세상에서 제일 쉬운 러스트 프로그래밍

6장. 자료구조와 이터레이터

윤인도

freedomzero91@gmail.com

예를 들어, 정수 10개를 다음과 같이 변수 10개에 저장할 수 있습니다.

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

...생략...

let num10 = 10;
```

하지만 여러 개의 변수를 관리하기가 번거로움! 따라서 값들을 묶어서 표현할 수 있는 자료구조를 사용

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

한 눈에 보기

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

참고로 이 외에도 다양한 자료구조가 러스트에 포함되어 있습니다.

https://doc.rust-lang.org/std/collections

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

벡터

힙 영역에 저장되는 가변 길이 배열 == 파이썬의 `list`

벡터 선언

- `Vec` 구조체의 `from` 메서드를 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.

파이썬

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

print(num)
```

러스트

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

    let num = vec1[1];

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

벡터에 값 추가하기

`push` 메서드를 사용해 원소를 벡터 마지막에 하나씩 추가

파이썬

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

print(vec1)
```

러스트

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

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

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

주의해야 하는 점

- 벡터 `vec1` 이 변경되기 때문에 처음에 `vec1`을 가변 변수로 선언
- 벡터를 프린트할 때는 디버그 모드를 사용하기 위해 서식을 `"{:?}"`로 사용

벡터에서 값 삭제하기

러스트 `pop` 은 항상 마지막 원소를 삭제

파이썬

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

print(num1, num2, vec1)
```

러스트

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

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

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

실행 결과

```
3 1 [2]
```

피보나치 수열 - 파이썬과 비슷하게 내부 함수로 구현하면 가능합니다.

```rust
fn fib(n: u32) -> u32 {
    fn _fib(n: u32, cache: &mut Vec<u32>) -> u32 {
        if n < cache.len() as u32 {
            cache[n as usize]
        } else {
            let result = _fib(n - 1, cache) + _fib(n - 2, cache);
            cache.push(result);
            result
        }
    }

    let mut cache = vec![0, 1];
    _fib(n, &mut cache)
}

fn main() {
    println!("{}", fib(10));
}
```

데크

- 파이썬의 리스트와 러스트의 벡터 모두 맨 앞의 원소를 제거하는 데 시간 복잡도가 $O(n)$ 만큼 소요
- 맨 앞에서 원소를 자주 제거해야 한다면 데크(deque)를 사용

```python
from collections import deque

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

```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 타입"의 값으로 구성된 "고정 길이" 자료형
- 파이썬에서 비슷한 내장 자료형은 없지만, 넘파이(numpy)의 배열(array)가 가장 비슷함
- 넘파이는 내부적으로 C로 구현된 배열을 가지고 있고, 파이썬에서 이 배열의 값을 꺼내서 사용하는 방식으로 동작

넘파이 배열을 이용해 열두 달을 나타내면 다음과 같습니다.

```python
import numpy as np

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

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

파이썬

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

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

러스트

`[3; 5]` 와 같이 값을 몇 번 반복할지 정의

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

실행 결과

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

러스트 배열 선언

- 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.

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 배열
- 데이터의 길이가 런타임에 정해지는 경우에는 벡터를 사용합니다.

튜플

튜플은 프로그래밍에서 가장 대표적인 열거형 자료형으로, 값들을 순서대로 나열해 저장하는 자료구조입니다. 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 명시할 수도 있습니다.

```
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

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
```

- 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"); // 튜플을 가변으로 선언
    tup1.0 = 3; // 튜플의 값 변경
    println!("{:?}", tup1);
}
```

실행 결과

```
(3, 0.1, "hello")
```

해시맵

- 해시맵은 키와 밸류를 묶어서 관리하는 자료형
- 키에 대응하는 밸류를 빠르게 찾을 수 있음
- 데이터를 인덱스로 관리하지 않는 경우에 유용

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.

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

fn main() {
    // 튜플의 배열로부터 해시맵 만들기
    let mut songs = HashMap::from([
        ("Toto", "Africa"),
        ("Post Malone", "Rockstar"),
        ("twenty one pilots", "Stressed Out"),
    ]);
    println!("----- Playlists -----");

    // 해시맵에 키와 밸류가 모두 있는지 확인
    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"); // 값 삽입
    songs.entry("Post Malone").and_modify(|v| *v = "Happier"); // 업데이트

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

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

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

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.

