CS315 - Homework 1 Report

Emine İrem Esendemir

ID: 22202103

Department of Computer Engineering

Bilkent University

Ankara, Turkey

April 6, 2025

1 Part A - Operations in each language

1.1 Dart

In Dart, there is no distinct "array" type. Instead, Dart uses `Lists`, which are ordered collections of elements. Unlike arrays, `Lists` in Dart are dynamic by default, meaning their size can grow or shrink during runtime. However, Dart also supports `fixed-length Lists`, which behave similarly to arrays in other languages—once created, their size cannot change [1].

- **What types are legal for subscripts?**
  In Dart, only integers or integer variables can be used as subscripts for Lists. Using a float or string as a subscript will result in a compile-time error as Dart enforces strict type-checking and expects subscripts to be of type `int`.

```dart
List<int> arr1 = [1, 2, 3, 4, 5];
// Integer subscripts
print(arr1[2]); // Outputs: 3
// Variable subscripts
int index = 3;
print(arr1[index]); // Outputs: 4
// String subscripts
// print(arr1["1"]); // Error: A value of type 'String' can't be
   assigned to a variable of type 'int'.
// Floating point subscripts
// print(arr1[2.5]); // Error: A value of type 'double' can't be
   assigned to a variable of type 'int'.
```

- **Are subscripting expressions in element references range checked?**
  Yes, Dart performs range checking on subscripting expressions. If it is tried to access an index that is out of the valid range (less than 0 or greater than or equal to the list's length), Dart throws a `RangeError`.

```dart
// print(arr1[10]); // Error: RangeError (length): Invalid value: Not
    in inclusive range 0..4: 10
// print(arr1[-1]); // Error: RangeError (length): Invalid value: Not
    in inclusive range 0..4: -1
```

- **Are ragged multidimensional arrays allowed?**
  Yes, Dart allows ragged multidimensional arrays through Lists of Lists. Each inner List can have a different length.

```dart
List<List<int>> raggedArray = [[1, 2, 3], [4, 5, 6, 7],[8, 9]];
print(raggedArray); // Outputs: [[1, 2, 3], [4, 5, 6, 7], [8, 9]]
```

- **Can array objects be initialized?**
  Yes, Dart provides various ways to initialize Lists. Lists can be declared with explicit types like

`List<int>` or using `var` thanks to Dart's type inference. It is possible to create Lists with predefined values, empty Lists `([])`, or fixed-length Lists using List.filled(length, value), which cannot grow or shrink but allow element updates.

Dart also supports dynamically generated Lists via List.generate(), where elements are created using a function. Read-only Lists can be made with List.unmodifiable(). These Lists prevents modifications. Dart's collection features further helps with the initialization process: the spread operator (...) allows combining multiple Lists; the null-aware spread (...?) safely handles null Lists; `if` includes elements conditionally; and `for` adds elements using loop logic.

```
List<int> arr2 = [1, 2, 3, 4, 5];
print("arr2: $arr2"); // Outputs: arr2: [1, 2, 3, 4, 5]
var numbers = [1, 2, 3];
print("numbers: $numbers"); // Outputs: numbers: [1, 2, 3]
List<int> empty = [];
print("empty: $empty"); // Outputs: empty: []
List<int> fixed = List.filled(5, 3);
print("fixed: $fixed"); // Outputs: fixed: [3, 3, 3, 3, 3]
// fixed.add(4); // Error: Unsupported operation: Cannot add to a
   fixed-length list
fixed[0] = 10; // This is allowed
print("fixed after modification: $fixed"); // Outputs: fixed after
   modification: [10, 3, 3, 3, 3]
// with a function
List<int> squares = List.generate(5, (index) => index * index); //
   [0, 1, 4, 9, 16]
print("squares: $squares"); // Outputs: squares: [0, 1, 4, 9, 16]
// initializing a read only list
final readOnly = List.unmodifiable([1, 2, 3]);
// readOnly[0] = 10; // Error: Unsupported operation: Cannot modify
   an unmodifiable list
// using spread operator
var base = [1, 2, 3, 4];
var combined = [0, ...base, 5]; // [0, 1, 2, 3, 4, 5]
print("combined: $combined"); // Outputs: combined: [0, 1, 2, 3, 4,
   5]
// using null aware spread operator
var nullableList = null;
var combinedNullable = [0, ...?nullableList, 5]; // [0, 5]
print("combinedNullable: $combinedNullable"); // Outputs:
   combinedNullable: [0, 5]
// using collection if
var condition1 = true;
var condition2 = false;
var listWithCondition = [0, if (condition1) 1, if (condition2) 2,];
```

```
print("listWithCondition: $listWithCondition"); // Outputs:
    listWithCondition: [0, 1]
// using collection for
var listWithFor = [for (var i = 0; i < 10; i++) i];
print("listWithFor: $listWithFor"); // Outputs: listWithFor: [0, 1,
    2, 3, 4, 5, 6, 7, 8, 9]
```

- **Are any kind of slices supported?**
  There is no dedicated operator as `:`, but there is a method called `sublist(start, end)`. It returns a new List from the start index (inclusive) to the end index (exclusive). There are other methods which might be useful for slicing. The method `where()` filters elements based on a condition, returning only those that match. A similar-named method `removeWhere()` mutate an existing list by removing elements based on a condition, this changes the original List. There is also the method `take(n)` to return the first n elements of a List and `skip(n)` to exclude the first n elements and return the remainder.

```
// sublist method
List<int> arr3 = [1, 2, 3, 4, 5];
List<int> slice = arr3.sublist(1, 3);
print("slice: $slice"); // Outputs: slice: [2, 3]
// using the where method
List<int> slice3 = arr3.where((element) => element > 1).toList();
print("slice3: $slice3"); // Outputs: slice3: [2, 3, 4, 5]
// using removeWhere method
List<int> arr4 = [1, 2, 3, 4, 5];
arr4.removeWhere((element) => element > 3);
print("arr4 after removeWhere: $arr4"); // Outputs: arr4 after
    removeWhere: [1, 2, 3]
// take method
List<int> arr5 = [1, 2, 3, 4, 5];
List<int> takeSlice = arr5.take(3).toList();
print("takeSlice: $takeSlice"); // Outputs: takeSlice: [1, 2, 3]
// skip method
List<int> skipSlice = arr5.skip(2).toList();
print("skipSlice: $skipSlice"); // Outputs: skipSlice: [3, 4, 5]
```

