Iterators & Generators

Iterables:

Iterables objects are a generalization of arrays. That's a concept that allows us to make any object useable in a **`for..of`** loop.

Of course, Arrays are iterables. But there are many other built-in objects that are iterables as well. For instance, strings are also iterables.

If an object isn't technically an array, but represents a collection (list, set) of something, then **`for..of`** is a great syntax to loop over it, so let's see how to make it work.

Array, strings, sets, maps are considered as iterables. On the hand, objects are not iterables.

```
const arr = [20, 23, 10]
const str = "HelloWorld"
for (let index=0; index <arr.length; index++0 {
const element = arr [index];
console.log(element)
}
for (const value of array) {
     console.log(value)
}
if we compare in string
for (const value of str) {
     console.log(value)
}
```

In JavaScript we use for-in loop

```
for (in)
     if (object.hasOwnProperty.call(object, key)) {
     const element = object [key];
     }
```

Let's show how an object cannot be used in for of loop

```
const obj= {
     name=" unnamed";
     age= -85
for (const value of Obj) {
     console.log(value)
```

Result: it shows that object is not iterables

→ Objects can be used in for in loop

For in loop

```
for (const key in arr) {
     console.log (key)
}
for (const key in obj ) {
     console.log(key)
     console.log(obj[key]);
}
```

For in loop reflects the property name.

Interaction protocol

In order to make a particular object you need to define the property and it should be a function it should return the object

→ The iterables protocol allows JavaScript objects to define or customize their iteration behavior

Property [symbol.iterator]
value - A zero-argument function that returns an object, confirming to the iterator protocol.

Obj [symbol.iterator]

Or

```
const obj = {
name:"unnamed",
path: "somevalue",
[symbol.iterator]: function() {
     return{}
}
```

-->To get the subsequent value we need to tell JavaScript to use "next" in iterator protocol to define it.

```
const obj = {
     name:"unnamed",
     age: -85,
     path: "somevalue",
     [Symbol.iterator]: function()  {
          return{
               next: function () {
                    propertyCount ++:
                    switch (propertyCount )  {
                    case1:
                         return {value: obj.age, done: false}

                    case2:
                         return {value: obj.name, done: false}

                    case3:
                         return {value: obj.path, done: false}

                    default:
                         return { value: undefined, done:true}

                    }
               }

          }

     }
}

for (const value of obj) {
     console.log(value)
}



const str = "This is my string"

// for (const iterator of str) {
//   console.log(iterator)
// }

const iterator = str [symbol.iterator] ()
// console.log(iterator)
console.log (iterator.next())
console.log (iterator.next())
console.log (iterator.next())
```

```
console.log (iterator.next())

console.log (iterator.next())
```

This is happening because of the above string

For the string type we can try this string

```
string.prototype[symbol.iterator] = function() {
     Let count = 4
     return {
          next: function () {

               if(count > 0) {
               count -- ;
               return { value: "Yash", done: false}
               }else {
                    return{ value: undefined, done: true}
               }
          }
     }
}
```

Here all the name "Yash" will be in a single line unless as in const iterator.

→ If we want to overwrite then we will use **next**

Ex:

```
class Vehicle{
     constructot(year) {
          this.year = year
     }

     [symbol.iterator] (){
     }
}

const v1 = new vehicle ()
for (const iterator of v1) {
}
```

Generators

The Generator object is returned by a generator function and it conforms to both the iterables protocol and the iterator protocol.

You can pause the execution of function

Generators are nothing but a simple function, only difference is we have to put * (star)

Ex:

```
function * numbersGen() {
```

Or

```
function * numbersGen() {

yield 1
yield 2
yield 3
yield 4
}

const numGen = numbersGen ()
console.log(numbersGen())
```

Generator uses the same principle as iterators.

→ When we call numbersGen it gives the iterator object, if we call next upon it we will be getting the same object value

```
console.log(numGen.next())
console.log(numGen.next())
console.log(numGen.next())
```

Since it is an iterator we can also use for of loop

```
for (const item of numGen ) {
    console.log(item)
```

→If const array numbers are huge amount then we can't store them to do it in a efficient manner we will use generator function

```
function* numbersGen() {
    let i=0
    while (true) {
        yield i++
    }
```

```
}

const numGen = numbersGen ()
```

→ By keeping * then the numbers function will become a generator function in JavaScript.
→ To generate a range of values like in python we can be able to generate here in very efficient way.

EX:

We do not have range function in JavaScript but let's create it

```
function* range(end=10, start=0, step=1) {
     for (let i= start; i < end; i += step) {
          yield i
     }
}

const rangeArr = [...range(50, 10, 5)] -using spread operator

Output:   (8) [10, 15, 20, 25, 30, 35, 40, 45]
```

Saving the iteration in other variable

```
const iter = range (5, 2)
```