Programming Methodology

Lab 8: Iterators and Generators

Wednesday October 22, 2025

Weekly Lab Agenda

- Go over reminders/goals
- Review past material
- Work in groups of 2-3 to solve a few exercises
    - Please sit with your group from last week.
- Discussion leaders will walk around and answer questions
- Solutions to exercises will be reviewed as a class
- Attendance taken at the end

Reminders

- HW 5 will be released this week, and will be due after the midterm (Sunday 11/2 to be exact).
- Midterm 2 is next Wednesday (10/29) from 7-9pm
  - Come to office hours for help!
  - You can also find past midterms on Canvas.
- If you need to miss lab and have a valid reason according to the syllabus (medical, other personal) please fill out the questionnaire on Canvas before the start time of your lab.
  - Waking up late, bus was late are NOT valid reasons to miss lab.

Today's Goals

- Iterators
- Generators

Iterators Review

```
interface Iterator<ValueType> {
  next(): IteratorResult<ValueType>;
}

interface IteratorResult<ValueType> {
  done: boolean; // true indicates Iterator has reached its end
  value?: ValueType; // could be undefined, specifically if done is true
}
```

An Iterator is any object that implements the above interface, which is built-in to JavaScript/TypeScript.

The next() method of an iterator yields the next item and moves the iterator forward. The result of next() is returned in the form of an IteratorResult, an object specifying the resulting item and whether the iterator has completed.

Generators Review

Implementing an Iterator requires us to define a class, where the **next()** method must return the next item and help to advance/prepare the iterator for future calls to **next()**. As we saw in last week's lab, keeping track of the current state of the iterator in preparation for future calls to **next()** is not always very straightforward.

As a result, we may prefer to implement a **Generator** instead, which often can be more readable and concise compared to implementing an Iterator.

Generators in JavaScript/TypeScript implement the IterableIterator interface. This means calling .next() on them will yield an item in the form of an IteratorResult. However, the main difference is how generators are defined and created.

Generators Review

To create a Generator, we write a generator function using **function***. This function uses **yield** instead of **return**, and creates a generator when it is called.

Each call to .next() of the generator returns the next item to be yielded, specifically in the same form as an IteratorResult.

Note that .next() returns IteratorResults despite us yielding numbers. This is because using **yield** instead of **return** automatically constructs the IteratorResult for us.

```
// declaring a generator function
function* myGenerator() {
  yield 1;
  yield 2;
  yield 3;
}
const gen = generator(); // create generator
console.log(gen.next().value); // prints 1
console.log(gen.next().value); // prints 2
console.log(gen.next().value); // prints 3
```

Exercise 1: Iterators

- Implement a class MergedIterator that merges two sorted number iterators into one. Each input iterator yields numbers in non-decreasing order.
- The `next()` method should return the smallest available value between the two iterators; in case of a tie, return the value from the first iterator.
- When one iterator is exhausted, continue returning values from the other until both are finished.
- Note: all values from both iterators must eventually be returned.

```
class MergedIterator implements Iterator<number> {
  // TODO: Implement this class
}
```

Exercise 1: Solution

```
class MergedIterator implements Iterator<number> {
  private _iter1: Iterator<number>;
  private _iter2: Iterator<number>;
  private _curr1: IteratorResult<number>;
  private _curr2: IteratorResult<number>;

  constructor(iterator1: Iterator<number>, iterator2: Iterator<number>) {
    this._iter1 = iterator1;
    this._iter2 = iterator2;
    this._curr1 = this._iter1.next();
    this._curr2 = this._iter2.next();
  }
  // ...
}
```

To merge the given iterators, we will want to store both iterators, as well as the current item we have obtained from each one. This allows us to compare the values from each iterator without advancing both iterators every time.

Exercise 1: Solution

```
next(): IteratorResult<number> {
  if (this._curr1.done && this._curr2.done) {
    // ...
  }
}
```

For the `next()` method, we can first begin by checking when at least one of the two iterators is out of items. First, we will check if both are out of items by examining `this._curr1` and `this._curr2`.

Exercise 1: Solution

```
next(): IteratorResult<number> {
  if (this._curr1.done && this._curr2.done) {
    return { done: true, value: undefined };
  }
}
```

If the done property is true for both, this means both iterators have ran out of items, so our MergedIterator has also ran out of items to return. We therefore return the IteratorResult indicating that our iterator has terminated.

Exercise 1: Solution

```
next(): IteratorResult<number> {
  if (this._curr1.done && this._curr2.done) {
    return { done: true, value: undefined };
  } else if (this._curr1.done) {
    // ...
  } else if (this._curr2.done) {
    // ...
  }
}
```

At this point, we know at least one iterator still has items to return. Here, we check if the first iterator is out of items. Afterward, we can then check if the second iterator is out of items.

