a un "contador", es un "acumulador". Es una variable que suma cantidades arbitrarias. La expresión a usar es la siguiente:

```
suma += cantidad # equivale a suma ← suma + cantidad
```

Habrán notado que es parecida al contador, excepto que en lugar de sumar 1, suma ciertas cantidades arbitrarias que se le asignan desde la derecha.

Veamos un ejemplo para sumar los elementos de un arreglo 'a':

```
a = [13, 17, 19, 29]          # inicia un array con numeros
suma = 0                      # inicia la suma en 0
a.each { |i| suma += i }      # cada elemento i se acumula en suma
puts "el total es #{suma}"    # produce el resultado
```

Los operadores *= y /= tienen efectos parecidos: acumulan productos, y divisiones, respectivamente.

Ejercicio 9.1: Verificar las expresiones citadas.

Lección 10. Expresiones Condicionales

Ciertas expresiones en Ruby resultan en valores de verdad:

true (verdadero) o false (falso).

Por ejemplo, las siguientes expresiones (en forma de comparación) evalúan a un valor de verdad:

```
a == b, x <= z, Math.sqrt(b) > c
```

Los operadores pueden ser los siguientes:

== (igualdad), < (menor), > (mayor),

<= (menor o igual), >= (mayor o igual).

El operador especial, <=>, resulta en -1, 0, 1, dependiendo de si el operando de la izquierda es menor, igual, o mayor que el de la derecha, respectivamente:

```
5 <=> 7          #→ -1
13 <=> 13         #→  0
7 <=> 3          #→  1
```

Alternativamente, el resultado de la evaluación de una función puede resultar en un valor de verdad. Ejemplos:

```
h.empty?, (r/q).finite?
```

Hay ocasiones en las que estas expresiones pueden llegar a ser compuestas:

```
expr1 or expr2, expr1 and expr2
```

Las siguientes tablas muestran los valores de verdad en estos casos:

```
nil and true      #→ nil
false and true    #→ false # falso y cualquier cosa, da falso
99 and false      #→ false
99 and nil        #→ nil
true and true     #→ true  # and es cierto cuando ambos son ciertos
false or nil      #→ nil
nil or false      #→ false
99 or false       #→ 99
true or false     #→ true  # or es cierto cuando alguno es cierto
not true          #→ false
not false         #→ true
```

Ejercicio 10.1: Verificar las expresiones citadas.

Ejercicio 10.2: Inventarse y evaluar expresiones de valores de verdad.

Parte III El Lenguaje

Para los ejercicios de las siguientes lecciones es necesario usar un editor de texto para guardar los programas que vamos a escribir en archivos.

Si bien es posible usar editores sencillos como el Notepad de Windows, es mejor usar editores especializados para programadores, pues estos tienen funciones especiales tales como indentación automática, macros, y sintaxis de colores. Todo esto ayuda a escribir programas más rápidamente y con menos errores.

Recomendamos usar [uno de] los siguientes editores:

- vim (VI improved): permite editar muchos archivos al mismo tiempo, y tiene sintaxis de colores para más de 100 lenguajes de programación. Es muy popular en Linux, pero su dificultad radica en aprender a usar los comandos internos, que son una combinación de teclas, pues trata de hacer todo con el teclado, en vez de usar menús gráficos.
<http://www.vim.org/>
- RDE (Ruby Development Environment): editor integrado de Ruby que permite hacer debugging y ejecutar los programas directamente.
<http://homepage2.nifty.com/sakazuki/rde/en/>
- SciTE (SCIntilla based Text Editor): viene incluido con Ruby en la instalación para Windows. Puede reconocer más de 60 lenguajes populares, incluyendo Ruby, Python, HTML, XML, Tcl/Tk, Scheme, SQL, etc.
 - En Windows, SciTE se encuentra ya instalado en el directorio: `c:\ruby\scite`.
 - En Ubuntu Linux, es fácil instalarlo con el Synaptic Package Manager.
 - Para otros sistemas, se puede bajar desde aquí:
<http://www.scintilla.org/>

Por conveniencia, también vamos a asumir que todos los programas los vamos a guardar bajo el directorio siguiente:

```
c:\code\ruby
```

Y si estáis usando Linux, el directorio es `~/code/ruby`.

[Asumimos que el lector sabe cómo crear estos directorios, si no existen]

Cómo usar SciTE

SciTE es suficientemente fácil e intuitivo, así que para este curso vamos a recomendarlo.

- Para guardar un programa en un archivo, hacer: `File > Save`.
- Para ejecutar un programa, hacer: `Tools > Go` [o presionar F5].
- SciTE normalmente no imprime resultados parciales; para examinar el estado de una variable, imprimirla con `puts` o `print`.

Para más información sobre el uso de SciTE, favor referirse al apéndice B, al final del libro.

Lección 11. Lógica Condicional

Todos los lenguajes de programación tienen una forma similar de controlar el flujo del programa por medio de condiciones. En Ruby, esto se logra con la estructura de control `if/elsif/else/end`.

Esta tiene la siguiente forma genérica:

<code>if expr1 [then :]</code>	si la expresión <i>expr1</i> es verdadera,
<code> bloque1</code>	entonces se evalúa el <i>bloque1</i> ;
<code>[elsif expr2 [then :]</code>	alternativamente, si <i>expr2</i> es cierta,
<code> bloque2]</code>	entonces se evalúa el <i>bloque2</i> ;
<code>[else</code>	alternativamente,
<code> bloque3]</code>	se evalúa el <i>bloque3</i> .
<code>end</code>	

En la expresión anterior, solamente uno de los bloques: *bloque1*, *bloque2* o *bloque3*, se ejecutan, dependiendo de los valores de verdad de *expr1* y *expr2*, respectivamente.

La palabra `then` se puede omitir, y, en vez, se pueden usar dos puntos, “:”, aunque también son opcionales.

La expresión `elsif expr2 ... bloque2` es opcional. La expresión `else bloque3` también es opcional. Es decir, si no se van a necesitar, se pueden omitir.

Veamos algunos ejemplos que ilustran usos específicos de esta estructura lógica.

Para comparar dos números, *a* y *b*, escribiríamos lo siguiente:

<code>if a > b</code>	<code>#caso a>b</code>
<code> puts "a es mayor que b"</code>	
<code>elsif a == b</code>	<code>#caso a==b</code>
<code> puts "a es igual a b"</code>	
<code>else</code>	<code>#caso a<b</code>
<code> puts "b es mayor que a"</code>	
<code>end</code>	

Hay también otras formas compactas de expresar condiciones sencillas:

```
x = a > b ? 1 : 0      # retorna x=1 si a>b;
                        # alternatively retorna x=0.
puts "a es mayor que b" if a>b
print total unless total.zero?
```

Ejercicio 11.1: Escriba un programa que compare tres números: a, b, c.

Ejercicio 11.2: Escriba un programa que saque el mínimo, y el máximo de los números de un arreglo.

Ejercicio 11.3: Imprima los números pares que se encuentren en un arreglo.
Ayuda: usar la división por módulo (%).

Lección 12. Ciclos Repetitivos

El siguiente paso lógico es estudiar las diferentes formas de repetir bloques de código. En la lección 8 estudiamos un caso especial, que es el de la función `each`, para iterar sobre los elementos de un array:

```
a = ["a","b","c"]
a.each { |v| puts v}      # resulta en a, b, c en cada línea separada
```

Otra manera de conseguir el mismo resultado es con la expresión `for/in`:

```
for e in a                # por cada elemento e de a, haga
  puts e                  # imprima el valor de e
end                        # fin del ciclo
```

Cuando queremos repetir un ciclo cierto número de veces, usamos un rango, donde escribimos los límites de la iteración separados por dos puntos, así:

```
for i in 5..10            # repetidamente va a asignar valores
  puts i                  # desde i=5 hasta i=10 y los va a imprimir
end
```

Cuando usamos tres puntos, excluye el último valor del rango. Esto es útil para iterar sobre los valores de un array (cuyo último índice es `n-1`):

```
a = ["a","b","c"]
for i in 0...a.size       # size=3; asigna valores i entre 0..2
  puts "#{i}:#{a[i]}"     # evalúa desde a[0] hasta a[2] y los imprime
end                       # imprime 0:a, 1:b, 2:c (un par por línea)
```

Cuando sabemos exactamente el número de repeticiones, podemos usar `times`, que es más fácil:

```
5.times do
  puts "hola"
end
```

Otra manera, usando la función `upto()` sobre los enteros:

```
1.upto(5) do
  puts "Hola"
end
```

También existe la función correspondiente `downto()` sobre los enteros:

```
5.downto(1) do |i|
  puts "#{i}:Hola" # resulta en 5:Hola, 4:Hola, etc por cada linea
end
```

Una cuenta que incremente por números diferentes de 1 se puede hacer con la función `step()`:

```
2.step(10, 2) do |i|      # desde 2 hasta 10, incrementando de a 2
  puts i                  # imprime i: 2, 4, 6, 8, 10
end
```

Aunque parezca increíble, la función `step()` también funciona sobre los reales, con incrementos fraccionarios:

```
2.step(10, 0.5) do |r|    # desde 2 hasta 10, incrementando de a 0.5
  puts r                  # imprime r: 2.0, 2.5, 3.0, 3.5, etc
end
```

Otra forma alternativa de hacer ciclos repetitivos es con la instrucción `while`. Esta evalúa una expresión que resulta en un valor de verdad, y repite el ciclo tantas veces como la expresión evaluada sea cierta. Veamos un ejemplo:

```
cuenta = 0
while (cuenta < 5) do
  puts cuenta              # imprime desde 0 hasta 4
  cuenta += 1
end
```

La última vez que repite el ciclo, $\text{cuenta} \leftarrow 5$, y la comparación $(5 < 5)$ es falsa, así que ya no repite el bloque más. El uso de la palabra `do` también es opcional.

La instrucción `until`, es similar al `while`, excepto que la expresión evaluada tiene una lógica negativa: el bloque se repite mientras que la condición sea falsa.

```
cuenta = 0
until cuenta >= 5 do
  puts cuenta              # imprime desde 0 hasta 4
  cuenta += 1
end
```

Otra instrucción, `loop`, crea un ciclo potencialmente infinito. Para salirse del ciclo se usa la instrucción `break`, junto con una condición. El ciclo anterior se expresaría de la siguiente manera:

```

cuenta = 0
loop
  break if cuenta >= 5
  puts cuenta          # imprime desde 0 hasta 4
  cuenta += 1
end

```

Ejercicio 12.1: Verificar todas las expresiones de esta lección.

Ejercicio 12.2: Escriba un programa que sume los números del 1 al 100. Comparar el resultado con la fórmula $(n+1) * n/2$. Dice la leyenda, que Carl Friedrich Gauss inventó esta fórmula a la edad de 7 años, cuando en el colegio su profesor les dio esa tarea: es el resultado de sumar 50 veces 101.

Ejercicio 12.3: Escribir un programa que calcule el factorial de un número, usando un ciclo de multiplicaciones. El factorial se define como:
 $n! = n * (n-1) * \dots * 1$

Ejercicio 12.4: Escribir un programa que nos diga cuantas iteraciones debemos hacer para calcular e con 3 decimales de precisión, usando la fórmula de la serie de Taylor, dada por la ecuación siguiente:

$$e^x = \lim_{n \rightarrow \infty} \left(1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} \right)$$

Ayuda: Usar Math::E como valor de referencia.

Ejercicio 12.5: Escribir un programa que imprima la tabla de la función $f(x) = \sin(x)$ para valores de x entre 0 y 360 grados, en incrementos de 5 grados.

Ejercicio 12.6: Escribir un programa que nos diga cuántas iteraciones son necesarias para acercarse a e con 3 decimales de precisión, usando la fórmula

siguiente:
$$e = \lim_{x \rightarrow \infty} \left(1 + \frac{1}{x} \right)^x$$

Ayuda: Usar Math::E como valor de referencia.

Ejercicio 12.7: Escribir un programa que sume dos matrices.

La suma de dos matrices bidimensionales A y B, se define como:

$$C = A + B, \text{ donde } c(i,j) = a(i,j) + b(i,j), \text{ con } 0 < i < m-1, 0 < j < n-1,$$

donde m es el número de filas, y n es el número de columnas.

Ayuda: para definir una matriz usar un arreglo de columnas y en cada uno de sus elementos, definir un arreglo de filas.

