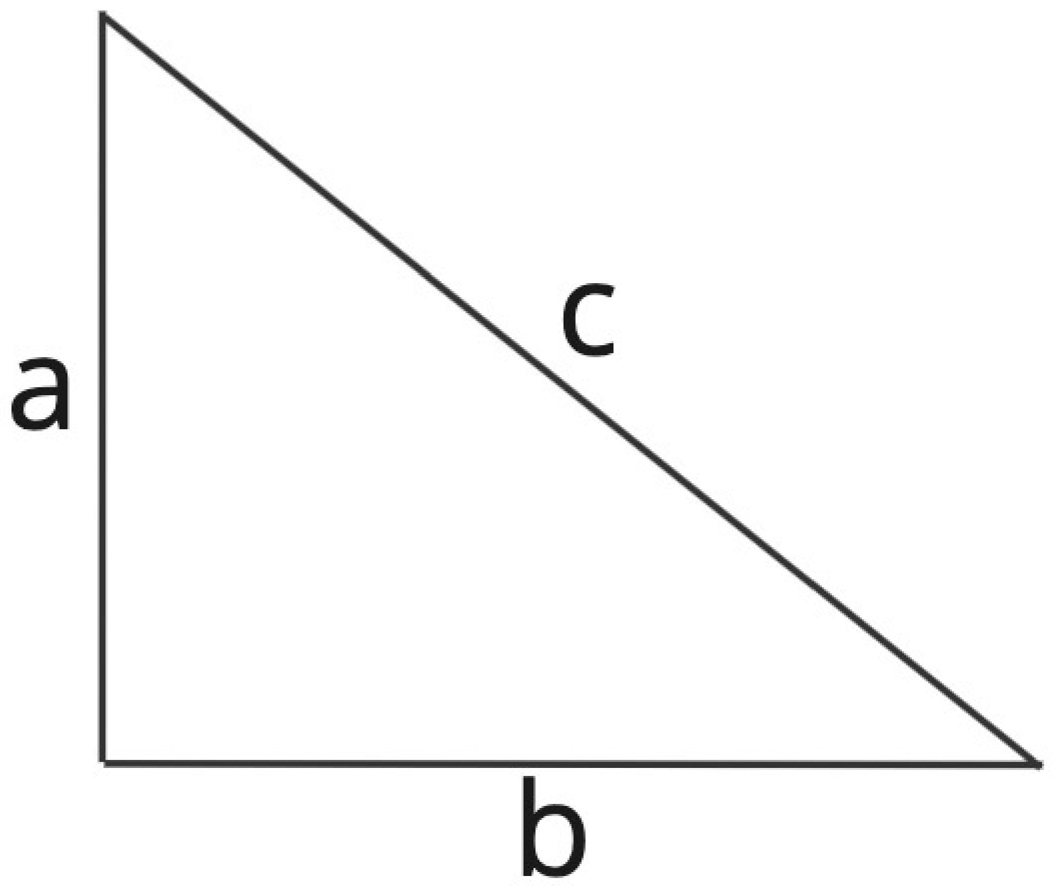