Exercise 1: Solution

```
next(): IteratorResult<number> {
  if (this._curr1.done && this._curr2.done) {
    return { done: true, value: undefined };
  } else if (this._curr1.done) {
    const result = this._curr2;
    this._curr2 = this._iter2.next();
    return result;
  } else if (this._curr2.done) {
    const result = this._curr1;
    this._curr1 = this._iter1.next();
    return result;
  }
}
```

If the first iterator is out of items, we will return the current item of `iterator2`. We also need to update `this._curr2` to store the next item from `iterator2`, so we store `this._curr2` in a temporary variable, update `this._curr2`, and lastly return the temporary variable.

Exercise 1: Solution

```
private _take_from_first() {
  const result = this._curr1;
  this._curr1 = this._iter1.next();
  return result;
}

private _take_from_second() {
  const result = this._curr2;
  this._curr2 = this._iter2.next();
  return result;
}
```

To reduce some code duplication, we will define two helper methods in the MergedIterators class.

Exercise 1: Solution

```
next(): IteratorResult<number> {
  if (this._curr1.done && this._curr2.done) {
    return { done: true, value: undefined };
  } else if (this._curr1.done) {
    return this._take_from_second();
  } else if (this._curr2.done) {
    return this._take_from_first();
  }
}
```

Our `next()` method is then updated to use the helper methods introduced in the previous slide.

Exercise 1: Solution

```
next(): IteratorResult<number> {
  if (this._curr1.done && this._curr2.done) {
    return { done: true, value: undefined };
  } else if (this._curr1.done) {
    return this._take_from_second();
  } else if (this._curr2.done) {
    return this._take_from_first();
  } else if (this._curr1.value <= this._curr2.value) {
    // ...
  }
}
```

At this point, we now know that neither `iterator1` nor `iterator2` have ran out of items. We can now safely compare the current values of both iterators.

Exercise 1: Solution

```
next(): IteratorResult<number> {
  if (this._curr1.done && this._curr2.done) {
    return { done: true, value: undefined };
  } else if (this._curr1.done) {
    return this._take_from_second();
  } else if (this._curr2.done) {
    return this._take_from_first();
  } else if (this._curr1.value <= this._curr2.value) {
    return this._take_from_first();
  }
}
```

As the problem description specifies, we will return the smaller of the two current iterator values, defaulting to the value from the first iterator if there is a tie.

Exercise 1: Solution

```
next(): IteratorResult<number> {
  if (this._curr1.done && this._curr2.done) {
    return { done: true, value: undefined };
  } else if (this._curr1.done) {
    return this._take_from_second();
  } else if (this._curr2.done) {
    return this._take_from_first();
  } else if (this._curr1.value <= this._curr2.value) {
    return this._take_from_first();
  } else {
    return this._take_from_second();
  }
}
```

Otherwise, it can only be that `iterator2` has a smaller current value than `iterator1`, so we will return the current value of `iterator2`.

Exercise 2: Generators

- Implement a generator for the MergedIterator exercise. It should return values the same way as the MergedIterator class.

```
function* mergedGenerator(
    iterator1: Iterator<number>,
    iterator2: Iterator<number>
): Generator<number> {
  // TODO: Implement this generator
}
```

Exercise 2: Solution

```
function* mergedGenerator(
  iterator1: Iterator<number>,
  iterator2: Iterator<number>
): Generator<number> {
  let curr1 = iterator1.next();
  let curr2 = iterator2.next();
  // ...
}
```

We begin by using `function*` to declare mergedGenerator as a generator function. Similar to our approach for Exercise 1, we want to store the current value of each iterator. This allows us to perform comparisons on the current values without advancing the two iterators unwantedly.

Small note: the type signatures for the input arguments have been omitted after this slide for spacing.

Exercise 2: Solution

```
function* mergedGenerator(iterator1, iterator2): Generator<number> {
  let curr1 = iterator1.next();
  let curr2 = iterator2.next();
  while (!curr1.done || !curr2.done) {
    // ...
  }
}
```

As long as one of the two iterators still has items to return, we will keep yielding values from our generator. Thus, we use the while loop above.

Exercise 2: Solution

```
function* mergedGenerator(iterator1, iterator2): Generator<number> {
  let curr1 = iterator1.next();
  let curr2 = iterator2.next();
  while (!curr1.done || !curr2.done) {
    if (curr1.done) {
      yield curr2.value;
      curr2 = iterator2.next();
    } else if (curr2.done) {
      yield curr1.value;
      curr1 = iterator1.next();
    }
    // ...
  }
}
```

Given that we have entered the while loop, we know that at least one iterator still has values to return. If `iterator1` is done, we will yield the current value of `iterator2`. If `iterator2` is done, we will yield the current value of `iterator1`.

