파이썬 프로그래머를 위한 러스트 입문

윤인도

freedomzero91@gmail.com

CH6. 데이터 구조와 이터레이터

데이터 구조(Data structure)란, 컴퓨터에서 어떠한 값의 모음을 효율적으로 나타내기 위한 방법을 의미합니다. 예를 들어, 정수 10개를 다음과 같이 변수 10개에 저장해 보겠습니다.

```
let num1 = 1;
let num2 = 2;
let num3 = 3;

...생략...

let num10 = 10;
```

이렇게 변수를 여러 개를 만들면 각 변수들이 독립적으로 존재하기 때문에 의미적으로 연결해서 생각하기가 어렵고, 다른 함수나 변수에 값들을 전달하려면 모든 변수를 전달해야 하기 때문에 번거롭습니다. 따라서 여러 개의 값을 하나로 묶어서 관리하면 편리합니다.

```
let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

이번 챕터에서는 다양한 용도로 값들을 묶어서 표현할 수 있는 데이터 구조와, 이 데이터 구조에서 값을 하나씩 꺼내 사용하는 이터레이터(iterator)에 대해서 알아보겠습니다.

한 눈에 보기

파이썬	러스트
list	Vec
np.array	array
tuple	()
Enum	Enum
dict	std::collections::HashMap
str	String, &str

참고로 이 외에도 다양한 데이터 구조가 러스트에 포함되어 있습니다.

- Sequences: `VecDeque`, `LinkedList`
- Maps: `BTreeMap`
- Sets: `HashSet`, `BTreeSet`
- Misc: `BinaryHeap`

벡터

벡터는 러스트에서 가장 널리 사용되는 자료형 중 하나로, 여러 개의 값을 하나로 묶어서 사용할 수 있습니다. 벡터의 특징은 길이를 런타임에 동적으로 변경 가능하다는 점입니다. 이러한 특징 때문에 런타임에서는 값이 힙 영역에 저장됩니다.

벡터 선언

- `Vec` 구조체의 `from` 메소드를 사용해 배열로부터 벡터를 만드는 방법입니다.
- 두 번째는 `vec!` 매크로를 사용해 벡터를 만드는 방법입니다.

값을 직접 입력해 벡터를 만드는 경우, 매크로를 사용하는 방법이 좀더 간결합니다.

```
fn main() {
    let vec1 = Vec::from([1, 2, 3]);
    let vec2 = vec![1, 2, 3];
}
```

비어 있는 벡터를 선언하는 경우는 원소로부터 타입을 추론할 수 없기 때문에 반드시 타입을 명시해야 합니다.

```
fn main() {
    let vec3: Vec<i32> = Vec::new();
    let vec4: Vec<i32> = vec![];
}
```

벡터 원소 접근하기

벡터의 원소는 인덱스(index)를 사용해 접근할 수 있습니다. 두 번째 원소 `2` 를 인덱스로 접근해 변수 `num` 에 할당하고, 출력하는 예제를 만들어 보겠습니다. 먼저 파이썬 코드는 다음과 같습니다.

```
vec1 = [1, 2, 3]
num = vec1[1]

print(num)
```

동일한 내용의 러스트 코드는 다음과 같습니다.

```
fn main() {
    let vec1 = vec![1, 2, 3];

    let num = vec1[1];

    println!("{}", num);
}
```

벡터에 값 추가하기

벡터를 선언하고 값을 추가해 보겠습니다. 먼저 파이썬에서 벡터와 비슷한 리스트로 같은 내용을 구현하면 다음과 같습니다. 리스트의 마지막에 4, 5, 6을 추가합니다.

```
vec1 = [1, 2, 3]
vec1.append(4)
vec1.append(5)
vec1.append(6)

print(vec1)
```

마찬가지로 벡터의 마지막에 값을 추가해 보겠습니다. `push` 메소드를 사용하면 원소를 벡터 마지막에 하나씩 추가할 수 있습니다.

주의해야 하는 점

- 벡터 `vec1` 이 변경되기 때문에 처음에 `vec1` 을 가변 변수로 선언해야 한다는 것입니다.
- 벡터를 프린트할 때는 디버그 모드를 사용하기 위해 서식을 `"{:?}"` 로 사용해야 합니다.

