DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 22

1. COLLECTIONS

2. VECTORS

WEBPAGE UPDATES

URL: https://onak.pl/ds210

- Related reading materials (added or to be added)
- More useful links at the bottom

WEBPAGE UPDATES

URL: https://onak.pl/ds210

- Related reading materials (added or to be added)
- More useful links at the bottom

FINAL PROJECT PROPOSAL

- see the pinned Piazza post
- two additional general direction have been added
- due this Friday

WEBPAGE UPDATES

URL: https://onak.pl/ds210

- Related reading materials (added or to be added)
- More useful links at the bottom

FINAL PROJECT PROPOSAL

- see the pinned Piazza post
- two additional general direction have been added
- due this Friday

HOMEWORKS

- emailing comments on the first Rust homework
- use available resources: discussion sections, office hours

1. COLLECTIONS

2. VECTORS

COLLECTIONS

Examples: lists/vectors, hash tables, stack, queue, balanced binary search trees

COLLECTIONS

Examples: lists/vectors, hash tables, stack, queue, balanced binary search trees

WHY USEFUL

- Storing multiple items, number unknown in advance
- Also the primary reason why generics exist: collections that work for different types of items

COLLECTIONS

Examples: lists/vectors, hash tables, stack, queue, balanced binary search trees

WHY USEFUL

- Storing multiple items, number unknown in advance
- Also the primary reason why generics exist: collections that work for different types of items

BADLY KEPT SECRET

- Most tasks: little memory management needed
- Collections will do all the work for you
- Caveat:
  - don't copy large amount of memory
  - use references

COLLECTIONS

Examples: lists/vectors, hash tables, stack, queue, balanced binary search trees

WHY USEFUL

- Storing multiple items, number unknown in advance
- Also the primary reason why generics exist: collections that work for different types of items

BADLY KEPT SECRET

- Most tasks: little memory management needed
- Collections will do all the work for you
- Caveat:
  - don't copy large amount of memory
  - use references

COLLECTION SELECTION

- Driven by efficiency and access needs

1. COLLECTIONS

2. VECTORS

VECTORS

Extandable and shrinkable array of items:

- Python: list
- C++: vector
- Rust: vector

Type: `Vec<T>` stores a collection of values of type `T`

CREATING VECTORS VIA MACRO `vec![...]`

In [2]:
```
// useful macro: vec![...]
// syntax similar to array

let small_primes = vec![2,3,5,7,11];
small_primes
```

Out[2]: [2, 3, 5, 7, 11]

CREATING VECTORS VIA MACRO `vec![...]`

In [2]:
```
// useful macro: vec![...]
// syntax similar to array

let small_primes = vec![2,3,5,7,11];
small_primes
```

Out[2]: [2, 3, 5, 7, 11]

In [3]:
```
// specifc length filled with a given value
let zeros = vec![0;10];
zeros
```

Out[3]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

CREATING VECTORS VIA MACRO `vec![...]`

In [2]:
```
// useful macro: vec![...]
// syntax similar to array

let small_primes = vec![2,3,5,7,11];
small_primes
```

Out[2]: [2, 3, 5, 7, 11]

In [3]:
```
// specifc length filled with a given value
let zeros = vec![0;10];
zeros
```

Out[3]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [4]:
```
// size doesn't have to be known in advance
fn get_ones(how_many:usize) -> Vec<i32> {
    vec![1;how_many]
}

let ones = get_ones(13);
ones
```

Out[4]: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

CREATING EMPTY VIA THE `new` CONSTRUCTOR

In [5]:
```
// creating a new empty vectors
// with explicit type specification
let v1 : Vec<f64> = Vec::new();
let mut v2 = Vec::<bool>::new();
```

CREATING EMPTY VIA THE `new` CONSTRUCTOR

In [5]:
```
// creating a new empty vectors
// with explicit type specification
let v1 : Vec<f64> = Vec::new();
let mut v2 = Vec::<bool>::new();
```

In [6]:
```
// won't work: no type specified
let v3 = Vec::new();
```

```
let v3 = Vec::new();
    ^^ consider giving `v3` the explicit type `Vec<T>`,
where the type parameter `T` is specified
let v3 = Vec::new();
         ^^^^^^^^ cannot infer type for type parameter
`T`
type annotations needed for `Vec<T>`
```

