CS315 Homework 1

Rüzgar Ayan 21801984

Question 1. What types are legal for subscripts?

Dart:

A quick try with string subscripts gives us a very informative error that shows only integer subscripts are legal:

```
var array = ['a','b','c','d','e'];
print("Initial array:"); print(array);
print("array[0]:"); print(array[0]); //Integer is legal

//When we try to use string, compilation error
// Error: A value of type 'String' can't be assigned to a variable of type
'int'.
print("array['xyz']:"); print(array["xyz"]);
```

Javascript:

Obviously integers are legal.

```
array = [1,2,3,4,5];
console.log("array[1]:", array[1]); //Prints 2
```

If we try to use strings, floating point numbers and even arrays or objects, we don't get any error and the result is just undefined as in the following code snippet.

```
another_array = [10,11];
object = {name: "Ruzgar", surname: "Ayan"};
console.log("array[1.2]:", array[1.2]); //Prints undefined
console.log("array['abc']:", array["abc"]); //Prints undefined
console.log("array[another_array]:", array[another_array]); //Prints
undefined
console.log("array[object]:", array[object]); //Prints undefined
```

We can also set the values using these non-integer types as the subscripts as follows:

```
array[true] = "Boolean index";
array[1.2] = "Double index";
array["abc"] = "String index";
array[another_array] = "Array index";
array[object] = "Object index";
console.log("Final state of the array", array);
console.log("Final length of the array", array.length);
```

And the output becomes:

```
Final state of the array [ 1,
  2,
  3,
  4,
  5,
  true: 'Boolean index',
  '1.2': 'Double index',
  abc: 'String index',
  '10,11': 'Array index',
  '[object Object]': 'Object index' ]
Final length of the array 5
```

As I understand, this is now the mix of an array and an object because it still says that its length is 5 but it also stored the non-integer subscripts by converting them to strings. Although this is not a pure array anymore, I would say that all types are legal for array subscripts because they don't give any errors.

PHP:

The arrays in PHP are associative, so it is possible to initialize arrays with boolean, floating point number and string keys as follows:

```php
$mixed_array = ["Spring", "Winter", "Fall", "Summer", 2.2 => "Double index",
"xyz" => "abc", true => "false"];
print_r($mixed_array);
echo "mixed_array[2] = $mixed_array[2]\n";
echo "mixed_array['xyz'] = "; echo $mixed_array["xyz"]; echo "\n";

//All below gives Undefined index error.
//echo "mixed_array[true] = $mixed_array[true]"
//echo "mixed_array[1.2] = $mixed_array[1.2]"
```

However, we can see from the output below that the floating points are rounded below and booleans are evaluated as 0 and 1.

```
(
    [0] => Spring
    [1] => false
    [2] => Double index
    [3] => Summer
    [xyz] => abc
)
mixed_array[2] = Double index
mixed_array['xyz'] = abc
```

Also later use of boolean and floating point number values in subscripts gives errors, so only integers and strings are legal for subscripts.

Python:

A quick try with string subscripts gives us a very informative error:

```python
#IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.newaxis
(`None`) and integer or boolean arrays are valid indices
print(arr["abc"])
```

We can see all the legal subscripts from this error and they are tried as follows:

```
arr = np.array([1, 2, 3, 10])
print(arr[True]) #[[1 2 3 10]]
print(arr[False]) #[]
print(arr[arr > 2]) #[3 10]
print(arr[[1,2]]) #[2 3]
print(arr[2, ...]) #[3]
```

Rust:

Only integer types are allowed in Rust. In the example below, when we try to use bool, float or string subscripts it gives the error **”`SliceIndex<[i32]>` is not implemented for xxx type”**

```
let mut array:[i32;5] =  [1,2,3,4,5];
println!("array is {:?}", array);
println!("array[0] is {}", array[0]);

// All below gives the error
// the trait `SliceIndex<[i32]>` is not implemented for
`bool`/`{float}`/`&str`
// This means only i32 is accepted for subscript tpye
//println!("array[true] is {}", array[true]);
//println!("array[1.2] is {}", array[1.2]);
//println!("array['abc'] is {}", array["abc"]);
```

Question 2. Are subscripting expressions in element references range checked?

