iteration

July 4, 2018

1 Iteration

Beautiful iteration in both `python` and `javascript`.

1.1 How to iterate

```
In [1]: numbers = [1, 2, 3, 4]
        for i in numbers:
            print(i)

1
2
3
4
```

```
In [1]: let numbers = [1, 2, 3, 4];

        for (let i of numbers) {
            console.log(i);
        }

1
2
3
4

Out[1]: undefined
```

1.2 Iterating the iterator once at a time

Two basic functions of iteration:

- `__iter__` in py and `[Symbol.iterator]()` in js, is to create an iterator, and being called on iterable
- `__next__` in py and `next()` in js, is to get the next element in a sequence of objects, and being called on iterator

```
In [2]: numbers = [1, 2, 3, 4]
        sum_numbers = 0
        iterator = iter(numbers)
        val = next(iterator)
        while val:
            sum_numbers += val
            try:
                val = next(iterator)
            except StopIteration:
                break
        # no break
        else:
            print("something wrong")
        print(f"The sum is {sum_numbers}")
```

```
The sum is 10
```

```
In [1]: let numbers = [1, 2, 3, 4];
        let sum_numbers = 0;
        let iterator = numbers[Symbol.iterator]();
        let next = iterator.next();

        while(!next.done) {
            sum_numbers += next.value;
            next = iterator.next();
        }
        console.log(`The sum is ${sum_numbers}`);
```

```
The sum is 10
```

```
Out[1]: undefined
```

1.3 Creating your own object iterable

```
In [2]: class FootballTeam:
            def __init__(self, players=None):
                self.players = players if players is not None else []

            def add_players(self, *players):
                self.players = self.players + list(players)

            def __iter__(self):
                return PlayerIterator(self.players)


        class PlayerIterator:
            def __init__(self, players):
```

```
            self.players = players
            self.index = 0

        def __next__(self):
            if self.index >= len(self.players):
                raise StopIteration
            player = self.players[self.index]
            self.index += 1
            return player


    fc = FootballTeam()
    fc.add_players(
        "boaz salossa",
        "kurniawan dwi yulianto",
        "bima sakti",
        "bambang pamungkas",
    )

    num_player = 0
    for p in fc:
        num_player += 1
    print(f"Total player: {num_player}")
```

```
Total player: 4
```

```
In [1]: class FootballTeam {
            constructor() {
                this.players = []
            }

            addPlayers(...names) {
                this.players = this.players.concat(names)
            }

            [Symbol.iterator]() {
                return new PlayerIterator(this.players);
            }

        }

        class PlayerIterator {
            constructor(players) {
                this.players = players;
                this.index = 0;
            }
```

```
            next() {
                let res = {value: undefined, done: true};
                if (this.index < this.players.length) {
                    res.value = this.players[this.index];
                    res.done = false;
                    this.index += 1;
                }
                return res;

            }
        }

        let fc = new FootballTeam();
        fc.addPlayers(
            "boaz salossa",
            "kurniawan dwi yulianto",
            "bima sakti",
            "bambang pamungkas",
        );

        let num_player = 0;
        for (let p of fc) {
            num_player += 1;
        }

        console.log(`Total player: ${num_player}`);
```

```
Total player: 4
```

```
Out[1]: undefined
```

1.4 Python iteration to infinity

Extra for python iteration, that `iter()` takes a callable and sentinel as its arguments, i.e. `iter(<callable>, <sentinel>)`

- callable is any object that returns True from a function call of `callable(object)`.
- sentinel is the value returned by the callable object that stops the iteration.

```
In [5]: # iterating for infinite sequences
        import datetime
        infinite_datetime = iter(datetime.datetime.now, None)
        next(infinite_datetime)
        next(infinite_datetime)
        # ... go on forever

        # please try it (by uncomment below),
        # to get to see the infinite sequence of time
```

```
        # and press CTRL + C, to end it like a boss!

        # now = next(infinite_datetime)
        # while now:
        #     print(now)
        #     now = next(infinite_datetime)
```

```
Out[5]: datetime.datetime(2018, 7, 3, 21, 44, 54, 814733)
```

```
In [10]: !cat text_with_end.txt
```

```
Here I am,
This is super important for you to read
END HERE
This should not be read
Not a chance!
```

```
In [1]: # read file, line by line until the callable produces the sentinel value,
        # in this example which is, "END HERE"
        with open("text_with_end.txt", "rt") as in_file:
            for l in iter(lambda: in_file.readline().strip(), "END HERE"):
                print(l)
```

```
Here I am,
This is super important for you to read
```

1.5 Generator

Generator is a function that generates an iterator in a lazy way, one at a time. The keyword of generator is `yield` instead of `return`. Let's talk about it, briefly.