```
fn main() {
    let mut vec1 = vec![1, 2, 3];

    vec1.push(4);
    vec1.push(5);
    vec1.push(6);

    println!("{:?}", vec1);
}
```

벡터에서 값 삭제하기

이번에는 리스트 `[1, 2, 3]` 에서 마지막 원소 3을 제거한 다음, 맨 앞의 원소 1을 제거해 보겠습니다. 파이썬의 `pop` 메소드는 실행 시 원소를 제거하고 제거된 값을 리턴합니다.

```
vec1 = [1, 2, 3]
num1 = vec1.pop()
num2 = vec1.pop(0)

print(num1, num2, vec1)
```

러스트는 `pop` 메소드에 인덱스를 넣을 수 없고, 무조건 마지막 원소가 제거됩니다. 마지막 원소가 아닌 다른 원소를 제거하려면 `remove` 메소드에 인덱스를 넣어야 합니다. 러스트의 `pop` 과 `remove` 모두 원소를 제거하고, 제거된 원소를 리턴합니다.

```
fn main() {
    let mut vec1 = vec![1, 2, 3];

    let num1 = vec1.pop().unwrap();
    let num2 = vec1.remove(0);

    println!("{} {} {:?}", num1, num2, vec1);
}
```

```
3 1 [2]
```

데크

참고로 파이썬의 리스트와 러스트의 벡터 모두 맨 앞의 원소를 제거하는 데 시간 복잡도가 $O(n)$ 만큼 소요되기 때문에 맨 앞에서 원소를 자주 제거해야 한다면 데크(deque)를 사용하는 것이 좋습니다. 파이썬은 `collections` 모듈의 `deque` 를 사용합니다.

```
from collections import deque

deq = deque([1, 2, 3])
print(deq.popleft())
```

러스트에서는 `VecDeque`를 사용합니다.

```rust
use std::collections::VecDeque;

fn main() {
    let mut deq = VecDeque::from([1, 2, 3]);
    println!("{}", deq.pop_front().unwrap());
}
```

실행 결과

```
1
```

배열

배열 선언

배열(array)이란, 같은 타입의 값이 모여 있는 길이가 고정된 자료형입니다.

파이썬에서 비슷한 내장 자료형은 없지만, 넘파이(numpy)의 배열(array)가 가장 이와 유사합니다. 넘파이는 내부적으로 C로 구현된 배열을 가지고 있고, 파이썬에서 이 배열의 값을 꺼내서 사용하는 방식으로 동작합니다.

넘파이 배열을 이용해 열두 달을 나타내면 다음과 같습니다.

```
import numpy as np

months = np.array(
    [
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December",
    ]
)
print(months)
```

`full` 함수를 사용하면 배열을 간단하게 한 번에 초기화할 수 있습니다.

```
nums = np.full(5, 3)
print(nums)
```

실행 결과

```
[3 3 3 3 3]
```

- 러스트의 배열의 길이는 처음 선언된 이후 변경할 수 없습니다.
- 메모리가 스택 영역에 저장되기 때문에 빠르게 값에 접근할 수 있습니다.
- 이때 배열의 원소들은 모두 같은 타입이어야 합니다.

```
fn main() {
    let months = [
        "January",
        "February",
        "March",
        ...
        "September",
        "October",
        "November",
        "December",
    ];
    println!("{:?}", months);
}
```

러스트에서도 편리한 배열 초기화를 지원합니다. `[3; 5]` 와 같이 표기하면 숫자 3을 5번 나열하라는 의미입니다.

```
fn main() {
    let nums = [3; 5];
    println!("{:?}", nums);
}
```

실행 결과

```
[3, 3, 3, 3, 3]
```

원소 참조

넘파이 배열의 원소들은 인덱스를 통해 접근이 가능합니다.

