Conceptos y Paradigmas de Lenguajes de Programación

Trabajo integrador 2024

Grupo N° 32 - Participantes:

Dellarupe Franco - 21239/0
Ponce Gregorio - 21361/2
Romagnoli Leandro - 19505/9
Spadari Pedro - 21586/8

Lenguajes asignados:

Principal: JavaScript
Secundario: Python

Bibliografía consultada:

https://developer.mozilla.org/en-US/docs/Web/JavaScript
https://tc39.es/ecma262/
https://docs.python.org/3.12/
https://developer.mozilla.org/en-US/docs/Glossary/Type_coercion
https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

Punto A:

Abstracción: capacidad de definir y usar estructuras u operaciones complejas sin necesidad de conocer todos sus detalles.

- JavaScript

```
1.    function calcularPromedio(numeros) {
2.        let suma = 0
3.        for (let i = 0; i < numeros.length; i++) {
4.            suma += numeros[i]
5.        }
6.        return suma / numeros.length
7.    }
8.    let listaNumeros = [10, 20, 30, 40, 50]
9.    let promedio = calcularPromedio(listaNumeros)
10.   console.log("El promedio es: ", promedio)
```

Este es un código en JavaScript el cual permite que una persona solo necesite saber qué parámetros debe enviar a la función para que esta calcule su promedio. No necesita saber cómo está implementada, solo utilizarla pasándole una lista de números.

- Python

```
1.    def calcular_promedio(numeros):
2.        suma = sum(numeros)
3.        return suma / len(numeros)
4.
5.    lista_numeros = [10, 20, 30, 40, 50]
6.    promedio = calcular_promedio(lista_numeros)
7.    print("El promedio es: ", promedio)
```

Este código hace lo mismo que el anterior, la única diferencia es la forma de declarar la función y que se utiliza una función ya integrada en el lenguaje, lo que demuestra también este principio, ya que no necesitamos saber cómo está definida la función para utilizarla y entender que suma todos los elementos de la lista de números pasada.

Ortogonalidad: un conjunto pequeño de constructores primitivos puede ser combinado a la hora de construir estructuras de control y datos.

- JavaScript

```
1.    let a = 5
2.    let b = 10
3.    let c = 15
4.
5.    if (a < b && b < c) {
6.        console.log("Los números están en orden
 ascendente.")
7.    } else {
```

```
8.        console.log("Los números no están en orden
 ascendente.")
9.    }
```

En este código se utilizan distintos operadores en una misma expresión booleana.

Se utiliza el operador lógico && que verifica que ambas sentencias sean verdaderas para dar paso a las instrucciones siguientes, pero dentro de cada uno de los lados del operador se utilizan operadores de comparación que sirven para verificar los valores.

Sigue el principio de ortogonalidad ya que, se pueden combinar los distintos constructores primitivos de manera de obtener expresiones complejas, sin perder la consistencia.

- Python

```
1.    a = 5
2.    b = 10
3.    c = 15
4.
5.    if a < b < c:
6.        print("Los números están en orden ascendente.")
7.    else:
8.        print("Los números no están en orden ascendente.")
```

Al igual que en el anterior, se utilizan dos tipos de operadores distintos para controlar el flujo del programa según el valor de las variables.

En este caso la instrucción a < b < c combina (a < b) and (b < c).

Sigue el principio de ortogonalidad porque construye expresiones complejas a partir de un conjunto de constructores primitivos de manera consistente.

Punto B:

Estructura if

- JavaScript

```
1.    let nota = 85
2.
3.    if (nota >= 90) {
4.        console.log("Aprobado - Excelente")
5.    } else if (nota >= 70) {
6.        console.log("Aprobado - Bien")
7.    } else {
8.        console.log("Desaprobado")
9.    }
```

- Python

```
1.    nota = 85
2.
3.    if nota >= 90:
4.        print("Aprobado - Excelente")
```

```
5.    elif nota >= 70:
6.        print("Aprobado - Bien")
7.    else:
8.        print("Desaprobado")
```

Las diferencias sintácticas con respecto a la estructura de control if son las siguientes:

1. En JavaScript los bloques de código que se ejecutan se encuentran encerrados entre llaves. En Python, luego del símbolo ":", es la indentación la que marca a qué instrucción corresponde cada bloque.
2. En JavaScript es obligatorio el uso de paréntesis para evaluar las condiciones. En Python si bien está permitido su uso, no es de carácter obligatorio.
3. En JavaScript para realizar un nuevo if en la sentencia else de un if previo, se debe usar 'else if', mientras que en Python se debe usar 'elif'.