Dart:

Yes, ranges are checked and a RangeError is thrown.

```
//Gives "Uncaught Error: RangeError (index): Index out of range: index should
be less than 5: 10"
print("array[5]:"); print(array[5]);
```

Javascript:

The answer to this is unclear since javascript does not throw an error in this case but it just returns undefined. Maybe, this allows the programmer to access invalid indices but then check if the value is undefined. I would consider this as range checking since it does not return garbage value as in C but actually knows that we are accessing outside of the range.

```
array = [1,2,3,4,5];
console.log("array[array.length]:", array[array.length]); //Does not give
error but gives undefined
console.log("array[-1]:", array[-1]); //Does not give error but gives
undefined
```

PHP:

Invalid integer and string indices give warnings during the compilation but still they are allowed. That is, range checking is done but it doesn't throw errors.

```
//PHP Notice:  Undefined offset: 10 in 762760953/source.php on line 14
echo "mixed_array[10] = $mixed_array[10]\n";
//PHP Notice:  Undefined index: xyz2 in 1330870507/source.php on line 19
echo "mixed_array['xyz2'] = "; echo $mixed_array["xyz2"]; echo "\n";
```

The outputs above are printed as empty.

Python:

Yes, ranges are checked and an IndexError is thrown.

```
print(arr[2]) #3
print(arr[-3]) # 2
#IndexError: index 4 is out of bounds for axis 0 with size 4
print(arr[4])
```

Rust:

Yes, ranges are checked and an error is thrown.

```
let mut array:[i32;5] =  [1,2,3,4,5];
//Throws "index out of bounds: the length is 5 but the index is 5" at
compilation
println!("array[5] is {}", array[5]);
```

Question 3. When are subscript ranges bound?

Dart:

Dart arrays are heap-dynamic and the subscript ranges are bound at run-time. In the documentation [1], it is stated that the default arrays in Dart language are growable lists. When we add another element to the array/list that normally gave error for "array[5]", we can now access that element because subscript ranges have changed:

```
array.add('x');
print("array[5]:"); print(array[5]); //x
```

Javascript:

Since Javascript uses Heap-dynamic arrays [2], both the storage allocation and the binding of subscripts is dynamic. In the example below, we can push a new element to the array and its size changes. Before the push, it gives undefined but after the push it gives the value. This means that the subscript ranges have changed with our push.

```
console.log("array[5]:", array[5]); //Gives undefined now
array.push(6);
console.log("array[5]:", array[5]); //Gives the new value now, subscript
range is dynamically changed
```

PHP:

Same as Javascript answer above, the arrays are Heap-dynamic. The subscript range bindings occur dynamically at runtime.

Python:

According to the numpy documentation [3], “NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.”
Since the array size is fixed, the subscripts are bound only once at the declaration of the array unlike the dynamic arrays from the other languages above [4].

Rust:

Rust arrays are of fixed-size and their sizes are known at compile-time [5], and because of that subscripts are bound at the compile time.

Question 4. When does allocation take place?

Dart:

As explained in the previous question, allocation is done at run-time since the arrays are heap-dynamic [1], they change during the run-time.

Javascript:

Again, same as Dart. Allocation is done at run-time.

PHP:

Again, same as Dart and Javascript. Allocation is done at run-time.

Python:

As explained in the previous question, numpy arrays have fixed size [3, 4]. They are still allocated in run-time but they are allocated only once because they cannot grow.

Rust :

Rust arrays are static and because of that allocation is done before run-time [5]. This causes fixed-size arrays and we cannot push new elements into the arrays as in the other languages above.

Question 5. Are ragged or rectangular multidimensional arrays allowed, or both?

For the answers to this question, programming languages that support ragged arrays are considered to also support rectangular arrays directly because rectangular arrays are a subset of ragged arrays. However, if there is different syntax for these two types of arrays, these are explained.

Dart:

Both are supported, there is no separate syntax for them.

```
List<List<int>> rectangular = [[1,2,3], [4,5,6], [7,8,9]];
List<List<int>> ragged = [[1], [2,3], [4,5,6]];
```

Javascript:

Both are supported, there is no separate syntax for them.

```
ragged = [[1,2],[3,4,5]];
rectangular = [[1,2],[3,4]];
console.log("ragged: ", ragged);
console.log("rectangular: ", rectangular);
```

PHP:

Both are supported, there is no separate syntax for them.

```
$multidim_array = [[1,2,3],[4,5]];
$multidim_array2 = [[1,2,3],[4,5,6]];
```

Python:

Both are supported and can be created using the same syntax.

```
rectangular = np.array([[1,2,3],[4,5,6],[7,8,9]])
ragged = np.array([[1],[2,3],[4,5,6]])

#[[1 2 3] [4 5 6] [7 8 9]]
print("rectangular :", rectangular)
#[list([1]) list([2, 3]) list([4, 5, 6])]
print("ragged: ", ragged)
```

However, a warning pops up saying that this way of defining ragged arrays is deprecated: “VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray ragged = np.array([[1],[2,3],[4,5,6]])”. So I have also added this ‘dtype=object’ part and it now works fine. Just note that they are printed differently, ragged array is shown as an array of lists.

