DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 16

RUST: BASIC TYPES. PROJECT MANAGER (`cargo`). FUNCTIONS. FLOW CONTROL. ARRAYS.

BASIC TYPES: BOOLEANS, CHARACTERS, AND STRINGS

- `bool` uses one byte of memory

In [2]:
```
let x = true;
let y: bool = false;

// x and (not y)
x && !y
```

Out[2]: true

BASIC TYPES: BOOLEANS, CHARACTERS, AND STRINGS

- `bool` uses one byte of memory

In [2]:
```
let x = true;
let y: bool = false;

// x and (not y)
x && !y
```

Out[2]: true

- `char` defined via single quote, uses four bytes of memory (Unicode scalar value)

In [3]:
```
let x = 'a';
let y = '🚦';
let z : char = '🦕';
```

BASIC TYPES: BOOLEANS, CHARACTERS, AND STRINGS

- `bool` uses one byte of memory

In [2]:
```
let x = true;
let y: bool = false;

// x and (not y)
x && !y
```

Out[2]: true

- `char` defined via single quote, uses four bytes of memory (Unicode scalar value)

In [3]:
```
let x = 'a';
let y = '🚦';
let z : char = '🦕';
```

- string slice defined via double quotes (not so basic actually!)

In [4]:
```
let s1 = "Hello! How are you, 🦕?";
let s2 : &str = "Zażółć gęślą jaźń.";
```

PROJECT MANAGER: `cargo`

- create a project: `cargo new PROJECT-NAME`
- main file will be `PROJECT-NAME/src/main.rs`

PROJECT MANAGER: `cargo`

- create a project: `cargo new PROJECT-NAME`
- main file will be `PROJECT-NAME/src/main.rs`

- to compile and run: `cargo run`
- to just compile: `cargo build`

PROJECT MANAGER: `cargo`

- create a project: `cargo new PROJECT-NAME`
- main file will be `PROJECT-NAME/src/main.rs`

- to compile and run: `cargo run`
- to just compile: `cargo build`

Add `--release` to create a "fully optimized" version:

- longer compilation
- faster execution
- some runtime checks not included (e.g., integer overflow)
- debugging information not included
- the executable in a different folder

PROJECT MANAGER: `cargo`

If you just want to **check** if your current version would compile: `cargo check`

- Much faster for big projects

CONDITIONAL EXPRESSIONS: `if`

Syntax:

```
if condition {
    DO-SOMETHING-HERE
} else {
    DO-SOMETHING-ELSE-HERE
}
```

- `else` part optional
- Compared to many C-like languages:
    - no parentheses around `condition` needed!
    - the braces mandatory

CONDITIONAL EXPRESSIONS: `if`

Syntax:

```
if condition {
    DO-SOMETHING-HERE
} else {
    DO-SOMETHING-ELSE-HERE
}
```

- `else` part optional
- Compared to many C-like languages:
  - no parentheses around `condition` needed!
  - the braces mandatory

```
In [5]: let x = 7;
        if x <= 15 {
            println!("x is not greater than 15");
        };

        x is not greater than 15
```

CONDITIONAL EXPRESSIONS: `if`

Syntax:

```
if condition {
    DO-SOMETHING-HERE
} else {
    DO-SOMETHING-ELSE-HERE
}
```

- `else` part optional
- Compared to many C-like languages:
  - no parentheses around `condition` needed!
  - the braces mandatory

In [5]:
```
let x = 7;
if x <= 15 {
    println!("x is not greater than 15");
};
```

```
x is not greater than 15
```

In [7]:
```
let threshold = 5;
if x <= threshold {
    println!("x is at most {}",threshold);
} else {
    println!("x is greater than {}", threshold);
};
```

```
x is greater than 5
```

CONDITIONAL EXPRESSIONS: `if`

Syntax:

```
if condition {
    DO-SOMETHING-HERE
} else {
    DO-SOMETHING-ELSE-HERE
}
```