- **Which operators are provided?**
  In Dart, several operators are supported for lists. Lists can be concatenated using the `+` operator, which creates a new list combining the elements of both operands. The spread operator (`...`) can also be used to combine arrays' operands and new values. Dart also supports equality comparison using the `==` operator. However, this checks for reference equality. `list1 == list2` will return `true` only if both variables refer to the same object in memory. By default, two different lists with the same operands are considered unequal. Relational comparison operators such as `<`, `>`, or arithmetic operations between lists are not supported and will result in an error.

```
List<int> list1 = [1, 2, 3, 4, 5];
List<int> list2 = [6, 7, 8, 9, 10];
List<int> list3 = [1, 2, 3, 4, 5];
List<int> list4 = list2;
List<int> concatenatedList = list1 + list2;
List<int> spreadList = [0, ...list1, ...list2];
print("concatenatedList: $concatenatedList"); // Outputs:
    concatenatedList: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("spreadList: $spreadList"); // Outputs: spreadList: [0, 1, 2,
    3, 4, 5, 6, 7, 8, 9, 10]
print("list1 == list3: ${list1 == list3}"); // Outputs: list1 ==
    list3: false, because they are different instances
print("list1 != list3: ${list1 != list3}"); // Outputs: list1 ==
    list3: true
print("list2 == list4: ${list2 == list4}"); // Outputs: list1 ==
    list4: true, because they are the same instance
// print("list1 < list2: ${list1 < list2}"); // Error: The operator
    '<' isn't defined for the class 'List<int>'.
```

1.2 Go

In Go, the `Array` data structure is similar to `Slice` structure. Slice can grow and shrink, while array cannot. Even though they have different property, their syntax looks quite similar. If the size is specified inside the bracket, it becomes `Array`. If it is not specified, it becomes `Slice` [3].

- **What types are legal for subscripts?**
  Only integers and integer variables are valid types for subscripts. Floats and strings are not allowed.

```
arr := [5]int{1, 2, 3, 4, 5}
fmt.Println(arr[2]) // integer subscript, Outputs: 3
i := 3
fmt.Println(arr[i]) // integer variable subscript, Outputs: 4
// fmt.Println(arr[2.5]) // Error: 2.5 (untyped float constant)
   truncated to int, floats are not allowed as subscripts
// fmt.Println(arr["2"]) // Error: cannot convert "2" (untyped string
    constant) to type int, strings are not allowed as subscripts
```

- **Are subscripting expressions in element references range checked?**
  Yes. In Go, subscripting expressions are range checked. Both negative indices and indices that are larger than the array size are not allowed. If it is tried to access an element with an index that is out of bounds, Go will return an error. For example, trying to access index 5 in an array of size 5 will result in the error message "index 5 out of bounds [0:5]". Similarly, using a negative index, such as arr[-1], will produce the error "index -1 (constant of type int) must not be negative." This ensures that only valid indices within the array's size can be used to access elements.

```
// fmt.Println(arr[5]) // Error: invalid argument: index 5 out of
   bounds [0:5]
// fmt.Println(arr[-1]) // Error: invalid argument: index -1 (
   constant of type int) must not be negative
```

- **Are ragged multidimensional arrays allowed?**
  In Go, ragged arrays are not directly supported with the array type. However, it can be achieved with a similar structure by using slices of slices.It means that arrays can have rows of varying lengths. Therefore, it is valid in Go, but it cannot be called array directly.

```
raggedArray := [3][]int{{1, 2, 3},{4, 5, 6, 7},{8, 9},}
fmt.Println(raggedArray) // Outputs: [[1 2 3] [4 5 6 7] [8 9]]
```

- **Can array objects be initialized?**
  Yes, they can be initialized either explicitly or implicitly.

```
var arr2 [5]int // implicit initialization
fmt.Println(arr2) // Outputs: [0 0 0 0 0]
arr3 := [6]int{1, 2, 3, 4, 5, 6} //explicit initialization
fmt.Println(arr3) // Outputs: [1 2 3 4 5 6]
```

- **Are any kind of slices supported?**
  Yes, there are some variations of slicing supported by Go.
  - Slice using a range with `[start:end]`: creates a new slice from the start index (inclusive) to the end index (exclusive).
  - Slice using a range with `[start:]`: creates a new slice from a start index to the end of the array.
  - Slice using a range with `[:end]`: creates a new slice from to the end index.
  - Slice using a range with `[:]`: creates a new slice of the entire array.
  - Slice using a range with `[start:end:cap]`: creates a new slice from the start index to the end index. The `cap` determines how much the slice can grow, starting from the start index.

```
arr4 := [5]int{1, 2, 3, 4, 5}
fmt.Println(arr4) // Outputs: [1 2 3 4 5]
fmt.Println(arr4[2:4]) // Outputs: [3 4]
fmt.Println(arr4[2:]) // Outputs: [3 4 5]
fmt.Println(arr4[:4]) // Outputs: [1 2 3 4]
fmt.Println(arr4[:]) // Outputs: [1 2 3 4 5]
fmt.Println(arr4[2:4:4]) // Outputs: [3 4]
fmt.Println(arr4[0:3:4]) // Outputs: [1 2 3]
```

- **Which operators are provided?**
  In Go, arrays are not designed to be resized, and slices are preferred when dynamic resizing is needed. So, there is no function to add or delete a certain part of the array. In terms of comparison, if the arrays are the same size, they can be compared using the == operator, but if they are in different sizes, it gives an error.

```
// comparison
arr5 := [5]int{1, 2, 3, 4, 5}
arr6 := [5]int{1, 2, 3, 4, 6}
// fmt.Println(arr == arr3) // Error: invalid operation: arr == arr3
   (mismatched types [5]int and [6]int)
fmt.Println(arr == arr5) // Outputs: true
fmt.Println(arr == arr6) // Outputs: false
```

1.3 JavaScript

JavaScript does not have a separate low-level array type like in some other languages. Instead, it provides a high-level, dynamic `Array` object. They are flexible and dynamic, allow elements of any type, including mixed types, and they can grow or shrink at runtime. Although they look like normal arrays in syntax, JavaScript arrays are not necessarily stored contiguously in memory [4].