```
import numpy as np

nums = np.full(5, 3)
nums[1] = 1
print(nums)
```

실행 결과

```
[3 1 3 3 3]
```

러스트 배열도 동일합니다. 이번에는 배열 원소를 수정해야 하기 때문에 `nums` 배열을 가변 변수로 선언합니다.

```
fn main() {
    let mut nums = [3; 5];
    nums[1] = 1;
    println!("{:?}", nums);
}
```

실행 결과

```
[3, 1, 3, 3, 3]
```

넘파이 배열의 길이보다 큰 값을 참조하려고 하면 에러가 발생합니다.

```
import numpy as np

nums = np.full(5, 3)
print(nums[5])
```

실행 결과

```
Traceback (most recent call last):
  File "/Users/code/temp/python/main.py", line 4, in <module>
    print(nums[5])
IndexError: index 5 is out of bounds for axis 0 with size 5
```

러스트 코드는 컴파일 시 인덱스가 범위를 벗어난다는 에러가 발생합니다.

```rust
fn main() {
    let nums = [3; 5];
    println!("{}", nums[5]);
}
```

실행 결과

```
Compiling rust_part v0.1.0 (/Users/code/temp/rust_part)
error: this operation will panic at runtime
 --> src/main.rs:3:20
  |
3 |     println!("{}", nums[5]);
  |                    ^^^^^^^ index out of bounds: the length is 5 but the index is 5
  |
  = note: `#[deny(unconditional_panic)]` on by default

error: could not compile `rust_part` due to previous error
```

하지만 이렇게 미리 참조할 배열 인덱스를 컴파일러가 알 수 없는 경우, 런타임에 에러가 발생할 수 있기 때문에 주의해야 합니다.

```
fn main() {
    let nums = [3; 5];
    for i in 0..nums.len() + 1 {
        println!("{}", nums[i]);
    }
}
```

실행 결과

```
3
3
3
3
3
thread 'main' panicked at
'index out of bounds: the len is 5 but the index is 5', src/main.rs:4:24
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

배열은 벡터와 자주 비교되는데, 데이터의 길이가 컴파일 타임에 정해지는 경우에는 배열을, 데이터의 길이가 런타임에 정해지는 경우에는 벡터를 사용합니다.

튜플

튜플은 프로그래밍에서 가장 대표적인 열거형 자료형으로, 값들을 순서대로 나열해 저장하는 데이터 구조입니다. 파이썬과 러스트 모두 튜플 자료형을 가지고 있습니다.

튜플 선언

파이썬의 튜플은 소괄호 안에 콤마로 구분된 값을 넣어서 선언합니다.

```
tup1 = (0, 0.1, "hello")
tup2 = (1, 1.01, "bye")

_, y, _ = tup2

print(f"tup1 is {tup1} and the value of y is {y}")
```

실행 결과

```
tup1 is (0, 0.1, 'hello') and the value of y is 1.01
```

러스트의 튜플도 소괄호 안에 콤마로 구분된 값을 넣어서 선언합니다.

- 튜플의 타입도 컴파일러가 추론하기 때문에 타입을 명시할 필요가 없습니다.
- 하지만 타입을 직접 명시해도 상관없습니다.

```
fn main() {
    let tup1 = (0, 0.1, "hello");
    let tup2: (i32, f64, &str) = (1, 1.01, "bye");

    let (_, y, _) = tup2;

    println!("tup1 is {:?} and the value of y is: {}", tup1, y);
}
```

실행 결과

```
tup1 is (0, 0.1, "hello") and the value of y is: 1.01
```

원소 참조

파이썬에서 튜플 원소를 참조하려면 인덱스를 넣으면 됩니다.

```
tup1 = (0, 0.1, ("hello", "world"))

print(tup1[2][0], tup1[2][1])
```

실행 결과

```
hello world
```

러스트에서 튜플 원소의 참조는 약간 특이한 방식으로 합니다. 튜플 이름 뒤에 점(`.`)을 붙이고 그 뒤에 인덱스를 입력합니다. 만일 다중 튜플인 경우, 점을 한번 더 찍고 인덱스를 입력하면 됩니다.

```
fn main() {
    let tup1 = (0, 0.1, ("hello", "world"));

    println!("{} {}", tup1.2 .0, tup1.2 .1);
}
```

