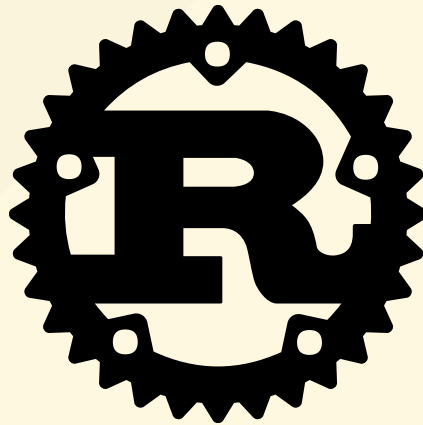


Rust

**A safe systems programming
language**



Features

Some of Rust's selling points:

- **guaranteed memory safety** (current [research topic](#))
- **threads without data races**
- generics
- pattern matching
- zero-cost abstractions ("What you don't use, you don't pay for.")
- Rust balances control (unmanaged) and safety

Quick Comparison

Rust	Python
compiled	interpreted (GIL)
<i>static</i> strong typing with <i>type inference</i>	<u><i>dynamic and</i></u> <u><i>strongly typed</i></u>
automatic ¹ /wo GC → real time capable	garbage collected
{ curly brace language }	indentation based

¹ you have to think about variable scopes

History

- in development since 2010
- first stable release (1.0) on May 15, 2015
- since then about every 6 weeks a new minor versions
 - 1.10 at the time of writing
- open source project backed by Mozilla, Samsung and others

Competition

Similar languages in order of common features:

- [nim](#)
- C++
- [D](#)
- [Swift](#)

Companies that use Rust

- [Servo](#), Mozilla's next generation parallel browser engine
- [Brotli decompressor](#) from Dropbox
- [... many more](#)

Rust is often used as a replacement for C to write safe and performant external modules for Ruby (RoR) or Python applications.