Estructura for

- JavaScript

```
1.    let palabras = ["Hola", "Mundo", "JavaScript"];
2.
3.    for (let i = 0; i < palabras.length; i++) {
4.        if (palabras[i].length > 5) {
5.            console.log(palabras[i].toUpperCase());
6.        } else {
7.            console.log(palabras[i].toLowerCase());
8.        }
9.    }
```

- Python

```
1.    palabras = ["Hola", "Mundo", "Python"]
2.
3.    for palabra in palabras:
4.        print(palabra)
5.    else:
6.        print("Fin del loop!")
```

Las diferencias sintácticas con respecto a la estructura de control for son las siguientes:

1. En JavaScript se necesita inicializar una variable índice, marcar el límite y definir de qué forma se modifica en cada iteración el índice, todo esto está encerrado entre paréntesis y el bloque de código a reiterar se encierra entre llaves. Mientras que en el caso de Python se utiliza una variable temporal la cual será llamada para ser utilizada y que en cada repetición se modificará por el siguiente valor en la lista, el bloque de código a ejecutar se encuentra luego de ":" y tendrá que estar indentado.
2. En el caso de Python, se puede indicar un bloque else opcional, el cual se ejecutará cuando el bucle termine (excepto cuando termine debido a una sentencia break).

Nota: en JavaScript hay múltiples maneras de realizar un loop for, como pueden serlo las sentencias for…of, que se puede utilizar para objetos iterables

como String, Array, Map, Set, entre otros; .forEach(), que se puede utilizar para realizar un loop entre los elementos de un array; o for…in, que itera sobre todas las propiedades enumerables de un objeto.

Punto C:

- JavaScript

```
1.    console.log(variable_no_declarada) // Error de
 semántica estática: error semántico estático al intentar
 acceder a una variable que nunca fue declarada.
2.
3.    let persona = {
4.        nombre: "Juan"
5.    };
6.
7.    console.log(persona.apellido.toUpperCase())// Error de
 semántica dinámica: error semántico dinámico al intentar
 acceder a la propiedad toUpperCase() de la propiedad
 apellido del objeto persona, la cual no está definida.
```

- Python

```
1.    def es_doble_digito(num):
2.        return (num > 9 & num < 100)
3.
4.    def resto(num1, num2):
5.        return num1 % num2
6.
7.    resultado1 = es_doble_digito() # Error de semántica
 estática: error semántico estático al llamar a la función
 es_doble_digito sin parámetros cuando tiene uno.
8.
9.    resultado2 = resto(10, 0)   # Error de semántica
 dinámica: error semántico dinámico al hacer módulo por 0.
10.   print(resultado1, resultado2)
```

Nota: si bien en el ejercicio se mencionan algunos errores como de semántica estática, en la práctica, dado que ambos lenguajes son interpretados, dichos errores recién se pueden detectar en ejecución.

Punto D:

Para este punto vamos a describir los tipos de variables permitidos tanto en JavaScript y Python según su tiempo de vida.

Comencemos definiendo que es tiempo de vida: El tiempo de vida de una variable o de un objeto es el periodo en el cual este se encuentra activo en memoria. Este periodo depende del tipo de objeto o de variable, cómo se crea y cómo es utilizado en el programa.

- JavaScript

```
1.    const variableGlobal1 = "Global solo lectura"
2.    var variableGlobal2= "Global lectura y escritura"
```

```
3.    function ejemplo(){
4.        var variableLocal= "Local a la función"
5.        if (true){
6.            let variableBloque= "Local al bloque if"
7.            console.log("Global 1: " + variableGlobal1)
8.            console.log("Global 2: " + variableGlobal2)
9.            console.log("Local 1: " + variableLocal)
10.           console.log("Local 2: " + variableBloque)
11.       }
12.   }
13.   ejemplo()
```

Nombre	Momento alocación	R-valor	Alcance	Tiempo vida
variableGlobal1	automático	"Global solo lectura"	2-13	1-13
variableGlobal2	automático	indefinido (basura)	1-13	1-13
ejemplo	-	-	4-13	3-12
variableLocal	automático	indefinido (basura)	3-12	3-12
variableBloque	automático	"Local al bloque if"	7-11	5-11