실행 결과

```
hello world
```

튜플 불변성

파이썬에서의 튜플과 러스트의 튜플은 차이점이 있는데 바로 불변성입니다. 파이썬의 튜플은 한 번 선언되면 원소의 내용을 바꾸거나, 튜플의 크기를 변경할 수 없습니다.

```
tup1 = (0, 0.1, "hello")

x = tup1[0]
_, y, _ = tup1

x = 1
y = 1.1

print(tup1, x, y)

tup1[0] = 3
```

실행 결과

```
(0, 0.1, 'hello') 1 1.1
Traceback (most recent call last):
  File "main.py", line 11, in <module>
    tup1[0] = 3
TypeError: 'tuple' object does not support item assignment
```

마찬가지로 러스트의 튜플도 한 번 선언되면 크기를 변경할 수 없지만, 원소의 내용은 바꿀 수 있습니다. 다만 처음 선언한 타입은 그대로 유지되어야 합니다.

```
fn main() {
    let mut tup1 = (0, 0.1, "hello");

    let mut x = tup1.0;
    let (_, mut y, _) = tup1;

    x = 1;
    y = 1.1;

    println!("{:?} {} {}", tup1, x, y);

    tup1.0 = 3;
}
```

실행 결과

```
(0, 0.1, "hello") 1 1.1
```

열거형

열거형은 여러 상수들의 집합으로 새로운 타입을 선언하는 방법입니다. 파이썬에서는 `Enum` 클래스를 상속해 열거형을 만들 수 있습니다.

```python
from enum import Enum

class Languages(Enum):
    PYTHON = "python"
    RUST = "rust"
    JAVASCRIPT = "javascript"
    GO = "go"

    def echo(self):
        print(self.name)

language = Languages.RUST
language.echo()

if language == Languages.PYTHON:
    print("I love Python")
elif language == Languages.GO:
    print("I love Go")
elif language == Languages.JAVASCRIPT:
    print("I love Javascript")
else:
    print("I love Rust🦀")
```

러스트의 열거형은 `enum` 키워드로 선언이 가능합니다. 이때 값이 없는 열거형과 값이 있는 열거형 두 가지를 만들 수 있는데, 먼저 값이 없는 열거형을 만들어 보면 다음과 같습니다.

> `impl` 블럭을 이용해 열거형에서 사용할 메소드를 만들 수 있습니다. 이에 관련한 자세한 문법은 나중에 객체지향을 배우면서 좀더 자세히 다루겠습니다.

```rust
fn main() {
    // Enum
    #[allow(dead_code)]
    #[derive(Debug)] // derive Debug trait, to print the enum
    enum Languages {
        Python,
        Rust,
        Javascript,
        Go,
    }

    impl Languages {
        fn echo(&self) {
            println!("{:?}", &self);
        }
    }
```

```
    let language = Languages::Rust;
    language.echo();

    // match
    match language {
        Languages::Python => println!("I love Python"),
        Languages::Go => println!("I love Go"),
        Languages::Javascript => println!("I love Javascript"),
        _ => println!("I love Rust🦀"),
    }
}
```

실행 결과

```
Rust
I love Rust🦀
```

열거형에 값을 지정하려면 열거형을 선언하면서 타입을 지정하면 됩니다.