The main difference that `yield` will return the thread of execution back to the caller (function) but only **temporarily** so that, it could **resume**. Think about it, like when you drive you need to yield at the roundabout (the triangle sign) so other cars could go circle through and you could resume driving at later time.

So to sum up, `return` statement will throw away **permanently** a function's local state, where `yield` keeps the local state so it could be **resumed** at any time by calling `next()`.

On top of that, in `js` the generator function is suffixed with an asterix or `function*` and uses `yield`. In python, it is simpler, just `yield`.

Takes time to play with the codes below to get your hands dirty with such *crazily* awesome concept! As this concept could help us understand many interesting programming *Abracadabras*, such as **async** programming, doing multiple things at once, like we normally do in our normal life :).

```
In [1]: function* number_gen() {
            yield 1;
            yield 2;
            yield 3;
```

```
        yield 4;
    }

    let sum_num = 0;
    let number_iterator = number_gen();
    let next_num = number_iterator.next();
    while(!next_num.done) {
        sum_num += next_num.value;
        next_num = number_iterator.next();
    }

    console.log(`The sum is ${sum_num}`);
```

```
The sum is 10
```

```
Out[1]: undefined
```

```
In [1]: def number_gen():
            yield 1
            yield 2
            yield 3
            yield 4


        sum_num = 0
        number_iterator = number_gen()
        next_num = next(number_iterator)
        while next_num:
            sum_num += next_num
            try:
                next_num = next(number_iterator)
            except StopIteration:
                break
        # no break
        else:
            print("something wrong")
        print(f"The sum is {sum_num}")
```

```
The sum is 10
```

1.6 Simplifying our iterable class with the concept of generator

```
In [1]: class LazyFootballTeam {
            constructor() {
                this.players = []
            }
```

```
        addPlayers(...names) {
            this.players = this.players.concat(names)
        }

        *[Symbol.iterator]() {
            for (let p of this.players) {
                yield p
            }
        }

    }

    let lfc = new LazyFootballTeam();
    lfc.addPlayers(
        "Ronaldo",
        "Ricky",
        "Messi",
        "Inzaghi",
        "Del Piero",
        "Nesta",
        "Iniesta",
        "Ricardo",
    );

    let lazy_num_player = 0;
    for (let p of lfc) {
        lazy_num_player += 1;
    }

    console.log(`Total lazy player: ${lazy_num_player}`);
```

```
Total lazy player: 8
```

```
Out[1]: undefined
```

```
In [1]: class LazyFootballTeam:
            def __init__(self, players=None):
                self.players = players if players is not None else []

            def add_players(self, *players):
                self.players = self.players + list(players)

            def __iter__(self):
                for _ in self.players:
                    yield _
```

```
        lfc = LazyFootballTeam()
        lfc.add_players(
            "Ronaldo",
            "Ricky",
            "Messi",
            "Inzaghi",
            "Del Piero",
            "Nesta",
            "Iniesta",
            "Ricardo",
        )
        lazy_num_player = 0
        for p in lfc:
            lazy_num_player += 1
        print(f"Total lazy player: {lazy_num_player}")
```

```
Total lazy player: 8
```

1.7 Being functionally lazy is good

1.7.1 Okay, why should we bother with being functionally lazy ?

Imagine that you have a collection of million objects, and you are only interested in a subset of it. And worse than that, you have a machine with only tiny memory to compute and you are a cheap guy, but smart one.

In this scenario, given your smart brain you could use generator concept functionally, **to the rescue! Being functionally lazy** with generator makes computing cheap and more efficient based on the *least amount of work done's* principle.