- Python

```
1.    variable_global = "Variable global"
2.
3.    def ejemplo():
4.        variable_local = "Variable local"
5.        print(variable_local)
6.        global variable_global # Se debe indicar con la
 palabra reservada global que se hace referencia a la
 variable previamente definida y no se quiere definir una
 nueva, local, con el mismo nombre
7.        variable_global += " (modificada dentro de
 ejemplo)"
8.
9.    print(variable_global)
10.   ejemplo()
11.   print(variable_global)
```

Nombre	Momento alocación	R-valor	Alcance	Tiempo vida
variable_global	automático	"Variable global"	2-11	1-11
ejemplo	-	-	4-11	3-7
variable_local	automático	"Variable local"	5-7	3-7

Parte 2:

Punto A:

- JavaScript

```
1.    function saludar(nombre = 'Invitado', saludo = 'Hola') {
2.        return `${saludo}, ${nombre}`;
3.    }
4.    console.log(saludar()); // Hola, Invitado
5.    console.log(saludar('Juan')); // Hola, Juan
6.    console.log(saludar('Ana', 'Buenos días')); // Buenos
 días, Ana
7.
8.
9.    function multiplicar(a, b) {
10.       return a * b;
11.   }
12.   console.log(multiplicar(5, 3)); // 15
13.
14.   function mostrarDetalles({ nombre, edad }) {
15.       console.log(`Nombre: ${nombre}, Edad: ${edad}`);
16.   }
17.   const persona = { nombre: 'Carlos', edad: 28 };
18.   mostrarDetalles(persona); // Nombre: Carlos, Edad: 28
19.
20.   function mostrarPrimeros([primero, segundo]) {
21.       console.log(`Primero: ${primero}, Segundo:
 ${segundo}`);
22.   }
23.   const numeros = [10, 20];
24.   mostrarPrimeros(numeros); // Primero: 10, Segundo: 20
25.
26.   function sumar(...numeros) {
27.       return numeros.reduce((acumulado, actual) => acumulado
 + actual, 0);
28.   }
29.   console.log(sumar(1, 2, 3, 4)); // 10
30.   console.log(sumar(5, 10, 15)); // 30
31.
32.   function mostrarDetalles2({ nombre, edad, ...titulos }) {
33.       console.log(`Nombre: ${nombre}, Edad: ${edad},
 Titulos: ` + JSON.stringify(titulos));
34.   }
35.   const persona2 = { nombre: 'Carlos', edad: 28, titulo1:
 'Doctor', titulo2: 'Herrero', titulo3: 'Maestro' };
36.   mostrarDetalles2(persona2); // Nombre: Carlos, Edad: 28,
 Titulos:
 {"titulo1":"Doctor","titulo2":"Herrero","titulo3":"Maestro"}
```

Antes de explicar los distintos tipos de parámetros que existen debemos aclarar que ningún tipo de parámetros necesita especificar tipos ya que este es un lenguaje de tipado dinámico.
Tipos de parámetros en javascript:

- Parámetros predefinidos: Los parámetros predefinidos son un tipo que nos permite indicar algún tipo de valor para cuando este no es indicado. La ligadura es posicional y los valores por defecto se evalúan en tiempo de llamada.
- Parámetros sin predefinidos: Son los parámetros estándar que se pasan a una función en el orden en que están definidos (ligadura posicional). Si un parámetro no recibe ningún valor este se define como “undefined”.
- Los parámetros pueden ser desestructurados directamente en la firma de la función, lo que permite una asignación más clara y legible de objetos y arrays. Su ligadura es posicional.
- Parámetros rest: Los parámetros rest en JavaScript proporcionan una forma flexible y concisa de manejar un número variable de argumentos en las funciones. Se definen usando el operador `...`, deben ser el último parámetro en la lista de parámetros y convierten los argumentos restantes en un array. Son especialmente útiles cuando se necesita trabajar con funciones que aceptan un número indefinido de argumentos, ofreciendo una sintaxis clara y la capacidad de utilizar métodos de array directamente.