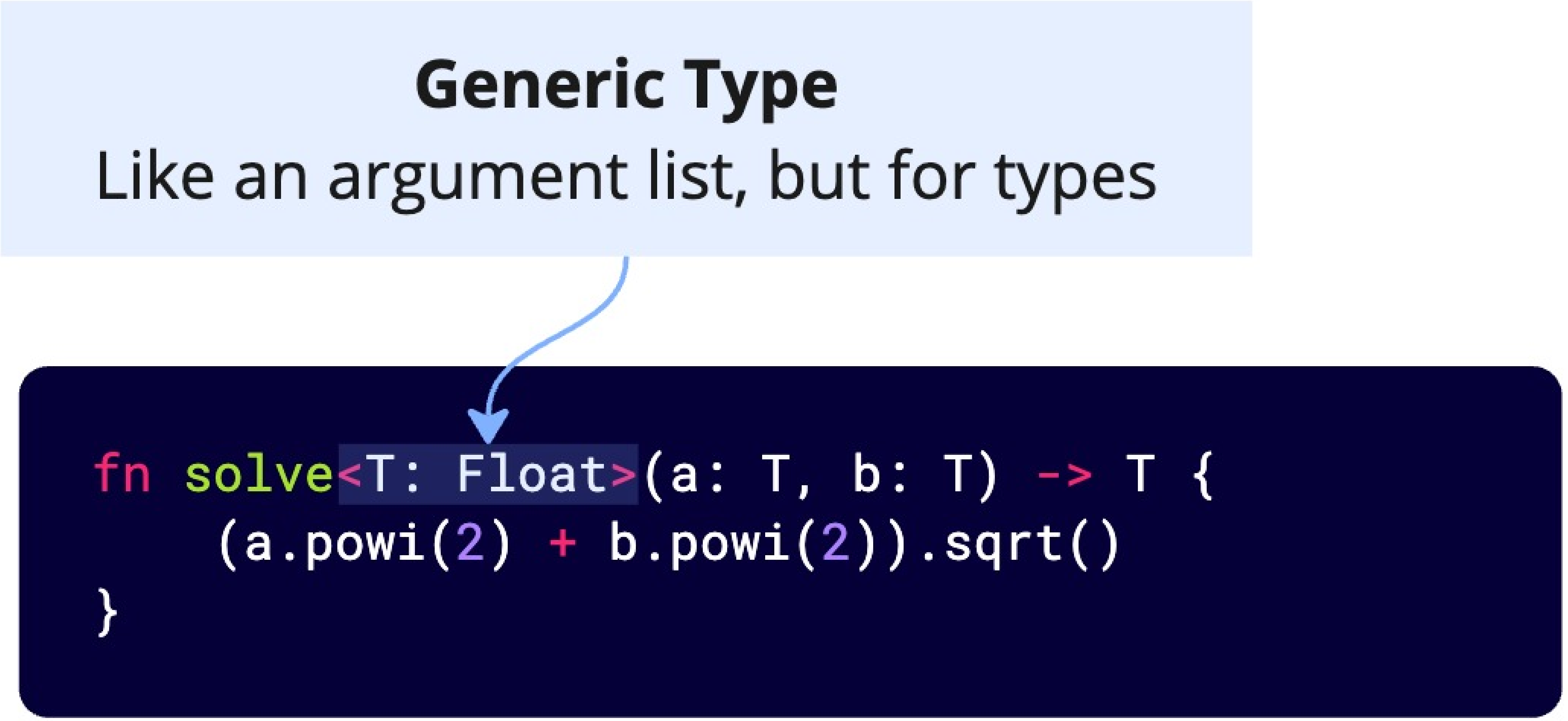