- 코드가 컴파일될 때 **스택 영역**에 만들어지는 `&str`
- 런타임에 **힙 영역**에 메모리가 할당되는 `String`

- `&str` 타입은 "문자열 리터럴"
- 한 번 만들어지면 값을 변경하거나 길이를 바꿀 수 없습니다.

```
fn main() {
    let s: &str = "hello";
    println!("{}", s);
}
```

> 스택 영역에 만들어지는 값을 참조하고 있어서 `str` 이 아닌 `&str` 로 나타냅니다!

문자열 또는 스트링이라고 불리는 `String` 타입은 여러 방법으로 선언이 가능

```rust
fn main() {
    // 비어 있는 스트링 만들기
    let mut s = String::new();

    // 스트링 리터럴로부터 스트링 만들기
    let data = "initial contents";
    let s = data.to_string();
    let s = "initial contents".to_string();

    // String::from()을 사용하여 스트링 만들기
    let s = String::from("initial contents");
}
```

정리하자면...

- 문자열 데이터의 소유권을 다뤄야 할 때 ➡️ `String`을 사용
- 문자열의 값만 필요할 때 ➡️ `&str`을 사용

문자열 슬라이스

`&str` 은 문자열의 일부분인 문자열 슬라이스라고도 합니다. `String` 타입 문자열로부터 문자열 슬라이스를 생성

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

실행 결과

```
buzzi!
```

러스트의 모든 문자열은 UTF-8로 인코딩되어 있다는 점을 주의!

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

실행 결과

```
thread 'main' panicked at 'byte index 4 is not a char boundary;
it is inside '😃' (bytes 2..6) of `Hi😃 buzzi!`', src/main.rs:4:17
```

일반적인 알파벳 문자는 바이트 스트림에서 1바이트를 차지하지만,

유니코드로 만들어진 이모지 의 경우는 4바이트를 차지함!

바이트 4에 해당하는 인덱스가 이모지 중간에 위치하므로 정상적으로 문자열을 잘라낼 수 없다!

문자열을 벡터로 만들어줘야 합니다.

```
fn main() {
    let greet = String::from("Hi😃 buzzi!");
    let greet_chars: Vec<char> = greet.chars().collect();
    let name = &greet_chars[4..].iter().collect::<String>();
    println!("{:?}", name);
}
```

열거형

열거형은 여러 상수들의 집합으로 새로운 타입을 선언하는 방법입니다.

파이썬에서는 `Enum` 클래스를 상속해 열거형을 만들 수 있습니다.

```
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()
```

실행 결과

```
RUST
```

각 변수를 직접 지정해줄 수도 있습니다.

```
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🦀")
```

실행 결과

```
I love Rust🦀
```

러스트의 열거형은 `enum` 키워드로 선언이 가능

값이 없는 열거형

```
#[allow(dead_code)]
#[derive(Debug)] // 콘솔 출력을 위해 Debug 트레이트 추가
enum Languages {
    Python,
    Rust,
    Javascript,
    Go,
}
```

> `#[...]` 은 애트리뷰트(attribute)라고 부르며 트레이트를 쉽게 추가할 수 있는 기능 (7장)

`impl` 블럭을 이용해 열거형에서 사용할 메서드를 생성

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

```rust
fn main() {
    let language = Languages::Rust;
    language.echo();
}
```

실행 결과

```
Rust
```

```rust
fn main() {
    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

```
I love Rust🦀
```

값이 있는 열거형(`Job`) → 선언 시 타입 정의

```
fn main() {
    #[derive(Debug)]
    enum Grade {
        A,
        B,
        C,
    }

    enum Job {
        Student(Grade, String), // 각 변수의 타입을 지정
        Developer(String),
    }
}
```

`match` 를 사용한 전체 코드

```
fn main() {
    // 객체를 만들 때도 적합한 타입을 전달
    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
```

Option 열거형

`Option<T>` : `T` 타입의 값이 있을 수도 있고 없을 수도 있다

- `Some(T)` : 값이 있다
- `None` : 값이 없다

```
enum Option<T> {
    Some(T),
    None,
}
```

- `Option`을 사용하려면, 열거형 변수 중 하나인 `Some`을 사용해 값을 감싸주어야 함
- 값이 없음을 나타내려면 `None`을 사용

```
fn main() {
    let some_number = Some(5);
    let some_string = Some("a string");
    let absent_number: Option<i32> = None;
    println!("{:?} {:?} {:?}", some_number, some_string, absent_number);
}
```

실행 결과

```
Some(5) Some("a string") None
```

match를 사용한 패턴 매칭

`Option`은 주로 `match`와 함께 사용