열거형 변수 뒤에 `(타입)` 과 같이 입력하면 됩니다. 이제 열거형 변수를 선언할 때, 해당 타입에 대한 정보를 추가로 입력해줘야 합니다.

```
let indo = Job::Student(Grade::A, "indo".to_string());
```

이제 `indo` 변수의 값에 따라 서로 다른 내용을 출력하도록 `match` 를 사용한 전체 코드는 다음과 같습니다.

```
#[allow(dead_code)]
fn main() {
    #[derive(Debug)] // derive Debug trait, to print the enum
    enum Grade {
        A,
        B,
        C,
    }

    enum Job {
        Student(Grade, String),
        Developer(String),
    }

    let indo = Job::Student(Grade::A, "indo".to_string());
```

```
    match indo {
        Job::Student(grade, name) => {
            println!("{} is a student with grade {:?}", name, grade);
        }
        Job::Developer(name) => {
            println!("{} is a developer", name);
        }
    }
}
```

실행 결과

```
indo is a student with grade A
```

if let

`let if`와 이름은 비슷하지만 전혀 다른 문법인 `if let` 이 있습니다. `if let` 문법은 조건문이 `let <표현식> = 변수` 로 구성되는데, 이때 표현식의 결과가 변수와 같은지를 검사합니다.

```
fn main() {
    let x = 3;

    if let 3 = x {
        println!("x is 3");
    } else if let 4 = x {
        println!("x is 4");
    } else {
        println!("x is not 3 or 4");
    };
}
```

해시맵

해시맵은 키와 밸류를 묶어서 관리하는 자료형으로, 키에 대응하는 밸류를 빠르게 찾을 수 있는 장점이 있습니다. 특히 데이터를 인덱스로 관리하지 않는 경우에 유용합니다.

파이썬에서는 해시맵을 딕셔너리로 구현하고 있습니다.

```
songs = {
    "Toto": "Africa",
    "Post Malone": "Rockstar",
    "twenty one pilots": "Stressed Out",
}
print("----- Playlists -----")
if "Toto" in songs and "Africa" in songs.values():
    print("Toto's africa is the best song!")

songs["a-ha"] = "Take on Me"  # Insert
songs["Post Malone"] = "Happier"  # Update

for artist, title in songs.items():
    print(f"{artist} - {title}")
print("---------------------")

songs.pop("Post Malone")  # Delete
print(songs.get("Post Malone", "Post Malone is not in the playlist"))
```

실행 결과

```
----- Playlists -----
Toto's africa is the best song!
Toto - Africa
Post Malone - Happier
twenty one pilots - Stressed Out
a-ha - Take on Me
---------------------
Post Malone is not in the playlist
```

러스트에서는 해시맵을 `HashMap` 을 이용해 구현이 가능합니다.

> 여기서 마지막에 `unwrap_or(&...)` 는 앞의 코드가 에러를 발생시켰을 때 처리하는 방법으로, 자세한 문법은 에러 처리 챕터에서 다루겠습니다.

```rust
use std::collections::HashMap;

fn main() {
    // Rust's HashMap does not keep the insertion order.
    let mut songs = HashMap::from([
        ("Toto", "Africa"),
        ("Post Malone", "Rockstar"),
        ("twenty one pilots", "Stressed Out"),
    ]);
    println!("----- Playlists -----");
    if songs.contains_key("Toto") &&
        songs.values().any(|&val| val == "Africa") {
        println!("Toto's africa is the best song!");
    }
```

```
    songs.insert("a-ha", "Take on Me"); // Insert
    songs.entry("Post Malone").and_modify(|v| *v = "Happier"); // Update

    for (artist, title) in songs.iter() {
        println!("{} - {}", artist, title);
    }

    println!("----------------------");
    songs.remove("Post Malone"); // Delete
    println!(
        "{:?}",
        songs
            .get("Post Malone")
            .unwrap_or(&"Post Malone is not in the playlist")
    );
}
```

실행 결과

```
----- Playlists -----
Toto's africa is the best song!
Post Malone - Happier
Toto - Africa
twenty one pilots - Stressed Out
a-ha - Take on Me
---------------------
"Post Malone is not in the playlist"
```

문자열

러스트에서는 문자열을 두 가지 방법을 사용해 선언할 수 있습니다.

- 첫 번째는 `String` 타입으로, 일반적인 문자열을 만들 때 사용합니다.
- 두 번째는 `&str` 타입으로, `String` 타입으로 선언된 문자열의 일부분을 의미합니다.

```rust
fn main() {
    let greet = String::from("Hi, buzzi!");
    // let name = "buzzi";
    let name = &greet[4..];
    println!("{}", name);
}
```

실행 결과

```
buzzi!
```

이터레이터

이터레이터란?