c
a
b

We're going to make two versions of 'solve'

First will allow you to pass in "**a**" and "**b**" **both** as f32 or f64

Second will allow mixing and matching of any type of number

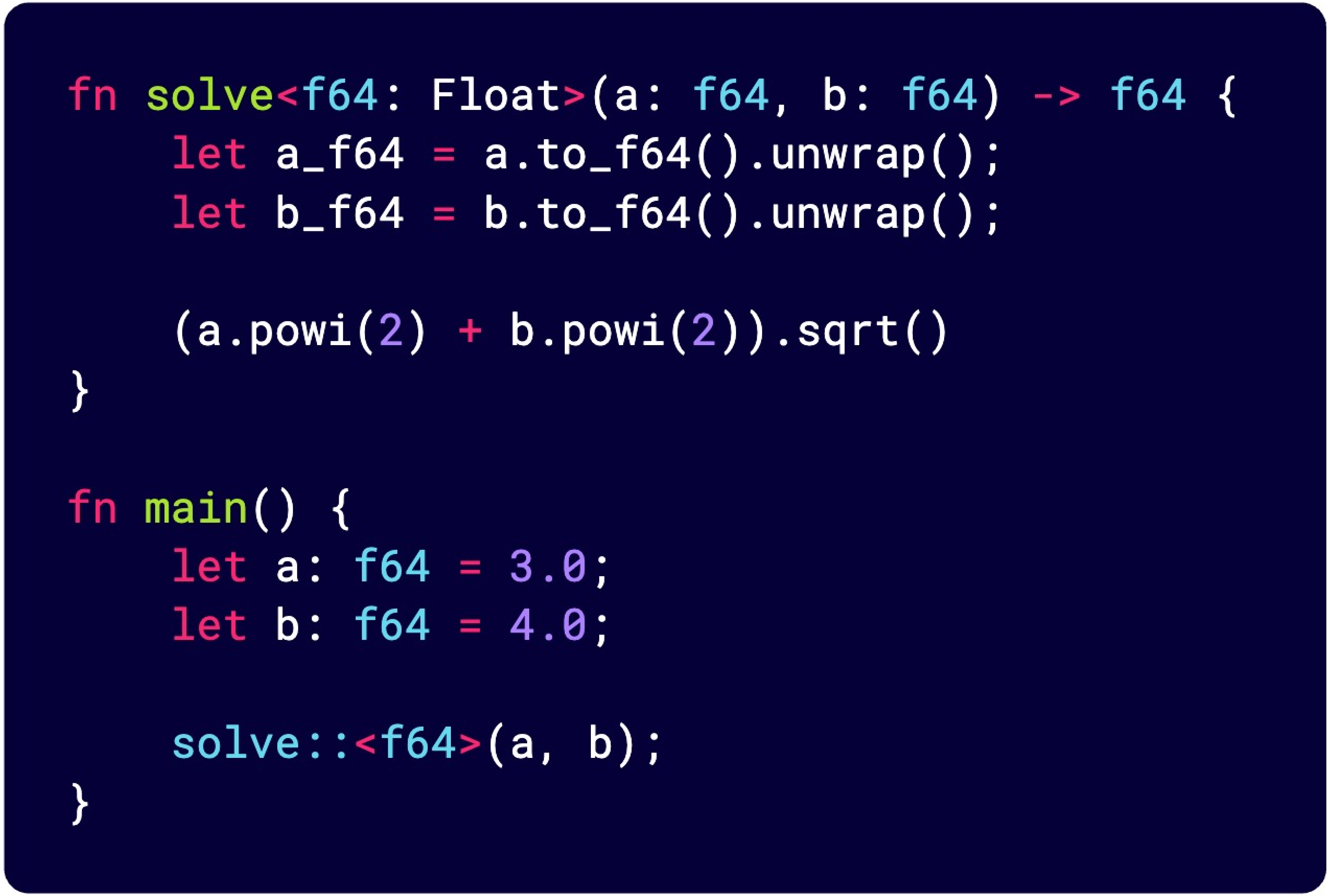


Generic Type

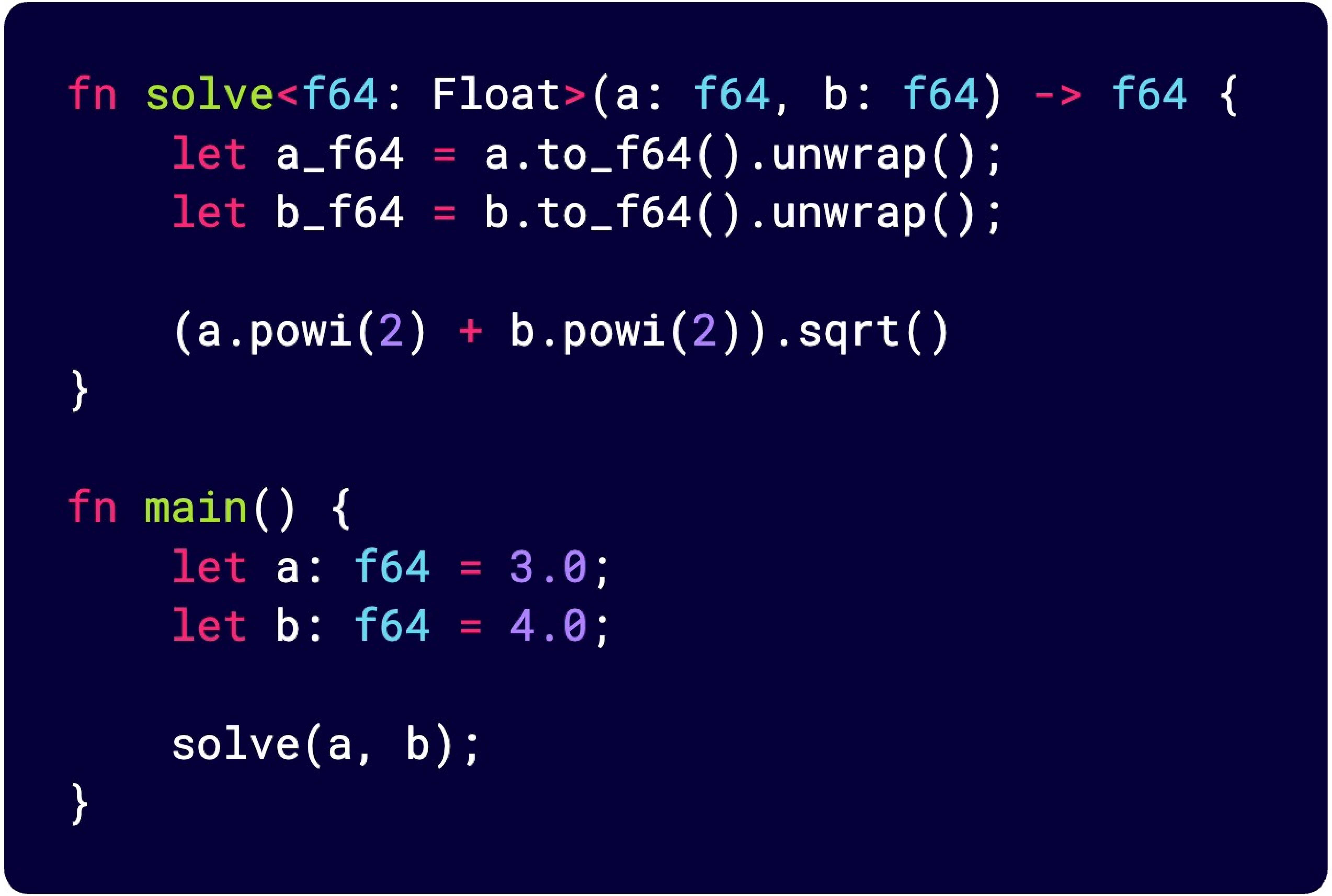
Like an argument list, but for types

```rust
fn solve<T: Float>(a: T, b: T) -> T {
    (a.powi(2) + b.powi(2)).sqrt()
}
```