Ensayarlo para producir la siguiente suma:

$$\begin{bmatrix} 132 \\ 100 \\ 122 \end{bmatrix} + \begin{bmatrix} 005 \\ 750 \\ 211 \end{bmatrix} = \begin{bmatrix} 137 \\ 850 \\ 333 \end{bmatrix}$$

Lección 13. Condiciones Múltiples (Case)

Como vimos en la lección 11 sobre condiciones, Ruby tiene una forma especial de controlar el flujo del programa, dependiendo de condiciones; esto es, con la instrucción `if/elsif/else`.

Un caso especial, es cuando queremos comparar una sola variable contra una cantidad de valores. Por ejemplo, si un semáforo tiene un determinado estado (color), queremos que un auto tome diversas acciones. Para eso, Ruby tiene definida la instrucción `case`, cuya sintaxis es como sigue:

```
case variable           # examina la variable
when valor1             # cuando valor1 == variable
  bloque1               # ejecuta el bloque1
[when valor2            # cuando valor2 == variable
  bloque2]              # ejecuta el bloque2
[else                   # en cualquier otro caso
  bloque3]              # ejecuta el bloque3
end
```

Cuando la variable tiene un valor igual a alguno de los examinados, entonces se ejecuta ese bloque solamente, y se descartan los otros. En el caso de que la variable no coincida con ninguno de los valores examinados, entonces se ejecuta la sección *else* y el *bloque3* (este último caso es conveniente para atrapar datos que están fuera de los rangos esperados). En cualquier caso, solamente uno de los bloques es ejecutado, y los otros son descartados.

En la expresión de sintaxis anterior, los bloques `when` y `else` son opcionales; los bloques `when` pueden ocurrir por lo menos una vez, y tantos como se necesiten.

Si los valores comparados son numéricos, la expresión `when` puede tener valores puntuales (discretos), o rangos, o expresiones regulares (que estudiaremos en otras lecciones).

Un ejemplo numérico:

```
i = 8
case i
when 1, 2..5            #evalua (1 == i), y luego (2..5 == i)
  print "i esta entre 1..5"
when 6..10              #evalua (6..10 == i)
  print "i esta entre 6..10"
else
  print "Error, dato fuera de rango" #atrapa otros valores
end
```

Un ejemplo con cadenas:

```
s = "abcde"
case s
when "aaa", "bbb"           #evalua ("aaa" == s) o ("bbb" == s)
  print "s es a o b"
when /c?e/                  #busca el patron (expresion regular) "c?e"
  print "s contiene el patron"
else
  print "No es comparable"
end
```

El caso de las expresiones regulares, se estudiará en otra lección.

Ejercicio 13.1: Escribir un programa que, dada una lista de calificaciones de estudiantes (en un array), con valores entre 0 y 100, cada uno, determine cuáles son excelentes ($\text{nota} \geq 90$), cuáles son buenos ($80 \leq \text{nota} < 90$), cuáles regulares ($60 \leq \text{nota} < 80$), y cuáles no pasan (< 60).

Ejercicio 13.2: Escribir un programa que, dada una lista que incluye nombres de compositores repetidas veces, nos cuente las veces que ocurre cada uno de los siguientes nombres: Mozart, Haydn, Bach. Ensayarlo con la siguiente lista: ["MOZART", "haydn", "mozart", "bach", "BACH", "Liszt", "Palestrina"]

Lección 14. Funciones

El concepto de funciones tiene el propósito de encapsular código que se va a necesitar para ejecutarlo varias veces. Es una forma para escribir código una sola vez y reusarlo varias veces.

Una función se define con la palabra clave `def`, definición, seguida de su nombre, y cualquier número de parámetros, o variables, a través de las cuales se le pasan valores a la parte interna de la función.

```
def miNombre(s)                # esta función se llama miNombre
    return "Mi nombre es #{s}" # recibe el parametro s, forma la frase
end                             # y regresa el valor de la frase nueva
s1 = miNombre("Antares")       # imprime "Mi nombre es Antares"
s2 = miNombre("Aldebaran")     # imprime "Mi nombre es Aldebaran"
```

Para que la función regrese un valor calculado, se usa la instrucción `return`. En realidad, la palabra `return` es también opcional.

Hay funciones que no regresan nada, pero que tienen efectos secundarios, tales como imprimir, o alterar valores de variables externas.

Las variables que se usan dentro de una función no son visibles por el código existente afuera de la función, y dejan de existir una vez que la función haya terminado su ejecución. Es decir, podemos pensar en una función como una “caja negra” que tiene entradas y salidas. Para que una variable externa sea visible dentro de una función, se puede pasar como parámetro, o se puede declarar esa variable como “global”, precediéndola de un signo `$`.

Veamos un ejemplo con comentarios:

```
a = [1,2,3]          # declaramos una variable externa
$b = 7               # declaramos una variable global, precedida por $
def f(x)             # declaramos una funcion
    puts a           # esta linea da error, porque a es invisible dentro f()
    puts $b          # esta linea accesa la variable global $b, es valido
    puts x           # esta linea accesa parametro x de la funcion, valido
    c = 99           # declaramos una variable local c, interna a la funcion
end                  # al terminar, las variables locales a f(x) desaparecen
puts c              # esta linea da error porque c no existe fuera de f()
```

Cuando una función se define en términos de sí misma, decimos que es “recursiva”. Por ejemplo, la función matemática, factorial de n , se define para los números naturales como:

$$\begin{aligned} \text{fact}(0) &= 1 \\ \text{fact}(n) &= n * \text{fact}(n-1), \text{ cuando } n > 0, \text{ para } n \in \mathbb{N} \end{aligned}$$

En Ruby se expresaría de la siguiente manera:

```
def fact(n)
  if 0 == n          # importante evaluar el punto limite primero,
    return 1         # para decidir cuando terminar
  else
    return n * fact(n-1) # la llamada recursiva viene despues
  end
end
puts fact(15)        #→ 1307674368000
```

Siempre que se definan funciones recursivas, es muy importante evaluar primero la condición de terminación, antes de invocar la llamada recursiva, para evitar que se produzca un loop infinito. Por esa razón, en inglés se habla de “tail-recursion”, es decir, la recursión se lleva a cabo por la cola.

Ejercicio 14.1: Escribir un programa que calcule la serie de Fibonacci con una función. Esta serie fue descubierta en occidente en el siglo XIII por Leonardo de Pisa, (ya se conocía desde el siglo VI en la India) y se define así:

```
fib(0) = 1
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), para n > 1, n ∈ ℕ
```

Ejercicio 14.2: La función recursiva anterior se hace lenta porque, cada que se la invoca, tiene que calcular una y otra vez todos los valores anteriores. Para optimizarla, escriba un programa que “recuerde” cada uno de los valores calculados, y los almacene en un arreglo, de manera que pueda responder rápidamente cuando se use consecutivamente. *Ayuda:* usar un arreglo global.

Ejercicio 14.3: Demostrar, por medio de un programa que use la función Fibonacci anterior, que el cociente de dos números sucesivos de Fibonacci, tiende hacia el “cociente áureo” (“Golden Ratio”): [demostrado por Kepler]

$$\lim_{n \rightarrow \infty} \frac{Fib(n+1)}{Fib(n)} = \varphi = \frac{1 + \sqrt{5}}{2}$$

Ejercicio 14.4: Escribir un programa que produzca una matriz con el triángulo de Pascal usando una función que reciba como parámetro el número de filas deseadas. Cada elemento del triángulo de Pascal tiene la siguiente relación:

$$a(i, j) = a(i-1, j) + a(i-1, j-1),$$

con unos en la primera columna, y en la diagonal:

$$\begin{array}{cccc}
 & & 1 & \\
 & 1 & 2 & 1 \\
 1 & 3 & 3 & 1 \\
 & & \text{etc} &
 \end{array}$$

Recordemos que la fila n del triángulo de Pascal produce los coeficientes de la expansión de la potencia binomial: $(x + y)^n$.

Ejercicio 14.5: Escribir un programa que use una función que nos dé el nombre de polígonos de n lados, con n entre 1 y 99. *Ayuda:* Consultar el enlace sobre polígonos citado en la sección “Curiosidades Matemáticas”, al final del libro.

Ejercicio 14.6: Escribir un programa con funciones para calcular los coeficientes de la potencia binomial: $(x + y)^n$

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k, \text{ donde } \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Lección 15. Clases

Por muchos años, se usaron lenguajes de programación basados en procedimientos y funciones (como C, y Pascal). Cuando los programas se hacen complejos, (de miles de líneas) esto resulta en código difícil de entender y modificar.

Poco a poco se fueron inventando otros lenguajes (como C++, y Smalltalk) que facilitaban la capacidad de abstraer y conceptualizar el mundo que nos rodea en términos de clases, o categorías. Esto es más intuitivo, y permite escribir código más comprensible. La razón de esto es que dentro de nuestro cerebro pensamos en categorías (taxonomías), y tenemos la tendencia a abstraer y agrupar todo lo que conocemos.

Podemos estar tranquilos, pues el concepto de clases no es nada especial. Hasta ahora, ya hemos manipulado variables en Ruby que son objetos, o instancias de determinadas clases predefinidas, tales como cadenas (Strings), números (Integer, o Float), arreglos (Array), y hashes (Hash). En esta lección vamos a aprender a definir clases y a darle el comportamiento que queramos.

Para empezar, podemos pensar en una clase, como una colección de funciones, y de variables, que describen un determinado concepto. En pseudo-código, podríamos pensar en algo así:
[ojo, esto todavía no es Ruby]

```
class MiClase
  variable1, variable2, etc
  funcion1, funcion2, etc
end
```

Una vez que ya esté definida, para usar una clase, primero hay que crear un objeto, o instancia, de la clase. Ya sabemos hacer esto con la función new:

```
h = Hash.new      # crea un objeto de la clase Hash
a = Array.new     # crea un objeto de la clase Array
m = MiClase.new   # crea un objeto de la clase MiClase
```

También ya hemos usado antes propiedades y funciones de un objeto:

```
n = a.size        # el tamaño de un array es una propiedad
h.sort            # sort() es una funcion de la clase
```

Algunas de las variables y funciones de la clase, solamente se usarán dentro de la clase, para ayudar a hacer computaciones internas, mientras que otras serán visibles desde afuera.

- A las variables internas les llamaremos “variables de la instancia” (“instance variables”, en inglés), y para distinguirlas, las precedemos con un signo de arroba.

Ejemplos: @temperatura.

- A las variables visibles desde afuera, les llamaremos atributos, como si fueran una propiedad del objeto que se puede leer y/o cambiar. Normalmente, los atributos son sustantivos, y sus valores son adjetivos, o números.

Ejemplos: perro.color, planeta.gravedad, motor.cilindros, paciente.edad.

- A las funciones de la clase le llamaremos métodos y normalmente usamos verbos para describir las acciones que van a desempeñar.

Ejemplos: perro.ladraz, motor.arrancar, paciente.crear.

Ahora veamos un ejemplo completo. Vamos a “modelar” el concepto de un perro que tiene nombre, y sabe ladraz. En Ruby, escribimos:

```
class Perro                                     # nombre de la clase
  def initialize(nombre)                       # metodo para inicializar
    @nombre = nombre                          # @nombre variable interna
    @ladrido = "guau"                        # @ladrido variable interna
  end
  def to_s                                     # representacion textual
    "#{@nombre}: #{@ladrido}"                # reporta valores interno
  end
  def nombre                                  # exporta nombre como propiedad publica
    @nombre                                  # retorna el valor de @nombre
  end
  def <=>(perro)                               # operador de comparacion para ordenar
    @nombre <=> perro.nombre                  # compara por nombre
  end
  def ladraz                                  # hacerlo ladraz
    @ladrido                                  # produce el ladrido
  end
end
```

En la definición anterior, vale la pena notar lo siguiente:

- Por convención, el nombre de la clase siempre empieza en mayúscula.
- Las variables que se declaran dentro de la clase tienen alcance local, por default, (es decir, no son visibles desde afuera de la clase), y son precedidas por el signo de arroba: @.
- El método initialize() se conoce como el constructor, porque es el primer método que se ejecuta cuando creamos una variable de tipo de la clase. El constructor es opcional, pero si lo definimos, lo usamos para efectuar cualquier tipo de inicialización dentro de la clase. Este método se ejecuta automáticamente (nunca se invoca explícitamente) al crear una variable de esta clase con el método new.
- Es conveniente también definir el método “to_s” donde expresamos qué hacer cuando nos piden la representación textual de la clase. Esto resulta útil cuando queremos examinar el contenido interno (o estado) de una clase.