Laziness could speed you up, once you are **functional** in understanding **generator**

For this fake demo, we would like to recruit only 2 players that start with **R** from our lazy football team. And of course, it is polluted with my favorite letter.

```
In [2]: function *filter_player(players, criteria) {
            for (let player of players) {
                if (criteria(player)) {
                    yield player
                }
            }
        }

        function *take_player(players, number) {
            if (number < 1) return;
            let count = 0;
            for (let player of players) {
                yield player;
                count += 1;
```

```
                if (count >= number) {
                    // stop the iterator permanently
                    return;
                }
            }
        }

        let r_player = [];
        for (let player of take_player(filter_player(lfc, p => p[0] === "R"), 2)) {
            r_player.push(player);
        }
        console.log(`The superstar players start with R: ${r_player}`);
```

```
The superstar players start with R: Ronaldo,Ricky
```

```
Out[2]: undefined
```

```
In [4]: def filter_player(players, criteria):
            for _player in players:
                if criteria(_player):
                    yield _player


        def take_player(players, number):
            if number < 1:
                return
            count = 0
            for _player in players:
                yield _player
                count += 1
                if count >= number:
                    return


        r_player = []
        for player in take_player(filter_player(lfc, lambda x: x[0] == "R"), 2):
            r_player.append(player)

        print(f"The superstar players start with R: {','.join(r_player)}")
```

```
The superstar players start with R: Ronaldo,Ricky
```

1.8 Generator not only can be paused, but also can send something back

In py, the method is called `.send(<value>)` and in js, is `next(<value>)` on its generator object.

```
In [1]: function *jumping_range(start, end) {
            let current = start;
```

```
        while(current <= end) {
            let jump = yield current;
            current += jump || 1;
        }
    }

    let res_odd = [];
    let odd_number_gen = jumping_range(start=1, end=10);
    let next_odd = odd_number_gen.next();
    while (!next_odd.done){
        res_odd.push(next_odd.value);
        // next(2) is to send back the value (in this case, 2),
        // where it stops its thread of execution
        // this value is stored as jump in the generator
        next_odd = odd_number_gen.next(2);
    }

    console.log(`Odd numbers: ${res_odd}`);
```

```
Odd numbers: 1,3,5,7,9
```

```
Out[1]: undefined
```

```
In [2]: def jumping_range(start, end):
            current = start
            while current <= end:
                jump = yield current
                current += jump if jump else 1


    res_odd = []
    odd_number_gen = jumping_range(start=1, end=10)
    next_odd = next(odd_number_gen)
    res_odd.append(next_odd)
    while next_odd:
        try:
            # send(2) is to send back the value (in this case, 2),
            # where it stops its thread of execution
            # this value is stored as jump in the generator
            next_odd = odd_number_gen.send(2)
            res_odd.append(next_odd)
        except StopIteration:
            break
    # no break
    else:
        print("something wrong")

    print(f"Odd numbers: {res_odd}")
```

```
Odd numbers: [1, 3, 5, 7, 9]
```

2 Generator comprehension

The idea of comprehension is to simplify the syntax, becoming terse to write for lazy programmers.

Although it's cool to be short, please always prefer to be readable. After all it will save "The-future-You" in maintaining your code.

Python and javascript uses `(<expression>)` as their syntatic-sugars for generator expression.

```
In [2]: square_num = (n*n for n in range(4))
        square_num

        for i in square_num:
            print(i)
```

```
0
1
4
9
```

```
In [4]: // generator comprehension syntax, however did not make it into es6 ;( !
        function *range(start, end) {
            let current = start ;
            while(current <= end -1 ) {
                yield current;
                current += 1
            }
        }

        square_num = (for n of range(0,4) n*n)

        for (let n of square_num) {
            console.log(n);
        }
```

```
        evalmachine.<anonymous>:8

        square_num = (for n of range(0,4) n*n)

                      ^^^



        SyntaxError: Unexpected token for
```

```
at createScript (vm.js:80:10)

at Object.runInThisContext (vm.js:139:10)

at run ([eval]:1002:15)

at onRunRequest ([eval]:829:18)

at onMessage ([eval]:789:13)

at emitTwo (events.js:126:13)

at process.emit (events.js:214:7)

at emit (internal/child_process.js:772:12)

at _combinedTickCallback (internal/process/next_tick.js:141:11)

at process._tickCallback (internal/process/next_tick.js:180:9)
```