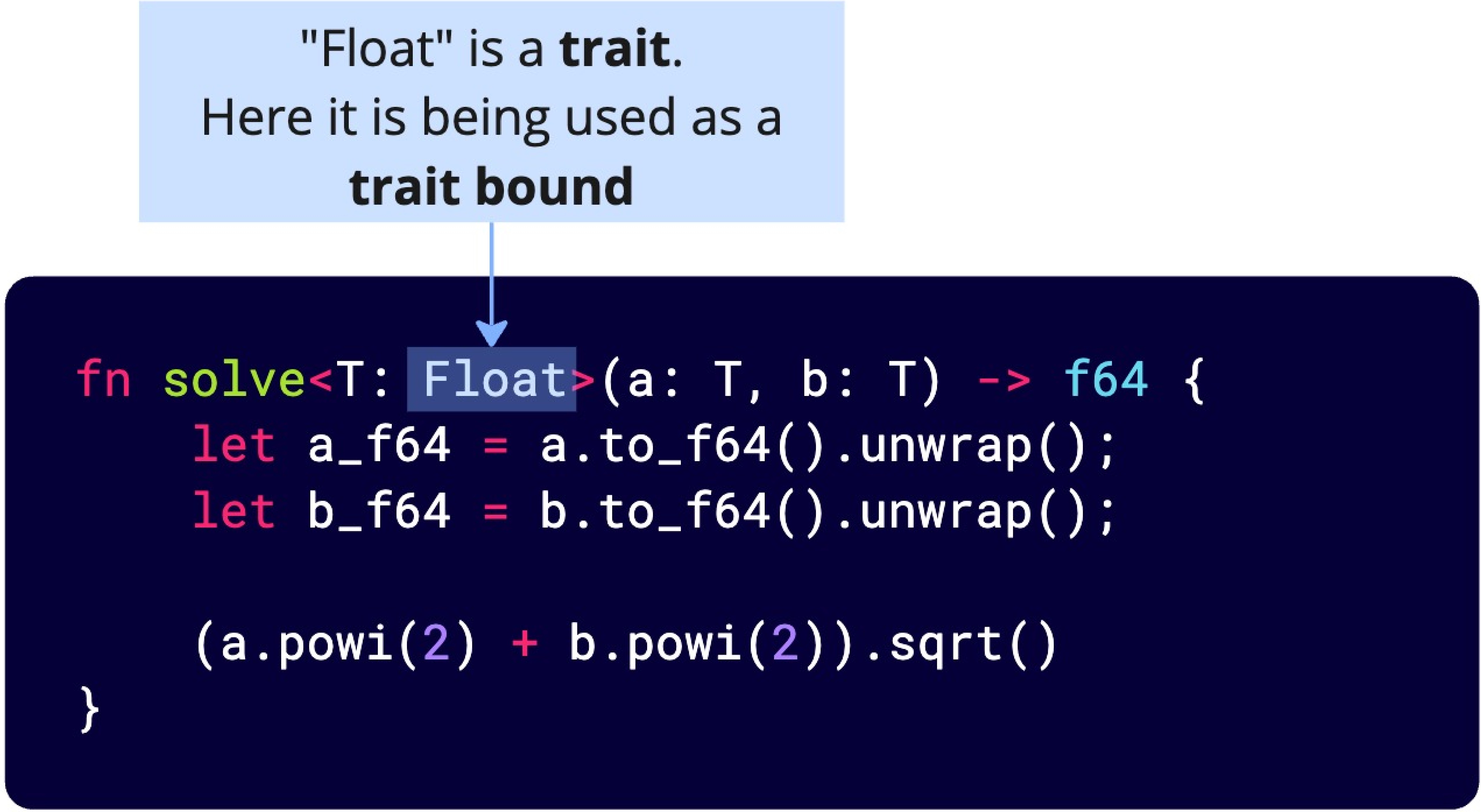
```
fn solve(3.0: f64, 4.0: f64) -> f64 {
    (3.0.powi(2) + 4.0.powi(2)).sqrt()
}


solve(3.0, 4.0)
```



```rust
fn solve<f64: Float>(a: f64, b: f64) -> f64 {
    let a_f64 = a.to_f64().unwrap();
    let b_f64 = b.to_f64().unwrap();

    (a.powi(2) + b.powi(2)).sqrt()
}

fn main() {
    let a: f64 = 3.0;
    let b: f64 = 4.0;

    solve::<f64>(a, b);
}
```