이터레이터(iterator)는 반복 가능한 시퀀스(sequence)를 입력으로 받아 각 원소에 특정 작업을 수행할 수 있도록 하는 기능입니다. 앞에서 배운 벡터를 이용해 값을 순서대로 출력하는 예제를 만들어 보겠습니다.

```rust
fn main() {
    let names = vec!["james", "cameron", "indo"];
    for name in names {
        println!("{}", name);
    }
    println!("{:?}", names);
}
```

실행 결과

```
error[E0382]: borrow of moved value: `names`
   --> src/main.rs:6:22
    |
2   |     let names = vec!["james", "cameron", "indo"];
    |         ----- move occurs because `names` has type `Vec<&str>`,
    which does not implement the `Copy` trait
3   |     for name in names {
    |                 -----
    |                 |
    |                 `names` moved due to this implicit call to `.into_iter()`
    |                 help: consider borrowing to avoid moving into
    the for loop: `&names`
...
6   |     println!("{:?}", names);
    |                      ^^^^^ value borrowed here after move
    |
```

- `for name in names` 에서 `names` 가 암묵적으로 `.into_iter()` 메소드를 호출
- `into_iter` 는 벡터 원소의 값과 소유권을 `for` 루프 안으로 가져와 반복합니다.
- 이미 이동된 소유권을 `println!("{:?}", names);` 에서 참조해서 에러가 발생합니다.

이를 해결하기 위해서는 명시적으로 `iter()` 메소드를 호출해 원소를 `for` 루프 안으로 전달해주어야 합니다.

```
fn main() {
    let names = vec!["james", "cameron", "indo"];
    for name in names.iter() {
        println!("{}", name);
    }
    println!("{:?}", names);
}
```

실행 결과

```
james
cameron
indo
["james", "cameron", "indo"]
```

`iter()` 메소드는 선언 즉시 원소를 내놓는 것이 아니라, 값이 필요해지면 그때 원소를 리턴합니다. 따라서 다음과 같은 코드가 가능합니다.

```rust
fn main() {
    let names = vec!["james", "cameron", "indo"];
    let names_iter = names.iter();
    for name in names_iter {
        println!("{}", name);
    }
    println!("{:?}", names);
}
```

실행 결과

```
james
cameron
indo
["james", "cameron", "indo"]
```

정리

- `iter` : 소유권을 가져오지 않고 원소를 반복
- `into_iter` : 소유권을 가져와 원소를 반복

이터레이터를 소비하는 메소드들

이번 단원에서는 이터레이터에 속한 메소드들을 이용해 원소에 여러 작업을 수행해 보겠습니다.

파이썬에서는 합계, 최대값, 최소값을 구하는 함수인 `sum` , `max` , `min` 을 리스트에 직접 사용합니다.

```
nums = [1, 2, 3]

sum = sum(nums)
max = max(nums)
min = min(nums)
print(f"sum: {sum}, max: {max}, min: {min}")
```

실행 결과

```
sum: 6, max: 3, min: 1
```

러스트에서는 이터레이터에서 `sum`, `max`, `min` 메소드를 호출합니다.

```
fn main() {
    let num = vec![1, 2, 3];

    let sum: i32 = num.iter().sum();
    let max = num.iter().max().unwrap();
    let min = num.iter().min().unwrap();
    println!("sum: {}, max: {}, min: {}", sum, max, min);
}
```

실행 결과

```
sum: 6, max: 3, min: 1
```

새로운 이터레이터를 만드는 메소드들

이터레이터 메소드 중에는 새로운 이터레이터를 만드는 메소드들이 있습니다.

