TTPS opción Ruby

Práctica 1

En esta práctica inicial del taller incorporaremos algunos conceptos de Git, la herramienta de control de versiones que utilizaremos a lo largo de la materia, y comenzaremos a tener contacto con el lenguaje de programación Ruby.

> Nota: para simplificar, todos los comandos que se muestran en esta práctica asumen un sistema operativo Ubuntu.

I. Repaso de Git

Prerequisitos

Para realizar esta parte de la práctica (y el resto de la materia también) necesitás tener Git instalado en la computadora donde vayas a hacer los ejercicios. Para saber si tenés Git instalado, podés ejecutar el comando `git` en una terminal y analizar la salida:

```
$ git --version
git version 2.9.2
```

> En esta y todas las prácticas, cuando estemos hablando de ejecutar comandos en una terminal vamos a denotar las líneas que tenés que ejecutar con un símbolo de prompt `$` si debés ejecutarlo con tu usuario o `#` si debés hacerlo con un usuario con privilegios de administrador (típicamente referenciado como el usuario `root`).

En el ejemplo anterior ejecutamos el comando `git --version` y obtuvimos la salida `git version 2.9.2`, lo cual indica que tenemos la versión `2.9.2` instalada y ya estamos listos para realizar los ejercicios de esta parte de la práctica.

Si al ejecutarlo recibiste un mensaje de error indicando que el comando `git` no fue encontrado, eso quiere decir que Git no está instalado en tu computadora, y por ende debés instalarlo ejecutando el siguiente comando:

```
# apt-get update -qq && apt-get install -y git
```

Una vez finalizada la instalación el comando `git` estará disponible para que lo uses.

Subcomandos Git

Git se maneja desde la línea de comandos mediante el uso de *subcomandos*. Cada tarea, orden o consulta que quieras hacer con Git la vas a poder realizar con el comando `git` y alguno de sus subcomandos, los cuales podés listar al ejecutar (solo) `git` o uno de los subcomandos más útiles que tiene Git: `git help`.

Cómo obtener ayuda

Git provee ayuda mediante las páginas del manual (o *man pages*) para cualquier subcomando que ofrezca, y para hacer aún más fácil la consulta de esa ayuda provee un subcomando especial que nos abre la página del manual del subcomando que querramos: `git help`.

Ejercicios

1. Ejecutá `git` o `git help` en la línea de comandos y mirá los subcomandos que tenés disponibles.
2. Ejecutá el comando `git help help`. ¿Cuál fue el resultado?
3. Utilizá el subcomando `help` para conocer qué opción se puede pasar al subcomando `add` para que ignore errores al agregar archivos.
4. ¿Cuáles son los estados posibles en Git para un archivo? ¿Qué significa cada uno?
5. Cloná el repositorio de materiales de la materia: `https://github.com/TTPS-ruby/capacitacion-ruby-ttps.git`. Una vez finalizado, ¿cuál es el *hash* del último commit que hay en el repositorio que clonaste?

   > Tip: `git log`.

6. ¿Para qué se utilizan los siguientes subcomandos?

   1. `init`
   2. `status`
   3. `log`
   4. `fetch`
   5. `merge`
   6. `pull`
   7. `commit`
   8. `stash`
   9. `push`
   10. `rm`
   11. `checkout`

7. Creá un archivo de texto en el repositorio que clonaste en el ejercicio **5** y verificá el estado de tu espacio de trabajo con el subcomando `status`. ¿En qué estado está el archivo que agregaste?

8. Utilizá el subcomando `log` para ver los commits que se han hecho en el repositorio, tomá cualquiera de ellos y copiá su *hash* (por ejemplo, `800dcba6c8bb2881d90dd39c285a81eabee5effa`), y luego utilizá el subcomando `checkout` para *viajar en el tiempo* (apuntar tu copia local) a ese commit. ¿Qué commits muestra ahora `git log`? ¿Qué ocurrió con los commits que no aparecen? ¿Qué dice el subcomando `status`?

9. Volvé al último commit de la rama principal (`master`) usando nuevamente el subcomando `checkout`. Corroborá que efectivamente haya ocurrido esto.