- Python

```python
import math

def calcular_area(figura, *args, precision=2):
    """
    Parámetros:
    - figura (str): Tipo de figura ('circulo' o 'rectangulo') (obligatorio, posicional).
    - *args: Argumentos adicionales dependientes de la figura
    - precision (int): Número de decimales para redondear el resultado (opcional, por nombre, valor por defecto: 2).
    """
    if figura == 'circulo' and len(args) == 1:
        radio = args[0]
        area = math.pi * radio ** 2
    elif figura == 'rectangulo' and len(args) == 2:
        base, altura = args
        area = base * altura
    else:
        raise ValueError("Figura no soportada o cantidad incorrecta de argumentos.")

    return round(area, precision)
```

```
20.
21.   # Ejemplos de uso
22.
23.   # Calcular el área de un círculo con radio 3
24.   area_circulo = calcular_area('circulo', 3)
25.   print(f"Área del círculo: {area_circulo}")  # Salida: Área
 del círculo: 28.27
26.
27.   # Calcular el área de un rectángulo con base 5 y altura 10
28.   area_rectangulo = calcular_area('rectangulo', 5, 10)
29.   print(f"Área del rectángulo: {area_rectangulo}")  #
 Salida: Área del rectángulo: 50
30.
31.   # Calcular el área de un círculo con radio 4 y especificar
 una precisión de 3 decimales
32.   area_circulo_preciso = calcular_area('circulo', 4,
 precision=3)
33.   print(f"Área del círculo con precisión:
 {area_circulo_preciso}")  # Salida: Área del círculo con
 precisión: 50.265
34.
35.   # Intentar calcular el área de una figura no soportada
 (lanza una excepción)
36.   # calcular_area('triangulo', 3)  # ValueError: Figura no
 soportada o cantidad incorrecta de argumentos.
```

Antes de explicar los distintos tipos de parámetros que existen debemos aclarar que ningún tipo de parámetros necesita especificar tipos ya que este es un lenguaje de tipado dinámico.

Tipos de parámetros:

- Parámetros opcionales: En una función Python se pueden indicar una serie de parámetros opcionales. Son parámetros que se indican con un valor por defecto y si no se pasan al invocar a la función entonces toman este valor.
- Parámetros posicionales: Los parámetros posicionales se asignan basándose en el orden en que se pasan a la función. Deben ser proporcionados en la llamada a la función a menos que tengan valores por defecto.
- Parámetros por nombre: Permiten a los desarrolladores pasar argumentos a una función utilizando el nombre del parámetro, lo que puede mejorar la claridad y la flexibilidad del código. Los argumentos se pasan utilizando el nombre del parámetro, lo que permite especificar explícitamente a qué parámetro se está asignando un valor. Los parámetros por nombre no requieren ser pasados en el orden en que se definen en la función, lo que puede mejorar la legibilidad. Los parámetros por nombre pueden tener valores por defecto, lo que permite omitir estos argumentos en la llamada a la función si no es necesario modificarlos.

Punto B:

La fortaleza de tipos se ve definida por las restricciones que implementa el lenguaje a la hora de evaluar cómo las operaciones que involucran valores de diferentes tipos pueden operarse. Un sistema de tipado fuerte define reglas estrictas para asegurar que las operaciones que se realizan sean válidas y compatibles entre tipos de datos, mientras que un sistema débil permite más flexibilidad, permitiendo conversiones implícitas entre tipos diferentes.

- JavaScript

```
1.    let x = '5';
2.    let y = 10;
3.    let result = x + y; // '510'
4.    console.log(result); // La cadena '5' se concatena con el
 número 10.
5.    let a = 1;
6.    let b = true;
7.    let sum = a + b; // 2
8.    console.log(sum); // true se convierte implícitamente en
 1.
```

Este es un lenguaje débilmente tipado, lo que significa que permite conversiones automáticas entre tipos de datos, lo que facilita el desarrollo, pero puede llevar a errores difíciles de detectar.

En este código se ve la concatenación de cadenas y números con conversión implícita de tipos.

Cuando una operación de suma (+) involucra una cadena, JavaScript convierte el otro operando a cadena también. En x + y, la variable y (que es un número) se convierte a cadena y se concatena con x, resultando en la cadena '510'.

Los valores booleanos true y false se convierten implícitamente a 1 y 0 respectivamente cuando se utilizan en contextos numéricos. Así, en a + b, b (que es true) se convierte a 1 y se suma a a, resultando en 2.