- **What types are legal for subscripts?**
  In JavaScript, valid array subscripts are non-negative integers or values that can be converted into such integers. Integer literals and variables are treated as valid indices. Strings that represent integers and floating-point numbers that represent whole numbers are also accepted, as they can be converted to integers. However, if a subscript is a non-integer float, a non-numeric string, or values like false or null, JavaScript does not throw an error but returns `undefined`.

```
let arr1 = [1,2,3,4,5];
let index = 0;

console.log(arr1[index]); // 1
console.log(arr1[0]); // 1
console.log(arr1["3"]); // 4
console.log(arr1[1.0]); // 2
console.log(arr1[false]); // undefined
console.log(arr1[null]); // undefined
console.log(arr1[1.5]); // undefined
console.log(arr1["a"]); // undefined
```

- **Are subscripting expressions in element references range checked?**
  No, JavaScript does not perform range checking when accessing array elements. If an index is outside the bounds of the array—either greater than its length or negative— it simply returns `undefined`, does not raise an error. Negative indices are not valid for accessing elements (unlike in some other languages like Python), and they do not wrap around.

```
let arr1 = [1,2,3,4,5];
console.log(arr1[10]); // undefined
console.log(arr1[-1]); // undefined
```

- **Are ragged multidimensional arrays allowed?**
  Yes, it is allowed to have ragged multidimensional arrays in JavaScript. As this arrays do not have to be homogenous, there is not such a rule of having inner arrays of same length.

```
let arr2 = [[1, 2], [3, 4, 5]];

console.log(Array.isArray(arr2)); // true
console.log(arr2); // [[1, 2], [3, 4, 5]]
```

- **Can array objects be initialized?**

  Yes, JavaScript allows array objects to be initialized in different ways. Arrays can be created using the `Array` constructor, array literals `[]`, or using methods like `Array.of()` and `Array.from()`.

  Using `new Array(length)` creates an array with the specified number of empty slots, without assigning any values. Using `new Array(val1, val2, ...)` creates an array with those values as elements. Using square brackets (`[]`) is most intuitive one and allows initialization with mixed data types.

  The `Array.of()` method provides a way to create an array from a list of arguments, if a single number is passed, it is treated as an array with 1 element, differently from the classic Constructor. The `Array.from()` method can be used to create an array from any iterable, such as a string, or with a mapping function to generate elements.

```
let arr3 = new Array(5);
let arr4 = new Array(1, 2, 3);
let arr5 = ["a", 42, true, null];  // Mixed types
let arr6 = Array.of(1, 2, 3);
let arr7 = Array.of(8);
let arr8 = Array.from("irem");
let arr9 = Array.from({ length: 5 }, (_, i) => i + 3);

console.log("arr3", arr3); // [ <5 empty items> ]
console.log("arr4", arr4); // [ 1, 2, 3 ]
console.log("arr5", arr5); // [ 'a', 42, true, null ]
console.log("arr6", arr6); // [ 1, 2, 3 ]
console.log("arr7", arr7); // [ 8 ]
console.log("arr8", arr8); // [ 'i', 'r', 'e', 'm' ]
console.log("arr9", arr9); // [ 3, 4, 5, 6, 7 ]
```

- **Are any kind of slices supported?**

  Yes, JavaScript supports slicing arrays through the built-in `slice()` method. The first argument is the starting index (inclusive), and the second is the ending index (exclusive). If the second argument is omitted, slicing continues to the end of the array. Negative indices are also supported and wraps up from the end of the array. The `slice()` method does not modify the original array, and any changes made to the resulting slice do not affect the original data. Arrays can also be filtered using the `filter()` method, which returns a new array containing elements that satisfy a specified condition.

```
let arr10 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let slice1 = arr10.slice(2, 5);
let slice2 = arr10.slice(2);
let slice3 = arr10.slice(-3);
let slice4 = arr10.slice(2, -2);
let filteredArr = arr10.filter(num => num > 5);
slice4[0] = 100; // Modifying the slice does not affect the original
    array
```

```
console.log("arr10", arr10); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
console.log("slice1", slice1); // [3, 4, 5]
console.log("slice2", slice2); // [3, 4, 5, 6, 7, 8, 9, 10]
console.log("slice3", slice3); // [8, 9, 10]
console.log("slice4", slice4); // [3, 4, 5, 6, 7, 8]
console.log("Filtered array:", filteredArr); // [6, 7, 8, 9, 10]
```

- **Which operators are provided?**

  JavaScript supports some operators on arrays, but their behavior can be different from what is expected.

  The `+` operator does not perform array concatenation. It first converts both arrays to strings and then concatenates those strings. To properly concatenate arrays, the spread operator (`...`) is used. For example, `[...arr1, ...arr2]` results in a new array combining the elements of both.

  Equality operators like `==` and `===` compare object references, not array contents. Therefore, even if two arrays have identical elements, they are considered unequal if they do not refer to the exact same object. For example, `arr1 === arr2` is only `true` if both variables point to the same array in memory.

  JavaScript allows relational operators such as `<` and `>` to be used in arrays, but their behavior is based on lexicographic string comparison of the arrays' string representations. This might lead to unintuitive results.

  The `in` operator in JavaScript checks whether an index exists in an array, not a value.

```
let arr11 = [1, 2, 3];
let arr12 = [4, 5, 6];
let arr13 = [1, 2, 3];
let arr14 = arr11 + arr12;
let arr15 = arr11;
let arr16 = [1, 3, 1];
let arr17 = [1, 2, 3, 4, 5, 6];

console.log("arr11 + arr12", arr14); // "1,2,34,5,6"
console.log(typeof arr14); // string
console.log("arr11 == arr13", arr11 == arr13); // false
console.log("arr11 == arr15", arr11 === arr15); // true
console.log("arr11 === arr13", arr11 === arr13); // false
console.log("arr11 === arr15", arr11 === arr15); // true

console.log("arr11 < arr12", arr11 < arr12); // true
console.log("arr11 > arr12", arr11 > arr12); // false
console.log("arr11 < arr16", arr11 < arr16); // true
console.log("arr11 < arr17", arr11 < arr17); // true

// spread operator
let arr18 = [...arr11, ...arr12];
console.log("Spread operator:", arr18); // [1, 2, 3, 4, 5, 6]
```

```
// in operator
console.log("0 in arr11:", 0 in arr11); // true
console.log("3 in arr11:", 3 in arr11); // false
```

1.4 Kotlin

Kotlin provides a built-in `Array` class for fixed-size, indexed collections. Arrays can store elements of any type but they need to be homogenous. Their size is defined at creation and cannot be changed. For dynamic or resizable collections, Kotlin has alternatives of `MutableList` and `ArrayList`, but they are out of scope for this homework [5].