- `else` part optional
- Compared to many C-like languages:
  - no parentheses around `condition` needed!
  - the braces mandatory

```
In [5]: let x = 7;
        if x <= 15 {
            println!("x is not greater than 15");
        };
```

```
x is not greater than 15
```

```
In [8]: let threshold = 20;
        if x <= threshold {
            println!("x is at most {}",threshold);
        } else {
            println!("x is greater than {}", threshold);
        };
```

```
x is at most 20
```

USING CONDITIONAL EXPRESSIONS AS EXPRESSIONS

Python:

```
100 if (x == 7) else 200
```

C++:

```
(x == 7) ? 100 : 200
```

Rust:

```
if x == 7 {100} else {200}
```

USING CONDITIONAL EXPRESSIONS AS EXPRESSIONS

Python:

```
100 if (x == 7) else 200
```

C++:

```
(x == 7) ? 100 : 200
```

Rust:

```
if x == 7 {100} else {200}
```

```
In [9]: let x = 1;
        println!("{}",if x == 7 {100} else {200});

        200
```

USING CONDITIONAL EXPRESSIONS AS EXPRESSIONS

Python:

```
100 if (x == 7) else 200
```

C++:

```
(x == 7) ? 100 : 200
```

Rust:

```
if x == 7 {100} else {200}
```

In [10]:
```
let x = 7;
println!("{}",if x == 7 {100} else {200});
```

```
100
```

USING CONDITIONAL EXPRESSIONS AS EXPRESSIONS

Python:

```
100 if (x == 7) else 200
```

C++:

```
(x == 7) ? 100 : 200
```

Rust:

```
if x == 7 {100} else {200}
```

```
In [10]: let x = 7;
         println!("{}",if x == 7 {100} else {200});

         100
```

USING CONDITIONAL EXPRESSIONS AS EXPRESSIONS

Python:

```
100 if (x == 7) else 200
```

C++:

```
(x == 7) ? 100 : 200
```

Rust:

```
if x == 7 {100} else {200}
```

```
In [10]: let x = 7;
         println!("{}",if x == 7 {100} else {200});
```

```
100
```

```
In [11]: // won't work: same type needed
         println!("{}",if x == 7 {100} else {1.2});
```

```
println!("{}",if x == 7 {100} else {1.2});
                                    ^^^ expected intege
r, found floating-point number
println!("{}",if x == 7 {100} else {1.2});
                                ^^^ expected because of this
`if` and `else` have incompatible types
```

USING CONDITIONAL EXPRESSIONS AS EXPRESSIONS

Python:

```
100 if (x == 7) else 200
```

C++:

```
(x == 7) ? 100 : 200
```

Rust:

```
if x == 7 {100} else {200}
```

In [10]:
```
let x = 7;
println!("{}",if x == 7 {100} else {200});
```

```
100
```

In [11]:
```
// won't work: same type needed
println!("{}",if x == 7 {100} else {1.2});
```

```
println!("{}",if x == 7 {100} else {1.2});
                                    ^^^ expected intege
r, found floating-point number
println!("{}",if x == 7 {100} else {1.2});
                                ^^^ expected because of this
`if` and `else` have incompatible types
```

In [12]:
```
// blocks can be more complicated
// last expression counts (no semicolon after)
let z = if x == 4 {
    let t = x * x;
    t + 1
} else {
    x + 1
};
println!("{}",z);
```

```
8
```

FUNCTIONS

Syntax:

```
fn function_name(argname_1:type_1,argname_2:type_2) -> type_ret {
    DO-SOMETHING-HERE-AND-RETURN-A-VALUE
}
```

FUNCTIONS

Syntax:

```
fn function_name(argname_1:type_1,argname_2:type_2) -> type_ret {
    DO-SOMETHING-HERE-AND-RETURN-A-VALUE
}
```

In [13]:
```
fn multiply(x:i32, y:i32) -> i32 {
    // note: no need to write "return x * y"
    x * y
}

multiply(10,20)
```

Out[13]: 200

FUNCTIONS

Syntax:

```
fn function_name(argname_1:type_1,argname_2:type_2) -> type_ret {
    DO-SOMETHING-HERE-AND-RETURN-A-VALUE
}
```