```
fn check_len(vec: Vec<i32>) -> Option<usize> {
    match vec.len() {
        0 => None,
        _ => Some(vec.len()),
    }
}

fn main() {
    let nums = vec![1, 2, 3];
    match check_len(nums) {
        Some(len) => println!("Length: {}", len),
    }
}
```

```
error[E0004]: non-exhaustive patterns: `None` not covered
   --> src/main.rs:11:11
    |
11  |     match check_len(nums) {
    |           ^^^^^^^^^^^^^^^ pattern `None` not covered
    |
```

👍 `Option`과 `match`를 함께 사용하면

- 값이 들어있는 경우
- 들어있지 않은 경우

두 가지를 반드시 체크해야 함!

두 가지 경우를 올바르게 처리한 코드

```
fn check_len(vec: Vec<i32>) -> Option<usize> {
    match vec.len() {
        0 => None,
        _ => Some(vec.len()),
    }
}

fn main() {
    let nums = vec![1, 2, 3];
    match check_len(nums) {
        Some(len) => println!("Length: {}", len),
        None => println!("No elements!"),
    }
}
```

실행 결과

```
Length: 3
```

if let 구문

`Option`의 결과에 따라서 특정 행동만 하고 싶다면, `if let` 구문을 사용

```
fn main() {
    let val = Some(3);
    match val {
        Some(3) => println!("three"),
        _ => (),
    }
    if let Some(3) = val {
        println!("three");
    }
}
```

실행 결과

```
three
three
```

`Result<T, E>` 열거형

- `Ok`는 결과값이 정상
- `Err`는 에러 발생

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

match를 사용한 패턴 매칭

파일이 존재하면 파일의 내용을 출력하고, 존재하지 않으면 패닉

```rust
use std::fs::File;
use std::io::Read;

fn main() {
    let mut s = String::new();
    match File::open("hello.txt") {
        Ok(mut file) => {
            file.read_to_string(&mut s).unwrap();
            println!("{}", s);
        }
        Err(error) => panic!("There was a problem opening the file: {:?}", error),
    };
}
```

if let구문

`match` ➡️ `if let`

```
use std::fs::File;
use std::io::Read;

fn main() {
    let mut s = String::new(); // 텍스트를 저장할 변수 생성

    if let Ok(mut file) = File::open("hello.txt") {
        file.read_to_string(&mut s).unwrap();
        println!("{}", s);
    } else {
        panic!("There was a problem opening the file");
    }
}
```

? 연산자

`?` 연산자는 `Result`을 리턴하는 함수에서만 사용 가능한 에러처리 연산자

- `Result` : `Ok` / `Err`

```rust
use std::fs::File;
use std::io::{self, Write};

fn write_info(name: &str) -> io::Result<()> {
    match File::create("my_best_friends.txt") {
        Err(e) => return Err(e), // 파일 생성시 에러가 발생하면 종료
        Ok(mut f) => {
            if let Err(e) = f.write_all(format!("name: {}\n", name).as_bytes()) {
                // 파일 쓰기시 에러가 발생하면 종료
                return Err(e);
            }
        }
    };
    Ok(()) // 파일 생성 및 쓰기가 성공하면 Ok(()) 리턴
}

fn main() {
    if let Ok(_) = write_info("John") {
        println!("Writing to file succeeded!");
    }
}
```

- 분기가 너무 많아서 가독성이 떨어지는 코드를 `?` 를 사용해 수정
- `Err(e) => return Err(e)` 같은 코드는 사실상 별 의미가 없음

```rust
use std::fs::File;
use std::io::{self, Write};

fn write_info(name: &str) -> io::Result<()> {
    let mut file = File::create("my_best_friends.txt")?;
    file.write_all(format!("name: {}\n", name).as_bytes())?;
    Ok(())
}

fn main() {
    if let Ok(_) = write_info("John") {
        println!("Writing to file succeeded!");
    }
}
```

`?` 연산자 사용 시 주의사항

- `?` 연산자를 사용하는 함수는 반드시 `Result`를 리턴해야 함
- 함수의 리턴 타입은 `?` 연산자를 사용하는 `Result`의 에러 타입과 일치해야 함

  예) 에러 종류를 `io::Error`로 명시한 경우