Rust:

With the syntax used so far for the arrays, we can only create rectangular arrays. However, **“Box”** in Rust can be used to create ragged multidimensional arrays. This is similar to using pointers in C.

```
let mut rectangular:[[i32;3];3] = [[1,2,3],[4,5,6],[7,8,9]];
println!("rectangular = {:?}", rectangular);

let boxes: [Box<[u8]>; 3] = [
    Box::new([1]),
    Box::new([2,3]),
    Box::new([4,5,6])
];
let ragged: &[Box<[u8]>] = &boxes;
println!("ragged = {:?}", ragged);
println!("rectangular[1][1] = {}", rectangular[1][1]);
println!("ragged[1][1] = {}", ragged[1][1]);
```

with output:

```
rectangular = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ragged = [[1], [2, 3], [4, 5, 6]]
rectangular[1][1] = 5
ragged[1][1] = 3
```

Question 6. Can array objects be initialized?

Dart:

Yes, this initialization was used in all the answers above as well. “List.filled(value, size, growable)” can also be used to initialize large arrays with the same element.

```
var initialized_arr = [1,2,3,4,5];
List<int> initialized_arr2 = List.filled(100, 10, growable: true);
```

Javascript:

Yes, this initilization was used in all the answers above as well. “Array(size).fill(value)” can also be used to initialize large arrays with the same element.

```
initialized_arr = [1,2,3];
initialized_arr2 = Array(100).fill("element"); // Array of 100 "element"s
```

PHP:

Yes, this initilization was used in all the answers above as well. “array_fill(start, length, value)” can also be used to initialize large arrays with the same element.

```
$initialized_arr = ["banana", "orange"];
print_r($initialized_arr);
```

```
$initialized_arr2 = array_fill(100, 10, "fruit");
print_r($initialized_arr2); //Keys 100..109 have the value "fruit"
```

Python:

Yes, this initilization was used in all the answers above as well. “np.full(dimensions, value)” can also be used to initialize large arrays (possibly multidimensional) with the same element.

```
initialized_arr = np.array([1, 2, 3, 10])
initialized_arr2 = np.full((2,3), "vegetable")
```

Rust:

Yes, this initilization was used in all the answers above as well. The syntax below for “initialized_arr2” can also be used to initialize large arrays with the same element.

```
let mut initialized_arr:[i32;5] =  [1,2,3,4,5];
let mut initialized_arr2:[i32;100] =  [10;100]; //Array of 10's of size 100
```

Question 7. Are any kind of slices supported?

Dart:

Slices are supported through the **sublist(start, end)** method provided for the arrays/lists.

```
var arr = [1,2,3,4,5];
var slice = arr.sublist(1,3);
print(slice); //[2,3]
```

Javascript:

Slices are supported through the **slice(start, end)** method provided for the arrays. However modifying the slices to modify the original array is not possible.

```
console.log("array: ", array);
console.log("array.slice(2,4): ", array.slice(2,4));

//Gives ReferenceError: Invalid left-hand side in assignment
//Cannot change after using the slice method
//array.slice(2,4) = [13,14];
```

The output to the code above is:

```
array:  [ 1, 2, 3, 4, 5, 6 ]
array.slice(2,4):  [ 3, 4 ]
```

Also, another method **splice** is provided for arrays to modify the elements in a slice of the array. An example is:

```
//splice is used to modify slices
array.splice(2,2, "New element 1", "New element 2");
```

```
console.log("array after splice: ", array);
```
with the output
```
array after splice:  [ 1, 2, 'New element 1', 'New element 2', 5, 6 ]
```

PHP:

There is a function called **array_slice**[6] provided by the language that returns a slice of the array as follows:

```
$original_array = [1,2,3,4,5];
$sliced_array = array_slice($original_array, 2, 2, true);
echo "sliced array ";
print_r($sliced_array);
$sliced_array[0] = "Modified element";
echo "original array after modifying its slice ";
print_r($original_array); //Doesn't change, array_slice just creates a copy
```
with output:
```
sliced array Array
(
    [2] => 3
    [3] => 4
)
original array after modifying its slice Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 4
    [4] => 5
)
```
Just like in Javascript, this slice cannot be used to modify the contents of the original array.