In [13]:
```
fn multiply(x:i32, y:i32) -> i32 {
    // note: no need to write "return x * y"
    x * y
}

multiply(10,20)
```

Out[13]: 200

In [14]:
```
fn and(p:bool, q:bool, r:bool) -> bool {
    if !p {
        return false;
    }
    if !q {
        return false;
    }
    r
}
and(true,true,true)
```

Out[14]: true

FUNCTIONS: RETURNING NO VALUE

How: skip the type of returned value part

```
In [15]: fn say_hello(who:&str) {
             println!("Hello, {}!",who);
         }

         say_hello("world");
         say_hello("Boston");
         say_hello("MCS B37");
```

```
Hello, world!
Hello, Boston!
Hello, MCS B37!
```

FUNCTIONS: RETURNING NO VALUE

How: skip the type of returned value part

Nothing returned equivalent to the unit type, `( )`

In [15]:
```
fn say_hello(who:&str) {
    println!("Hello, {}!",who);
}

say_hello("world");
say_hello("Boston");
say_hello("MCS B37");
```

```
Hello, world!
Hello, Boston!
Hello, MCS B37!
```

In [16]:
```
fn say_good_night(who:&str) -> () {
    println!("Good night {}",who);
}

say_good_night("room");
say_good_night("moon");
say_good_night("cow jumping over the moon");

let z : () = ();
```

```
Good night room
Good night moon
Good night cow jumping over the moon
```

LOOPS: `for`

Usage: iterate over an iterator, range, or collection

In [17]:
```
for i in 1..5 {
    println!("{}",i);
};
```

```
1
2
3
4
```

LOOPS: `for`

Usage: iterate over an iterator, range, or collection

In [17]:
```
for i in 1..5 {
    println!("{}",i);
};
```

```
1
2
3
4
```

In [18]:
```
// reverse order
for i in (1..5).rev() {
    println!("{}",i)
};
```

```
4
3
2
1
```

LOOPS: `for`

Usage: iterate over an iterator, range, or collection

In [17]:
```
for i in 1..5 {
    println!("{}",i);
};
```

```
1
2
3
4
```

In [18]:
```
// reverse order
for i in (1..5).rev() {
    println!("{}",i)
};
```

```
4
3
2
1
```

In [19]:
```
// closed range
for i in 1..=5 {
    println!("{}",i);
};
```

```
1
2
3
4
5
```

LOOPS: `for`

Usage: iterate over an iterator, range, or collection

In [17]:
```
for i in 1..5 {
    println!("{}",i);
};
```

```
1
2
3
4
```

In [18]:
```
// reverse order
for i in (1..5).rev() {
    println!("{}",i)
};
```

```
4
3
2
1
```

In [19]:
```
// closed range
for i in 1..=5 {
    println!("{}",i);
};
```

```
1
2
3
4
5
```

In [21]:
```
// every other element
for i in (1..5).rev().step_by(2) {
    println!("{}",i)
};
```

```
4
2
```

ARRAYS AND `for` OVER AN ARRAY

- Arrays in Rust are of fixed length (we'll learn about more flexible `Vec` later)
- All elements of the same type

In [22]:
```
// simplest definition
// compiler guessing element type to be i32
// indexing starts at 0
let mut arr = [1,7,2,5,2];
arr[1] = 13;
println!("{} {}",arr[0],arr[1]);
```

```
1 13
```

ARRAYS AND `for` OVER AN ARRAY

- Arrays in Rust are of fixed length (we'll learn about more flexible `Vec` later)
- All elements of the same type

In [22]:
```
// simplest definition
// compiler guessing element type to be i32
// indexing starts at 0
let mut arr = [1,7,2,5,2];
arr[1] = 13;
println!("{} {}",arr[0],arr[1]);
```

```
1 13
```

In [23]:
```
arr.sort();
// loop over the array
for x in arr {
    println!("{}",x);
};
```

```
1
2
2
5
13
```