```rust
fn read_file_to_string(filename: &str) -> Result<String, io::Error> {
    let mut file = File::open(filename)?;
    let mut s = String::new();
    file.read_to_string(&mut s)?;
    Ok(s)
}
```

- 함수에서 리턴되는 에러가 다양하거나, 에러 종류별로 특별한 처리를 해줘야 할 때는 부적합

> 에러 종류가 다양한 경우 별도의 열거형을 정의하는 것도 가능

이터레이터

이터레이터란?

이터레이터(iterator)는 반복 가능한 시퀀스(sequence)를 입력으로 받아 각 원소에 특정 작업을 수행할 수 있도록 하는 기능

벡터를 이용해 값을 순서대로 출력하는 예제

```
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()` 메서드를 호출
- `into_iter` 는 벡터 원소의 값과 소유권을 `for` 루프 안으로 가져와 반복
- 이미 이동된 소유권을 `println!("{:?}", names);` 에서 참조해서 에러가 발생

이를 해결하기 위해서는 명시적으로 `iter()` 메서드를 호출해 원소를 `for` 루프 안으로 전달

```
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()` 메서드는 선언 즉시 값을 만들지 않고, 값이 필요해지면 그때 원소를 생성

```
fn main() {
    let names = vec!["james", "cameron", "indo"];
    let names_iter = names.iter(); // 이터레이터를 변수로 저장
    for name in names_iter { // 나중에 사용
        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 메서드

파이썬에서는 합계, 최대값, 최소값을 구하는 함수인 `sum`, `max`, `min`을 리스트에 직접 사용 가능

```
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` 메서드를 호출합니다.

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

    let sum: i32 = num.iter().sum();
    // min/max는 Option을 리턴
    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
```

`Option`에서 `None`이 리턴되면 `unwrap`은 패닉

새로운 이터레이터를 만드는 메서드들

이터레이터 메서드 중에는 새로운 이터레이터를 만드는 메서드들이 있습니다.

파이썬에서는 `enumerate` 와 `zip`

```
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 있음
- 이때 `collect` 를 사용해 이터레이터 결과를 즉시 벡터로 변환

```rust
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`

> 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]
```

- 러스트에서는 클로저를 이용해 동일한 내용을 구현
- `filter`는 소유권 이동이 필요해서 `into_iter` 메서드로 이터레이터를 생성

```
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 사용하는 방법?

1. 원본 벡터를 복사(`clone`)하는 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.clone(). // 복사
        into_iter().filter(|x| x % 2 == 1).collect();
    println!("{:?}", filters);

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

2. 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();  // 복사
    println!("{:?}", filters);

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

연습문제

1. 다음 코드를 완성해서 주어진 배열 `fruits` 로부터 각 과일의 갯수를 세어서 `HashMap` 에 저장해보세요.

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

fn main() {
    let fruits = vec!["apple", "banana", "apple", "banana", "orange", "pear", "orange"];

    ...

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

실행 결과

```
{"apple": 2, "banana": 2, "orange": 2, "pear": 1}
```

정답

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

fn main() {
    let fruits = vec!["apple", "banana", "apple", "banana", "orange", "pear", "orange"];

    let mut counts: HashMap<&str, i32> = HashMap::new();

    for fruit in fruits {
        let count = counts.entry(fruit).or_insert(0);
        *count += 1;
    }

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

2. match를 사용한 다음 코드가 컴파일되도록 수정해보세요.

```
fn divide(numerator: f64, denominator: f64) -> Result<f64, &'static str> {
    if denominator == 0.0 {
        Err("Cannot divide by zero")
    } else {
        Ok(numerator / denominator)
    }
}

fn main() {
    let result = divide(2 as f64, 3 as f64);

    match result {
        Ok(x) => println!("Result: {}", x),
    }
}
```

정답

```rust
fn divide(numerator: f64, denominator: f64) -> Result<f64, &'static str> {
    if denominator == 0.0 {
        Err("Cannot divide by zero")
    } else {
        Ok(numerator / denominator)
    }
}

fn main() {
    let result = divide(2 as f64, 3 as f64);

    match result {
        Ok(x) => println!("Result: {}", x),
        Err(e) => println!("Error: {}", e),
    }
}
```

3. 입력 받은 실수의 제곱근을 구하는 함수 `get_square_root` 를 만들어보세요.

```
fn get_square_root(number: f64) -> Option<f64> {
    ...
}

fn main() {
  let result = get_square_root(9.0);
  match result {
    Some(value) => println!("9^0.5 = {}", value),
    None => println!("음수는 입력할 수 없습니다!"),
  }
}
```

정답

```
fn get_square_root(number: f64) -> Option<f64> {
  if number >= 0.0 {
    Some(number.sqrt())
  } else {
    None
  }
}
```

4. `iter`를 사용해서 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

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

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

5. `filter`를 사용해서 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

```rust
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);
}
```