DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 17

RUST: FLOW CONTROL (CONTINUED). ALGEBRAIC DATA TYPES (TUPLES AND ENUMS).

POSTPONE THE PROJECT PROPOSAL DATE?

LOOPS: `while`

```
while condition {
    // DO SOMETHING HERE
}
```

LOOPS: `while`

```
while condition {
    // DO SOMETHING HERE
}
```

In [2]:
```
// find largest integer x s.t. x * x < 250
let mut x = 1;
while (x+1) * (x+1) < 250 {
    x += 1;
}
x
```

Out[2]: 15

INFINITE LOOP: `loop`

```
loop {
    // DO SOMETHING HERE
}
```

Need to use `break` to jump out of the loop!

In [3]:
```
let mut x = 1;
loop {
    if (x + 1) * (x + 1) >= 250 {break;}
    x += 1;
}
x
```

Out[3]: 15

INFINITE LOOP: `loop`

```
loop {
    // DO SOMETHING HERE
}
```

Need to use `break` to jump out of the loop!

```
In [3]: let mut x = 1;
        loop {
            if (x + 1) * (x + 1) >= 250 {break;}
            x += 1;
        }
        x

Out[3]: 15
```

- `loop` can return a value!
- `break` can act like `return`

```
In [4]: let mut x = 1;
        let x = loop {
            if x * x >= 250 {break x - 1;}
            x += 1;
        };
        x

Out[4]: 15
```

`break` AND `continue`

- work in all loops
- `break`: terminate the execution
    - can return a value in `loop`
- `continue`: terminate this iteration and jump to the next one
    - in `while`, the condition will be checked
    - in `for`, there may be no next iteration

`break` AND `continue`

- work in all loops
- `break`: terminate the execution
    - can return a value in `loop`
- `continue`: terminate this iteration and jump to the next one
    - in `while`, the condition will be checked
    - in `for`, there may be no next iteration

```
In [5]: for i in 1..=10 {
            if i % 3 != 0 {continue;}
            println!("{}",i);
        };
```

```
3
6
9
```

`break` AND `continue`

- work in all loops
- `break`: terminate the execution
    - can return a value in `loop`
- `continue`: terminate this iteration and jump to the next one
    - in `while`, the condition will be checked
    - in `for`, there may be no next iteration

```
In [5]: for i in 1..=10 {
            if i % 3 != 0 {continue;}
            println!("{}",i);
        };
```

```
3
6
9
```

`break` and `continue` can use labels

```
In [6]: 'outer_loop: loop {
            loop {
                break 'outer_loop;
            }
        };
        println!("Managed to escape! :-)");
```

```
Managed to escape! :-)
```

`break` AND `continue`

- work in all loops
- `break`: terminate the execution
    - can return a value in `loop`
- `continue`: terminate this iteration and jump to the next one
    - in `while`, the condition will be checked
    - in `for`, there may be no next iteration

In [5]:
```
for i in 1..=10 {
    if i % 3 != 0 {continue;}
    println!("{}",i);
};
```

```
3
6
9
```

`break` and `continue` can use labels

In [6]:
```
'outer_loop: loop {
    loop {
        break 'outer_loop;
    }
};
println!("Managed to escape! :-)");
```

```
Managed to escape! :-)
```

In [7]:
```
let x = 'outer_loop: loop {
    loop { break 'outer_loop 1234;}
};
println!("{}",x);
```

```
1234
```

TUPLES

- Syntax: `(value_1,value_2,value_3)`
- Type: `(type_1,type_2,type_3)`

```
In [8]: let mut tuple = (1,1.1);

        let another = ("abc","def","ghi");

        let yet_another: (u8,u32) = (255,4_000_000_000);
```

TUPLES

- Syntax: `(value_1,value_2,value_3)`
- Type: `(type_1,type_2,type_3)`

Accessing elements via index (0 based):

In [8]:
```
let mut tuple = (1,1.1);

let another = ("abc","def","ghi");

let yet_another: (u8,u32) = (255,4_000_000_000);
```

In [9]:
```
println!("({}, {})",tuple.0,tuple.1);
tuple.0 = 2;
println!("({}, {})",tuple.0,tuple.1);
```

```
(1, 1.1)
(2, 1.1)
```

TUPLES

- Syntax: `(value_1,value_2,value_3)`
- Type: `(type_1,type_2,type_3)`

In [8]:
```
let mut tuple = (1,1.1);

let another = ("abc","def","ghi");

let yet_another: (u8,u32) = (255,4_000_000_000);
```

Accessing elements via index (0 based):

In [9]:
```
println!("({}, {})",tuple.0,tuple.1);
tuple.0 = 2;
println!("({}, {})",tuple.0,tuple.1);
```

```
(1, 1.1)
(2, 1.1)
```

Accessing via matching:

In [10]:
```
let (integer,float) = tuple;
println!("({},{})",integer,float);
```

```
(2,1.1)
```

ENUMS

- Data type allowing for capturing a small set of options

In [11]:
```
enum Direction {
    North,
    East,
    South,
    West,
}

let dir = Direction::North;
let dir_2: Direction = Direction::South;
```

ENUMS

- Data type allowing for capturing a small set of options

In [11]:
```
enum Direction {
    North,
    East,
    South,
    West,
}

let dir = Direction::North;
let dir_2: Direction = Direction::South;
```

In [12]:
```
// Avoiding specifying "Direction::"
use Direction::East;
let dir_3 = East;
```

ENUMS

- Data type allowing for capturing a small set of options

In [11]:
```
enum Direction {
    North,
    East,
    South,
    West,
}

let dir = Direction::North;
let dir_2: Direction = Direction::South;
```

In [12]:
```
// Avoiding specifying "Direction::"
use Direction::East;
let dir_3 = East;
```

In [13]:
```
// Bringing two options into the current scope
use Direction::{East,West};
let dir_3 = West;
```

ENUMS

- Data type allowing for capturing a small set of options

In [11]:
```
enum Direction {
    North,
    East,
    South,
    West,
}

let dir = Direction::North;
let dir_2: Direction = Direction::South;
```

In [12]:
```
// Avoiding specifying "Direction::"
use Direction::East;
let dir_3 = East;
```

In [13]:
```
// Bringing two options into the current scope
use Direction::{East,West};
let dir_3 = West;
```

In [14]:
```
// Bringing all options in
use Direction::*;
let dir_4 = South;
```

ENUMS: PATTERN MATCHING VIA `match`

In [15]:
```
// print the direction
match dir {
    // if things not in scope,
    // have to use "Direction::"
    Direction::North => println!("N"),
    // but they are, so we don't have to
    South => println!("S"),
    West => println!("W"),
    East => println!("E"),
};
```

```
N
```

ENUMS: PATTERN MATCHING VIA `match`

In [15]:
```
// print the direction
match dir {
    // if things not in scope,
    // have to use "Direction::"
    Direction::North => println!("N"),
    // but they are, so we don't have to
    South => println!("S"),
    West => println!("W"),
    East => println!("E"),
};
```

```
N
```

In [16]:
```
// won't work
match dir_2 {
    North => println!("N"),
    South => println!("S"),
    // East and West not covered
};
```

```
match dir_2 {
      ^^^^^ patterns `East` and `West` not covered
non-exhaustive patterns: `East` and `West` not covered
help: ensure that all possible cases are being handled,
possibly by adding wildcards or more match arms
```

ENUMS: PATTERN MATCHING VIA `match`

In [15]:
```
// print the direction
match dir {
    // if things not in scope,
    // have to use "Direction::"
    Direction::North => println!("N"),
    // but they are, so we don't have to
    South => println!("S"),
    West => println!("W"),
    East => println!("E"),
};
```

```
N
```

In [16]:
```
// won't work
match dir_2 {
    North => println!("N"),
    South => println!("S"),
    // East and West not covered
};
```

```
match dir_2 {
      ^^^^^ patterns `East` and `West` not covered
non-exhaustive patterns: `East` and `West` not covered
help: ensure that all possible cases are being handled,
possibly by adding wildcards or more match arms
```

In [17]:
```
match dir_2 {
    North => println!("N"),
    South => println!("S"),

    // match anything left
    _ => (),
};
```

```
S
```

ENUMS: PATTERN MATCHING VIA `match`

In [15]:
```
// print the direction
match dir {
    // if things not in scope,
    // have to use "Direction::"
    Direction::North => println!("N"),
    // but they are, so we don't have to
    South => println!("S"),
    West => println!("W"),
    East => println!("E"),
};
```

```
N
```

In [16]:
```
// won't work
match dir_2 {
    North => println!("N"),
    South => println!("S"),
    // East and West not covered
};
```

```
match dir_2 {
      ^^^^^ patterns `East` and `West` not covered
non-exhaustive patterns: `East` and `West` not covered
help: ensure that all possible cases are being handled,
possibly by adding wildcards or more match arms
```

In [17]:
```
match dir_2 {
    North => println!("N"),
    South => println!("S"),

    // match anything left
    _ => (),
};
```

```
S
```

In [18]:
```
match dir_2 {
    _ => (),

    // will never get here!!
    North => println!("N"),
    South => println!("S"),
};
```

DISPLAYING ENUMS

By default Rust doesn't know how to display it

```
In [20]: println!("{}",dir);
```

```
println!("{}",dir);
             ^^^ `Direction` cannot be formatted with
the default formatter
`Direction` doesn't implement `std::fmt::Display`
help: the trait `std::fmt::Display` is not implemented
for `Direction`
```