CREATING EMPTY VIA THE `new` CONSTRUCTOR

In [5]:
```
// creating a new empty vectors
// with explicit type specification
let v1 : Vec<f64> = Vec::new();
let mut v2 = Vec::<bool>::new();
```

In [6]:
```
// won't work: no type specified
let v3 = Vec::new();
```

```
let v3 = Vec::new();
    ^^ consider giving `v3` the explicit type `Vec<T>`,
where the type parameter `T` is specified
let v3 = Vec::new();
         ^^^^^^^^ cannot infer type for type parameter
`T`
type annotations needed for `Vec<T>`
```

In [7]:
```
// here Rust can infer what the type is
let mut v4 = Vec::new();
v4.push(123); // <= add element at the end
```

BASIC OPERATIONS

In [8]:
```
// adding elements at the end
println!("{:?} (length={})", v2, v2.len());
v2.push(true);
v2.push(false);
println!("{:?} (length={})", v2, v2.len());
```

```
[] (length=0)
[true, false] (length=2)
```

BASIC OPERATIONS

In [8]:
```
// adding elements at the end
println!("{:?} (length={})", v2, v2.len());
v2.push(true);
v2.push(false);
println!("{:?} (length={})", v2, v2.len());
```

```
[] (length=0)
[true, false] (length=2)
```

In [9]:
```
// accessing specific element
v2[0] = v2[1];
println!("{:?}",v2);

// works because of bool values are copied
// by default
```

```
[false, false]
```

BASIC OPERATIONS

In [8]:
```
// adding elements at the end
println!("{:?} (length={})", v2, v2.len());
v2.push(true);
v2.push(false);
println!("{:?} (length={})", v2, v2.len());
```

```
[] (length=0)
[true, false] (length=2)
```

In [9]:
```
// accessing specific element
v2[0] = v2[1];
println!("{:?}",v2);

// works because of bool values are copied
// by default
```

```
[false, false]
```

In [10]:
```
// won't work
struct Seconds(i64);
let mut v = Vec::new();
v.push(Seconds(123));
v.push(Seconds(321));
let z = v[0];
```

```
let z = v[0];
        ^^^^ move occurs because value has type `Second
s`, which does not implement the `Copy` trait
cannot move out of index of `Vec<Seconds>`
help: consider borrowing here

&v[0]
```

BASIC OPERATIONS

In [8]:
```
// adding elements at the end
println!("{:?} (length={})", v2, v2.len());
v2.push(true);
v2.push(false);
println!("{:?} (length={})", v2, v2.len());
```

```
[] (length=0)
[true, false] (length=2)
```

In [9]:
```
// accessing specific element
v2[0] = v2[1];
println!("{:?}",v2);

// works because of bool values are copied
// by default
```

```
[false, false]
```

In [10]:
```
// won't work
struct Seconds(i64);
let mut v = Vec::new();
v.push(Seconds(123));
v.push(Seconds(321));
let z = v[0];
```

```
let z = v[0];
        ^^^^ move occurs because value has type `Second
s`, which does not implement the `Copy` trait
cannot move out of index of `Vec<Seconds>`
help: consider borrowing here

&v[0]
```

In [14]:
```
// references do work
struct Minutes(i64);
{
    let mut v = Vec::new();
    v.push(Minutes(123));
    v.push(Minutes(321));
    let z1 : &Minutes = &v[1];
    let z2 : &mut Minutes = &mut v[0];
};
```

BASIC OPERATIONS

In [8]:
```
// adding elements at the end
println!("{:?} (length={})", v2, v2.len());
v2.push(true);
v2.push(false);
println!("{:?} (length={})", v2, v2.len());
```

```
[] (length=0)
[true, false] (length=2)
```

In [9]:
```
// accessing specific element
v2[0] = v2[1];
println!("{:?}",v2);

// works because of bool values are copied
// by default
```

```
[false, false]
```

In [10]:
```
// won't work
struct Seconds(i64);
let mut v = Vec::new();
v.push(Seconds(123));
v.push(Seconds(321));
let z = v[0];
```

```
let z = v[0];
        ^^^^ move occurs because value has type `Second
s`, which does not implement the `Copy` trait
cannot move out of index of `Vec<Seconds>`
help: consider borrowing here

&v[0]
```