- **What types are legal for subscripts?**
  In Kotlin, only integers, integer constants and integer variables, are legal for subscripts when accessing elements in an array. Using a non-integer type, as string or floating-point number, will result in a compilation error. Kotlin is statically typed and enforces that array indices must be of type `Int`.

```
val arr1 = arrayOf(1, 2, 3, 4, 5)
// Integer subscripts
println(arr1[2])  // Outputs: 3
// Variable subscripts
val index = 3
println(arr1[index])  // Outputs: 4
// String subscripts
// println(arr1["1"])  // Error: The integer index expected, not
   String
// Floating point subscripts
// println(arr1[2.5])  // Error: The index must be Int, not Double
```

- **Are subscripting expressions in element references range checked?**
  Yes, Kotlin checks array indices at runtime. If an index that is out of bounds (negative or beyond the array size) is accessed, it throws an `ArrayIndexOutOfBoundsException`.

```
// println(arr1[5])  // Error: ArrayIndexOutOfBoundsException: 5
// println(arr1[-1])  // Error: ArrayIndexOutOfBoundsException: -1
```

- **Are ragged multidimensional arrays allowed?**
  Yes, Kotlin allows ragged arrays by using arrays of arrays. Each inner array can have a different length.

```
val arr2 = arrayOf(
    arrayOf(1, 2, 3),
    arrayOf(4, 5),
    arrayOf(6, 7, 8, 9)
)
// print ragged array
println("Ragged array:")
for (i in arr2.indices) {
    for (j in arr2[i].indices) {
        print("${arr2[i][j]} ")
    }
    println()
```

```
}
```

- **Can array objects be initialized?**
  Yes, array objects in Kotlin can be initialized in various ways using functions like `arrayOf()`, `intArrayOf()`, `arrayOfNulls()`, and the `Array()` constructor. `arrayOf()` can be used to create arrays with elements of any type, including mixed types. If the generic `Array` class with primitive values (like `Int`) is used, Kotlin will wrap these values as objects. This introduces a slight performance overhead. To avoid this, Kotlin provides specialized primitive arrays like `intArrayOf()`, `doubleArrayOf()`, etc., which store values more efficiently without boxing. [5] It is also possible to initialize arrays with null values using `arrayOfNulls<Int>(5)`, or fill arrays using a lambda with `Array(size){initializer}`.

```
// mixed type
val arr3 = arrayOf(1, "two", 3.0)
println(arr3.joinToString())  // Outputs: 1, two, 3.0
// intArray example
val intArray = intArrayOf(1, 2, 3, 4, 5)
println(intArray.joinToString())  // Outputs: 1, 2, 3, 4, 5
// Null array
val arr5 = arrayOfNulls<Int>(5)
println(arr5.joinToString())  // Outputs: null, null, null, null,
    null
// print array
val initArray = Array<Int>(3) { 0 }
println(initArray.joinToString())  // Outputs: 0, 0, 0
val asc = Array(5) { i -> (i * i)}
println(asc.joinToString())  // Outputs: 0, 1, 4, 9, 16
```

- **Are any kind of slices supported?**
  Kotlin provides various functions support slicing, though not in the same way as some other languages like Python. The `slice()` function allows selecting elements within a specified index range and returns a `List`. The `copyOfRange()` function extracts a range of elements and returns a new `Array`. In addition to these, functions such as `drop()`, `take()`, and `filterIndexed()` can be used to create new lists based on specific criteria or positions. These operations do not modify the original array but return new collections.

```
val arr4 = arrayOf(1, 2, 3, 4, 5)
// slice array
val slice = arr4.slice(1..3)
println("Slice of arr4: ${slice.joinToString()}")  // Outputs: 2, 3,
    4
println("Type of slice: ${slice::class.simpleName}")  // Outputs:
    ArrayList
// copyRange array
val copyRange = arr4.copyOfRange(1, 4)
```

```
println("Copy range of arr4: ${copyRange.joinToString()}")  //
    Outputs: 2, 3, 4
println("Type of copyRange: ${copyRange::class.simpleName}")  //
    Outputs: Array
// drop array
val drop = arr4.drop(2)
println("Drop of arr4: ${drop.joinToString()}")  // Outputs: 3, 4, 5
println("Type of drop: ${drop::class.simpleName}")  // Outputs:
    ArrayList
// take array
val take = arr4.take(3)
println("Take of arr4: ${take.joinToString()}")  // Outputs: 1, 2, 3
println("Type of take: ${take::class.simpleName}")  // Outputs:
    ArrayList
// filteredIndexed()
val filteredIndexed = arr4.filterIndexed { index, value -> index % 2
    == 0 }
println("Filtered indexed of arr4: ${filteredIndexed.joinToString()}"
    )  // Outputs: 1, 3, 5
println("Type of filteredIndexed: ${filteredIndexed::class.simpleName
    }")  // Outputs: ArrayList
```

- **Which operators are provided?**

  Kotlin supports several operators and methods for working with arrays. The `==` operator checks for structural equality, but for the arrays, it behaves like referential equality. It returns `true` only if both references point to the same array object. The `===` operator checks for referential equality, also returning `true` only if both variables refer to the exact same object in memory. To check for content equality between two arrays, the `contentEquals()` method is used, which compares the elements of the arrays element-by-element.

  Kotlin also supports the `in` and `!in` operators to check for the presence or absence of elements in an array.

  For concatenation, there are `+` and `+=` operators. The `+` operator returns a new array with the contents of the two operands combined, while the `+=` operator appends elements or arrays to an existing array by creating a new array behind.

```
var arr6 = arrayOf(1, 2, 3)
val arr7 = arrayOf(1, 2, 3)
val arr8 = arr6
println("arr6: ${arr6.joinToString()}")  // Outputs: 1, 2, 3
println("arr7: ${arr7.joinToString()}")  // Outputs: 1, 2, 3
println("arr8: ${arr8.joinToString()}")  // Outputs: 1, 2, 3

// == operator
```

```
println("arr6 == arr7: ${arr6 == arr7}")  // Outputs: false
println("arr6 == arr8: ${arr6 == arr8}")  // Outputs: true
// === operator
println("arr6 === arr7: ${arr6 === arr7}")  // Outputs: false
println("arr6 === arr8: ${arr6 === arr8}")  // Outputs: true
// contentEquals
println("arr6 contentEquals arr7: ${arr6 contentEquals arr7}")  //
    Outputs: true
println("arr6 contentEquals arr8: ${arr6 contentEquals arr8}")  //
    Outputs: true
// in operator
println("Does arr6 contain 2? ${2 in arr6}")  // Outputs: true
println("Does arr6 not contain 4? ${4 !in arr6}")  // Outputs: true

//+ operator
arr6 += 4
println("arr6 after += operator: ${arr6.joinToString()}")  // Outputs
    : 1, 2, 3, 4
arr6 += arrayOf(5, 6, 7)
println("arr6 after += operator: ${arr6.joinToString()}")  // Outputs
    : 1, 2, 3, 4, 5, 6, 7
val arr9 = arr6 + arr7
println("arr9 = arr6 + arr7: ${arr9.joinToString()}")  // Outputs: 1,
     2, 3, 4, 5, 6, 7, 1, 2, 3
```

1.5 PHP

PHP's arrays are originally implemented as ordered maps. Keys can be strings or ints, values can be any type. They can store different types of values in the same array, also they can grow and shrink [7].