DISPLAYING ENUMS

By default Rust doesn't know how to display it

```
In [21]: println!("{:?}",dir);
```

```
println!("{:?}",dir);
                 ^^^ `Direction` cannot be formatted usi
ng `{:?}`
`Direction` doesn't implement `Debug`
help: the trait `Debug` is not implemented for `Directi
on`
```

DISPLAYING ENUMS

By default Rust doesn't know how to display it

In [21]:
```
println!("{:?}",dir);
```

```
println!("{:?}",dir);
                ^^^ `Direction` cannot be formatted usi
ng `{:?}`
`Direction` doesn't implement `Debug`
help: the trait `Debug` is not implemented for `Directi
on`
```

In [22]:
```

dir
```

```
`Direction` cannot be formatted using `{:?}`
`Direction` doesn't implement `Debug`
help: the trait `Debug` is not implemented for `Directi
on`
```

DISPLAYING ENUMS

By default Rust doesn't know how to display it

In [21]:
```
println!("{:?}",dir);
```

```
println!("{:?}",dir);
                ^^^ `Direction` cannot be formatted usi
ng `{:?}`
`Direction` doesn't implement `Debug`
help: the trait `Debug` is not implemented for `Directi
on`
```

In [22]:
```

dir
```

```
`Direction` cannot be formatted using `{:?}`
`Direction` doesn't implement `Debug`
help: the trait `Debug` is not implemented for `Directi
on`
```

In [23]:
```
#[derive(Debug)]
enum Direction {
    North,
    East,
    South,
    West,
}

use Direction::*;
```

DISPLAYING ENUMS

By default Rust doesn't know how to display it

In [21]:
```
println!("{:?}",dir);
```

```
println!("{:?}",dir);
                ^^^ `Direction` cannot be formatted usi
ng `{:?}`
`Direction` doesn't implement `Debug`
help: the trait `Debug` is not implemented for `Directi
on`
```

In [22]:
```

dir
```

```
`Direction` cannot be formatted using `{:?}`
`Direction` doesn't implement `Debug`
help: the trait `Debug` is not implemented for `Directi
on`
```

In [23]:
```
#[derive(Debug)]
enum Direction {
    North,
    East,
    South,
    West,
}

use Direction::*;
```

In [24]:
```
dir
```

Out[24]: North

DISPLAYING ENUMS

By default Rust doesn't know how to display it

In [21]:
```
println!("{:?}",dir);
```

```
println!("{:?}",dir);
                ^^^ `Direction` cannot be formatted usi
ng `{:?}`
`Direction` doesn't implement `Debug`
help: the trait `Debug` is not implemented for `Directi
on`
```

In [22]:
```

dir
```

```
`Direction` cannot be formatted using `{:?}`
`Direction` doesn't implement `Debug`
help: the trait `Debug` is not implemented for `Directi
on`
```

In [23]:
```
#[derive(Debug)]
enum Direction {
    North,
    East,
    South,
    West,
}

use Direction::*;
```

In [24]:
```
dir
```

Out[24]:
```
North
```

In [25]:
```
println!("{:?}",dir);
```

```
North
```

`match` AS EXPRESSION

In [26]:
```
// swap east and west
let dir_4 = West;
println!("{:?}", dir_4);

let dir_4 = match dir_4 {
    East => West,
    West => {
        println!("Switching West to East");
        East
    }

    // variable mathching anything else
    other => other,
};

println!("{:?}", dir_4);
```

```
West
Switching West to East
East
```

ENUMS

- Each option can come with additional information

In [27]:
```
enum DivisionResult {
    Ok(u32),
    DivisionByZero,
}

fn divide(x:u32, y:u32) -> DivisionResult {
    if y == 0 {
        DivisionResult::DivisionByZero
    } else {
        DivisionResult::Ok(x / y)
    }
}

let (a,b) = (9,3);
match divide(a,b) {
    DivisionResult::Ok(result)
        => println!("the result is {}",result),
    DivisionResult::DivisionByZero
        => println!("noooooo!!!!"),
};
```

```
the result is 3
```

ENUMS

- Each option can come with additional information

In [27]:
```
enum DivisionResult {
    Ok(u32),
    DivisionByZero,
}

fn divide(x:u32, y:u32) -> DivisionResult {
    if y == 0 {
        DivisionResult::DivisionByZero
    } else {
        DivisionResult::Ok(x / y)
    }
}

let (a,b) = (9,3);
match divide(a,b) {
    DivisionResult::Ok(result)
        => println!("the result is {}",result),
    DivisionResult::DivisionByZero
        => println!("noooooo!!!!"),
};
```

```
the result is 3
```