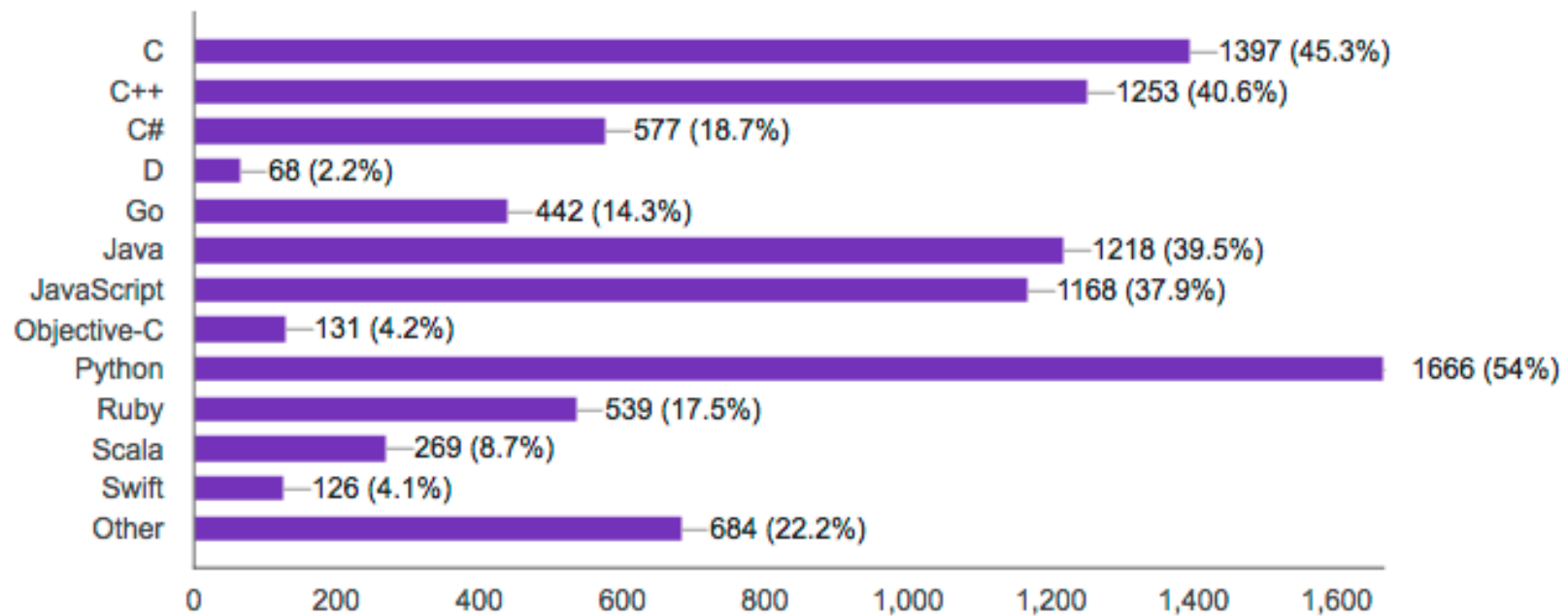
Trivia

- Rust programmers are called *rustaceans* (like pythonistas)
- the language is named after a fungus
- idiomatic Rust code is called *rusty* or *rustic*
- [Ferris the Crab](#)



Rust Python

What programming languages are you most comfortable with? (3085 responses)



Compiled Language

- Python is interpreted (at runtime)
- Rust's compilation steps (MIR since 1.9):

Source \rightarrow (HIR \rightarrow [MIR](#) \rightarrow LLVM IR \rightarrow) Machine Code

- [C]Python: source (.py) \rightarrow bytecode (.pyc) \rightarrow runtime
- [LLVM](#) intermediate representation (IR) is used by many compilers for a lot of different languages: clang (C), [Swift](#), [D](#), [Haskell](#), [Pyston](#), ...
- Rust benefits from improvements of LLVM's optimization steps

Hello World

```
fn main() {  
    let words = ["Hello", "pythonistas"];  
    for word in words.iter() {  
        print!("{}", word);  
    }  
    println!("!");  
}
```

Compile & Run

```
$ rustc hello.rs -o hello && ./hello  
Hello pythonistas !
```

Installation

- Linux, Win (e.g. cygwin bash), OSX:

```
curl https://sh.rustup.rs -sSf | sh
```

- Rust comes in different flavours: *stable*, *beta*, *nightly*
- `rustup` (former multirust) lets you install different Rust versions side-by-side
- `rustup` also manages target platforms for cross-compilation
- Rust's package manager called `cargo` is also part

Cargo

- Rust's package manager and build tool
- Packages are called *crates*
- ~5.6k crates on crates.io
- **No** batteries included but there is a crate for everything
- test runner, dependency management (pip), documentation generator, ...
- additional features through plugins

```
$ cargo --list
Installed Commands:
  bench
  build
  check
  clean
  clippy
  doc
  fetch
  fmt
  generate-lockfile
  git-checkout
  graph
  help
  init
  install
  locate-project
  login
  ...
```

Setup a new project (`--bin` application, default is library):

```
$ cargo new --bin example
$ tree example
example
├── Cargo.toml // ~requirements.txt && setup.py
└── src/
    └── main.rs
```

```
$ cargo run
   Compiling example v0.1.0 (file:///home/andreas/personal/...)
   Running `target/debug/example`
Hello, world!
```

Variables

- immutable by default
- create a variable binding: `let name: type = value;`
- mutable binding: `let mut x: type = value;`
- must be initialized with some value
- `type` can often be inferred

Ownership

- variable *bindings have ownership*
- if a binding goes out of scope its resources are freed
- Rust ensures that there is only **one** binding to any resource

Borrowing

- *borrowing* is lending a reference `&T` to a resource

```
sum(&[1, 2, 3, 4, 5]);
```

