1) `no method named 'push_str' for reference '&str'`

`input` 的类型是 `&str`（**字符串切片**），它是**只读引用**，本身**不能增长/修改**，因此没有 `push_str` 方法。`push_str` 是 `String` 的方法，因为 `String` 拥有并管理一段可增长的堆内存。

怎么修：先把 `&str` 变成 `String`，再 `push_str`。

---

2) `expected 'String', found '()'`

你的函数签名是 `fn compose_me(input: &str) -> String`，但函数体最后没有返回字符串（没有尾表达式/return），所以编译器认为返回了 `()`（unit 类型），于是类型不匹配。

在标准库里有这样的实现（概念上等价）：

rust 复制 编辑

```rust
impl From<&str> for String {
    fn from(s: &str) -> String {
        // 把 &str 的字节拷贝到堆上，构造一个新的 String
    }
}
```

`String::from("hi")` 实际等价于：

rust 复制 编辑

```rust
<String as From<&str>>::from("hi")  // 返回类型明确是 String
```

所以它返回 `String`。

下面几种都生成 `String`，只是调用路径不同：

rust 复制 编辑

```rust
String::from("hi")
"hi".to_string()     // 通过 ToString，内部也会走 From
"hi".to_owned()      // 拥有化：&str -> String
"hi".into()          // 需要上下文类型是 String 才能推断
format!("{}","hi")   // 格式化产生 String
```

`to_owned()` 详解

Trait：`ToOwned`
签名：`fn to_owned(&self) -> Self::Owned`

- 典型用途：把借用的不可变视图，变成拥有所有权的可变对象（需要 **分配 & 拷贝**）。
- 常见映射：
  - `&str.to_owned()` → `String`
  - `&[T].to_owned()` → `Vec<T>`（逐元素拷贝）
  - `&Path.to_owned()` → `PathBuf`
- 对已经"拥有"的类型，`to_owned()` 通常等同于 `clone()`（因为 `Owned=Self`）。

例子

rust 复制 编辑

```rust
let s1: &str = "hi";
let s2: String = s1.to_owned(); // 分配并拷贝

let v1: &[i32] = &[1,2,3];
let v2: Vec<i32> = v1.to_owned(); // 分配并拷贝元素
```

`into()` 详解

Trait：`Into<T>`（通常由 `From<U> for T` 自动提供）
签名：`fn into(self) -> T`

- **消耗**当前值（move），转换为 **目标类型** `T`。是否分配取决于具体转换实现。
- 目标类型需要**能被推断**，否则用涡轮鱼：`expr.into::<Target>()`。

例子 1：需要分配的转换

```rust
let s: String = "hi".into();        // &str -> String（分配并拷贝）
```

例子 2：零拷贝/零分配的转换（move 即可）

```rust
let s: String = String::from("hi");
let bytes: Vec<u8> = s.into();      // String -> Vec<u8>（复用底层缓冲区，通常不再分配）
```

例子 3：目标类型由上下文决定

```rust
fn take_string(x: String) { /* ... */ }

take_string("hello".into());        // 因为形参是 String，这里推断成 &str -> String
```

这两句之所以**返回的是引用** `&str`，因为它们得到的只是**对原字符串的一段"视图/切片"**，没有新开内存、没有复制数据。

---

1) `string_slice(&String::from("abc")[0..1]);`

- `&String::from("abc")`：先把 `String` 借用成 `&String`；通过 *Deref* 自动变成 `&str`。
- `[0..1]`：对底层 `str` 做**字节区间切片**，返回 `&str` 指向原缓冲区的那一段（这里是 `"a"`）。
- 这是**零拷贝**的借用，所以类型是 `&str` 而不是 `String`。

  > 注意切片按 **字节** 计数，必须落在 UTF-8 字符边界；例如 `"你好"` 的 `"你"` 要用 `[0..3]`，`[0..1]` 会 panic。

> 顺便说明：表达式里创建的临时 `String` 会活到语句结束，因此把 `&str` 传进函数是安全的，但**不能把这个引用返回到外面**。

---

2) `string_slice(" hello there ".trim());`

- `" hello there "` 是 `&'static str`（字面量切片）。
- `trim()` **不会分配**，它只是找出首尾非空白的**下标范围**，然后返回该范围的 `&str` **子切片**。
- 仍然是对原数据的借用，所以类型是 `&str`。