ARRAYS AND `for` OVER AN ARRAY

- Arrays in Rust are of fixed length (we'll learn about more flexible `Vec` later)
- All elements of the same type

In [22]:
```
// simplest definition
// compiler guessing element type to be i32
// indexing starts at 0
let mut arr = [1,7,2,5,2];
arr[1] = 13;
println!("{} {}",arr[0],arr[1]);
```

```
1 13
```

In [23]:
```
arr.sort();
// loop over the array
for x in arr {
    println!("{}",x);
};
```

```
1
2
2
5
13
```

In [25]:
```
// create array of given length
// and fill it with a specific value
let arr2 = [15;3];
for x in arr2 {
    print!("{} ",x);
}
println!();
```

```
15 15 15
```

ARRAYS AND `for` OVER AN ARRAY

- Arrays in Rust are of fixed length (we'll learn about more flexible `Vec` later)
- All elements of the same type

In [22]:
```
// simplest definition
// compiler guessing element type to be i32
// indexing starts at 0
let mut arr = [1,7,2,5,2];
arr[1] = 13;
println!("{} {}",arr[0],arr[1]);
```

```
1 13
```

In [23]:
```
arr.sort();
// loop over the array
for x in arr {
    println!("{}",x);
};
```

```
1
2
2
5
13
```

In [25]:
```
// create array of given length
// and fill it with a specific value
let arr2 = [15;3];
for x in arr2 {
    print!("{} ",x);
}
println!();
```

```
15 15 15
```

In [28]:
```
// with type definition included
let arr3 : [u8;3] = [15;3];
println!("{}", arr3[1] * 200);
```

```
thread '<unnamed>' panicked at 'attempt to multiply wit
h overflow', src/lib.rs:147:16
stack backtrace:
   0: rust_begin_unwind
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
5102f57a/library/std/src/panicking.rs:498:5
   1: core::panicking::panic_fmt
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
5102f57a/library/core/src/panicking.rs:116:14
   2: core::panicking::panic
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
5102f57a/library/core/src/panicking.rs:48:5
   3: run_user_code_26
```

ARRAYS AND `for` OVER AN ARRAY

- Arrays in Rust are of fixed length (we'll learn about more flexible `Vec` later)
- All elements of the same type

In [22]:
```
// simplest definition
// compiler guessing element type to be i32
// indexing starts at 0
let mut arr = [1,7,2,5,2];
arr[1] = 13;
println!("{} {}",arr[0],arr[1]);
```

```
1 13
```

In [23]:
```
arr.sort();
// loop over the array
for x in arr {
    println!("{}",x);
};
```

```
1
2
2
5
13
```

In [25]:
```
// create array of given length
// and fill it with a specific value
let arr2 = [15;3];
for x in arr2 {
    print!("{} ",x);
}
println!();
```

```
15 15 15
```

In [29]:
```
// with type definition included
let arr3 : [u8;3] = [15;3];
println!("{}", arr3[1]);
```

```
15
```

ARRAYS AND `for` OVER AN ARRAY

- Arrays in Rust are of fixed length (we'll learn about more flexible `Vec` later)
- All elements of the same type

In [22]:
```
// simplest definition
// compiler guessing element type to be i32
// indexing starts at 0
let mut arr = [1,7,2,5,2];
arr[1] = 13;
println!("{} {}",arr[0],arr[1]);
```

```
1 13
```

In [23]:
```
arr.sort();
// loop over the array
for x in arr {
    println!("{}",x);
};
```

```
1
2
2
5
13
```

In [25]:
```
// create array of given length
// and fill it with a specific value
let arr2 = [15;3];
for x in arr2 {
    print!("{} ",x);
}
println!();
```

```
15 15 15
```

In [29]:
```
// with type definition included
let arr3 : [u8;3] = [15;3];
println!("{}", arr3[1]);
```

```
15
```

In [30]:
```
// get the length
arr3.len()
```

Out[30]:
```
3
```

DISCUSSION SECTION TODAY: EXAMPLES IN R