# Rust cheatlist

## Cargo

- `cargo new project_name`: Initializes a new Rust project named *project_name* in the current directory.
- `cargo build`: Build program without optimizations. Output is stored in *./target/debug*.
- `cargo build –release`: Build program with runtime optimizations. Output is stored in *./target/release*.

## Data types

- Scalar types:
    - Integer: Internally represented in 2-components notation when signed (-($2^n$ ) → $2^n$ -1)

| Length | Signed | Unsigned |
|---|---|---|
| 8-bit | i8 | u8 |
| 16-bit | i16 | u16 |
| 32-bit | i32 | u32 |
| 64-bit | i64 | u64 |
| 128-bit | i128 | u128 |
| arch | isize | usize |

- Floating point: `f32`, `f64`
- Boolean: `bool`
- Character: `char`
- Compound types
- Tuple: Fixed size (defined at declaration), elements may differ in type
    - `let tup: (i32, f64, u8) = (500, 6.4, 1);`
    - Values can be retrieved by either pattern matching: `let (x, y, z) = tup; // x, y and z are now accessible as variables` or by using a period `let x = tup.0;`
    - Array: Fixed size, elements should be of the same type
    - `let a = [1, 2, 3];`
    - `let a: [f64; 3] = [1.0, 2.0, 3.0];`
    - `let a = [0; 5];`: Creates an array of size 5 with all elements initialized to 0
    - `let first = a[0];`: Accessing elements of array
    - `for element in a.iter() { … }`: Iterates over elements in array
    - Rust panics on index out of bounds situations

## Variables

- `let foo = bar;`: Creates immutable variable `foo` and assigns it value `bar`.
- `let mut foo = bar;`: Creates mutable variable `foo` and assigns it value `bar`.
- `let foo: type = false;`: Creates immutable variable `foo%` with explicit type definition.

## Functions

- `fn function_name() { … }`
- `fn function_name(x: i32, y: char) { … }`: Parameterized function
- `fn function_name(x: i32) → i32 { … }`: Function with return value. Returned value is last evaluated expression of the function body.

## Terminology

- Associated function: function implemented on a type rather than on a particular instance of the type. Similar as a *static method* in Java.
- Destructing: splitting a tuple in individual parts by pattern matching
- Expression: instructions that evaluate to a resulting value. No semicolon at end of line!
- Macro:
- Prelude:
- Statement: instructions that do not return a value
- Trait:

## Syntax

- `&var`: Passes `var` as a reference. Allows a function to access a variable without the need to copy it to the function's stack.
- `&mut var`: Passes `var` as a mutable reference. Allows a function to access and alter the variable's value.