- El método de comparación `<=>` se define para hacer que la clase sea comparable, es decir, que se le puedan aplicar operaciones tales como `sort()`, etc. Consiste en declarar qué variable interna vamos a usar para hacer comparaciones. En este caso, lo vamos a hacer por nombre.
- El método `nombre()` nos permite examinar (leer) el nombre del objeto desde afuera.
- El método `ladrar()` hace que el perro ladre (simplemente retorna el valor de `@ladrido`).

Ahora, para crear un objeto de tipo `Perro`, como ya sabemos, usamos el nombre de la clase, seguido de la palabra `new`, seguido de los argumentos que le vamos a pasar al método `initialize()`:

```
f = Perro.new("fifi") # creamos el perro fifi, en la variable f
m = Perro.new("milu") # creamos el perro milu, en la variable m
puts f.ladRAR        # los hacemos ladRAR
puts m.ladRAR        # los expresamos textualmente
puts f.to_s          # los expresamos textualmente
puts m.to_s          # los expresamos textualmente
m <=> f               #→ 1          # compara por nombre
```

En el ejemplo anterior, `Perro` es la clase, y `f` y `m` son instancias de clase `Perro`. Es importante distinguir entre estos dos conceptos:

- la clase (que consiste en la definición), y,
- los objetos, o instancias, que son variables del tipo de la clase.

Ejercicio 15.1: Modificar la clase `Perro` anterior para que incluya propiedades tales como el color, la edad, y el sexo.

Ejercicio 15.2: Ensayar qué pasa cuando se hace

```
milu.ladrido = "wan wan".
```

Se puede acceder esta variable interna?

Ejercicio 15.3: Redefinir el método de comparación, de manera que compare por edad, en vez de por nombre.

Ejercicio 15.4: Escribir un programa que ponga una colección de perros en un arreglo, que examine sus propiedades, que los haga ladrar a todos, y que los ordene por edad.

Lección 16. Atributos de Clases

En la lección anterior mencionamos que las variables de una clase, por default, tienen visibilidad local, es decir, solamente son visibles para los métodos de la clase (y más adelante veremos cómo afecta esto a las subclases).

Así que en la clase Perro, de la lección anterior, no se puede acceder al ladrido directamente:

```
fifi.ladrido           #produce un error
```

Ahora, para “exportar” un atributo, tal como el nombre, tenemos la opción de hacerlo legible y/o escribible. Para hacerlo legible, (es decir que se pueda acceder para leer desde afuera) como hicimos en la lección anterior, declaramos un método que retorne su valor:

```
class Perro
  def initialize(nombre)           # nombre de la clase
    @nombre = nombre              # metodo para inicializar
    ...                           # asignar el nombre
  end

  def nombre                       # propiedad publica legible
    @nombre                       # retorna el nombre
  end
  ...
end
f = Perro.new("fifi")
f.nombre                          # retorna el valor sin producir error
```

Para hacerlo escribible, añadimos un método que modifique su valor:

```
class Perro
  ...
  def nombre=(nombreNuevo)        # metodo para hacerlo escribible
    @nombre = nombreNuevo        # modifica el nombre
  end
end
f = Perro.new("fifi")
f.nombre = "fifirucho"           # invoca el metodo escribible
```

Ruby nos permite hacer esto de otra manera más fácil. Si queremos declarar atributos legibles, los podemos declarar usando la palabra clave `attr_reader`:

```
class Perro
  attr_reader :nombre, :fecha_de_nacimiento, :color
  #atributos legibles
  ...
end
```

Ahora, hay que modificar el constructor para que acepte valores iniciales:

```

class Perro
  ...
  def initialize(nombre, nacimiento, color) # metodo constructor
    @nombre = nombre                     # @nombre
    @fecha_de_nacimiento = nacimiento   # @edad
    @color = color                       # @color ... variables internas
  end
  ...
end

```

Lo anterior permite hacer uso de las propiedades nuevas así:

```

f = Perro.new("fifi", "2000-06-21", "gris")
f.color           #=> "gris"

```

Para hacer un atributo escribible (modificable), hay que añadir la siguiente declaración:

```

class Perro
  attr_writer :sobrenombre      #atributo escribible
  ...
end

```

Esto permite cambiarle el valor desde afuera, como si fuera una propiedad, así:

```

f = Perro.new("fifi", "2000-06-21", "gris")
f.sobrenombre = "fifirucho"

```

La declaración alternativa, `attr` se puede usar para declarar atributos que son legibles y escribibles:

```

attr :nombre, true

```

Le sigue un valor de verdad, que cuando es verdadero, lo hace también modificable.

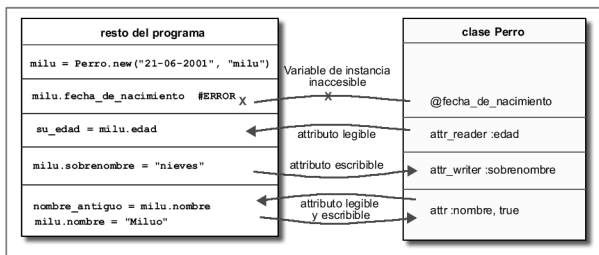


Figura 16.1 Acceso a atributos desde fuera de la clase

Las variables precedidas por dos puntos se conocen como “símbolos”, y representan solamente eso. Son constantes cuyo valor es una cadena con el mismo nombre. No se les puede cambiar su valor.

Internamente, el interpretador Ruby identifica la presencia de las palabras claves, `attr_reader`, `attr_writer`, o `attr`, y los símbolos asociados, y convierte todo esto a métodos de acceso y escritura. Los símbolos son usados para crear variables de la clase que llevan el mismo nombre.

Ejercicio 16.1: Modificar la clase Perro anterior para que incluya las siguientes propiedades legibles y escribibles: color, edad, y sexo.

Lección 17. Control de Acceso a la Clase

El acceso a los métodos de la clase se puede restringir usando tres diferentes niveles de protección:

- *Métodos públicos*: son los métodos que se pueden acceder por cualquiera, sin restringir su acceso. Por default, los métodos son públicos.
- *Métodos protegidos*: solamente se pueden acceder por objetos de la clase que los define, y sus subclases; es decir, el acceso queda “entre la familia”.
- *Métodos privados*: solamente se pueden acceder dentro de la clase que los define.

Para poner estas restricciones en efecto, se usan las palabras claves: `public`, `protected`, y `private`.

Hay dos formas de usarlas. La primera forma es escribir estas palabras entre los métodos de la clase:

```
class MiClase
  def metodo1  #público por defecto
  end
  protected
  def metodo2  #protegido
  end
  private
  def metodo3  #privado
  end
  public
  def metodo4  #público
  end
end
```

La segunda forma es una alternativa para expresar lo mismo, y consiste en definir los métodos primero, y cualificarlos después de la siguiente manera:

```
class MiClase
  def metodo1
  end
  def metodo2
  end
  def metodo3
  end
  def metodo4
  end
  public :metodo1, :metodo4
  protected :metodo2
  private :metodo3
end
```


La siguiente figura muestra los diferentes niveles de acceso a los métodos de la clase. La lección siguiente explica el mecanismo de herencia.

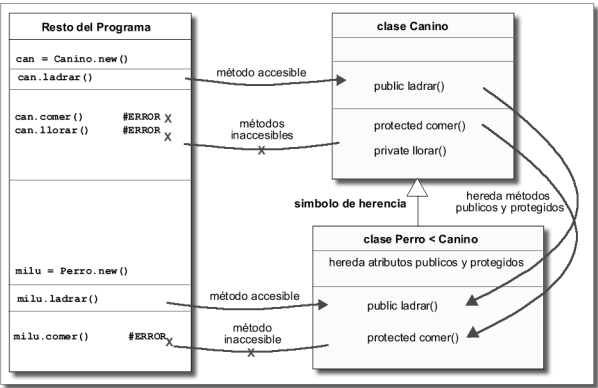


Figura 17.1 Acceso a los métodos de la clase

Lección 18. Herencia y Taxonomías

Taxonomía es el arte de clasificar cosas en una estructura jerárquica, de manera que queden en una relación padre-hijo, o superclase-subclase.

Por ejemplo, para una aplicación gráfica de objetos tridimensionales, podríamos pensar en una superclase genérica de cajas, con propiedades tales como color.

```
class Caja      # Caja es un objeto generico
  def initialize(color)
    @color = color      #guarda el valor en una variable local
  end
end
```

Después podemos pensar en otros objetos que consideramos subclases de Caja, tales como: cilindro, cubo, esfera, cuyos volúmenes vienen dados por fórmulas distintas. Decimos que las subclases heredan ciertas propiedades de las super-clases, como color, etc. También decimos que las subclases se especializan más en el objeto que representan. De esta manera, vamos modelando, o conceptualizando el problema que tratamos de resolver.

Ruby permite expresar relaciones de taxonomía, al definir clases. Usando la sintaxis SubClase < SuperClase, podemos definir clases así:

```
class Cilindro < Caja      # Cilindro es una subclase de Caja
  attr_reader :radio, :lado, :volumen
  def initialize(color, radio, lado)
    super(color)      # invoca el constructor de la superclase
    @radio, @lado = radio, lado
    @volumen = 2 * Math::PI * radio * lado
  end
end
```

La ventaja de usar taxonomías, es que podemos ir desarrollando una aplicación donde las subclases heredan de las superclases, y el código está organizado de una manera lógica, incremental y extensible, evitando duplicación y minimizando esfuerzo.

Ejercicio 18.1: Completar el ejemplo de esta lección escribiendo las clases para cubo y esfera. *Ayuda:* volumen de la esfera = $\frac{4}{3} * \pi * r^3$
Crear un tubo, un dado, y una pelota, con radios y lados =2, y reportar sus respectivos volúmenes.

Lección 19. Expresiones Regulares

Las expresiones regulares constituyen un mini-lenguaje cuyo propósito es crear patrones genéricos que luego se buscan dentro de una cadena de caracteres. La tabla siguiente explica algunos de los caracteres especiales usados en expresiones regulares:

Caracter	Significado
\	Caracteres especiales: \t , \n , \s (espacio), \d (dígito), \007 (octal), \x7f (hex), \ (línea vertical)
.	cualquier caracter, excepto \n
^	el principio de una cadena
\$	el final de una cadena
*	el elemento anterior ocurre 0 o mas veces
+	el elemento anterior ocurre 1 o mas veces
?	el elemento anterior ocurre 0 o 1 veces
{ }	especifica un rango de elementos
[]	una clase de caracteres contenidos dentro de los parentesis
()	agrupa expresiones regulares
	expresiones alternativas

Ejemplos:

En la tabla siguiente, el patrón de la columna izquierda hace juego con las cadenas de la derecha:

Expr Regular	Ejemplo
/\d\d:\d\d:\d\d/	"12:34:56"
/\w+ /	una o mas palabras
/\s+ /	uno o mas espacios
/a.c/	"xabcz", "xamcz", "xa9cz", ... (cualquier caracter entre a y c)
/Jair(o íto)/	"Jairo", "Jairito"
/[Ee]lla/	"Ella", "ella"
/^Salud/	"Saludando" (cadena que empiece con esa palabra)
/Salud\$/	"Para su Salud" (cadena que termine con esa palabra)
/ab*c/	"xacz", "xabcz", "xabbbbcz", ... (cero o mas ocurrencias de b, precedidas por a, y seguidas por c)
/ab+c/	"xabcz", "xabbbbbcz", ... (una o mas ocurrencias de b, precedidas por a y seguidas por c)

/a?c/	"xcz", "xacz" (cero o una ocurrencia de a, seguida de c)
/[0-9]/	cualquier numero de un dígito
/[a-zA-Z]/	cualquier palabra de una letra
/tu yo el ella/	cualquiera de las cuatro palabras entre //

Para evaluar el valor de verdad de una expresión regular, se usa el operador `==`. Se puede utilizar en expresiones `if`, y `while`. Ejemplo:

```
if "xabcz" == /abc/ # busca el patron "abc" en la cadena "xabcz"
  puts "si esta"    # imprime "si esta"
end
```

Se pueden usar expresiones regulares con los métodos de sustitución en las cadenas, `sub()` y `gsub()`:

```
s = "no no no"
s.sub(/no/, "si") # produce "si no no", reemplaza la primera ocurrencia
s.gsub(/no/, "si") # produce "si si si", reemplaza todas las ocurrencias
```

El método `scan()` de las cadenas de caracteres, toma una expresión regular como parámetro. El efecto es que itera sobre la cadena buscando ese patrón, y retorna un arreglo con el resultado:

```
s = "Hassuna Samarra Halaf" #ciudades antiguas del medio oriente
s.scan(/\w+/) #→ ["Hassuna", "Samarra", "Halaf"] #separa palabras
```

El método `split()` de las cadenas de caracteres, también puede tomar una expresión regular como parámetro. En tal caso, usa el patrón dado como separador para quebrar la cadena, y retorna un arreglo con el resultado:

```
s = "Aldebaran, Regulus, Antares, Fomalhaut" #estrellas cardinales
s.split(/\s*/) #→ ["Aldebaran", "Regulus", "Antares", "Fomalhaut"]
#la expresion regular anterior busca comas,
#seguidas de cero o mas espacios
```

Ejercicio 19.1: Confirmar las expresiones regulares de esta lección.