Exercise 2: Solution

```
function* mergedGenerator(iterator1, iterator2): Generator<number> {
  let curr1 = iterator1.next();
  let curr2 = iterator2.next();
  while (!curr1.done || !curr2.done) {
    if (curr1.done) {
      yield curr2.value;
      curr2 = iterator2.next();
    } else if (curr2.done) {
      yield curr1.value;
      curr1 = iterator1.next();
    }
    // ...
  }
}
```

```
// Exercise 1
const result = this._curr2;
this._curr2 = this._iter2.next();
return result;
```

Notice how using a generator means we can first return `curr2`, then update `curr2` to the next value of `iterator2`. This is different from what we had to do in Exercise 1, as we had to prepare the iterator before calling return. In this generator, we can prepare the iterator after we yield.

Exercise 2: Solution


```
function* mergedGenerator(iterator1, iterator2): Generator<number> {
  let curr1 = iterator1.next();
  let curr2 = iterator2.next();
  while (!curr1.done || !curr2.done) {
    if (curr1.done) {
      yield curr2.value;
      curr2 = iterator2.next();
    } else if (curr2.done) {
      yield curr1.value;
      curr1 = iterator1.next();
    } else if (curr1.value <= curr2.value) {
      yield curr1.value;
      curr1 = iterator1.next();
    } else {
      yield curr2.value;
      curr2 = iterator2.next();
    }
  }
}
```

The rest of the while loop follows similarly in reasoning as the `next()` method in Exercise 1.

Note that the generator will automatically return the IteratorResult for a terminated iterator once we have stopped yielding values (i.e. exited the while loop).

Exercise 3: More Generators

- Implement a generator (chainGenerator) for the ChainIterator problem. It should behave the same way as ChainIterator from last week's lab.
- This means chainGenerator should take in an array of iterables. It should then yield items from the first iterable until it runs out. Then, it should yield items from the next iterable in the given array, repeating this process until reaching the end of the array of iterables. You may assume the given array is not empty.

```
function* chainGenerator<T>(iterables: Iterable<T>[]): Generator<T> {
  // TODO: Implement this generator
}
```

Exercise 3: Solution


```
function* chainGenerator<T>(iterables: Iterable<T>[]): Generator<T> {
  for (const iterable of iterables) {
    // ...
  }
}
```

We will first iterate through the array of iterables provided. Our goal is to move from the start of the array to the end, yielding all items from the current iterable before moving on to the next iterable.

Exercise 3: Solution


```
function* chainGenerator<T>(iterables: Iterable<T>[]): Generator<T> {
  for (const iterable of iterables) {
    const currIterator = iterable[Symbol.iterator]();
    let currVal = currIterator.next();
    // ...
  }
}
```

For the current iterable, we will use `[Symbol.iterator]()` to retrieve an iterator for the iterable. We then store the first item as the variable `currVal`.

Exercise 3: Solution


```
function* chainGenerator<T>(iterables: Iterable<T>[]): Generator<T> {
  for (const iterable of iterables) {
    const currIterator = iterable[Symbol.iterator]();
    let currVal = currIterator.next();
    while (!currVal.done) {
      yield currVal.value;
      currVal = currIterator.next();
    }
  }
}
```

While the current iterator still has items remaining, we will yield the current value from the iterator. We then advance the iterator afterwards and update `currVal` to be the next value.

Once the outer for loop terminates, we will have gone through all iterables and have no more items to yield. The generator will then automatically yield `{done: true, value: undefined}` to indicate that our generator has ran out of values.

Exercise 3: Solution


```
function* chainGenerator<T>(iterables: Iterable<T>[]): Generator<T> {
  for (const iterable of iterables) {
    for (const item of iterable) {
      yield item;
    }
  }
}
```

What about this implementation? Does this still work?

...

This actually works too! Since we have an array of iterables, we can also just loop through each iterable, yielding its items. This implementation further demonstrates how generators can sometimes be easier to read and understand than iterators.

Exercise 3: Solution


```
next(): IteratorResult<T> { // From Lab 7 Exercise 2
  let iter_result = this._curr_iterator.next();
  while (iter_result.done) {
    this._idx++;
    if (this._idx >= this._iterables.length) {
      return { done: true, value: undefined };
    }
    this._curr_iterator = this._iterables[this._idx][Symbol.iterator]();
    iter_result = this._curr_iterator.next();
  }
  return { done: false, value: iter_result.value };
}
```

When we quickly compare our generator implementation to our `.next()` implementation from ChainIterator (Exercise 2 from Lab 7), we again see how generators provide an easier to understand implementation of the same problem compared to iterators.

Exercise 3: Solution


```
next(): IteratorResult<T> { // From Lab 7 Exercise 2
  let iter_result = this._curr_iterator.next();
  while (iter_result.done) {
    this._idx++;
    if (this._idx >= this._iterables.length) {
      return { done: true, value: undefined };
    }
    this._curr_iterator = this._iterables[this._idx][Symbol.iterator]();
    iter_result = this._curr_iterator.next();
  }
  return { done: false, value: iter_result.value };
}
```

Notice that a lot of the details here where we track our current index in the `iterables` array and the iterator of the current iterable are not necessary using generators.