10. Creá un directorio vacío en el raiz del proyecto clonado. ¿En qué estado aparece en el `git status`? ¿Por qué?

11. Creá un archivo vacío dentro del directorio que creaste en el ejercicio anterior y volvé a ejecutar el subcomando `status`. ¿Qué ocurre ahora? ¿Por qué?

12. Utilizá el subcomando `clean` para eliminar los archivos no versionados (*untracked*) y luego ejecutá `git status`. ¿Qué información muestra ahora?

13. Actualizá el contenido de tu copia local mediante el subcomando `pull`.

II. Ruby: sintaxis y tipos básicos

Prerequisitos

Antes de realizar los ejercicios de esta parte, necesitás tener instalada la última versión de Ruby en tu computadora. Al momento de publicar esta práctica la última versión estable de Ruby es la `2.3.1`, por lo que instalaremos esa versión utilizando Rbenv y su *plugin* ruby-build. Rbenv permite usar distintas versiones de Ruby en nuestra computadora, y ruby-build es una extensión de Rbenv que simplifica la instalación de las versiones del lenguaje.

> Los pasos para instalar todas las partes que detallamos a continuación son un resumen rápido de los pasos que debés realizar para tener un ambiente de Ruby funcionando en tu computadora. Aquí obviaremos los detalles de cada paso, por lo que te recomendamos que leas los `README` de las dos herramientas que antes mencionamos para conocer más en profundidad cómo funcionan.

Ejecutá la siguiente secuencia de comandos para instalar Ruby 2.3.1 en tu computadora:

```
# apt-get install -y autoconf bison build-essential lib{ssl,yaml,sqlite3}-dev libread
line6{,-dev} zlib1g{,-dev}
$ git clone https://github.com/sstephenson/rbenv.git ~/.rbenv
$ cd ~/.rbenv && src/configure && make -C src
$ git clone https://github.com/sstephenson/ruby-build.git ~/.rbenv/plugins/ruby-build
$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
$ echo 'eval "$(rbenv init -)"' >> ~/.bashrc
$ source ~/.bashrc
$ rbenv install 2.3.1
$ rbenv global 2.3.1
```

Ejercicios

> Nota: al realizar estos ejercicios no utilices las sentencias de control `while`, `for` ni `repeat`.

1. Investigá y probá en un intérprete de Ruby cómo crear objetos de los siguientes tipos básicos usando literales y usando el constructor `new` (cuando sea posible):
    1. Arreglo (`Array`)
    2. Diccionario o *hash* (`Hash`)
    3. String (`String`)
    4. Símbolo (`Symbol`)
2. ¿Qué devuelve la siguiente comparación? ¿Por qué?

    ```
    'TTPS Ruby'.object_id == 'TTPS Ruby'.object_id
    ```
3. Escribí una función llamada `reemplazar` que reciba un `String` y que busque y reemplace en el mismo cualquier ocurrencia de `{` por `do\n` y cualquier ocurrencia de `}` por `\nend`, de modo que convierta los bloques escritos con llaves por bloques multilínea con `do` y `end`. Por ejemplo:

    ```
    reemplazar("3.times { |i| puts i }")
    # => "3.times do\n |i| puts i \nend"
    ```
4. Escribí una función que convierta a palabras la hora actual, dividiendo en los siguientes rangos los minutos:
    - Si el minuto está entre 0 y 10, debe decir "en punto",
    - si el minuto está entre 11 y 20, debe decir "y cuarto",
    - si el minuto está entre 21 y 34, debe decir "y media",
    - si el minuto está entre 35 y 44, debe decir "menos veinticinco" (de la hora siguiente),

- si el minuto está entre 45 y 55, debe decir "menos cuarto" (de la hora siguiente),
- y si el minuto está entre 56 y 59, debe decir "casi las" (y la hora siguiente)

Tomá como ejemplos los siguientes casos:

```
# A las 10:01
en_palabras(Time.now)
# => "Son las 10 en punto"
# A las 9:33
en_palabras(Time.now)
# => "Son las 9 y media"
# A las 9:45
en_palabras(Time.now)
# => "Son las 10 menos cuarto"
# A las 6:58
en_palabras(Time.now)
# => "Casi son las 7"
```