- resource is not freed when reference gets out of scope
- **borrowing rules:**
 1. one or more references `&T` to a resource (shared borrow)
 2. exactly one (even across threads) mutable reference `&mut T` (mutable borrow)

Lifetimes

- references have lifetimes
- a reference can't outlive the resource it is pointing at
- if the compiler can't infer the lifetime then it must be declared

```
// implicit  
fn foo(x: &i32) {  
}  
  
// explicit  
fn bar<'a>(x: &'a i32) {  
}
```

Enums

- Rust's enums are *algebraic datatypes*
 - **algebraic**: build from product types (tuples, structs) and sum types (only one variant at any one time, e.g. enum variants)

```
enum Message {  
    Quit, // variant /wo data  
    ChangeColor(i32, i32, i32), // tuple variant  
    Move { x: i32, y: i32 }, // struct (≈dict) variant  
    Write(String), // single value variant  
}
```

Pattern Matching

```
enum E {  
    A,  
    B(i32)  
}  
  
fn main() {  
    let e = E::B(4);  
    match e {  
        E::A => println!("I'm an A!"),  
        E::B(x) => println!("I'm an B with value: {}!", x),  
    }  
}
```

Iterators



- list/dict comprehensions on steroids
- lazy evaluated, must be consumed to get the value
- get a vector with values from 0 to 9:

```
let v = (0..10).collect::<Vec<_>>::();
```
- [iterator cheat sheet](#)

Traits



- defines functionality an implementing type must provide

```
use std::ops::Add;

struct Pair { a: f64, b: f64 }

impl Add for Pair {
    type Output = Pair;

    fn add(self, other: Pair) -> Pair {
        Pair {
            a: self.a + other.a,
            b: self.b + other.b,
        }
    }
}
```

Generics

- parameterized type

```
use std::ops::Add;

// looks a bit strange, but
// the output can be different from the input
// type.
fn add<T: Add<T>>(x: T, y: T) -> T::Output {
    x + y
}

fn main() {
    println!("{}", add(3.0, 0.14159265359));
    println!("{}", add(30_000u64, 40_000));
}
```

Error Handling

- **no exceptions**, errors are values (like in Go)

```
fn read_file(path: Into<String>) -> Result<Vec<u8>, IOErrors>

let data = match read_file("./some.file") {
    Ok(data) => data,
    Err(error) => panic!(err),
}
```

- there are certain macros and methods to avoid explicit matching of common `Option` and `Result` types (`try!`)

```
// shorter version from above
```


Macros

- end with an `!:` `println!`
- expandend at compile time
- hygienic macros: expand always to valid code at compile time (unlike C's text based preprocessor macros)

Tools

rustfmt

- `cargo install rustfmt`
- de-facto standard for code formatting, like go's `gofmt` but a bit less opinionated
- `autopep8` ☐ `rustfmt`
- the [rusti project](#) aims to provide a REPL for Rust

clippy

- sophisticated linter
- `> 160` lints
- really nice for beginners to see antipatterns (index based loops instead of iterators, etc.)
- requires Rust nightly build (`rustup` 🎉) because the compiler plugin API is not stable yet

Pros

- very friendly and helpful community - [code of conduct](#)
- incredibly good [documentation](#)
- great tooling
- transparent language development, core team that decides about [RFCs](#), [internals.rust-lang.org](#)
- [play.rust-lang.org](#)

Cons

- infrastructure too immature for productive use (depends)
- steep learning curve (fight against the borrow checker)
 - (+) learning Rust will give you a new perspective on how your code is executed
- debugger support could be better

Resources

- [The Rust Programming Language](#) - The official Rust book
- [users.rust-lang.org](#)
- [rustplatz](#)
- [/r/rust](#)
- [#rust-beginners](#) IRC
- [community overview](#) or [rustaceans.org](#)



width: 300%

See you @ [Rustfest?](#)

First european Rust conference

Questions?

Thank you for listening!

Made with ❤️ and [marp editor](#)

[klingt.net](#)

Andreas Linz - 2016