Ejercicio 19.2: Escribir una instrucción que separe las palabras de la siguiente cadena:

```
"Paris|Roma|Madrid|Estambul|Damasco"
```

Ejercicio 19.3: Escribir un programa que separe los elementos de la siguiente cadena:

```
"http://sitioweb.net?var1=1&var2=2&var3=3"
```

Lección 20. Archivos

Los archivos se usan para almacenar datos de forma permanente. Todos los lenguajes de programación ofrecen la manera de acceder archivos para crearlos, leerlos, escribirlos, cerrarlos, y borrarlos.

Ruby tiene dos clases relacionadas para este propósito, File y IO (Input/Output). File es una subclase de IO y hereda muchos de sus métodos.

Veamos primero cómo abrir un archivo existente, y cómo leer su contenido:

```
archivo="datos.txt" # usar una variable si lo usamos varias veces
if File.exists?(archivo) # nos aseguramos de que exista
  f = File.open(archivo) # abrimos el archivo datos.txt
  f.each do |linea| # procesamos cada línea como un arreglo
    puts "#{f.lineno}: #{linea}" # imprimimos número y línea
  end
  f.close # lo cerramos al final del proceso
else
  puts "#{archivo} no existe" # reportamos un error si no existe
end
```

En el programa anterior, se hubiera podido pasar el parámetro “r” al método File.open(), así: File.open(archivo, "r"), para indicar que lo vamos a leer (“read”), pero este es el default, así que podemos omitir este parámetro.

Recordemos que Ruby fue diseñado para ser fácil y entretenido, así que ofrece muchas formas de hacer lo mismo. Por ejemplo, el método IO.each es sinónimo de IO.each_lines.

Otra alternativa es leer todo el contenido del archivo y guardarlo en un arreglo, con el método IO.readlines():

```
archivo="datos.txt" # nombre del archivo
if File.exists?(archivo) # nos aseguramos de que exista
  f = File.open(archivo) # abrimos el archivo datos.txt
  a = f.readlines(archivo) # lo leemos y asignamos a un arreglo
  f.close # una vez leído, ya lo podemos cerrar
  a.each do |linea| # ahora procesamos cada línea del arreglo
    puts "#{linea}" # imprimimos la línea
  end
else
  puts "#{archivo} no existe" # reportamos un error si no existe
end
```

Esta segunda alternativa ofrece varias ventajas:

- usa solamente una instrucción para acceder el archivo y leerlo;
- procesa directa y rápidamente en memoria.

La posible desventaja es que si el archivo es bastante grande (del orden de Megabytes), puede ocupar mucha memoria.

Si queremos leer una línea a la vez, usamos el método `IO.read()`:

```
linea = f.read(archivo)
```

Para leer una línea en particular, podemos usar la propiedad `IO.lineno`:

```
f.lineno = 1000    # va a la línea 1000
linea = f.gets     # la lee y asigna a una variable
```

Ahora, vamos a crear un archivo, le escribimos datos, y lo cerramos:

```
archivo2="nuevosdatos.txt"    # nombre en variable para uso frecuente
f = File.new(archivo2, "w")    # abrimos el archivo nuevosdatos.txt
f.puts("línea con texto")      # escribe una línea de texto al archivo
pi = Math::PI
f.print("pi aproximado ",pi,"\n") # otra forma de escribir al archivo
e = Math::E
f.write("e vale "+e.to_s+"\n") # otra forma de escribir al archivo
f.close
```

Como muestra el ejemplo anterior, el método `File.new()` permite crear archivos. El primer parámetro es el nombre del archivo, y el segundo parámetro es el modo de apertura. La tabla siguiente muestra los modos de apertura disponibles:

Modo	Significado
r	Solo-lectura; comienza desde el principio del archivo (default).
r+	Lectura/Escritura: comienza desde el principio del archivo.
w	Solo-escritura: trunca el archivo si existe, o crea uno nuevo para escritura.
w+	Lectura/Escritura: trunca el archivo si existe, o crea uno nuevo para lectura/escritura.
a	Solo-escritura: comienza desde el final del archivo si existe, o lo crea nuevo para escritura.
a+	Lectura/Escritura: comienza desde el final del archivo si existe, o lo crea nuevo para lectura/escritura.
b	(Solo en Windows). Modo binario, y se puede combinar con cualquiera de las letras anteriores.

La siguiente figura muestra los diferentes tipos de acceso disponibles para archivos; también muestra la relación de los archivos con el resto del sistema:

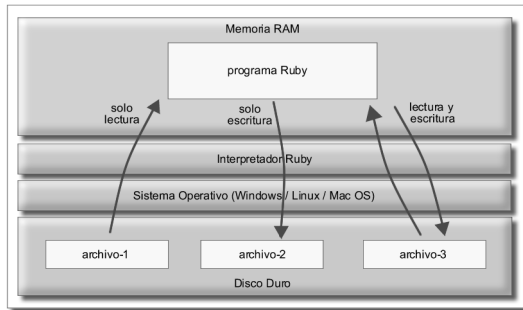


Figura 20.1 Archivos: (1) de lectura, (2) de escritura, y (3) de ambos: lectura y escritura

Ejercicio 20.1: Verificar los programas de esta lección.

Ejercicio 20.2: Escribir un programa que lea un archivo de texto, y que cuente las veces que ocurre cada palabra del texto.

Ayuda: Usar un hash donde cada llave es una palabra, y el valor es la cuenta.

Ejercicio 20.3: Escribir un programa que lea un texto de un archivo, que reemplace ciertas palabras, y escriba el texto nuevo en otro archivo.

Ejercicio 20.4: Un archivo contiene una secuencia de letras que representan una cadena de [ARN](#) (ácido ribonucleico): Adenina (A), Citosina (C), Guanina (G), y Uracil (U). Escribir un programa que lea el archivo, y que, utilizando el código genético, produzca otro archivo con la secuencia de proteínas que se genera a partir del ARN dado.

Ayuda: Seguir el enlace sobre genética en la sección de “Curiosidades matemáticas” al final del libro. El algoritmo debe hacer lo siguiente: leer el ARN en grupos de tres nucleótidos (llamados codones) y se traducen con la tabla para producir una proteína. *Ejemplo:* UGG produce Triptofán (Trp/P). Verificar los códigos de iniciación y terminación de la cadena.

Lección 21. Directorios

La clase Dir se usa para representar y manipular directorios.

Algunos de los métodos que soporta son los siguientes:

```
Dir.chdir("c:/code/ruby")      # cambia el directorio corriente
Dir.pwd                        # nombre del directorio corriente
Dir.getwd                      # sinonimo de Dir.pwd
Dir.delete("c:/code/old")      # borra un directorio
Dir.mkdir("c:/code/ruby/new")  # crea un directorio
```

Ruby ofrece varias formas de procesar la lista de archivos en un directorio: Podemos, por ejemplo, crear un arreglo con la lista de archivos:

```
a = Dir.entries("c:/code/ruby") # lista de archivos en directorio
```

Alternativamente, podemos iterar sobre la lista directamente, con foreach:

```
Dir.foreach("c:/code/ruby") {|x| puts x} #itera sobre lista de archivos
```

Otra forma alternativa con each:

```
d = Dir.open("c:/code/ruby") # usa un objeto Dir en una variable
d.each {|x| puts x}          # itera sobre lista de archivos
d.close
```

Con el método glob() se puede filtrar la lista de archivos con una expresión regular:

```
a = Dir.glob("*.rb")          # lista de archivos con terminacion rb
```

También es posible leer un directorio con d.pos y d.read:

```
d = Dir.new("c:/code/ruby")   # sinonimo de Dir.open
d.pos = 1                     # posiciona sobre ler elemento de la lista
s = d.read                    # lee el directorio
puts s                         # reporta lo que leyo
d.close
```

Ejercicio 21.1: Verificar el código de esta lección.

Lección 22. Entrada/Salida, Corrientes y Tubos

A veces deseamos poder pasar datos hacia un programa desde afuera, digamos, desde la línea de comando, y ver los resultados ahí mismo. Esto es bastante conveniente para conectar un programa con otro, usando el concepto de tubos (“pipes”, en inglés). La idea es que ciertos programas son diseñados de manera que se pueden invocar secuencialmente: la salida del primero es la entrada al segundo, etc. como conectando un tubo detrás de otro.

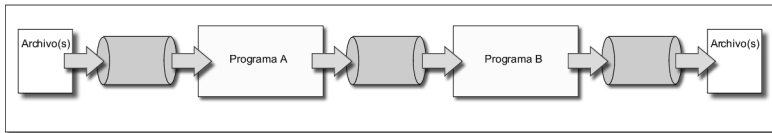


Figura 22.1 Conectando programas

Desde la consola de comandos, esto se invocaría de la siguiente manera:

```
c:\code\ruby> ruby ProgramaA.rb < archivo1.txt | ProgramaB.rb >
archivo2.txt
```

Para hacer esto posible, Ruby (igual que muchos otros lenguajes) tiene predefinidas las constantes globales `STDIN`, `STDOUT`, y `STDERR`, de tipo IO, que sirven para pasar “streams”, o “corrientes” de datos hacia dentro y hacia afuera del programa. `STDIN` se usa para leer datos hacia adentro, `STDOUT` para escribir resultados hacia afuera, y `STDERR` para reportar errores.

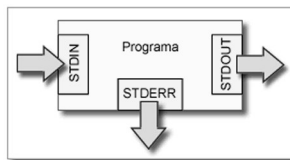


Figura 22.2 `STDIN`, `STDOUT`, `STDERR`

Estas corrientes están siempre disponibles (abiertas) así que no hay necesidad de abrirlas ni de cerrarlas.

Para leer una línea de datos por `STDIN`:

```
s = STDIN.gets
```

También existe una variable `$stdin`, que normalmente tiene el valor `STDIN`, pero se puede redefinir para leer datos desde diferentes fuentes:

```
s = $stdin.gets # lee desde STDIN
archi = File.new("misdatos.txt","r")
$stdin = archi # reasigna el valor de $stdin
s = $stdin.gets # ahora lee linea desde archivo archi
archi.close
$stdin = STDIN # reasigna al default
s = $stdin.gets # lee desde STDIN
```

STDOUT se puede usar de la siguiente manera:

```
STDOUT.puts("Las constantes son:") # escribe texto hacia afuera
pi = Math::PI
STDOUT.print("pi aproximado ",pi,"\n")# otra forma de escribir a STDOUT
e = Math::E
STDOUT.write("e = " + e.to_s + "\n") # otra forma de escribir a STDOUT
```

También existe la variable \$stdout, que normalmente tiene el valor STDOUT, pero se puede redefinir para enviar resultados hacia un archivo.

```
archi = File.new("misresultados.txt","w")
puts "Hola..." # escribe a STDOUT
$stdout = archi
puts "Adios!" # escribe al archivo archi
archi.close
$stdout = STDOUT # reasigna al default
puts "Eso es todo." # escribe a STDOUT
```

De igual manera, existe la variable \$stderr, con valor STDERR, y se puede redefinir. Su uso es similar al de \$stdout, y STDOUT, respectivamente. STDERR se usa para redireccionar mensajes de error.

STDIN, STDOUT, y STDERR se portan como archivos, porque también son descendientes de IO. Así que lo siguiente es válido:

```
$stdin.each { |linea| $stdout.puts linea }
```

Ejercicio 22.1: Verificar el código sobre STDIN, STDOUT y STDERR, de esta lección. *Ayuda:* Para ensayar STDIN, escribir un programa sencillo e invocarlo directamente con Ruby desde la línea de comando haciendo:

```
ruby programa.rb
```

Lección 23. Formatos de salida

Hasta ahora hemos estado usando las funciones `puts()` y `print()` para imprimir resultados de una manera sencilla.