Python:

Yes, slices are supported and they are much more useful in this language. We can actually modify a slice of the array and see its effects in the original array as below:

```
print("arr ", arr) #[1, 2, 3, 10]
print("arr[1:3] ", arr[1:3]) #[2, 3]
print("arr[::2] ", arr[::2]) #[1, 3]
arr[::2] = [25, 25]
print("arr ", arr) #[25, 2, 25, 10]
```

Rust:

Yes, there is even a separate type for slices that is different from the regular arrays. We cannot directly set one to each other. Also, changing the elements of the slice does not change the original array elements as illustrated in the code snippet below.

```
let mut array:[i32;5] =  [1,2,3,4,5];
```

```
println!("array is {:?}", array); //[1, 2, 3, 4, 5]
let mut slice = &array[2..4];
println!("slice = array[2..4] is {:?}", slice); //[3, 4]
slice = &array[3..5];
println!("Now slice is {:?}", slice); //[4, 5]
println!("array is {:?}", array); //Still not changed after modifying the
slice.
```

Question 8. Which operators are provided?

Dart:

According to [1], there are only “+”, “==”, “[]”, “[]=” where the last two are already obvious. The first two are shown below. We see that the “==” operator do not actually check for the equality of the elements, but it checks for the equality of the references.

```
print([1,2] + [3,4]); //[1,2,3,4]

var arr1 = [1,2];
var arr2 = [1,2];
print(arr1 == arr2); //false
print(arr1 == arr1); //true
```

Javascript:

There are operators such as “+”, “*” which are defined on any objects and thus they are also usable on arrays. However, the result of these operators do not make much sense with the arrays. In the following example code, the arrays are just converted to strings and the operations are done on these strings. There is also the destructuring operator, which makes sense for arrays as well. These are shown in the following example.

```
[a,b] = [123,456];
console.log("a =", a, ", b =", b);
console.log("([3,4] + [1,2]): ", ([3,4] + [1,2]));
console.log("([3,4] * [1,2]): ", ([3,4] * [1,2]));
```

with the output

```
a = 123 , b = 456
([3,4] + [1,2]):  3,41,2
([3,4] * [1,2]):  NaN
```

There are also a lot of **methods** provided for the arrays, below are some frequently used ones of them:

```
array = [1,2,3,4]
console.log("array: ", array);

array.pop();
```

```
console.log("after array.pop(): ", array);
array.shift();
console.log("after array.shift(): ", array);
array.push(10);
console.log("after array.push(10): ", array);

console.log("[3,4].concat([1,2]): ", [3,4].concat([1,2]));
```

PHP:

The provided operators are "+" (union) operator and several different comparison operators [7] all shown in the code snippet below.

```
echo "[1,2] + [3,4,5] = \n";
print_r([1,2] + [3,4,5]);
//Result is [1,2,5], the first array overwrites the same indices when finding
the union

$a = [1 => "First", 2 => "Second"];
$b = [2 => "Second", 1 => "First"];
var_dump($a == $b); //TRUE
var_dump($a === $b); //FALSE
var_dump($a != $b); //FALSE
var_dump($a <> $b); //FALSE
var_dump($a !== $b); //TRUE
```

Union does not work as first expected. Since PHP arrays are associative arrays, it actually takes the union of the keys and not the values. Since in the above example first array has the keys {0,1} and the second one has the keys {0,1,2}, the result will have the union of these key sets and the values of common keys comes from the first array as illustrated above.

There is also a distinction between "==" and "===". The first operator just checks whether or not each key has the same value where the second operator also cares about the order of these keys in the associative array.

Python:

The regular operations for numbers such as addition, multiplication, exponentitation and many more are supported on numpy arrays [7]. These can be the elementwise addition of two arrays or the addition of an array with a scalar as both illustrated below. Also, matrix multiplication is provided through the "@" operation.

```
print("[1, 2, 3, 4] + [4, 3, 2, 1] = ", np.array([1, 2, 3, 4]) +
np.array([4,3,2,1])) #[5,5,5,5]
print("[1, 2, 3, 4] + 10 = ", np.array([1, 2, 3, 4]) + 10) #[11,12,13,14]
print("[1, 2, 3, 4] ** 2 = ", np.array([1, 2, 3, 4]) ** 2) #[1,4,9,16]

#Matrix product
print("[[1, 2], [3, 4]] * [[0, 1], [1, 0]] = ", np.array([[1, 2], [3, 4]]) @
np.array([[0, 1], [1, 0]]))
#[[2,1],[4,3]]
```