- **What types are legal for subscripts?**
  In PHP, array subscripts can be of various types, including integers, strings, floats, booleans and null. PHP converts these types to integers or strings by using them as keys. Integer and string subscripts are used as they are defined. Float subscripts are converted to integers, potentially losing precision as shown in the warning. Boolean values are converted to 0 (false) and 1 (true). Also, null indices are possible and returns its value. Negative integers are allowed if explicitly defined; otherwise PHP throws warnings.

```
$arr1 = array(1, 2, 3, 4, 5);
$arr2 = array("a" => 1, "b" => 2, "c" => 3);
$arr3 = array(1 => "a", 2 => "b", 3 => "c", -1 => "d", "" =>"e");
echo $arr1[0] . " integer subsript\n"; // Outputs: 1
echo $arr2["c"] . " string subscript\n\n"; // Outputs: 3
echo $arr3[2.5] . " float subscript\n\n"; // Outputs: b
// PHP Deprecated: Implicit conversion from float 2.5 to int loses
   precision
echo $arr1[true] . " true subscript\n"; // Outputs: 2
echo $arr1[false] . " false subscript\n"; // Outputs: 1
echo $arr3[null] . " null subscript\n"; // Outputs: e
// echo $arr1[-1] . " negative subscript\n"; // Warning: Undefined
   array key -1 in
echo $arr3[-1] . " negative subscript\n"; // Outputs: d
```

- **Are subscripting expressions in element references range checked?**
  PHP arrays do not require sequential or continuous integer indices, so the concept of a strict "range" does not apply. Instead, when accessing an element, PHP checks whether the specified key exists in the array. If it does, the value is returned. If not, a warning is issued and the result is `null`. Therefore, PHP performs "existence checking", not "range checking", when accessing array elements.

```
$arr4 = array(1 => 11, 2 => 12, 1000 => 1010);
echo $arr4[1000] . " = arr1[1000]\n"; // Outputs: 1010
// echo $arr1[1001] . "arr1[1001]\n"; // Warning: Undefined array key
    1001 in
// echo $arr1[999] . "$arr1[999]\n"; // Warning: Undefined array key
   999 in
```

- **Are ragged multidimensional arrays allowed?**
  Yes, PHP allows ragged multidimensional arrays. Since a PHP arrays can hold elements of any type, including other arrays, it is valid to have inner arrays of different lengths.

```
$arr5 = array(
```

```
    1 => array(1, 2, 3),
    2 => array(4, 5, 6),
    3 => array(7, 8, 9, 10)
);
print_r($arr5);
```

- **Can array objects be initialized?**

  Yes, arrays in PHP can be initialized in various ways. They can be created using the `array()` constructor or the short syntax `[]`. If indices are not explicitly specified, they default to sequential integers starting from zero. PHP also provides utility functions for initialization, such as `array_fill($start_index, $num, $value)` to fill an array with repeated values, and `range($start, $end, $step)` to create arrays with sequential values.

```
$arr6 = array(12,13,14,15);
$arr7 = array(1 => 12, 2 => 13, 3 => 14, 4 => 15);
$arr8 = [12, 13, 14, 15];
$arr9 = array_fill(2, 4, 6);
print_r($arr9);
echo "\n";
$arr10 = range(7, 10); // 7, 8, 9, 10
$arr11 = range("a", "e"); // a, b, c, d, e
$arr12 = range(1, 10, 2); // 1, 3, 5, 7, 9
$arr13 = array();
print_r($arr10);
echo "\n";
print_r($arr11);
echo "\n";
print_r($arr12);
echo "\n";
print_r($arr13);
echo "\n";
```

- **Are any kind of slices supported?**

  Yes, PHP supports slicing arrays through the built-in function `array_slice(array $array, int $offset, int $length = null, bool $preserve_keys = false)`. The original array remains same, new array is created.

  The `$offset` defines the starting position of the slice. If it is positive, counting starts from the beginning of the array. If it is negative, the slice starts that far from the end of the array. The `$length` parameter determines how many elements to include. When `$length` is omitted, the slice continues to the end of the array. If `$length` is a negative value, the slice will stop that many elements before the end. By default, PHP changes indices of the numeric keys in the resulting array, starting from zero. However, this can be changed by specifying `$preserve_keys` as true. In that case, the original keys (including non-sequential or associative keys) are maintained. String keys are always preserved.

```php
$arr14 = array(11 => 1, 12 => 2, 13 => 3, 14 => 4, 15 => 5);
print_r(array_slice($arr14, 1, 3)); // Outputs: Array ( [0] => 2 [1]
    => 3 [2] => 4 )
print_r(array_slice($arr14, 1)); // Outputs: Array ( [0] => 2 [1] =>
    3 [2] => 4 [3] => 5 )
print_r(array_slice($arr14, 1, -1, true)); // Outputs: Array ( [12]
    => 2 [13] => 3 [14] => 4 )
```

- **Which operators are provided?**

  PHP provides several operators for arrays. The `==` operator checks whether two arrays have the same key/value pairs, without considering the order or type. As long as the keys and values match, the result is `true`, even if the types or order differ.

  The `===` operator performs more strict comparison. The arrays must have the same key/value pairs in the same order and with the same types. If any of these conditions are not met, the result is `false`.

  PHP also allows the comparison operators such as `<` and `>`. When comparing two arrays with these operators, PHP compares elements in order until a difference is found. If all elements match but one array has more elements, the longer array is considered greater.

  The `+` operator is known as the union operator in PHP. It combines two arrays, but only adds elements from the second array if their keys do not already exist in the first. If the keys overlap, the values from the first array are kept.

```php
echo "\n\n6. Which operators are provided?\n";
// == Returns true if $x and $y have the same key/value pairs
// === Returns true if $x and $y have the same key/value pairs in the
    same order and of the same types
$arr15 = [1, 2, 3];
$arr16 = [1, 2, 3];
$arr17 = ["1", "2", "3"];
$arr18 = [2 => 3, 1 => 2, 0 => 1];

var_dump($arr15 == $arr16); // bool(true)
var_dump($arr15 == $arr17); // bool(true)
var_dump($arr15 == $arr18); // bool(true)
echo "\n";
var_dump($arr15 === $arr16); // bool(true)
var_dump($arr15 === $arr17); // bool(false)
var_dump($arr15 === $arr18); // bool(false)
echo "\n";


// < and >: Comparison operators
$compare1 = [1, 2, 3];
```

```php
$compare2 = [1, 3, 1];
$compare3 = [1, 2, 3, 4];
var_dump($compare1 < $compare2); // bool(true) - First differing
    element: 2 < 3
var_dump($compare1 > $compare2); // bool(false)
var_dump($compare1 < $compare3); // bool(true) - All elements match
    but $arr3 has more elements
echo "\n";

// +: Union operator
$add1 = [1, 2, 3];
$add2 = [4, 5, 6];
$add3 = [2 => 12, 3 => 13, 4 => 14];
print_r($add1 + $add2); // Outputs: Array ( [0] => 1 [1] => 2 [2] =>
    3)
print_r($add1 + $add3); // Outputs: Array ( [0] => 1 [1] => 2 [2] =>
    3 [3] => 13 [4] => 14)
```

1.6 Python - NumPy

In standard Python, the main sequence data structure is the `list`. Lists are flexible, heterogeneous, and dynamic. However, the `ndarray` type provided by the NumPy library is main data structure behaving like array. A `NumPy array` is a fixed-size and homogeneous, and stored in contiguous memory [6].