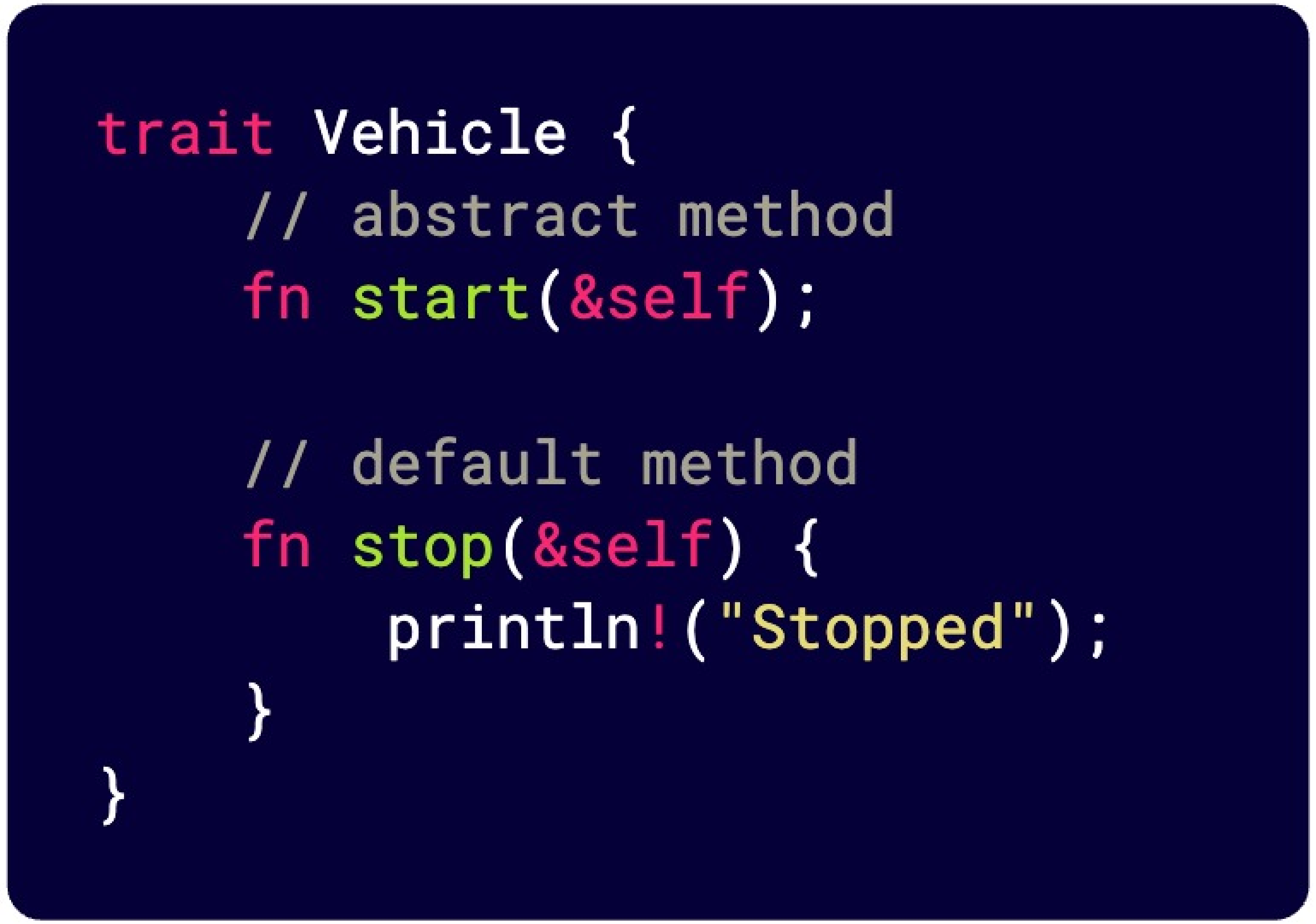


```rust
fn solve<f64: Float>(a: f64, b: f64) -> f64 {
    let a_f64 = a.to_f64().unwrap();
    let b_f64 = b.to_f64().unwrap();

    (a.powi(2) + b.powi(2)).sqrt()
}

fn main() {
    let a: f64 = 3.0;
    let b: f64 = 4.0;

    solve(a, b);
}
```



"Float" is a **trait**.
Here it is being used as a
trait bound

```rust
fn solve<T: Float>(a: T, b: T) -> f64 {
    let a_f64 = a.to_f64().unwrap();
    let b_f64 = b.to_f64().unwrap();

    (a.powi(2) + b.powi(2)).sqrt()
}
```