Queremos, de ahora en adelante, poder formatear cualquier dato de manera que aparezca como deseemos. Por ejemplo, datos textuales suelen ser alineados a la izquierda, derecha, o centrados, mientras que datos numéricos suelen ser truncados a cierto número de decimales.

Ruby tiene la función `sprintf()` (heredada del lenguaje C), con la siguiente forma genérica:

```
sprintf( formato, var1, var2, ...)
```

La tabla siguiente muestra las letras aceptadas:

Formato	Descripción
b	binario
i	entero
d	decimal
f	real (punto flotante)
s	cadena de caracteres
e	usa representación científica (exponencial)
E	similar a e, pero usa la letra mayuscula
g	similar a e, pero default a 4 decimales, o menos.
o	convierte a octal
s	cadena alfanumérica
u	trata el argumento como decimal sin signo
X	convierte a hexadecimal usando letras mayusculas
x	convierte a hexadecimal

El formato para números es una cadena de la forma “%n[m]x” donde:

- x es una letra de la tabla anterior,
- n es el número total de dígitos que ocupa (incluyendo el cero), y,
- m es el número de decimales [solamente en el caso de números de punto flotante].

Ejemplos:

```
sprintf("%12.9f", Math::PI)      #→ " 3.141592654" #pi con 9 decimales
sprintf("%4i", 557)              #→ " 557"          #entero a 4 cifras
sprintf("%1.1e", 1230000)        #→ "1.2e+06"
sprintf("%lg", 1230000)          #→ "1.23e+06"
sprintf("%b %o", 127, 127)       #→ "1111111 177"  #binario, octal
sprintf("%x", 127)               #→ "7f"           #hexadecimal.
```

El formato para cadenas de caracteres tiene la forma “%-ns” donde:

- n es el número de caracteres a mostrar (trunca si es menor que la longitud de la cadena en la variable)
- Si se usa un signo negativo(-), la cadena es justificada hacia la izquierda; de otra manera, se justifica a la derecha.

Ejemplos:

```
sprintf("%-10s", "hola")         #→ "hola      "
sprintf("%10s", "hola")          #→ "      hola"
```

Ejercicio 23.1: Verificar el código de esta lección.

Ejercicio 23.2: Escribir una expresión que imprima una tabla de la función seno(x), con valores de x desde 0 hasta 3.14, en incrementos de 0.02. Debe imprimir el valor de x, a dos decimales, y el valor de seno(x), a tres decimales.

Lección 24. Yield

Hay una instrucción especial de Ruby llamada `yield`, que no encontramos en otros lenguajes modernos. Por eso decidimos dejar su estudio para esta lección. Normalmente `yield` se usa dentro de funciones/métodos, y transfiere control al bloque asociado con el programa que le llama. Por ejemplo, si tenemos la siguiente función:

```
def dosVeces
  yield
  yield
end

dosVeces { puts 'Hola' } # hace dos veces lo que le pasemos desde aqui
#→ Hola
#→ Hola
```

`Yield` se puede usar para darle la oportunidad, al programa que le llama, a hacer algo con las variables internas que está procesando:

```
def fib_hasta(max)
  i1, i2 = 1, 1
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end

# funcion Fibonacci hasta un numero dado
# asignacion paralela i1=1, i2=1
# transfiere control al que llama

fib_hasta(1000) { |i| print i, ' ' } # imprime resultados parciales
#→ 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

La función anterior permite que el programa que le llama examine/imprima los valores parciales.

Visto de otra manera, podemos decir que `yield` es una forma de permitir que una función o método, pueda recibir un bloque como parámetro. Eso le permite al usuario, de la función o método, interactuar y extender/modificar su comportamiento.

Ejercicio 24.1: Verificar el código de esta lección.

Lección 25. Módulos y Mixins

Los módulos son una forma adicional de agrupar código para poderlo reutilizar luego. Para crear un módulo, simplemente agrupamos funciones y clases bajo la siguiente sintaxis, y lo guardamos todo en un archivo:

```
module MiModulo

  PHI = 1.61803398874989    #el cociente aureo
  def mifuncion
    puts "Un saludo desde mifuncion"
  end

  class MiClase
    def mimetodo
      puts "Un saludo desde MiClase.mimetodo"
    end
  end
end
```

Después, para utilizar este código en otros archivos, lo hacemos disponible con la directiva `require`, como lo hemos hecho ya en otras lecciones (con módulos tales como `'Math'`, etc).

```
require 'MiModulo'

cociente = MiModulo::PHI    #→ 1.61803398874989

c = MiModulo::MiClase.new   # crea una instancia de MiClase
c.mimetodo                  #→ "Un saludo desde MiClase.mimetodo"
```

No se pueden crear instancias de módulos. Sin embargo, se puede incluir las funciones de un módulo dentro de una clase que estemos definiendo, para extender su funcionalidad.

```
require 'MiModulo'

class MiClase1
  include MiModulo    # extiende la clase
end

class MiClase2
end

m1 = MiClase1.new
m1.mifuncion          #→ "Un saludo desde mifuncion"

m2 = MiClase2.new
m2.extend(MiModulo)   # extiende la instancia
m2.mifuncion          #→ "Un saludo desde mifuncion"
```

Esto se conoce técnicamente como “polimorfismo de interfaces”, y permite extender clases dándoles un comportamiento especial, o compartido. En Ruby, esta forma de extender clases se conoce como “mixins”.

Uno de los módulos predefinidos que se utiliza normalmente de esta manera es Enumerable. Definimos los métodos initialize() y each(). Automáticamente otros métodos tales como collect(), detect(), map(), each_with_index() quedan a nuestra disposición.

```
class MultiArray
  include Enumerable
  def initialize(*arrays) #acepta una coleccion de arreglos
    @arrays = arrays
  end
  def each                #por cada uno, itera su contenido
    @arrays.each { |a| a.each { |x| yield x } }
  end
end
ma = MultiArray.new([1, 2], [3], [4])
ma.collect                #→ [1, 2, 3, 4]
ma.detect { |x| x > 3 }   #→ 4
ma.map { |x| x ** 2 }     #→ [1, 4, 9, 16]
ma.each_with_index { |x, i| puts "El elemento #{i} es #{x}" }
#→ El elemento 0 es 1
#→ El elemento 1 es 2
#→ El elemento 2 es 3
#→ El elemento 3 es 4
```

Ejercicio 25.1: Verificar el código de esta lección.

Esta página se ha dejado intencionalmente en blanco.

Parte IV Usos Avanzados (omitido)

La parte IV, “Usos Avanzados”, se omite en esta versión del libro.

Esta sección está incluida en la versión comercial del libro, que se encuentra disponible en :

<http://www.lulu.com/content/1759456>

Esta sección incluye los siguientes capítulos:

Parte IV. Usos Avanzados

- Lección 26. Administración de Errores
- Lección 27. Logger
- Lección 28. Biblioteca de Funciones Matemáticas
- Lección 29. Documentación Automática
- Lección 30. Programación Web Básica con Ruby/CGI
- Lección 31. Programación Web con Plantillas (Templates)
- Lección 32. Programación Web con WEBrick
- Lección 33. Programación Web con Mod_Ruby
- Lección 34. Programación Web con eRuby
- Lección 35. Programación Web con Rails
- Lección 36. Acceso a Bases de Datos
- Lección 37. Interfaces Gráficas
- Lección 38. XML
- Lección 39. El Patrón Observador-Observable

Soluciones a Ejercicios Selectos

```
2.2.1: piHiparco = 377.0 / 120    #→ 3.14166666666667
      #Nota: con que uno de los términos en la expresión sea
      #expresado como número real, el resultado será real.

2.2.2: piArquimedes1 = 3 + 10.0/71 #→ 3.14084507042254
      piArquimedes2 = 3 + 1.0/7    #→ 3.14285714285714

2.2.3: piLiuHui = 355.0 / 113     #→ 3.14159292035398

2.2.4: piFibonacci = 864.0 / 275  #→ 3.14181818181818

2.3:  fahrenheit = (celcius * 9 / 5) + 32

2.4:  celcius = (fahrenheit - 32) / 1.8

2.5:  radianes = grados * 2 * Math.PI / 360.0

2.6:  grados = radianes * 360.0 / (2 * Math.PI)

3.3:  a, b, c = 3, 6, 2
x1 = -b + Math.sqrt(b**2 - 4 * a * c) / (2 * a)
# da error cuando b**2 < 4ac
x2 = -b - Math.sqrt(b**2 - 4 * a * c) / (2 * a)

3.6:  piEuler1 = 4 * (5 * Math.atan(1.0/7) + 2 * Math.atan(3.0/79))
      #→ 3.14159265358979
      piEuler2 = 4 * ( 2 * Math.atan(1.0/3) + Math.atan(1.0/7) )
      #→ 3.14159265358979
      #Nota: dentro de las funciones trigonométricas
      #hay que usar números reales.

3.7:  piWrenchShanks = 24 * Math.atan(1.0/8) + 8 * Math.atan(1.0/57) +
      4 * Math.atan(1.0/239)
      #→ 3.14159265358979

3.8:  piRuby = Math::PI           #→ 3.14159265358979
      #tiene el mismo valor que piEuler1, piEuler2, y piWrenchShanks.

8.2:  a = [1,2,3,4,5]
      a.each {|i| puts i.to_s + ", " + (i**2).to_s + ", " +
        Math.sqrt(i).to_s }

8.3:  a = ["estas", "son", "algunas", "palabras"]
      a.each do |p|
        puts p + ": tiene #{p.length} letras."
      end

9.1:  a=[1,2,3,4,5]
      total=0
      a.each{|n| total += n}      #equivale a decir: total = total + n
      #acumula la suma en la variable total
      puts "La suma es : " + total.to_s
```

11.1:

```
# asignar valores a las variables, a, b, c

a, b, c = 5, 3, 7

# fijense como hay que considerar todos los casos, uno por uno
# la indentacion se usa para hacer el programa mas facil de leer
# poner comentarios libremente para ayudar a seguir la logica del
programa
if (a > b)

    if (a > c)

        if (b > c)
            puts "a=#{a} > b=#{b} > c=#{c}"
        elsif (b == c)
            puts "a=#{a} > b=#{b} = c=#{c}"
        else
            #este es un comentario: (a > c > b)
            puts "a=#{a} > c=#{c} > b=#{b}"
        end

    elsif (a == c)
        puts "a=#{a} = c=#{c} > b=#{b}"
    else
        #otro comentario: (c > a > b)
        puts "c=#{c} > a=#{a} > b=#{b}"
    end

elsif (a == b)

    if (a > c)
        puts "a=#{a} = b=#{b} > c=#{c}"
    elsif (a == c)
        puts "a=#{a} = b=#{b} = c=#{c}"
    else
        # (c > a=b)
        puts "c=#{c} > a=#{a} = b=#{b}"
    end

else # (b > a)

    if (b > c)

        if (a > c)
            puts "b=#{b} > a=#{a} > c=#{c}"
        elsif (a == c)
            puts "b=#{b} > a=#{a} = c=#{c}"
        else # (b > c > a)
            puts "b=#{b} > c=#{c} > a=#{a}"
        end

    elsif (b == c)
        puts "b=#{b} = c=#{c} > a=#{a}"
    else # (c > b > a)
        puts "c=#{c} > b=#{b} > a=#{a}"
    end

end
```

11.2:

```
a=[2,8,-4,7,1] # arreglo con datos a comparar

min = a[0]      #asumir que el primer elemento es el menor
max = a[0]      #asumir que el primer elemento es el mayor
a.each do |v|   # por cada elemento v del arreglo
  if v < min     # si es menor
    min = v     # asigne un nuevo minimo
  end
  if v > max     # si es mayor
    max = v     # asigne un nuevo maximo
  end
end             # fin del ciclo
puts "Resultado final:"
puts "El minimo es: #{min}"
puts "El maximo es: #{max}"
```

11.3:

```
a=[1,2,3,4,5,6]
a.each do |i|
  if (i % 2 == 0)      #si la division modulo 2 da cero, es par
    puts "#{i} es par."
  else
    puts "#{i} es impar."
  end
end
end
```

12.2:

```
n = 100
s = 0                                # inicia el acumulador en zero
1.upto(n) { |i| s += i}
puts "el total es: #{s}"            #→ 5050
s2 = (n + 1) * n/2
puts "usando la formula nos da: #{s2}" #→ 5050
```

12.3:

```
n = 5                                # objetivo
fact_n = 1                          # valor inicial de acumulador
1.upto(n) do |i|                    # cuenta entre 1 y 5
  fact_n *= i                       # equivale a fact_n = fact_n * i
                                     # fact_n = 1 * 2 * 3 * 4 * 5
  puts "#{n}: #{fact_n}"            # imprime resultados parciales (opcional)
end
puts "fact( #{n} ) = #{fact_n}"     # resultado final
```

12.4:

```
# exp(1) = 1 + 1/1! + 1/2! + ... + 1/n!

e1 = Math::E # 2.71828182845905 valor de referencia
#Queremos calcular hasta que obtengamos el valor 2.718; esto es:
e2 = (e1 * 1000).to_i / 1000.0 # metodo para truncar a 3 decimales

s = 1 # s lleva la suma de la serie
i = 1 # i cuenta las iteraciones del ciclo
while s < e2 # el ciclo se repite mientras que s < e2
  fact_n = 1
  1.upto(i) { |j| fact_n *= j} # aqui calculamos el factorial
  s += 1.0 / fact_n # y aqui lo acumulamos a s
  i += 1 # contamos las iteraciones en i
  puts "#{i}: #{s}" # imprimimos resultados parciales
end
puts "Se requieren #{i} iteraciones" # imprimimos el resultado final

# De hecho, la serie converge rapidamente.
# Se requieren 7 iteraciones para llegar a e2 (3 decimales de precision)
# y se requieren 15 iteraciones para llegar a e1 (14 decimales)
```

12.5:

```
0.step(360,5) do |i| # desde 0 hasta 360, incrementando de a 5
  rad = i * 2.0 * Math::PI / 360 #conversion grados a radianes
  puts i.to_s + ":" + Math.sin(rad).to_s # imprime i:sin(i)
end
```

12.6:

```
e1 = Math::E # e1 = 2.71828182845905
e2 = (e1 * 1000).to_i / 1000.0 # e2 = 2.718
x = 1 # x es un contador y empieza en 1
loop do # empieza el ciclo
  f = (1 + 1.0/x)** x # calcula f
  puts "#{x}:#{f}" # imprime el resultado parcial
  break if f>e2 # examina y termina el ciclo
  x += 1 # incrementa el contador
end # repite el ciclo desde aqui

# esta formula requiere 4,822 repeticiones para acercarse a e2;
# requiere 67,095,913 ciclos para acercarse a e1;
# diriamos que converge lentamente hacia e.
```

```

12.7: # suma de dos matrices
a = [[1, 3, 2], [1, 0, 0], [1, 2, 2]] #matriz a
b = [[0, 0, 5], [7, 5, 0], [2, 1, 1]] #matriz b
c = Array.new                          #matriz de filas c

m, n = a.size, a[1].size               #m: filas, n: columnas
for i in 0...m do                      #iterador sobre filas
  c[i] = Array.new                     #la 2da dimension (cols) de c
  for j in 0...n do                   #iterador sobre columnas
    c[i][j] = 0                       #las iniciamos en 0
  end
end

for i in 0...m do                      #iterar sobre las filas
  for j in 0...n do                   #por cada fila, itera columnas
    c[i][j] = a[i][j] + b[i][j]       #y calculamos la suma
  end
end

c.each do |fila|                       # por cada fila
  puts fila.join(" ")                 # imprime el resultado
end

13.1:
calificaciones = [20, 55, 61, 70, 98, 48, 87, 120, -3]
resultados = Array.new
for nota in calificaciones do
  case nota
  when 0..59                               #valores posibles quedan cubiertos
    resultados.push("malo")               #preparar lista nueva
  when 60..79
    resultados.push("regular")
  when 80..89
    resultados.push("bueno")
  when 90..100
    resultados.push("excelente")
  else
    resultados.push("error")
  end
end
print resultados.join(", ")               #imprime la lista con los resultados

13.2:
lista = ["MOZART", "haydn", "mozart", "bach", "BACH", "Liszt",
"Palestrina"]
cuenta = {"Mozart"=>0, "Haydn"=>0, "Bach"=>0, "otros"=>0}
lista.each do |nombre|
  nombre = nombre.downcase.capitalize     #asegura formato comparable
  case nombre
  when "Mozart"
    cuenta["Mozart"] += 1                 #acumula cuenta en hash
  when "Haydn"
    cuenta["Haydn"] += 1
  when "Bach"
    cuenta["Bach"] += 1
  else
    cuenta["otros"] += 1
  end
end
cuenta.each {|k,v| puts "#{k}:#{v}"}      #imprime la cuenta de cada uno

```

```

14.1:
def fib(n)
  case n
  when 1, 2
    return 1
  else
    return fib(n-1) + fib(n-2) #calcula recursivo y lento
  end
end
fib(20)          #→ 6765
fib(30)          #→ 832040      #se hace lento para numeros grandes

```

14.2: # Algoritmo Fibonacci con memoria:

```

$f = [1,1]          # declara un array global
def fibm(n)
  if $f[n]           # si ya tiene el valor
    return $f[n]     # lo devuelve
  else               # sino
    $f[n] = fibm(n-1) + fibm(n-2) # lo calcula recursivamente
    return $f[n]     # y lo devuelve
  end
end                  # efecto secundario, $f crece
fibm(100)            #→ 354224848179261915075

```

14.3:

```

def limgolden(n)
  phi = (1 + Math.sqrt(5)) / 2.0      # 1.61803398874989
  phi2 = (phi * 10**n).to_i / (10.0 ** n) # precision a n decimales
  i = 1

  loop do
    m = 1.0 * fibm(i+1) / fibm(i)    # calcula el cociente aureo
    m = (m * 10**n).to_i / (10.0 ** n) # aproxima a n decimales
    puts "#{i}:#{m}"                  # imprime resultado parcial
    break if (m == phi2)              # precision deseada?
    i += 1                            # contador de iteraciones
  end

  print "#{n} decimales toma #{i} iteraciones"
end

limgolden(3)          #→ 3 decimales toma 10 iteraciones
limgolden(6)          #→ 6 decimales toma 17 iteraciones
limgolden(14)         #→ 14 decimales toma 36 iteraciones
#podemos concluir que converge rapidamente a phi

```

```

14.4:
def pascal(n)
  m = Array.new          # crea la matriz, de una dimension
                        # vamos a considerar: i filas, y j columnas
  for i in 0..n do      # por cada fila
    m[i] = Array.new    # crea la segunda dimension
    m[i].fill(0, 0..n)  # inicia la fila en ceros
    m[i][0] = 1         # inicia la primera columna en unos
    m[i][i] = 1         # inicia la diagonal en unos
  end
  for i in 0..n do      # por cada fila
    for j in 0...i do   # por cada columna, hasta la diagonal excl.
      m[i][j] = m[i-1][j] + m[i-1][j-1] # calcula
    end
    puts m[i].join(" ") # imprime la fila
  end
end

#para ensayarlo intentar lo siguiente:

pascal(10)

```

```

14.5:
$prefijos0 = {1=>"hena", 2=>"di", 3=>"tri", 4=>"tetra", 5=>"penta",
6=>"hexa", 7=>"hepta", 8=>"octa", 9=>"enea", 10=>"deca", 20=>"icosa"} # +
sufijo "gono"
$prefijos10 = {1=>"hen", 2=>"dode", 3=>"tri", 4=>"tetra", 5=>"penta",
6=>"hexa", 7=>"hepta", 8=>"octa", 9=>"enea"} # + sufijo "decagono"
$prefijos100 = {2=>"icosi", 3=>"triaconta", 4=>"tetraconta",
5=>"pentaconta", 6=>"hexaconta", 7=>"heptaconta", 8=>"octaconta",
9=>"eneaconta"} # + kai + prefijo0 (1..9) + gono

```

```

def poligono(n)
  if (n < 1) or (n > 99)
    return "error: #{n} esta fuera de limites"
  elsif (n < 11) or (20 == n)
    return $prefijos0[n] + "gono"
  elsif (n > 10) and (n < 20)
    m = n - 10
    return $prefijos10[m] + "decagono"
  elsif (n > 20)
    decenas = n / 10
    unidades = n - (decena * 10)
    if (unidades > 0)
      return $prefijos100[decenas] + "-kai-" +
        $prefijos0[unidades] + "gono"
    else
      return $prefijos100[decenas] + "gono"
    end
  end
end

```

```

#ahora algunos ejemplos para ensayarlo:
poligono(0)  #→ "error: 0 esta fuera de limites"
poligono(5)  #→ "pentagono"
poligono(13) #→ "tridecagono"
poligono(20) #→ "icosagono"
poligono(30) #→ "triacontagono"
poligono(54) #→ "pentaconta-kai-tetragon"

```



```

14.6:
def fact(x)                                #calculamos fact(x) recursivamente
    if (0 == x) or (1 == x)
        1
    else
        x * fact(x-1)
    end
end

def coef(n, k)                             #definimos coef(n,k) en terminos de fact()
    fact(n) / (fact(k) * fact(n-k))
end

def binom(n)
    a = Array.new                          # un array para guardar las expresiones parciales
    for k in 0..n                          # la sumatoria implica que va a haber un ciclo
        c = coef(n,k)                     # calculamos el coeficiente para (n,k)
        m = n - k                         # guardamos el valor de n-k
        a[k] = "#{c} * x^{m} * y^{k}"     #calculamos la expresion parcial
    end
    puts a.join(" + ") # concatenamos todo con el signo +
end

binom(3) # produce el siguiente resultado:
#1 * x^3 * y^0 + 3 * x^2 * y^1 + 3 * x^1 * y^2 + 1 * x^0 * y^3

15.1, 15.3:
class Perro
    def initialize(nombre, color, edad, sexo) # constructor
        @nombre = nombre
        @color = color
        @edad = edad
        @sexo = sexo
        @ladrido = "guau"
    end
    def to_s # representacion textual
        "Nombre=#{@nombre}, Ladrido=#{@ladrido}, " +
        "Color=#{@color}, Edad=#{@edad}, Sexo=#{@sexo}"
    end
    def nombre
        @nombre
    end
    def color
        @color
    end
    def edad
        @edad
    end
    def sexo
        @sexo
    end
    def <=>(perro) #15.3 permite ordenarlo por edad
        @edad <=> perro.edad
    end
    def ladra
        @ladrido
    end
end

```

15.4

```
a = Array.new
a[0] = Perro.new("fifi", "negro", 3, "femenino" )
a[1] = Perro.new("milu", "blanco", 5, "masculino" )
a[2] = Perro.new("goofy", "cafe", 7, "masculino" )

a.each do |p|
  puts p.to_s      #examina las propiedades de cada uno
  puts p.ladra     #lo hace ladrar
end
a.sort             #los ordena por edad
```

16.1:

```
class Perro
  attr_reader :nombre, :color, :edad, :sexo
  attr_writer :nombre, :color, :edad, :sexo
  def initialize(nombre, color, edad, sexo) # constructor
    @nombre = nombre
    @color = color
    @edad = edad
    @sexo = sexo
    @ladrido = "guau"
  end
  def to_s # representacion textual
    "Nombre=#{@nombre}, Ladrido=#{@ladrido}, " +
    "Color=#{@color}, Edad=#{@edad}, Sexo=#{@sexo}"
  end
  def <=>(perro) # permite ordenarlo por edad
    @edad <=> perro.edad
  end
  def ladra
    @ladrido
  end
end
```

```

18.1:
class Caja      # Caja es superclase de todas
  attr_reader :color
  def initialize(color)
    @color = color
  end
end

class Cilindro < Caja      # Cilindro es subclase de Caja
  attr_reader :radio, :lado, :volumen
  def initialize(color, radio, lado)
    super(color)      # pide a superclase que guarde color
    @radio, @lado = radio, lado
    @volumen = 2 * Math::PI * radio * lado
  end
end

class Cubo < Caja      # Cubo es subclase de Caja
  attr_reader :lado, :volumen
  def initialize(color, lado)
    super(color)      # pide a superclase que guarde color
    @lado = lado
    @volumen = lado * lado * lado
  end
end

class Esfera < Caja      # Esfera es subclase de Caja
  attr_reader :radio, :volumen
  def initialize(color, radio)
    super(color)      # pide a superclase que guarde color
    @radio = radio
    @volumen = (4.0/3.0) * Math::PI * (radio ** 3)
  end
end

tubo = Cilindro.new("gris", 2, 2)
puts "tubo:#{tubo.volumen}"      #->25.1327412287183
dado = Cubo.new("rojo", 2)
puts "dado:#{dado.volumen}"      #->8
pelota = Esfera.new("azul", 2)
puts "pelota:#{pelota.volumen}"  #->33.5103216382911

