Práctica 1: Sintaxis y conceptos básicos

En esta primera práctica del lenguaje Ruby vamos a comenzar a conocer su sintaxis, las construcciones que nos ofrece para trabajar con sus tipos básicos y estos tipos en sí.

> Tené en cuenta que a partir de esta práctica es deseable que realices los ejercicios en computadora, para lo cual vas a necesitar un ambiente de desarrollo de Ruby funcionando. Si aún no lo hiciste, te recomendamos que revises primero la Explicación 0: Instalación de Ruby y cuando termines de configurar el ambiente, vuelvas a esta práctica para hacer los ejercicios. Si bien en general es indistinto de qué forma tengas instalado Ruby y qué versión del lenguaje utilices, es importante que utilices una versión relativamente reciente (`3.3` o posterior).

Ejercicios

> Los ejercicios de esta práctica están pensados para ser resueltos sin utilizar las estructuras de control **`while`**, **`for`** o `repeat`. En su lugar, cuando tengas que iterar sobre elementos probá los métodos de iteración que el lenguaje provee.

1. Investigá y probá en un intérprete de Ruby (`irb`, por ejemplo) cómo crear objetos de los siguientes tipos básicos, tanto mediante el uso de literales como utilizando el constructor **`new`** (cuando sea posible):
   - `Symbol`
   - `String`
   - `Array`
   - `Hash`
2. ¿De qué forma(s) se puede convertir un objeto (cualquiera fuere su tipo o clase) en `String`?
3. ¿De qué forma(s) se puede convertir un objeto `String` en un símbolo?
4. ¿Qué devuelve la siguiente comparación? ¿Por qué?

   ```ruby
   'TTPS Ruby'.object_id == 'TTPS Ruby'.object_id
   ```

5. Escribí una función llamada `reemplazar` que, dado un `String` que recibe como parámetro, busque y reemplace en éste 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:

   ```ruby
   reemplazar("3.times { |i| puts i }")
   # => "3.times do\n |i| puts i \nend"
   ```

6. Escribí una función que exprese en palabras la hora que recibe como parámetro, según las siguientes reglas:

- 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” con la hora siguiente,
- si el minuto está entre 45 y 55, debe decir “menos cuarto” con la hora siguiente,
- y si el minuto está entre 56 y 59, debe decir “Casi son las” con la hora siguiente

Tomá como ejemplos los siguientes casos:

```
tiempo_en_palabras(Time.new(2025, 10, 21, 10, 1))
# => "Son las 10 en punto"
tiempo_en_palabras(Time.new(2025, 10, 21, 9, 33))
# => "Son las 9 y media"
tiempo_en_palabras(Time.new(2025, 10, 21, 8, 45))
# => "Son las 9 menos cuarto"
tiempo_en_palabras(Time.new(2025, 10, 21, 6, 58))
# => "Casi son las 7"
tiempo_en_palabras(Time.new(2025, 10, 21, 0, 58))
# => "Casi es las 1"
```

Es importante considerar que cuando la hora es 1, la forma correcta de expresarla no es *“Son las 1 en punto”*, sino *“Es la 1 en punto”*. Esto debe tenerse en cuenta en cada uno de los casos expresados en el enunciado de este ejercicio.

> Tip: resolver utilizando rangos numéricos

7. 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, en una búsqueda *case-insensitive* (sin distinguir mayúsculas o minúsculas). Por ejemplo:

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

8. 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
```

9. Dada una cadena cualquiera, y utilizando los métodos que provee la clase `String`, realizá las siguientes operaciones sobre dicha cadena, implementando métodos que funcionen de la siguiente forma:

   - `string_reverso`: retorna el string con los caracteres en orden inverso.

- `string_sin_espacio`: elimina los espacios en blanco que contenga.
- `string_a_arreglo_ascii`: retorna un arreglo con cada uno de los caracteres convertidos a su correspondiente valor ASCII.
- `string_remplaza_vocal`: cambia las vocales por números:
  - `"a"` o `"A"` por `"4"`,
  - `"e"` o `"E"` por `"3"`,
  - `"i"` o `"I"` por `"1"`,
  - `"o"` u `"O"` por `"0"`,
  - y `"u"` o `"U"` por `"6"`.

10. ¿Cuál es el valor de retorno del siguiente código?

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

11. Tomando el ejercicio anterior como referencia, ¿en qué situaciones usarías los métodos `send` y `public_send` definidos en la clase `Object`? ¿Cuál es la principal diferencia entre esos dos métodos?

12. Escribí una función `longitud` 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:

```ruby
longitud(['TTPS', 'Opción', 'Ruby', 'Cursada 2025'])
# => [4, 6, 4, 12]
```

13. Escribí una función llamada `listar` que reciba un `Hash` y retorne un `String` con los pares de clave/valor formateados en una lista ordenada en texto plano. Por ejemplo:

```ruby
listar({ perros: 2, gatos: 2, peces: 0, aves: 0 })
# => "1. perros: 2\n2. gatos: 2\n3. peces: 0\n4. aves: 0"
```

14. Mejorar la función anterior en una nueva llamada `listar_mejorada` para que además reciba opcionalmente un parámetro llamado `pegamento` (su valor por defecto debe ser `": "`) que sea el que utilice para unir los pares de clave/valor. Por ejemplo:

```ruby
listar_mejorada({ perros: 2, gatos: 2, peces: 0, aves: 0 }, " -> "
    )
# => "1. perros -> 2\n2. gatos -> 2\n3. peces -> 0\n4. aves -> 0"
```

15. Escribí un método que reciba un argumento y retorne un valor booleano indicando si la cadena recibida como argumento es pentavocálica o panvocálica (contiene todas las vocales). El chequeo no debe ser sensible a minúsculas y mayúsculas.

16. Escribí un script en Ruby que le pida a quien lo ejecute que ingrese su nombre por entrada estándar (el teclado), y que lo utilice para saludarl@ imprimiendo en pantalla, por ejemplo:

```
$ ruby script.rb
Por favor, ingresá tu nombre:
Pepe
Hola, Pepe
```

17. Escribí un nuevo script, que de manera similar al implementado en el punto anterior haga el saludo usando un nombre que se provea, pero que en lugar de solicitar que el nombre se ingrese por entrada estándar, éste se reciba como argumento del script. Por ejemplo:

```
$ ruby script.rb Pepe
Hola, Pepe
```

> Tip: investigá cómo se puede trabajar con los argumentos que recibió el script Ruby en su ejecución.

18. Implementá las funciones necesarias para que, dado un color expresado en notación RGB, se pueda obtener su representación en las notaciones entera y hexadecimal. La notación entera se define como `red + green * 256 + blue * 256 * 256` y la hexadecimal como el resultado de expresar en base 16 el valor de cada color y concatenarlos en orden. Por ejemplo:

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

19. Investigá qué métodos provee Ruby para:

    - Obtener la lista de ancestros (superclases) de una clase.
    - Conocer la lista de métodos de una clase.
    - Conocer la lista de métodos de instancia de una clase.
    - Conocer las variables de instancia de una clase.
    - Obtener el valor de una variable de instancia de un objeto (sin utilizar un método generado con `attr_reader` o similar) accediéndolo desde fuera de éste.
    - Establecer el valor de una variable de instancia de un objeto (sin utilizar un método generado con `attr_writer` o similar) desde fuera de éste.

20. Escribí una función que encuentre la suma de todos los números naturales múltiplos de 3 y 5 (ambos) que sean menores que un número `tope` que reciba como parámetro. Por ejemplo:

```
multiplos_de_3_y_5(100)
# => 315
```

21. Creá otra función, similar a la anterior, que busque los números naturales múltiplos de *N* números enteros que recibirá como parámetro en un arreglo. Por ejemplo:

```
multiplos_de([3, 5], 100)
# => 315
multiplos_de([3, 5, 17, 28, 65], 100_000)
# => 92820
```