Rust:

Classical relational operators are provided for the rust arrays. From the second error in the answer, we see that these operators are only provided when both used arrays are of the same size. These operators compare the arrays element by elements.

```
let mut array:[i32;5] =  [1,2,3,4,5];
let mut array2:[i32;5] =  [1,2,3,4,5];
let mut array3:[i32;5] =  [5,4,3,2,1];
let mut array4:[i32;3] =  [1,2,3];
println!("{}", array == array2); //true
println!("{}", array == array3); //false
println!("{}", array3 > array); //true
println!("{}", array3 < array); //false
println!("{}", array <= array2); //true

//error[E0369]: cannot add `[i32; 5]` to `[i32; 5]`
//println!("{:?}", array + array2);

//Error no implementation for `[i32; 5] == [i32; 3]`
//println!("{}", array == array4); Must be the same size
```

Discussion Question 1. Best Language for Array Operations

I have two candidates depending on the requirements. If we can take advantage of an associative array, then PHP is obviously a very good candidate since the language's array design and array methods are totally on top of building associative arrays. Note that there are other alternatives of associative arrays in the other languages (such as maps and dictionaries), but I don't consider them since we are talking about the arrays.

If we don't need associative arrays, then my answer would be Python. I would directly eliminate Rust since it is very limited with the fixed arrays and a low number of functionalities related to arrays. This is expected since Rust is probably designed for more performance and more low-level programming. The rest, Dart, Javascript and Python are mostly similar dynamic arrays in terms of their capabilities but I would choose Python because of its better readibilitiy/writabilitiy and some extra functionalities. I especially liked that we can change the slices of Python to change the original array, which I couldn't do in the other languages.

Discussion Question 2. Learning Strategy

Since the whole homework was about arrays in these languages, I started by learning the usage and syntax of the arrays in these languages. My first source of learning was programming tutorial websites such as “tutorialspoint.com” or “w3schools.com”. I believe these websites are more helpful than the actual programming language documentations most of the time, because they have clearer explanations and examples for those learning the language for the first time. After getting the basics, I opened an online compiler for the language and just repeated what I saw in the tutorials. Then, I modified and reran these example codes to really understand what is happening. Even if I got an error, that was helpful because I learned new things from that error explanation. For example, I learned that only integer array subscripts are legal in Dart through an error explanation.

When I was going to answer a specific question in this homework, I first read the official documentations this time. Because these are more reliable and concise than tutorial websites. Again, I have tried what I have read from these documentations. Whenever there were some unclear points in the document, I could find out the truth by trying the possibilities and seeing the results/errors.

Since I was trying to learn the same concepts in several different languages, I was able to retry the mistakes I’ve done in one language in the others to see whether or not it will work with this one. This helped me compare the languages with each other in different areas.

For some more specific questions/problems that I couldn’t find the answer of in the documentations, I have looked into solutions from websites such as Stack Overflow. These answers were mostly too complex for this homework and using libraries and other non-array related classes/methods. I have understood the ideas from these answers and tried to reduce them to whatever level I need for this homework.

The hardest question for me was the last one asking for the provided array operators. Most of the documentations for the languages did not give a direct list of operators and I needed to look at many different resources to find all the possible operators. For some languages, I have found some operators but I am not sure if I’ve found all of them since there was no official documentation listing “all the operators”. In that sense, some languages were much better in terms of documentation. For example, I was able to find all the information I needed in PHP whereas there were much less resources for Dart and Rust. I believe this was an important factor for the learning process of a programming language.

References

[1] “List class - dart:core library - Dart API”, https://api.dart.dev/stable/2.14.4/dart-core/List-class.html

[2] “Array - JavaScript - MDN Web Docs”, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

[4] “What is NumPy? — NumPy v1.22.dev0 Manual”, https://numpy.org/devdocs/user/whatisnumpy.html

[3] “numpy.append — NumPy v1.21 Manual”, https://numpy.org/doc/stable/reference/generated/numpy.append.html

[5] “array - Rust”, https://doc.rust-lang.org/std/primitive.array.html

[6] “array_slice - Manual - PHP”, https://www.php.net/manual/en/function.array-slice.php
[7] “Array Operators - Manual - PHP”, https://www.php.net/manual/en/language.operators.array.php
[8] “NumPy quickstart — NumPy v1.21 Manual”, https://numpy.org/doc/stable/user/quickstart.html
[9] “Rust - Operators - Tutorialspoint”, https://www.tutorialspoint.com/rust/rust_operators.htm