19.2:
s = "Paris|Roma|Madrid|Estambul|Damasco"
s.split(/\|/)
#-> ["Paris", "Roma", "Madrid", "Estambul", "Damasco"]

19.3:
s = "http://sitioweb.net?var1=1&var2=2&var3=3"
url, vars = s.split(/\?/)      #separa "url?vars"
v = vars.split(/\&/)           #separa expresiones usando & como separador
h = Hash.new
h["url"] = url

v.each do |par|
  izq, der = par.split(=)      #separa cada lado del =
  h[izq] = der
end

#produce:
#h = [url => http://sitioweb.net, var1 => 1, var2 => 2, var3 => 3]

```

```

20.2:
archivo="datos.txt"                # archivo a leer
if File.exists?(archivo)           # lo procesamos si existe
  f = File.open(archivo)
  a = f.readlines(archivo)         # lo leemos
  f.close                          # lo cerramos
  dic = Hash.new                   # contaremos en un diccionario
  a.each do |linea|                # procesamos cada linea del arreglo
    palabras = linea.downcase.split(/[.,"!;:\s]) # separar las palabras
    palabras.each do |palabra|     # procesar cada palabra
      if dic[palabra]              # si esta en el diccionario
        dic[palabra] += 1          # aumentar el contador
      else                          # si no esta en el diccionario
        dic[palabra] = 1          # anadirla y contarla
      end
    end
  end
  sa = dic.sort                    # ordena arreglo de arreglos
  sa.each do |par|                 # para cada elemento
    puts "#{par[0]}:#{par[1]}"     # imprime el par: palabra, cuenta
  end
else
  puts "#{archivo} no existe"      # reportamos un error si no existe
end

20.3:
archivo="datos2.txt"              # archivo a leer
if File.exists?(archivo)          # lo procesamos si existe

  f = File.open(archivo)
  a = f.readlines(archivo)         # lo leemos
  f.close                          # lo cerramos

  dic = {"@nombre@"=>"Cesar", "@mensaje@"=>"vencimos"} #a reemplazar

  a.each do |linea|                # procesamos cada linea
    dic.each do |w1, w2|
      linea.gsub!(w1, w2)          # reemplaza en linea
    end
  end

  f = File.new("resultados.txt", "w")
  a.each { |linea| f.puts "#{linea}" } # imprime en el archivo
  f.close
else
  puts "#{archivo} no existe"      # reportamos un error si no existe
end

Datos2.txt:
Este es un mensaje para @nombre@: @mensaje@; quedese tranquilo.
Estimado Sr @nombre@, sus tropas dicen: "@mensaje@, @mensaje@, @mensaje@".

Resultados.txt:
Este es un mensaje para Cesar: vencimos; quedese tranquilo.
Estimado Sr Cesar, sus tropas dicen: "vencimos, vencimos, vencimos".

```

```

20.4:
def esComienzo(codon) #verifica el codon de comienzo
  return ("AUG" == codon)
end
def esFinal(codon) #verifica el codon de terminacion
  return (("UAG" == codon) or ("UGA" == codon) or ("UAA" == codon))
end

# datos para traducir de codon a proteina
$codigoGenetico = {"UUU" => "F", "UUC" => "F", "UUA" => "L", "UUG" => "L",
"UCU" => "S", "UCC" => "S", "UCA" => "S", "UCG" => "S", "UAU" => "Y",
"UAC" => "Y", "UAA" => "-[stop]", "UAG" => "-[stop]", "UGU" => "C",
"UGC" => "C", "UGA" => "-[stop]", "UGG" => "W", "CUU" => "L",
"CUC" => "L", "CUA" => "L", "CUG" => "L", "CCU" => "P", "CCC" => "P",
"CCA" => "P", "CCG" => "P", "CAU" => "H", "CAC" => "H", "CAA" => "Q",
"CAG" => "Q", "CGU" => "A", "CGC" => "A", "CGA" => "A", "CGG" => "A",
"AUU" => "I", "AUC" => "I", "AUA" => "I", "AUG" => "M", "ACU" => "T",
"ACC" => "T", "ACA" => "T", "ACG" => "T", "AAU" => "N", "AAC" => "N",
"AAA" => "K", "AAG" => "K", "AGU" => "S", "AGC" => "S", "AGA" => "R",
"AGG" => "R", "GUU" => "V", "GUC" => "V", "GUA" => "V", "GUG" => "V",
"GCU" => "A", "GCC" => "A", "GCA" => "A", "GCG" => "A", "GAU" => "D",
"GAC" => "D", "GAA" => "E", "GAG" => "E", "GGU" => "G", "GGC" => "G",
"GGA" => "G", "GGG" => "G"}

def toRNA (linea) #convierte cadena larga a array de codones
  return linea.scan(/.../) #parte cada tres letras
end

def estaBienConstruida(codons) #funcion para verificar comienzo y final
  return esComienzo(codons[0]) and esFinal(codons[-1])
end

archivo="arn.txt" # archivo a leer
if File.exists?(archivo) # lo procesamos si existe
  f = File.open(archivo)
  a = f.readlines(archivo) # lo leemos
  f.close # lo cerramos
  result = Array.new # nuevo arreglo con resultados

  a.each do |linea| # procesamos cada linea
    linea.lstrip! # remueve espacios a la izquierda
    linea.rstrip! # remueve espacios a la derecha
    linea.chomp! # remueve \n al final

    if not linea.empty?
      linea.upcase! # convierte a mayusculas
      codons = toRNA(linea) # convierte cadena a arreglo
      if estaBienConstruida(codons)
        proteina = String.new # prepara cadena para resultado
        codons.each do |codon|
          proteina += $codigoGenetico[codon] # reemplaza proteina
        end
        result.push(proteina)
      else
        result.push("Error: cadena no comienza o termina bien")
      end
      result.push(proteina) # anade la nueva cadena al arreglo
    end
  end
end
#(continua ...)

```

```

#(... continua)

f = File.new("proteinas.txt", "w")
result.each { |linea| f.puts "#{linea}" } # imprime en el archivo
f.close
else
  puts "#{archivo} no existe"          # reportamos un error si no existe
end

#este es el archivo de datos (ponerlo separado): arn.txt:
AUGGCACUCGCGGAGGCCGACGACGGCGCGUGGUCUUCGGCGAGGAGCAGUGA

# este es el resultado: proteinas.txt:
MALAEADDGAVVFGEEQ-[stop]

23.2:
0.step(3.14, 0.02) {|x| puts sprintf("%3.2f %5.3f", x, Math.sin(x))}

```

Bibliografía

Libros sobre Ruby

- “Learning Ruby”, Michael Fitzgerald, (O'Reilly, 2007), ISBN 0-596-52986-4.
- “Programming Ruby: The Pragmatic Programmer's Guide”, Dave Thomas (The Pragmatic Programmer, 2005), ISBN 0-974-51405-5.
- “Ruby Cookbook”, Leonard Richardson; Lucas Carlson (O'Reilly, 2006) ISBN 0-596-52369-6.
- “Ruby Developer's Guide”, Robert Feldt, Lyle Johnson, Michael Neumann (Syngress Publishing 2002), ISBN 1-928-99464-4.
- “Ruby in a Nutshell ”, Katz Matsumoto, (O'Reilly, 2001), ISBN 0-596-00214-9.
- “The Ruby Way”, Hal Fulton (Addison Wesley, 2006), ISBN 0-672-32884-4.

Otros Enlaces sobre Ruby en Internet

- Versión completa del libro “Ruby Fácil”
<http://www.lulu.com/content/1759456>
- Ruby Programming Language
<http://www.ruby-lang.org/en/>
- Versión en línea del libro: “Programming Ruby”
<http://www.rubycentral.com/pickaxe/>
- Ruby on Rails Tutorial
<http://wiki.rubyonrails.org/rails/pages/Tutorial>
- Learn to Program
<http://pine.fm/LearnToProgram/>
- Learn Ruby
<http://www.math.umd.edu/~dcarrera/ruby/0.3/>
- Rubypedia: fuente de recursos Ruby
<http://www.rubypedia.com/>
- RubyForge: proyectos open source sobre Ruby
<https://rubyforge.org/>
- RubyMatters: más recursos sobre Ruby
<http://www.rubymatters.com/>
- Descripción de Ruby en la wikipedia
http://en.wikipedia.org/wiki/Ruby_%28programming_language%29
- Ruby Application Archive: Colección de aplicaciones/programas y herramientas para Ruby
<http://raa.ruby-lang.org/>
- Technorati: blogs recientes sobre Ruby
<http://technorati.com/tag/ruby>
- O'Reilly Ruby: libros sobre Ruby por la editorial O'Reilly
<http://www.oreillynet.com/ruby/>
- “Getting Real”: sobre la filosofía detrás de Rails
<http://www.37signals.com/>

Curiosidades Matemáticas

Algunos de los ejercicios hacen referencia a temas, en su mayoría, matemáticos, con los que quizás el lector no esté familiarizado. Los siguientes enlaces ofrecen más información sobre estos temas.

- Cociente Aureo
http://en.wikipedia.org/wiki/Golden_ratio
- Código Genético
http://en.wikipedia.org/wiki/Genetic_code
- Coeficiente Binomial
http://en.wikipedia.org/wiki/Binomial_coefficient
- Número de Euler, e
http://en.wikipedia.org/wiki/E_%28mathematical_constant%29
- Serie de Fibonacci
http://en.wikipedia.org/wiki/Fibonacci_number#Limit_of_consecutive_quotients
- Fórmula de Euler
http://en.wikipedia.org/wiki/Euler%27s_formula
- Carl Friedrich Gauss
http://en.wikipedia.org/wiki/Carl_Friedrich_Gauss
- Límites
http://en.wikipedia.org/wiki/Limit_%28mathematics%29
- Matrices
http://en.wikipedia.org/wiki/Matrix_%28mathematics%29
- Series de Taylor
http://en.wikipedia.org/wiki/Taylor_series
- Triángulo de Pascal
http://en.wikipedia.org/wiki/Pascal%27s_triangle
- Pi
<http://en.wikipedia.org/wiki/Pi>
- Polígonos
<http://en.wikipedia.org/wiki/Polygon>

Apéndices

Apéndice A. El Código ASCII

El código ASCII, American Standard Code for Information Interchange, (Código Estándar Americano para Intercambio de Información) fue definido en 1960 para teletipos (máquinas que mandaban mensajes eléctricos por cable), y es un código de 7 bits (de 0 a 127) para los caracteres impresos del inglés.

Este código tuvo muchas limitaciones: por ejemplo, no se podían hacer los caracteres especiales del español (áéíóúñ), así que luego tuvo que ser extendido a 8 bits. Pero tampoco se podían hacer los caracteres de otros idiomas, así que hubo necesidad de desarrollar otros códigos, tales como Unicode y UTF, que son los que actualmente se usan, de 2 bytes, y que cubren todos los idiomas del mundo, incluyendo símbolos especiales de matemáticas, financieros, etc.

El código ASCII se puede generar fácilmente con un ciclo que itere sobre la función chr: [del 0 al 31, y el 127, se omiten por ser caracteres de control]

