[[questions]]
type = "ShortAnswer"
prompt.prompt = """
Imagine using a third-party function whose implementation you don't know, but whose type signature is this:
```
fn mystery<T>(x: T) -> T {
  // ????
}
```
Then you call `mystery` like this:
```
let y = mystery(3);
```
Then, the value of `y` must be :
"""
answer.answer = "3"
context = """
The only possible function that has the signature `T -> T` is the identity function:
```
fn mystery<T>(x: T) -> T {
  x
}
```
The function could of course panic or print, but the return value can only be the input. `mystery` does not know
what type `T` is, so there is no way for `mystery` to generate or mutate a value of `T`. See [Theorems for free!](https://dl.acm.org/doi/pdf/10.1145/99370.99404) for more examples of this idea.
3 really is the correct answer!
"""
[[questions]]
id = "40ae0cfe-3567-4d05-b0d9-54d612a2d654"
type = "Tracing"
prompt.program = """
fn print_slice<T>(mut v: Span<T>) {
    while let Option::Some(x) = v.pop_front() {
        println!("{x}");
    }
}
fn main() {
    let arr = array![1, 2, 3, 4];
    print_slice(arr.span().slice(1, 3));
}
"""
answer.doesCompile = false

answer.lineNumber = 3
context = """
If a type is generic (like `T`), we cannot assume anything about it, including the ability to display it. Therefore `println!("{x}")` is invalid
because `x: @T`.
"""
[[questions]]
id = "694bb2d0-f2e6-4b0b-a3e7-2d9f9e8b3d09"
type = "Tracing"
prompt.program = """
#[derive(Drop)]
struct Point<T> {
    x: T,
    y: T
}
#[generate_trait]
impl PointImpl of PointTrait {
    fn f(self: Point<u32>) -> u32 {
        self.y
    }
}
#[generate_trait]
impl PointImplGeneric<T> of PointTraitGeneric<T> {
    fn f(self: Point<T>) -> T {
        self.x
    }
}
fn main() {
    let p: Point<u32> = Point { x: 1, y: 2 };
    println!("{}", p.f());
}
"""
answer.doesCompile = false
answer.lineNumber = 23
context = """
These definitions of `f` conflict, and there is no way for to determine which `f` should be used when `p.f()` is called. Therefore this is a compiler error.
Moreover, the generic implementation would require `T` to be droppable, as `self.y` is dropped when the function returns.
"""
[[questions]]
type = "Tracing"
prompt.prompt = """
#[derive(Drop)]
struct Wallet<T>{
    balance: T,

```
    address: W
}
fn main() {
    let mut account = Wallet{ balance: 10, address: '0xAbDeFG' };
    print!("{}", account.balance);
    print!("{}", account.address);
}
"""
answer.doesCompile = false
context = """
The code fails to compile because the struct `Wallet` is defined with a single generic type `T`, but
the `address` field uses an undefined type `W`. To fix this, you need to introduce another generic
type `W` in the struct definition to represent the type of the `address` field.  To fix the code,
you can modify the struct definition as follows:
```rust
struct Wallet<T, W> {
    balance: T,
    address: W,
}
```
"""
```