- Python

```
1.    x = '5'
2.    y = 10
3.    try:
4.        z = x + y
5.    except TypeError as e:
6.        print(e)   # No se puede concatenar 'str' y 'int'.
7.
8.    a = 1
9.    b = True
10.   c = a + b
11.   print(c)   # 2 (aunque Python permite esta suma, sigue
 siendo un tipado fuerte ya que no hay conversión implícita
 entre str y int)
```

Este es un lenguaje con un sistema de tipos fuerte, lo que significa que no permite operaciones entre tipos de datos diferentes sin una conversión explícita.

En la primera parte del código se lanza la excepción TypeError ya que al no haber una conversión explícita de tipos, por ejemplo con la instrucción int(x) que cambiaría el tipo de la variable x, el lenguaje no permite la concatenación de una cadena y un número.

En Python, aunque True se convierte a 1 cuando se suma con un número, esto es porque en Python True y False son en realidad subtipos de int, y no porque el lenguaje permita conversiones implícitas entre tipos arbitrarios.

Punto C:

El paradigma que utilizan ambos lenguajes es el de TERMINACIÓN, donde el controlador de excepciones realiza las acciones necesarias para manejar la excepción, pero no se retorna al punto donde se produjo la excepción (invocador), sino que continúa su ejecución a partir de la finalización del manejador.

- JavaScript

```
1.    function f() {
2.        try {
3.            throw "excepción";
4.        } catch (e) {
5.            console.log('captura "falso" interno');
6.            throw e; // esta instrucción throw se suspende
 hasta
7.            // que el bloque finally se haya completado
8.        } finally {
9.            return false; // sobrescribe el "throw" anterior
10.       }
11.       // "return false" se ejecuta ahora
12.    }
13.
14.   try {
15.       console.log(f());
16.   } catch (e) {
17.       // ¡esto nunca se alcanza!
18.       // mientras se ejecuta f(), el bloque `finally`
 devuelve false,
19.       // que sobrescribe el `throw` dentro del `catch`
 anterior
20.       console.log('"falso" externo capturado');
21.   }
22.
23.   // Produce
24.   // captura "falso" interno
25.   // false
```

- Python

```
1.    def intento():
2.        try:
3.            try:
4.                a = int(input("Ingrese un divisor: "))
5.                print("Resultado: " + 20/a);
6.            except ZeroDivisionError: // Opción para un tipo
 de error
7.                print("No se puede dividir por 0.")
8.        except (ValueError, NameError): // Opción para
 múltiples tipos de errores bajo un único manejador
9.            print("Hubo un error y estamos manejando de a 2
 errores con el mismo manejador!")
10.
11.   try:
12.         print("Hola, esto es un ejemplo de excepciones en
 python!")
13.         intento()
14.   except: // Opción que actúa como comodín
15.       print("Hubo un error y lo está manejando el comodín!")
16.   else: // Se ejecuta si NO hubo errores
17.         print("Todo fue de maravillas!")
18.   finally: // Se ejecuta siempre al final
19.         print("Se acabó todo todillo.")
```

Punto D:

La realización de este trabajo nos permitió profundizar en los detalles técnicos de los lenguajes de programación seleccionados, ampliando nuestra comprensión sobre sus características distintivas y su aplicación práctica en el desarrollo de software.

Al investigar y ejemplificar los tipos de parámetros soportados por cada lenguaje, comprendimos mejor las diferencias en el modo de ligadura, como los parámetros posicionales y nombrados.

La exploración de la fortaleza del sistema de tipos de ambos lenguajes nos permitió apreciar cómo la rigidez o flexibilidad en la declaración de tipos puede influir en la detección temprana de errores y en la facilidad de mantenimiento del código.

El estudio del manejo de excepciones nos mostró cómo ambos lenguajes implementan los mecanismos para la gestión de errores, permitiendo un control más fino y robusto de las situaciones excepcionales que pueden surgir en la ejecución del programa.

En conclusión, este trabajo nos proporcionó una visión más profunda y comparativa de las características y capacidades tanto de JavaScript como de Python, fortaleciendo nuestra habilidad para poder seleccionar el lenguaje adecuado según las necesidades del proyecto y aplicar las mejores prácticas de programación para optimizar el desarrollo y mantenimiento del software.