파이썬에서는 `enumerate` 와 `zip` 입니다.

```
nums1 = [1, 2, 3]
nums2 = [4, 5, 6]

enumer = list(enumerate(nums1))
print(enum)
zip = list(zip(nums1, nums2))
print(zip)
```

실행 결과

```
[(0, 1), (1, 2), (2, 3)]
[(1, 4), (2, 5), (3, 6)]
```

마찬가지로 러스트에서도 원소와 인덱스를 동시에 반복하거나 두 시퀀스의 원소를 동시에 반복할 수 있습니다.

```
fn main() {
    let nums1 = vec![1, 2, 3];
    let nums2 = vec![4, 5, 6];

    let enumer: Vec<(usize, &i32)> = nums1.iter().enumerate().collect();
    println!("{:?}", enumer);

    let zip: Vec<(&i32, &i32)> = nums1.iter().zip(nums2.iter()).collect();
    println!("{:?}", zip);
}
```

실행 결과

```
[(0, 1), (1, 2), (2, 3)]
[(1, 4), (2, 5), (3, 6)]
```

가장 중요한 이터레이터 `map` 과 `filter` 입니다.

> 파이썬에서는 성능상 이유로 리스트 컴프리헨션이 더 선호됩니다.

```
nums = [1, 2, 3]

f = lambda x: x + 1

print(list(map(f, nums)))
print(list(filter(lambda x: x % 2 == 1, nums)))
```

실행 결과

```
[2, 3, 4]
[1, 3]
```

러스트 코드에서는 클로저를 이용해 동일한 내용을 구현했습니다. 이때 `filter` 의 경우, 기존의 원소의 값을 이동해서 새로운 벡터를 만들기 때문에 `into_iter` 메소드로 이터레이터를 만들었습니다.

```
fn main() {
    let nums: Vec<i32> = vec![1, 2, 3];

    let f = |x: &i32| x + 1;

    let maps: Vec<i32> = nums.iter().map(f).collect();
    println!("{:?}", maps);

    let filters: Vec<i32> = nums.into_iter().filter(|x| x % 2 == 1).collect();
    println!("{:?}", filters);
}
```

실행 결과

```
[2, 3, 4]
[1, 3]
```

원본 벡터를 필터 이후에도 사용하기 위해서는 두 가지 방법이 있습니다.

원본 벡터를 복사(clone)하는 방법

```
fn main() {
    let nums: Vec<i32> = vec![1, 2, 3];

    let f = |x: &i32| x + 1;

    let maps: Vec<i32> = nums.iter().map(f).collect();
    println!("{:?}", maps);

    let filters: Vec<i32> = nums.clone().into_iter().filter(|x| x % 2 == 1).collect();
    println!("{:?}", filters);

    println!("{:?}", nums);
}
```

이터레이터를 복사하는 방법

```rust
fn main() {
    let nums: Vec<i32> = vec![1, 2, 3];

    let f = |x: &i32| x + 1;

    let maps: Vec<i32> = nums.iter().map(f).collect();
    println!("{:?}", maps);

    let filters: Vec<i32> = nums.iter().filter(|x| *x % 2 == 1).cloned().collect();
    println!("{:?}", filters);

    println!("{:?}", nums);
}
```

Quiz

1. 입력 받은 벡터의 모든 원소를 2배로 만들어서 새로운 벡터를 만들어보세요.

```
fn main() {
    let nums: Vec<i32> = vec![1, 2, 3];

    let maps: Vec<i32> = ?
    println!("{:?}", maps);
}
```

정답

```
fn main() {
    let nums: Vec<i32> = vec![1, 2, 3];

    let maps: Vec<i32> = nums.iter().map(|x| x * 2).collect();
    println!("{:?}", maps);
}
```

2. 입력 받은 벡터의 모든 원소 중 짝수만 골라서 새로운 벡터를 만들어 보세요.

```
fn main() {
    let nums: Vec<i32> = vec![1, 2, 3];

    let filters: Vec<i32> = ?
    println!("{:?}", filters);
}
```

정답

```
fn main() {
    let nums: Vec<i32> = vec![1, 2, 3];

    let filters: Vec<i32> = nums.into_iter().filter(|x| x % 2 == 0).collect();
    println!("{:?}", filters);
}
```