- **What types are legal for subscripts?**
  Integers, and integer variables are valid as subscripts. Floats and strings cause *IndexError*. Negative integer subscripts can be used to access elements starting from the end of the array.

```
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1[2])  # Outputs: 3
index = 3
print(arr1[index])  # Outputs: 4
# String subscripts
# arr1['a']  # IndexError: only integers, slices (':'), ellipsis
   ('...'), numpy.newaxis ('None') and integer or boolean arrays are
   valid indices
# Negative subscripts
print(arr1[-1])  # Outputs: 5
print(arr1[-2])  # Outputs: 4
# float subscripts
# arr1[1.5]  # IndexError: only integers, slices (':'), ellipsis
   ('...'), numpy.newaxis ('None') and integer or boolean arrays are
   valid indices
```

- **Are subscripting expressions in element references range checked?**
  Yes, they are range-checked. When an integer is given equal to size or greater than that number, it will raise an *IndexError*. Python supports negative subscripts as said above; these negative integers are also range-checked. If they are in the range, their absolute value is subtracted from the sequence length to locate the correct element; otherwise, an *IndexError* is thrown.

```
arr1 = np.array([1, 2, 3, 4, 5])
# print(arr1[5])  # IndexError: index 5 is out of bounds for axis 0
   with size 5
# print(arr1[-6])  # IndexError: index -6 is out of bounds for axis 0
    with size 5
print(arr1[-2])  # Outputs: 4
```

- **Are ragged multidimensional arrays allowed?**
  By default, NumPy does not support ragged (non-rectangular) multidimensional arrays, as it expects arrays to have rectangular shapes. If an attempt is made to create a ragged array without explicitly specifying the data type, NumPy raises a *ValueError*, stating: *inhomogeneous shape*. However, it is possible to create ragged arrays by explicitly setting the data type to `object`. This allows the array to store subarrays of different lengths.

```
ragged = np.array([np.array([1, 2, 3]), np.array([4, 5]), np.array
    ([6, 7, 8, 9])], dtype=object)
print(ragged) # Outputs: [array([1, 2, 3]) array([4, 5]) array([6, 7,
    8, 9])]
# ragged2 = np.array([[1, 2, 3], [4, 5], [6, 7, 8, 9]]) # ValueError:
    setting an array element with a sequence. The requested array has
    an inhomogeneous shape after 1 dimensions.
```

- **Can array objects be initialized?**
  Yes, NumPy provides various ways for initializing the arrays. Arrays can be initialized with zeros, ones, specific values, uninitialized memory, ranges, random numbers, or explicitly defined values. There are more ways to initialize to serve different demands, below ones are the some examples of it.

```
# zero initialization
arr2 = np.zeros(5)
print(arr2)  # Outputs: [0. 0. 0. 0. 0.]
# one initialization
arr3 = np.ones(5)
print(arr3)  # Outputs: [1. 1. 1. 1. 1.]
# empty initialization
arr4 = np.empty(5)
print(arr4)  # Outputs: values may vary, as they are uninitialized
# initialization with a specific value
arr5 = np.full(5, 7)
print(arr5)  # Outputs: [7 7 7 7 7]
# initialization with a range of values
arr6 = np.arange(5)
print(arr6)  # Outputs: [0 1 2 3 4]
# initialization with a range of values with a step
arr7 = np.arange(0, 10, 2)
print(arr7)  # Outputs: [0 2 4 6 8]
# random initialization
arr8 = np.random.rand(5)
print(arr8)  # Outputs: random values between 0 and 1
# explicit initialization
arr9 = np.array([1, 2, 3, 4, 5])
print(arr9)  # Outputs: [1 2 3 4 5]
```

- **Are any kind of slices supported?**
  Yes, NumPy supports various slicing techniques. Boolean arrays can be used as masks to select elements where the condition is *True*. Integer arrays can select specific positions directly. Basic slicing using the start:stop notation extracts elements between the specified indices. If the start or stop is omitted, it defaults to the beginning or end of the array respectively. Adding a third value specifies the step.

Negative step values allow slicing in reverse order. For example, `arr[::-1]` reverses the array, and `arr[5:2:-1]` selects elements from index 5 down to 3. Negative indices count from the end of the array. For multidimensional arrays, NumPy supports the ellipsis ... to simplify selection across dimensions. For example, `arr[..., 1]` selects the second column across all rows.

```
arr1 = np.array([1, 2, 3, 4, 5])
bool_subscript = np.array([True, False, True, True, True])
print(arr1[bool_subscript])  # Outputs: [1 3 4 5]

int_subscript = np.array([0, 2, 4])
print(arr1[int_subscript])  # Outputs: [1 3 5]
test_array = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Basic slicing
print(test_array[2:5])  # Outputs: [2 3 4]
print(test_array[:2])  # Outputs: [0 1], from start, to specified end
print(test_array[2:])  # Outputs: [2 3 4 5 6 7 8 9 10], from
   specified start to the end

# with steps
print(test_array[::2])  # Outputs: [0 2 4 6 8 10]
print(test_array[1:8:2])  # Outputs: [1 3 5 7]

# with negative steps
print(test_array[::-1])  # Outputs: [10 9 8 7 6 5 4 3 2 1 0]
print(test_array[5:2:-1])  # Outputs: [5 4 3]

# with negative indices
print(test_array[-5:-2])  # Outputs: [5 6 7]
print(test_array[-2:-5])  # Outputs: []

# ellipsis
ragged_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(ragged_array[..., 1])  # Outputs: [2]
```

- **Which operators are provided?**

  Numpy-ndarrays have various operators defined on them. It supports element-wise arithmetic operations such as `+`, `-`, `*`, `/`, `//`, `%`, and `**`. These operators work between arrays of the same shape and size or between arrays and scalars. For example, `arr + 5` adds 5 to each element in the array.

  Comparison operators such as `==`, `!=`, `<`, `<=`, `>`, and `>=` are also applied element-wise and return Boolean arrays of the same shape.