In [14]:
```
// references do work
struct Minutes(i64);
{
    let mut v = Vec::new();
    v.push(Minutes(123));
    v.push(Minutes(321));
    let z1 : &Minutes = &v[1];
    let z2 : &mut Minutes = &mut v[1];
};
```

In [16]:
```
// reaching out of bounds (panics in debug mode, continues in release mode)
let v = vec![1,2,3,4,5];
v[100]
```

```
thread '<unnamed>' panicked at 'index out of bounds: the len is 5 but the index is 100', src/lib.rs:129:40
stack backtrace:
   0: rust_begin_unwind
             at /rustc/9d1b2106e23b1abd32fce1f17267604a5102f57a/library/std/src/panicking.rs:498:5
   1: core::panicking::panic_fmt
```

BASIC OPERATIONS

In [17]:
```
// With bound checking
// .get(index) returns Option<&T>
let mut v = vec![0,1,2];
v.get(2)
```

Out[17]: Some(2)

In [18]:
```
v.get(10)
```

Out[18]: None

BASIC OPERATIONS

In [17]:
```
// With bound checking
// .get(index) returns Option<&T>
let mut v = vec![0,1,2];
v.get(2)
```

Out[17]: Some(2)

In [18]:
```
v.get(10)
```

Out[18]: None

In [19]:
```
// removing last element
// returns `Option<T>)
let last = v.pop();
last
```

Out[19]: Some(2)

BASIC OPERATIONS

In [17]:
```
// With bound checking
// .get(index) returns Option<&T>
let mut v = vec![0,1,2];
v.get(2)
```

Out[17]: Some(2)

In [18]:
```
v.get(10)
```

Out[18]: None

In [20]:
```
// removing last element
// returns `Option<T>)
let last = v.pop();
last
```

Out[20]: Some(1)

BASIC OPERATIONS

In [17]:
```
// With bound checking
// .get(index) returns Option<&T>
let mut v = vec![0,1,2];
v.get(2)
```

Out[17]: Some(2)

In [18]:
```
v.get(10)
```

Out[18]: None

In [21]:
```
// removing last element
// returns `Option<T>)
let last = v.pop();
last
```

Out[21]: Some(0)

BASIC OPERATIONS

In [17]:
```
// With bound checking
// .get(index) returns Option<&T>
let mut v = vec![0,1,2];
v.get(2)
```

Out[17]: Some(2)

In [18]:
```
v.get(10)
```

Out[18]: None

In [22]:
```
// removing last element
// returns `Option<T>)
let last = v.pop();
last
```

Out[22]: None

ITERATING OVER VECTOR

- Iterating: `.iter`
- Mutable iterating: `.iter_mut`

ITERATING OVER VECTOR

- Iterating: `.iter`
- Mutable iterating: `.iter_mut`

In [23]:
```
let mut v = vec![2,1,0];
for z in &v { // z in v.iter()
    println!("{}",z);
};
```

```
2
1
0
```

ITERATING OVER VECTOR

- Iterating: `.iter`
- Mutable iterating: `.iter_mut`

In [23]:
```
let mut v = vec![2,1,0];
for z in &v { // z in v.iter()
    println!("{}",z);
};
```

```
2
1
0
```

In [24]:
```
for z in &mut v { // z in v.iter_mut()
    *z += 1;
}
println!("{:?}",v);
```

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

ITERATING OVER VECTOR

- Iterating: `.iter`
- Mutable iterating: `.iter_mut`

In [23]:
```
let mut v = vec![2,1,0];
for z in &v { // z in v.iter()
    println!("{}",z);
};
```

```
2
1
0
```

In [24]:
```
for z in &mut v { // z in v.iter_mut()
    *z += 1;
}
println!("{:?}",v);
```

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

In [25]:
```
// if you need index as well
for (i,z) in v.iter().enumerate() {
    println!("{}: {}",i,z);
};
```

```
0: 3
1: 2
2: 1
```

UNDER THE HOOD: SELECT IMPLEMENTATION DETAILS

HOW TO MAKE PUSH/POP OPERATIONS EFFICIENT?

(TO BE CONTINUED...)