TTPS opción Ruby

Práctica 3

Esta práctica incorpora ejercicios sobre excepciones, tanto definición y manejo de las misma como un breve sondeo de las principales clases de excepción, y a su vez introduce algunos ejercicios sobre testing en Ruby.

Excepciones

1. Investigá la jerarquía de clases que presenta Ruby para las excepciones. ¿Para qué se utilizan las siguientes clases?
    - `IOError`
    - `NameError`
    - `RuntimeError`
    - `NotImplementedError`
    - `StopIteration`
    - `TypeError`
    - `SystemExit`
2. ¿Cuál es la diferencia entre `raise` y `throw`? ¿Para qué usarías una u otra opción?
3. ¿Para qué sirven `begin .. rescue .. else` y `ensure`? Pensá al menos 2 casos concretos en que usarías estas sentencias en un script Ruby.
4. ¿Para qué sirve `retry`? ¿Cómo evitarías caer en un loop infinito al usarla?
5. ¿Cuáles son las diferencias entre los siguientes métodos?

```
def opcion_1
 a = [1, nil, 3, nil, 5, nil, 7, nil, 9, nil]
 b = 3
 c = a.map do |x|
   x * b
 end
 puts c.inspect
rescue
 0
end

def opcion_2
 c = begin
       a = [1, nil, 3, nil, 5, nil, 7, nil, 9, nil]
       b = 3
       a.map do |x|
         x * b
       end
     rescue
       0
     end
 puts c.inspect
end

def opcion_3
 a = [1, nil, 3, nil, 5, nil, 7, nil, 9, nil]
 b = 3
 c = a.map { |x| x * b } rescue 0
 puts c.inspect
end

def opcion_4
 a = [1, nil, 3, nil, 5, nil, 7, nil, 9, nil]
 b = 3
 c = a.map { |x| x * b rescue 0 }
 puts c.inspect
end
```

6. Suponé que tenés el siguiente script escrito en Ruby y se te pide que lo hagas *resiliente* (tolerante a fallos), intentando siempre que se pueda recuperar la situación y volver a intentar la operación que falló. Agregá las modificaciones que consideres necesarias para lograr que el script sea más robusto.

```
# Este script lee una secuencia de no menos de 15 números desde teclado y lueg
o imprime el resultado de la división
# de cada número por su entero inmediato anterior.

# Como primer paso se pide al usuario que indique la cantidad de números que i
ngresará.
cantidad = 0
while cantidad < 15
 puts '¿Cuál es la cantidad de números que ingresará? Debe ser al menos 15'
 cantidad = gets.to_i
end

# Luego se almacenan los números
numeros = 1.upto(cantidad).map do
 puts 'Ingrese un número'
 numero = gets.to_i
end

# Y finalmente se imprime cada número dividido por su número entero inmediato
anterior
resultado = numeros.map { |x| x / (x - 1) }
puts 'El resultado es: %s' % resultado.join(', ')
```

7. Partiendo del script del inciso anterior, implementá una nueva clase de excepción que se utilizará para indicar que la entrada del usuario no es un valor numérico entero válido. ¿De qué clase de la jerarquía de `Exception` heredaría?

8. Sea el siguiente código:

```
def fun3
 puts "Entrando a fun3"
 raise RuntimeError, "Excepción intencional"
 puts "Terminando fun3"
rescue NoMethodError => e
 puts "Tratando excepción por falta de método"
rescue RuntimeError => e
 puts "Tratando excepción provocada en tiempo de ejecución"
rescue
 puts "Tratando una excepción cualquiera"
ensure
 puts "Ejecutando ensure de fun3"
end

def fun2(x)
 puts "Entrando a fun2"
 fun3
 a = 5 / x
 puts "Terminando fun2"
end

def fun1(x)
 puts "Entrando a fun1"
 fun2 x
rescue
 puts "Manejador de excepciones de fun1"
 raise
ensure
 puts "Ejecutando ensure de fun1"
end

begin
 x = 0
 begin
   fun1 x
 rescue Exception => e
   puts "Manejador de excepciones de Main"
   if x == 0
     puts "Corrección de x"
     x = 1
     retry
   end
 end
 puts "Salida"
end
```

1. Seguí el flujo de ejecución registrando la traza de impresiones que deja el programa y

justificando paso a paso.

2. ¿Qué pasaría si se permuta, dentro de `fun3`, el manejador de excepciones para `RuntimeError` y el manejador de excepciones genérico (el que tiene el rescue vacío)?