```
32.upto(126) {|c| puts c.chr}
```

32 : (espacio)	51 : 3	71 : G	91 : [111 : o
33 : !	52 : 4	72 : H	92 : \	112 : p
34 : "	53 : 5	73 : I	93 :]	113 : q
35 : #	54 : 6	74 : J	94 : ^	114 : r
36 : \$	55 : 7	75 : K	95 : _	115 : s
37 : %	56 : 8	76 : L	96 : `	116 : t
38 : &	57 : 9	77 : M	97 : a	117 : u
39 : '	58 : :	78 : N	98 : b	118 : v
40 : (59 : ;	79 : O	99 : c	119 : w
41 :)	60 : <	80 : P	100 : d	120 : x
42 : *	61 : =	81 : Q	101 : e	121 : y
43 : +	62 : >	82 : R	102 : f	122 : z
44 : ,	63 : ?	83 : S	103 : g	123 : {
45 : -	64 : @	84 : T	104 : h	124 :
46 : .	65 : A	85 : U	105 : i	125 : }
47 : /	66 : B	86 : V	106 : j	126 : ~
48 : 0	67 : C	87 : W	107 : k	
49 : 1	68 : D	88 : X	108 : l	
50 : 2	69 : E	89 : Y	109 : m	
	70 : F	90 : Z	110 : n	

Tabla del código ASCII (de 32 a 126)

- Más información sobre el código ASCII:
<http://en.wikipedia.org/wiki/ASCII>

Apéndice B. Cómo usar SciTE

SciTE es el editor de texto usado para programar en Ruby. Permite escribir programas, guardarlos, corregirlos, y ejecutarlos.

En esta pequeña guía vamos a demostrar cómo se usa SciTE.

B.1 Primer paso – Edición

Empezamos por ejecutar Scite, que se consigue siguiendo los siguientes menús de Windows:

[Comienzo] -> Programas -> Ruby-186-25 -> SciTE

Seguidamente, escribimos el programa en el espacio en blanco del editor. Podemos notar, como en la figura 1, que cada línea del programa obtiene automáticamente un número secuencial, que se muestra en la columna de la izquierda. Esto es útil cuando nos salen errores, para saber que línea ir a corregir.

También podemos observar que el código tiene colores. Otra ventaja, son las líneas verticales que aparecen indicando la indentación, o subordinación relativa del código. Todo esto es para facilitar leerlo y escribirlo correctamente.

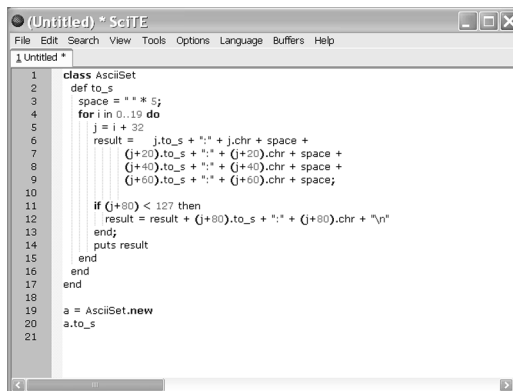


Figura B.1. Editando programas Ruby en SciTE

B.2 Segundo paso – Guardar el trabajo

Una vez que hayamos terminado de escribir el programa, lo guardamos en el disco para mantener una copia permanente de este. De esta manera, podemos volver otro día y seguir modificando nuestro trabajo. El comando para guardarlo es el menú siguiente, como muestra la figura 2:

File -> Save

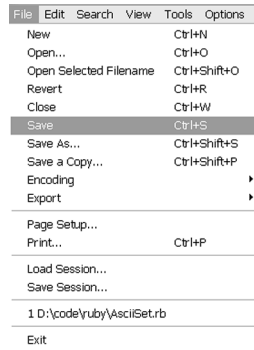


Figura B.2. Guardar el programa con File -> Save

Aparece una ventana donde le damos un nombre al programa. La terminación del nombre de programas Ruby, por convención, es rb. La figura 3 ilustra esto.

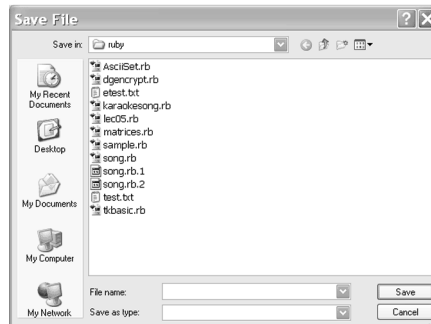


Figura B.3. Guardar el programa

B.3 Tercer paso – Ejecutar el programa

Una vez que el programa ha sido guardado, entonces se puede ejecutar para probarlo.

El comando para ejecutar un programa desde SciTE es el menú siguiente (figura 4):

Tools -> Go

Tools	Options	Language	Buffers
Compile			Ctrl+F7
Build			F7
Go			F5
Stop Executing			Ctrl+Break
Next Message			F4
Previous Message			Shift+F4
Clear Output			Shift+F5
Switch Pane			Ctrl+F6

Figura B.4. Ejecutar el programa

El equivalente es presionar la tecla [F5] que se encuentra encima de las teclas numéricas, en el teclado alfa-numérico. Ver Figura 5.

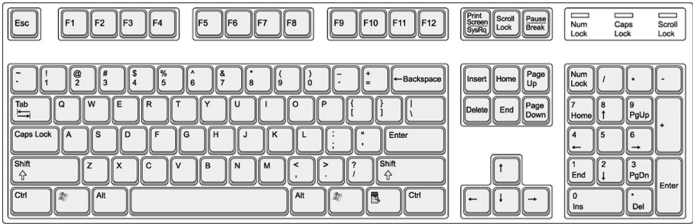


Figura B.5. [F5] en el teclado alfa-numérico

Finalmente, los resultados aparecen en una ventana nueva, a la derecha, como se muestra en la figura 6.

Si hay errores, estos se reportan donde salen los resultados, a la derecha. Después hay que corregirlos en la ventana de la izquierda, volver a guardar, y volver a correr el programa. Esto se repite tantas veces cuantas sean necesarias hasta garantizar que el programa funcione correctamente.

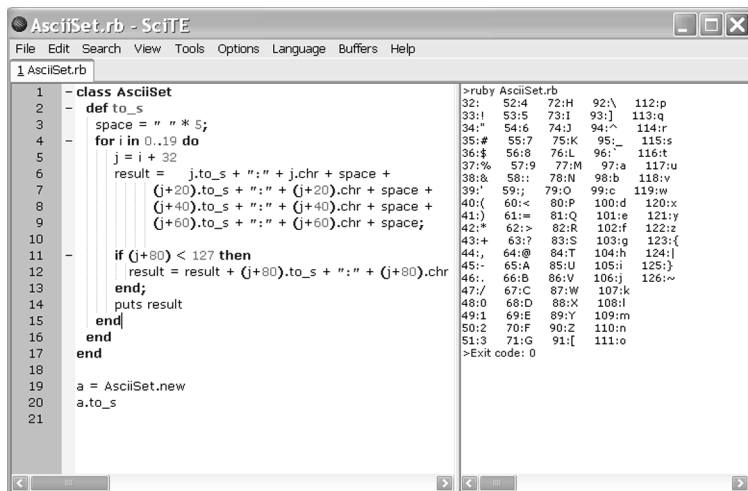


Figura B.6. Examinar los resultados

En la ventana de la izquierda, los signos +/- de la segunda columna permiten colapsar y expandir el código para facilitar su lectura y entenderlo mejor.

Sobre el Autor



Diego F Guillén Nakamura tiene un BSc en ingeniería Eléctrica y Electrónica de la Universidad de Los Andes (Bogotá, Colombia), y un MSc con especialización en Inteligencia Artificial de la Universidad de Soffa (Tokyo, Japón). Diego lleva ejerciendo su profesión más de 25 años, durante el cual ha usado más de 15 lenguajes de programación. Actualmente, Diego trabaja en el área de software para una multinacional japonesa, y vive en Gold Coast, Australia. En el presente, sus intereses académicos están en las áreas de matemáticas, bioinformática, lenguajes de programación, y Linux. En sus ratos libres Diego disfruta escuchando y practicando las armonías de la música choro y bossanova de Brazil en la guitarra clásica.

Este libro fue hecho, en su mayoría, con tecnologías Open Source, gratuitas:

- El sistema operativo usado fue Ubuntu Linux 7.10 (*Gutsy Gibbon*).
- Se usó OpenOffice 2.3 para la edición del libro, con tipos de letra FreeSerif 10 para el texto, y Courier 10 Pitch 7 para los ejemplos.
- Se usó GIMP 2.4.2 para la cubierta y la resolución de las imágenes.
- Algunas imágenes fueron hechas con Macromedia Fireworks MX versión 6, en Windows XP. [software commercial]
- Todo el código se desarrolló usando la versión 1.8.6 de Ruby (ensayada en ambos Windows XP y Ubuntu Linux).
- El browser utilizado fue Firefox 2.0.0.12.

Índice Alfanumérico

abs.....	9	detect.....	57
acos().....	11	Dir.....	50
acumuladores.....	20	directorios.....	50
and.....	22	do.....	19
append.....	12	downcase.....	13
aprender haciendo.....	v	downto().....	28
archivos.....	47	each.....	18, 27, 57
arrays.....	14	each_with_index().....	57
arreglo.....	14	editor de texto.....	23
ASCII.....	9, 76	else.....	25, 31
asignación.....	6	elsif.....	25
asin().....	11	empty?.....	14
atan().....	11	enteros,	8
atributo escribible.....	39	Enumerable.....	57
atributo legible.....	39	exp().....	11
atributos.....	37	expresiones regulares.....	45
attr.....	40	factorial.....	29, 33
attr_reader.....	39	falso.....	22
attr_writer.....	40	Fechas.....	16
binario.....	12	File.....	47
break.....	28	File.exists?().....	47
cadena.....	7	File.new().....	48
cadenas.....	12	File.open().....	47
capitalize.....	13	finite?.....	9
case.....	31	first.....	14
categorías.....	36	flatten.....	14
ceil.....	9	floor.....	9
chr.....	9	flujo del programa.....	25
Ciclos.....	27	for ... in.....	27
Clases.....	36	funciones.....	33
collect.....	57	funciones hiperbólicas.....	11
comentario.....	6	funciones trigonométricas.....	11
complejos.....	9	gsub().....	46
concat().....	14	has_key?().....	17
constructor.....	37	has_value?().....	17
contadores.....	20	Hash.....	17
corrientes.....	51	Hash.delete.....	17
cos().....	11	Hash.keys.....	17
cosh().....	11	Hash.sort.....	17
Date.....	16	Hash.values.....	17
day.....	16	Hash.values_at.....	17
def.....	33	Herencia.....	44
delete.....	13	hex.....	12

Horas.....	16	Open Source.....	1
hour.....	16	or.....	22
HTML.....	23	parámetros.....	33
hypot().....	11	patrones genéricos.....	45
if.....	25	polimorfismo de interfaces.....	57
igualdad.....	8	Pop.....	15
incremento.....	28	private.....	42
índice.....	14	propiedades.....	36
infinite?	9	protected.....	42
initialize().....	37	public.....	42
insert.....	13	punto flotante.....	8
insert().....	14	push.....	15
Instalar.....	3	puts.....	7
instancias.....	36, 38	Python.....	1
IO.....	47, 51	racionales.....	9
IO.lineno.....	48	Rails.....	1
IO.read().....	48	rango.....	27
IO.readlines().....	47	RDE.....	23
irb.....	6	reales, racionales, y complejos.....	8
iterar.....	18	recursividad.....	33
join().....	15	return.....	12, 33
last.....	14	reverse.....	13 s.
length.....	12	round.....	9
llaves.....	18	Ruby.....	1
log().....	11	Ruby interactivo.....	6
log10().....	11	scan().....	46
Lógica Condicional.....	25	SciTE.....	23 s., 77
loop.....	28	sec.....	16
map().....	57	serie de Taylor.....	29
matemática viva.....	vi	shift.....	15
Math.....	11	símbolos.....	41
Math::E.....	11	sin().....	11
Math::PI.....	11	sinh().....	11
matrices.....	29	sort.....	14
menor sorpresa.....	2	sprintf().....	53
métodos.....	37	sqrt.....	11
min.....	16	STDERR.....	51
mixins.....	57	STDIN.....	51
modulo.....	8	STDOUT.....	51
módulos.....	56	step().....	28
month.....	16	streams.....	51
new.....	36	strftime.....	16
now.....	16	Strings.....	12
números.....	8	sub().....	46
objeto.....	8, 36	subcadena.....	13
oct.....	12	subclase.....	44

superclase.....	44	year.....	16
swapcase.....	13	yield.....	55
tab.....	12	zero?	9
tan().....	11	14
tanh().....	11	-.....	7
Taxonomía.....	44	-=.....	20
taxonomías.....	36	@.....	37
Time.....	16	*.....	7
times.....	27	**.....	7
Tk.....	23	*=.....	21
to_f.....	8	/.....	7
to_s.....	8	/=.....	21
unless.....	26	\n.....	12
unshift.....	15	\t.....	12
until.....	28	#.....	6
upcase.....	13	%.....	8
upto().....	27	+.....	7
valores de verdad.....	22	+=.....	20
variable.....	6	<.....	22
variables de la instancia.....	36	<<.....	12, 15
variables globales.....	33	<=.....	22
vector.....	14	<=>.....	22
verdadero.....	22	=.....	7
vim.....	23	=().....	39
visibilidad.....	39	==.....	8, 22
wday.....	16	=>.....	17
when.....	31	=~.....	46
while.....	28	>.....	22
yday.....	16	>=.....	22