  Bitwise logical operators, `&`, `|`, `~`, `^`, `<<`, and `>>`, can be used on integer arrays and work elementwise as well.

The `is` operator checks object identity, not array contents. Thus, two arrays with the same content are not the same object unless explicitly assigned. The `in` operator checks for membership by comparing references, not values. Therefore, `arr in another_array` is not reliable for checking if all elements are present. `np.isin()` should be used for that purpose.

For matrix operations, NumPy supports the `@` operator for matrix multiplication, and it also works as a dot product when used with 1D arrays.

```
arr10 = np.array([11, 22, 33])
arr11 = np.array([44, 55, 66])
arr12 = np.array([11, 22, 33])
arr13 = arr12
arr14 = np.array([11, 22, 33, 44, 55, 66])
print(f"arr10: {arr10}")
print(f"arr11: {arr11}")
print(f"arr12: {arr12}")
print(f"arr13: {arr13}")
print()


# Arithmetic operations
print(f"arr10 + arr11: {arr10 + arr11}")  # Outputs: [55 77 99]
print(f"arr10 - arr11: {arr10 - arr11}")  # Outputs: [-33 -33 -33]
print(f"arr10 * arr11: {arr10 * arr11}")  # Outputs: [484 1210 2178]
print(f"arr10 / arr11: {arr10 / arr11}")  # Outputs: [0.25 0.4  0.5 ]
print(f"arr10 // arr11: {arr10 // arr11}")  # Outputs: [0 0 0]
print(f"arr10 % arr11: {arr10 % arr11}")  # Outputs: [11 22 33]
print(f"arr10 ** arr11: {arr10 ** arr11}")  # Outputs:
    [8288407272263152657 4719772409484279808 5794255510723202113]
print(f"arr10 + 5: {arr10 + 5}")  # Outputs: [16 27 38]
#print(f"arr11 + arr14: {arr11 + arr14}")  # ValueError: operands
    could not be broadcast together with shapes (3,) (6,)
print()

# Comparison operators
print(f"arr10 == arr11: {arr10 == arr11}")  # Outputs: [False False
    False]
print(f"arr10 != arr11: {arr10 != arr11}")  # Outputs: [ True  True
    True]
print(f"arr10 < arr11: {arr10 < arr11}")    # Outputs: [ True  True
    True]
print(f"arr10 <= arr11: {arr10 <= arr11}")  # Outputs: [ True  True
    True]
print(f"arr10 > arr11: {arr10 > arr11}")    # Outputs: [False False
    False]
```

```
print(f"arr10 >= arr11: {arr10 >= arr11}")  # Outputs: [False False
    False]
print()

# Logical operators
print(f"arr10 & arr11: {arr10 & arr11}")  # Outputs: [0 0 0]
print(f"arr10 | arr11: {arr10 | arr11}")  # Outputs: [55 55 55]
print(f"~arr10: {~arr10}")  # Outputs: [-12 -23 -34]
print(f"arr10 ^ arr11: {arr10 ^ arr11}")  # Outputs: [55 55 55]
print(f"arr10 >> 3: {arr10 >> 3}")  # Outputs: [1 2 4]
print(f"arr10 << 3: {arr10 << 3}")  # Outputs: [88 176 264]
print()

# is operator
print(f"arr12 is arr10: {arr12 is arr10}")  # Outputs: False
print(f"arr12 is arr13: {arr12 is arr12}")  # Outputs: True
print()

# in operator
print(f"arr10 in arr11: {arr10 in arr11}")  # Outputs: False
print(f"arr10 in arr12: {arr10 in arr12}")  # Outputs: True
print(f"44 in arr11: {44 in arr11}")  # Outputs: True
print()

# Matrix multiplication
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[9, 8], [7, 6]])
print(f"matrix1 @ matrix2: {matrix1 @ matrix2}") # Outputs: [[23 20]
    [55 48]]
print(f"arr10 @ arr11: {arr10 @ arr11}")  # Outputs: 3872
```

1.7 Rust

Rust provides a built-in `array` type for fixed-size collections of elements with the same type. Arrays in Rust are defined with a specific length and stored on the stack [8].

- **What types are legal for subscripts?**
  In Rust, the only legal type for array subscripting is `usize`, which is an unsigned integer type used for indexing and memory-related operations. While integer literals like `arr[2]` work fine because they are automatically inferred as `usize`, variables like `u32`, `i32`, or any other type must be explicitly cast to `usize`. Using a variable with type `u32` will cause compile time error.

```
let arr1 = [1, 2, 3, 4, 5];
// Integer subscript
println!("{}", arr1[2]); // Outputs: 3
// Variable subscript
let index = 3;
println!("{}", arr1[index]); // Outputs: 4
let index2 : u32 = 0;
// println!("{}", arr1 [index2]) ; // Error: the type [{integer}]
   cannot be indexed by u32
// String subscript
// println!("{}", arr1[" 1 "]); // Error: the type `[{integer}]`
   cannot be indexed by `&str`
// Float index - Not allowed
// println!("{}", arr1[1.5]); // Error: the type `[{integer}]` cannot
    be indexed by `{float}`
```

- **Are subscripting expressions in element references range checked?**
  Yes. In Rust, subscripting expressions are range checked. Both negative indices and indices that are larger than the array size are not allowed and cause error.

```
// println!("{}",arr1[10]); // Error: index out of bounds: the length
    is 5 but the index is 10
// println!("{}",arr1[-1]); // Error: cannot use a negative integer
   for indexing on `[{integer}; 5]`
```

- **Are ragged multidimensional arrays allowed?**
  No, Rust does not support ragged arrays using its built-in array type. All inner arrays must be of the same length. Attempting to define arrays with rows of different lengths results in a `mismatched type` error. Only rectangular, fixed-size multidimensional arrays are allowed.

```
// let ragged = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]; // Error:
   mismatched types
let multidimensional = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
println!("{:?}", multidimensional); // Outputs: [[1, 2, 3], [4, 5,
   6], [7, 8, 9]]
```

- **Can array objects be initialized?**
  Yes, array objects in Rust can be initialized in different ways. Arrays have a fixed size and elements of the same type. Initialization can be done either by explicitly listing values or by using a repeated value with the syntax `[value; size]`. Arrays are immutable by default, their elements cannot be changed after creation. If it is specified as mutable by `mut`, its elements can be modified.

```
let explicit: [i32; 3] = [1, 2, 3];
println!("{:?}", explicit); // Outputs: [1, 2, 3]
let arr1: [i32; 3] = [111; 3]; // immutable array
println!("{:?}", arr1); // Outputs: [111, 111, 111]
// arr1[1] = 20; //Error: cannot assign to `arr1[_]`, as `arr1` is
   not declared as mutable
let mut arr2: [i32; 3] = [0; 3]; // Mutable array
arr2[0] = 10;
```