3. ¿La palabra reservada `retry` que función cumple? ¿Afectaría el funcionamiento del programa si se mueve la línea `x = 0` dentro del segundo `begin` (inmediatamente antes de llamar a `fun1` con `x`)?

Testing

> Nota: Para esta práctica utilizaremos `MiniTest` en cualquiera de sus variantes (`minitest/unit` o `minitest/spec`).

1. ¿En qué consiste la metodología TDD? ¿En qué se diferencia con la forma tradicional de escribir código y luego realizar los tests?

2. Dado los siguientes tests, escribí el método correspondiente (el que se invoca en cada uno) para hacer que pasen:

```
require 'minitest/autorun'
require 'minitest/spec'

describe '#incrementar' do
  describe 'cuando el valor es numérico' do
    it 'incrementa el valor en un delta recibido por parámetro' do
      x = -9
      delta = 10
      assert_equal(1, incrementar(x, delta))
    end

    it 'incrementa el valor en un delta de 1 unidad por defecto' do
      x = 10
      assert_equal(11, incrementar(x))
    end
  end

  describe 'cuando el valor es un string' do
    it 'arroja un RuntimeError' do
      x = '10'
      assert_raises(RuntimeError) do
        incrementar(x)
      end
      assert_raises(RuntimeError) do
        incrementar(x, 9)
      end
    end
  end
end

describe '#concatenar' do
  it 'concatena todos los parámetros que recibe en un string, separando por es
pacios' do
    class Dummies; end

    assert_equal('Lorem ipsum 4 Dummies', concatenar('Lorem', :ipsum, 4, Dummi
es))
  end

  it 'Elimina dobles espacios si los hubiera en la salida final' do
    assert_equal('TTPS Ruby', concatenar('TTPS', nil, '      ', "\t", "\n", 'R
uby'))
  end
end
```

3. Implementá al menos 3 tests para cada uno de los siguientes ejercicios de las prácticas anteriores:

    1. De la práctica 1:

2. 4 (`en_palabras`)
3. 5 (`contar`)
4. 6 (`contar_palabras`)
5. 9 (`longitud`)
6. De la práctica 2:
7. 1 (`ordenar_arreglo`)
8. 2 (`ordenar`)
9. 4 (`longitud`)
10. 14 (`opposite`)
11. 16 (`da_nil?`)

4. Implementá los tests que consideres necesarios para probar el *Mixin* `Countable` que desarrollaste en el ejercicio 11 de la práctica 2, sin dejar de cubrir los siguientes puntos:

    - Testear en una clase existente
    - Testear en una clase creada únicamente con el propósito de testear
    - Testear qué ocurre antes de que se invoque el método del que se está contando las invocaciones
    - Testear la inicialización correcta del *Mixin*
    - Testear algún caso extremo que se te ocurra
5. Suponé que tenés que desarrollar una función llamada ‘expansor’ la cual recibe un string (conformado únicamente con letras) y devuelve otro string donde cada letra aparezca tantas veces según su lugar en el abecedario. Un ejemplo simple sería:

    ```
    expansor 'abcd'
    # => 'abbcccdddd'
    ```

    A continuación se presentará su especificación (sin implementar):

```
require 'minitest/autorun'
require 'minitest/spec'

describe 'expansor' do
  # Casos de prueba con situaciones y/o entradas de datos esperadas
  describe 'Casos felices' do
    describe 'cuando la entrada es el string "a"' do
      it 'debe devolver "a"'
    end

    describe 'cuando la entrada es el string "f"' do
      it 'debe devolver "ffffff"'
    end

    describe 'cuando la entrada es el string "escoba"' do
      it 'debe devolver "eeeeessssssssssssssssscccooooooooooooooobba"'
    end
  end

  # Casos de pruebas sobre situaciones inesperadas y/o entradas de datos anóma
las
  describe 'Casos tristes' do
    describe 'cuando la entrada no es un string' do
      it 'debe disparar una excepción estándar con el mensaje "La entrada no e
s un string"'
    end

    describe 'cuando la entrada es el string vacío' do
      it 'debe disparar una excepción estándar con el mensaje "El string es va
cío"'
    end

    describe 'cuando la entrada es el string "9"' do
      it 'debe disparar un excepción estándar con el mensaje "El formato del s
tring es incorrecto"'
    end

    describe 'cuando la entrada es el string "*"' do
      it 'debe disparar una excepción estándar con el mensaje "El formato del
string es incorrecto"'
    end
  end
end
```

1. Completar la especificación de los casos de prueba.
2. Implementar la función `expansor` y verificar que todos los casos pasen.