In [28]:
```
enum DivisionResult {
    Ok(u32,u32),
    DivisionByZero,
}

fn divide(x:u32, y:u32) -> DivisionResult {
    if y == 0 {
        DivisionResult::DivisionByZero
    } else {
        DivisionResult::Ok(x / y, x % y)
    }
}

let (a,b) = (9,3);
match divide(a,b) {
    DivisionResult::Ok(result,reminder) => {
            println!("the result is {}",result);
            println!("the reminder is {}",reminder);
    }
    DivisionResult::DivisionByZero
        => println!("noooooo!!!!"),
};
```

```
the result is 3
the reminder is 0
```

SIMPLIFIED MATCHING `if let`

Consider the following example (in which we want to use just one branch):

In [29]:
```
match divide(8,4) {
    DivisionResult::Ok(result,reminder) => println!("{} (reminder {})",result,reminder),
    _ => (), // <--- how to avoid this?
};
```

```
2 (reminder 0)
```

SIMPLIFIED MATCHING `if let`

Consider the following example (in which we want to use just one branch):

In [29]:
```
match divide(8,4) {
    DivisionResult::Ok(result,reminder) => println!("{} (reminder {})",result,reminder),
    _ => (), // <--- how to avoid this?
};
```

```
2 (reminder 0)
```

`if let` allows for matching just one branch

In [30]:
```
if let DivisionResult::Ok(result,reminder) = divide(8,7) {
    println!("{} (reminder {})",result,reminder);
};
```

```
1 (reminder 1)
```

SIMPLIFIED MATCHING `if let`

Consider the following example (in which we want to use just one branch):

In [29]:
```
match divide(8,4) {
    DivisionResult::Ok(result,reminder) => println!("{} (reminder {})",result,reminder),
    _ => (), // <--- how to avoid this?
};
```

```
2 (reminder 0)
```

`if let` allows for matching just one branch

In [30]:
```
if let DivisionResult::Ok(result,reminder) = divide(8,7) {
    println!("{} (reminder {})",result,reminder);
};
```

```
1 (reminder 1)
```

In [31]:
```
let dir = North;
if let North = dir {
    println!("North");
};
```

```
North
```

SIMPLIFIED MATCHING `if let`

Consider the following example (in which we want to use just one branch):

In [29]:
```
match divide(8,4) {
    DivisionResult::Ok(result,reminder) => println!("{} (reminder {})",result,reminder),
    _ => (), // <--- how to avoid this?
};
```

```
2 (reminder 0)
```

`if let` allows for matching just one branch

In [30]:
```
if let DivisionResult::Ok(result,reminder) = divide(8,7) {
    println!("{} (reminder {})",result,reminder);
};
```

```
1 (reminder 1)
```

In [31]:
```
let dir = North;
if let North = dir {
    println!("North");
};
```

```
North
```

In [32]:
```
if let North = dir {
    println!("North");
} else {
    println!("Something else");
};
```

```
North
```

ALGEBRAIC DATA TYPES

Algebraic operations on types:

- product ($\times$) $\equiv$ tuples
- disjoint union ($\uplus$) $\equiv$ enums

$$((\texttt{u32} \times \texttt{f32} \times \texttt{bool}) \uplus (\texttt{bool} \times \texttt{u8})) \times \texttt{u32} \uplus ()$$

ALGEBRAIC DATA TYPES

Algebraic operations on types:

- product ($\times$) $\equiv$ tuples
- disjoint union ($\uplus$) $\equiv$ enums

$$((\texttt{u32} \times \texttt{f32} \times \texttt{bool}) \uplus (\texttt{bool} \times \texttt{u8})) \times \texttt{u32} \uplus ()$$

- Inspired by functional programming languages
- Explicitly supported in Rust, ML, OCaml, Haskell, Schema, TypeScript, ...

ALGEBRAIC DATA TYPES WITH RECURSION

- Could be very useful for expressing some concepts!
- Idealized not working example of a list

```
enum List {
  Element(i32,List),
  End,
}
```

- Easier in pure functional languages, but can be implemented in Rust too

ALGEBRAIC DATA TYPES WITH RECURSION

- Could be very useful for expressing some concepts!
- Idealized not working example of a list

```
enum List {
  Element(i32,List),
  End,
}
```

- Easier in pure functional languages, but can be implemented in Rust too

In [33]:

```
// actual recursive list in Rust
// you don't have to understand it at this point
enum List {
    Element(i32,Box<List>),
    End,
}

let mut list = List::End;
for t in [3,2,5,1,13,15].iter().rev() {
    list = List::Element(*t,Box::new(list));
}

fn show(list: &List) {
    let mut list = list;
    loop {
        match &list {
            List::End => break,
            List::Element(x,l) => {
                print!("{} ",x);
                list = &*l;
            }
        }
    }
    println!();
}

show(&list);
```

```
3 2 5 1 13 15
```

NEXT TIME: THINGS WILL GET REAL

MEMORY MANAGEMENT IN GENERAL AND IN RUST