> Tip: resolver utilizando rangos numéricos

5. Escribí una función llamada `contar` que reciba como parámetro dos `string` y que retorne la cantidad de veces que aparece el segundo `string` en el primero, sin importar mayúsculas y minúsculas. Por ejemplo:

```
contar("La casa de la esquina tiene la puerta roja y la ventana blanca.", "la")
# => 5
```

6. Modificá la función anterior para que sólo considere como aparición del segundo `string` cuando se trate de palabras completas. Por ejemplo:

```
contar_palabras("La casa de la esquina tiene la puerta roja y la ventana blanca."
, "la")
# => 4
```

7. Dada una cadena cualquiera, y utilizando los métodos que provee la clase `String`, realizá las siguientes operaciones sobre el `string`:

    1. Imprimilo con sus caracteres en orden inverso.
    2. Eliminá los espacios en blanco que contenga.
    3. Convertí cada uno de sus caracteres por su correspondiente valor ASCII.
    4. Cambiá las vocales por números (`a` por `4`, `e` por `3`, `i` por `1`, `o` por `0`, `u` por `6`).

8. ¿Qué hace el siguiente código?

```
[:upcase, :downcase, :capitalize, :swapcase].map do |meth|
 "TTPS Ruby".send(meth)
end
```

9. Escribí una función que dado un arreglo que contenga varios `string` cualesquiera, retorne un nuevo arreglo donde cada elemento es la longitud del `string` que se encuentra en la misma posición del arreglo recibido como parámetro. Por ejemplo:

```
longitud(['TTPS', 'Opción', 'Ruby', 'Cursada 2015'])
# => [4, 6, 4, 12]
```

10. Escribí una función llamada `a_ul` que reciba un `Hash` y retorne un `String` con los pares de clave/valor del hash formateados en una lista HTML `<ul>`. Por ejemplo:

```
a_ul({ perros: 1, gatos: 1, peces: 0})
# => "<ul><li>perros: 1</li><li>gatos: 1</li><li>peces: 0</li></ul>"
```

11. Escribí una función llamada `rot13` que *encripte* un `string` recibido como parámetro utilizando el algoritmo `ROT13`. Por ejemplo:

```
rot13("¡Bienvenidos a la cursada 2015 de TTPS Opción Ruby!")
# => "¡Ovrairavqbf n yn phefnqn 2015 qr GGCF Bcpvóa Ehol!"
```

12. Escribí una función más genérica, parecida a la del ejercicio anterior, que reciba como parámetro un `string` y un número `n`, y que realice una *rotación* de `n` lugares de las letras del `string` y retorne el resultado. Por ejemplo:

```
rot("¡Bienvenidos a la cursada 2015 de TTPS Opción Ruby!", 13)
# => "¡Ovrairavqbf n yn phefnqn 2015 qr GGCF Bcpvóa Ehol!"
```

13. Escribí un *script* en Ruby que le pida al usuario su nombre y lo utilice para saludarlo imprimiendo en pantalla `¡Hola, <nombre>!`. Por ejemplo:

```
$ ruby script.rb
Por favor, ingresá tu nombre:
Matz
¡Hola, Matz!
```

14. Dado un color expresado en notación RGB, debés calcular su representación entera y hexadecimal,

donde la notación *entera* se define como `red + green*256 + blue*256*256` y la *hexadecimal* como el resultado de expresar en hexadecimal el valor de cada color y concatenarlos en orden. Por ejemplo:

```
notacion_hexadecimal([0, 128, 255])
# => '#0080FF'
notacion_entera([0, 128, 255])
# => 16744448
```

15. Investigá qué métodos provee Ruby para:

    1. Conocer la lista de métodos de una clase.
    2. Conocer la lista de métodos de instancia de una clase.
    3. Conocer las variables de instancia de una clase.
    4. Obtener la lista de ancestros (*superclases*) de una clase.

16. Escribí una función que encuentre la suma de todos los números naturales múltiplos de `3` ó `5` menores que un número `tope` que reciba como parámetro.

17. Cada nuevo término en la secuencia de Fibonacci es generado sumando los 2 términos anteriores. Los primeros 10 términos son: `1`, `1`, `2`, `3`, `5`, `8`, `13`, `21`, `34`, `55`. Considerando los términos en la secuencia de Fibonacci cuyos valores no exceden los 4 millones, encontrá la suma de los términos pares.