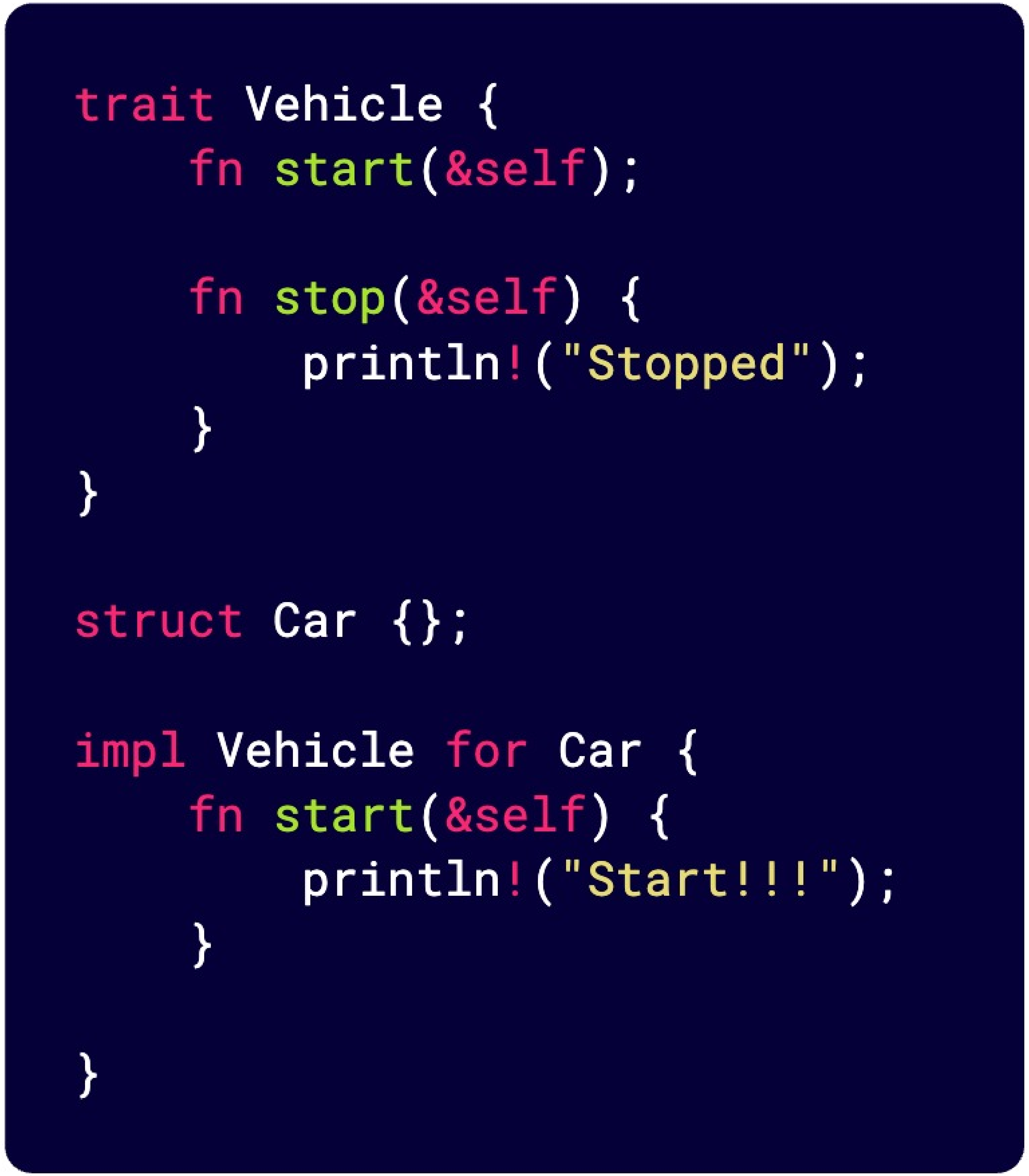


```rust
trait Vehicle {
    // abstract method
    fn start(&self);

    // default method
    fn stop(&self) {
        println!("Stopped");
    }
}
```

