Don't panic

...or how should your Rust program behave when faced a critical error.

A situation which unfortunately happens too often...

```
// This function returns Order, so we are forced to return
// an Order even for incorrect inputs.
fn create_order(num_dishes: usize) -> Order {
    if num_dishes < Order::MAXIMUM_NUM_DISHES {
        Ok(Order::new(num_dishes))
    } else {
        // ??? Order::INVALID ???
    }
}
```

LET'S PANIC!

```
// This function returns Order, so we are forced to return
// an Order even for incorrect inputs.
fn create_order(num_dishes: usize) -> Order {
    if num_dishes < Order::MAXIMUM_NUM_DISHES {
        Ok(Order::new(num_dishes))
    } else {
        panic!("Too many dishes for a single Order")
        // This either unwinds the stack or aborts the program immediately.
        // (configurable in Cargo.toml of your project)
    }
}
```

Hmm, maybe let's reconsider this...

```rust
/// Error signifying that there are too many dishes to fit in an Order.
struct TooManyDishes;

// This function returns Result, so that we are not forced to return
// an Order for incorrect inputs - we just return Err.
fn create_order(num_dishes: usize) -> Result<Order, TooManyDishes> {
    if num_dishes < Order::MAXIMUM_NUM_DISHES {
        Ok(Order::new(num_dishes))
    } else {
        Err(TooManyDishes)
    }
}
```

Another common case - `Option` / `Result`

```rust
struct DivisionByZero;

fn div(dividend: i32, divisor: i32) -> Result<i32, DivisionByZero> {
    if divisor == 0 {
        // It is idiomatic to return errors after failed checks early in an imperative way, using explicit `return`.
        return Err(DivisionByZero)
    }

    // It is idiomatic to have the "happy path" (the no-errors scenario) linear and using functional syntax.
    Ok(dividend / divisor)
}
fn main() {
    let res: Result<i32, DivisionByZero> = div(2137, 42);

    // We are 100% sure division by 42 can't fail, so let's use this shorthand.
    let quotient = res.unwrap();

    // This is equivalent to:
    let quotient = match res {
        Ok(x) => x,
        Err(err) => panic!("called `Result::unwrap()` on an `Err` value: {:?}", err),
    }
}
```

Let's encode more guarantees in the type system!

```rust
use std::num::NonZeroI32;

fn div(dividend: i32, divisor: NonZeroI32) -> i32 {
    dividend / divisor // Nothing can get broken here!
}
fn main() {
    // let quotient = div(2137, 42); // This would not type check, because 42 is not NonZeroI32.

    // We have to create a NonZeroI32 instance:
    let non_zero_divisor_opt: Option<NonZeroI32> = NonZeroI32::new(42);

    // We are 100% sure 42 is not 0, so let's use this shorthand.
    let non_zero_divisor = non_zero_divisor_opt.unwrap();

    // This is equivalent to:
    let non_zero_divisor = match non_zero_divisor_opt {
        Some(x) => x,
        None => panic!("called `Option::unwrap()` on a `None` value"),
    }

    let quotient = div(2137, non_zero_divisor);
}
```

But wait, we ended up with an `unwrap()` anyway. Did we then improve at all?

Actually, yes. Now, the function (here: `div()`) with some (possibly complex) logic **only accepts** (so that the compiler verifies that in compile-time) valid arguments. This way, the function's code can be simpler (no invalid-input-related error handling). Also, it's easier to convince yourself that your number is nonzero that to make sure that it upholds all guarantees required in the docs of the function containing logic.

To `panic` or not to `panic`

Don't panic:

- if the failure is caused by bad user input - you don't want to open up a highway for DoS attackers, do you?
- if the failure only affects some task and not the program in general, e.g. when the server returned bad data for a request of one of the users; others are unaffected;
- if the failure is recoverable (there is a reasonable action to be done in such situation, e.g. give user the default value when the server can't be queried for the actual value);

To `panic` or not to `panic`

Do panic:

- if the failure is for sure caused by a bug in your code. It makes sense to inform the whole world that you wrote deficient software, by yelling at them. More seriously, this shortens the feedback loop and bugs are fixed earlier, instead of silently damaging production;
- if the failure is not recoverable (there is no hope, the program is broken, *R.I.P.*, only the famous *restart* could help here);