- **Are any kind of slices supported?**
  Yes, Rust supports slices, which are references to parts of arrays. A slice does not create a new array. The slice is still part of the original array and depends on it. If the original array goes out of scope, the slice becomes invalid. When slicing, both the beginning and end can be specified, one of them can be specified, or none of them can be specified, which does not exclude any element.

```
let testarr = [1, 2, 3, 4, 5];
let slice1: &[i32] = &testarr[0..2];
println!("{:?}", slice1); // Outputs: [1, 2]
let slice2: &[i32] = &testarr[1..];
println!("{:?}", slice2); // Outputs: [2, 3, 4, 5]
let slice3: &[i32] = &testarr[..3];
println!("{:?}", slice3); // Outputs: [1, 2, 3]
```

- **Which operators are provided?**
  Rust supports less operators compared to most of the languages. The `==` operator is supported for arrays of the same type and size, and it performs element-wise comparison. If all elements match, it returns `true`. Comparing arrays of different lengths using `==` is a compile-time error. Rust allows use of the `<`, `>`, `<=`, and `>=` operators for arrays of the same size. These perform lexicographical comparison. The `===` and `+` operators are not supported for the arrays.

```
let arr3 = [1, 2, 3];
let arr4 = [4, 5, 6];
let arr5 = [1, 2, 3];
let arr6 = [1, 3, 1];
let arr7 = [1, 2, 3, 4];
// let arr5 = arr3 + arr4; // Error: the type `[{integer}; 3]` cannot
    be added to `[{integer}; 3]`
// == operator
println!("arr3 == arr4: {:?}", arr3 == arr4); // Outputs: false
```

```
println!("arr3 == arr5: {:?}", arr3 == arr5); // Outputs: true
// println!("arr3 == arr7: {:?}", arr3 == arr7); // Error: no
   implementation for `[{integer}; 3] == [{integer}; 4]`
// === operator
// println!("arr3 === arr5: {:?}", arr3 === arr5); // Error: no
   operator `===` found for type `[{integer}; 3]`
// < operator
println!("arr3 < arr4: {:?}", arr3 < arr4); // Outputs: true
println!("arr3 < arr6: {:?}", arr3 < arr6); // Outputs: true
// println!("arr3 < arr7: {:?}", arr3 < arr7); // expected an array
   with a size of 3, found one with a size of 4
```

2 Part B - Evaluation

This section includes evaluation of these languages in terms of readability and writability of array operations.

2.1 Dart

Creating and manipulating arrays in Dart is straightforward, which contributes positively to both readability and writability. Methods such as `List.generate()`, `where()`, and `skip()` make slicing and initializing operations easy to follow. The use of the `+` operator and spread syntax (`...`) for concatenation also improves clarity. However, creating fixed-size or unmodifiable lists requires more explicit constructs like `List.filled()` or `List.unmodifiable()`, which can reduce writability in scenarios where immutability is needed.

2.2 Go

In Go, slice syntax is clear and intuitive, contributing positively to readability. However, creating array and slice is very similar, which might be confusing.Additionally, array comparisons are only allowed when the lengths match exactly, which can decrease flexibility when working with differently sized arrays.

2.3 JavaScript

In Javascript, arrays are easy to create with `Array.from()` and `Array.of()` functions, which increases both its readability and writability. However, the `+` operator returning a string instead of performing concatenation can be confusing. Equality checks using `==` or `===` only compare references, not contents, which may reduce both readability and writability when comparing arrays.

2.4 Kotlin

Kotlin arrays are readable due to clear syntax and methods like `slice()` and `copyOfRange()`. Writability is supported by multiple initialization options, such as `arrayOf()`, `intArrayOf()`, and `Array()`. Kotlin lacks specific operator to slice, such as `:`; however, functions like `drop()` and `take()` gives similar results. Operators like `+`, `+=`, and `in` increases writability.

2.5 PHP

PHP arrays are writable due to their flexible syntax. Arrays can be easily created using `[]` or `array()`, and filled with values using `array_fill()` or `range()`. The ability to mix value types makes writing code quick. In terms of readability, PHP's behavior of converting subscript types (like float, null, or boolean) can be confusing, and warnings rather than errors may lead to bugs. Difference between `==` and `===` does not meet the general expectation, which might decrease its writability.

2.6 Python - NumPy

NumPy arrays offer very good readability thanks to consistent syntax for slicing, indexing, and arithmetic operations. Functions like `arange()`, `zeros()`, and `full()` make initialization intuitive. Slicing is readable, with support for negative indices, steps, and boolean masking. Writability is strong due to element-wise support for most operators, like `+`, `*`, `@`, and comparison operators.

2.7 Rust

In Rust, Arrays are fixed-size and immutable by default; to modify elements, they must be explicitly declared with `mut`, which is not intuitive. Ragged arrays are not allowed, decreasing writability. Comparisons like `arr1 == arr2` are only allowed if both arrays have the same size and type. This type safety increases clarity but reduces writability.

2.8 Final Decision

Among all the languages evaluated, I believe `Python - NumPy` provides the most readable and writable array operations. Its support for element-wise arithmetic, wide set of built-in operators, and simple, flexible array initialization make it easier to use and rich at the same time compared to the others.

3 Part C - Learning Strategy

To run the programs, I used my own computer with local compilers and interpreters. For answering the questions in Part A and gaining a general understanding of each language, I firstly consulted the official documentation for each language and tried what comes to mind and what I saw in the other programming languages as interesting. In the case of JavaScript, where official resources were harder to find, I relied on trusted sources like Mozilla's developer site [4]. When certain reasons behind the behaviors or functions were unclear, I referred to websites such as Stack Overflow [9], W3Schools [10], and GeeksforGeeks [2] to clarify specific details.

References

[1] Dart documentation. https://dart.dev/. Accessed: 2025-04-01.

[2] Geeksforgeeks. https://www.geeksforgeeks.org. Accessed: 2025-04-06.

[3] Go documentation. https://go.dev/doc/. Accessed: 2025-04-02.

[4] Javascript — mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed: 2025-04-06.

[5] Kotlin documentation. https://kotlinlang.org/. Accessed: 2025-04-02.

[6] Numpy documentation. https://numpy.org/doc/2.1/reference/arrays.ndarray.html. Accessed: 2025-04-04.

[7] Php documentation. https://www.php.net/docs.php. Accessed: 2025-04-04.

[8] Rust documentation. https://www.rust-lang.org/learn. Accessed: 2025-04-04.

[9] Stack overflow. https://stackoverflow.com. Accessed: 2025-04-06.

[10] W3schools. https://www.w3schools.com. Accessed: 2025-04-06.