A trait is a set of methods

It can contain **abstract methods** which don't have an implementation

It can contain **default methods**, which have an implementation

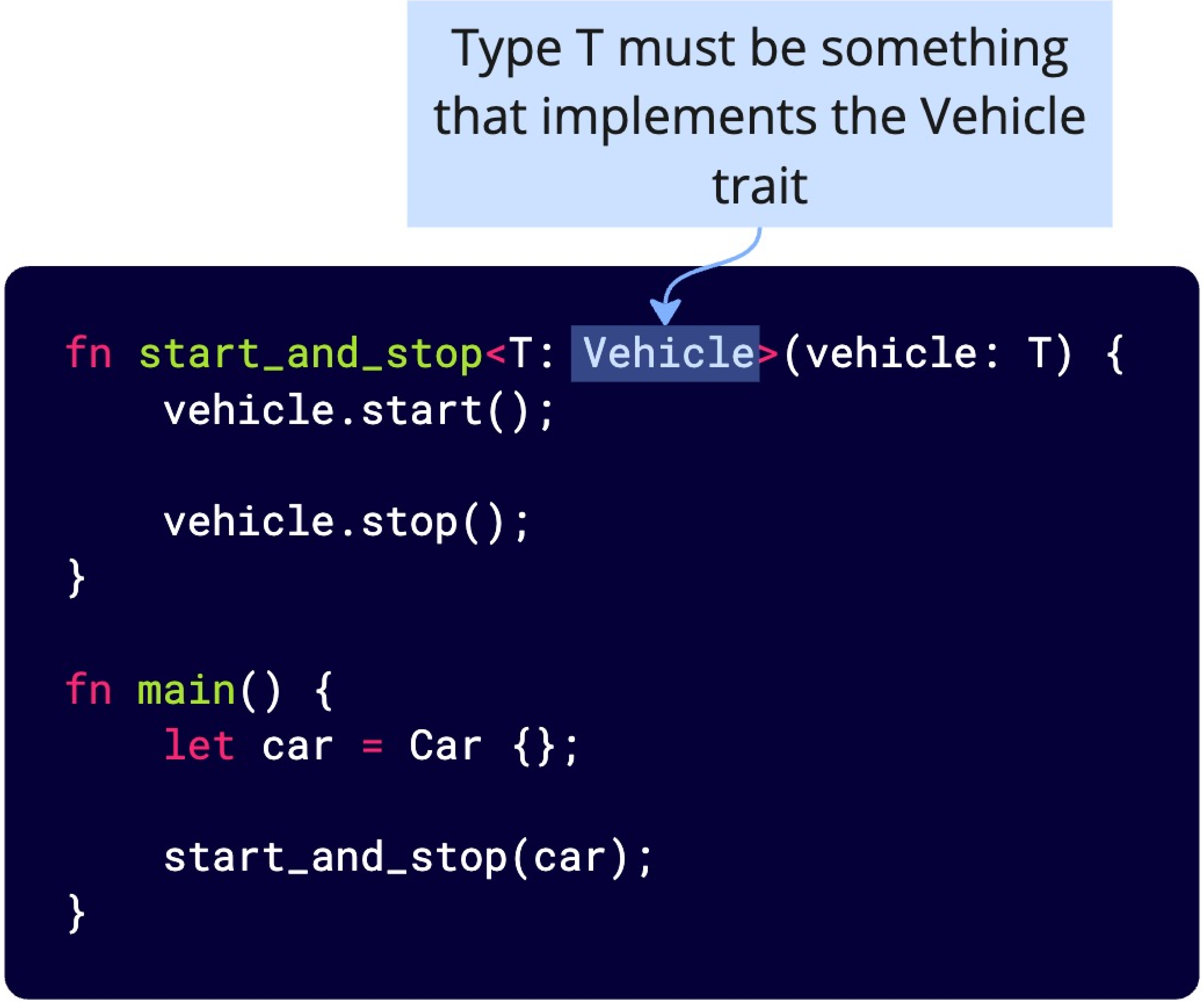


```
trait Vehicle {
    fn start(&self);

    fn stop(&self) {
        println!("Stopped");
    }
}

struct Car {};

impl Vehicle for Car {
    fn start(&self) {
        println!("Start!!!");
    }

}
```

A struct/enum/primitive can **implement** a trait

The implementor has to provide an implementation for all of the **abstract methods**

The implementor can **optionally** override the default methods



Type T must be something that implements the Vehicle trait

```
fn start_and_stop<T: Vehicle>(vehicle: T) {
    vehicle.start();

    vehicle.stop();
}

fn main() {
    let car = Car {};

    start_and_stop(car);
}
```