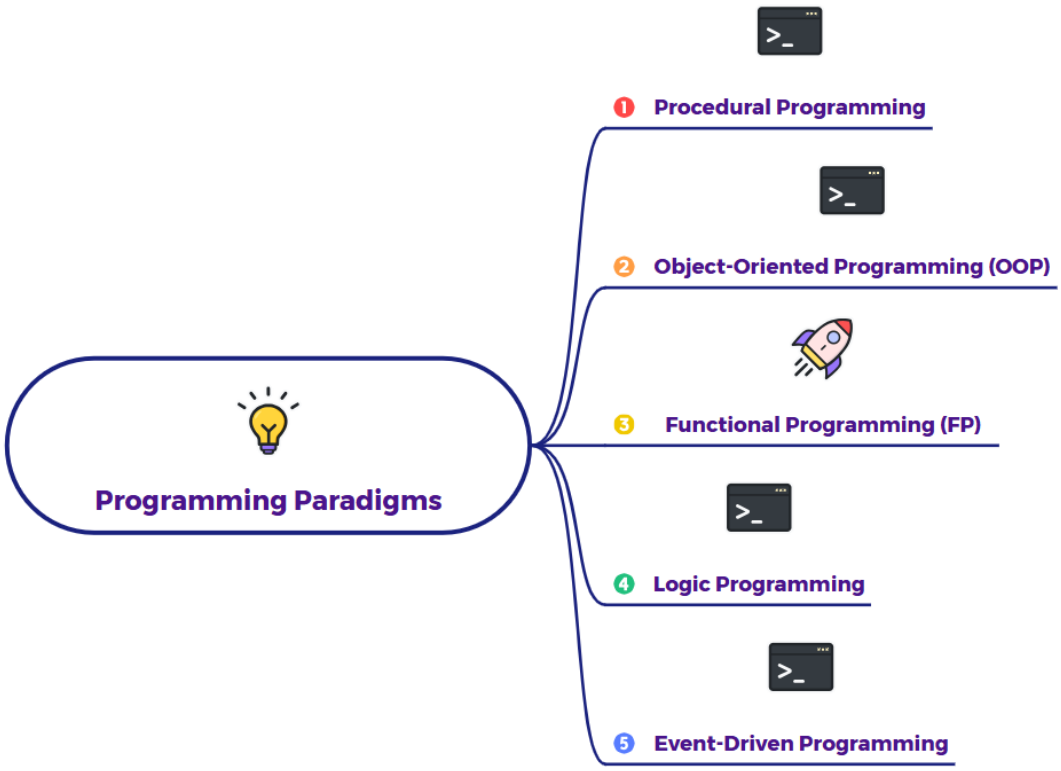
Explain functional programming with an example



The "Functional Fruit Salad" Way:

Imagine you want to make a fruit salad, but with a rule: **you never directly *change* any fruit once it's cut**. Instead, every action you perform creates a *new version* of the fruit or salad.

Here's how it works:

1. Pure Functions: The Predictable Kitchen Gadgets

- **Analogy:** Think of your kitchen gadgets as "pure functions."
    - You have a peeler gadget. If you put an apple into it, it *always* gives you a peeled apple. It never changes the peeler itself, and it never changes anything else in the kitchen.
    - You have a slicer gadget. If you put a whole peeled apple into it, it *always* gives you a bowl of sliced apple pieces. It doesn't modify the original whole apple, and it doesn't mess with your knife or cutting board.

- In Functional Programming, a **"pure function"** is like this. It takes an input, produces an output, and has no "side effects" – it doesn't change anything outside of itself, and it doesn't rely on anything external that might change. Given the same input, it always produces the same output.

2. Immutability: Never Alter the Original Fruit

- **Analogy:** When you cut an apple, you don't take the *original* whole apple and magically turn it into slices. You take the whole apple, and the *slicing action* creates a *new set of slices*. The original whole apple, conceptually, is still there (or you discard it, but you don't *transform* it in place).
- In Functional Programming, **"immutability"** means that once a piece of data (like our apple) is created, it cannot be changed. If you need a modified version, you create a *new* piece of data based on the old one. This avoids confusion about "who changed what when."

3. Function Composition: Chaining Your Gadgets

- **Analogy:** To make a perfect fruit salad, you might chain your gadgets together:
    - Peeled Apple = peeler(Whole Apple)
    - Sliced Apple = slicer(Peeled Apple)
    - Diced Mango = dicer(peeler(Whole Mango))
    - Sweetened Berries = add_honey(Washed Berries)
- Then, Final Salad = combine_fruits(Sliced Apple, Diced Mango, Sweetened Berries)
- You're taking the output of one function and feeding it directly as the input to the next, like an assembly line of pure gadgets.

Example: Making the Fruit Salad

Let's say we start with:

- Whole Apple
- Whole Mango
- Bag of Berries

Functional Steps:

1. **Peel Apple:** Use the peel() function on Whole Apple to get Peeled Apple. (peel(Whole Apple) = Peeled Apple)
2. **Slice Apple:** Use the slice() function on Peeled Apple to get Sliced Apple. (slice(Peeled Apple) = Sliced Apple)
3. **Peel Mango:** Use the peel() function on Whole Mango to get Peeled Mango.
4. **Dice Mango:** Use the dice() function on Peeled Mango to get Diced Mango.
5. **Wash Berries:** Use the wash() function on Bag of Berries to get Washed Berries.
6. **Sweeten Berries:** Use the add_honey() function on Washed Berries to get Sweetened Berries.
7. **Combine:** Use the mix_all() function on (Sliced Apple, Diced Mango, Sweetened Berries) to get Final Fruit Salad.

Key Points of Functional Programming in this Example:

- **No Side Effects:** None of your gadgets (functions) ever mess up another part of the kitchen or change their own settings. They just take an input and give an output.
- **Immutability:** You never modified the Whole Apple. You simply created a Peeled Apple and then a Sliced Apple from it. The original (conceptually) remains untouched.
- **Chainable Operations:** You can easily string together operations (dice(peel(mango))) because each one reliably produces a new, clean output.

Why is this "Functional" way useful?

- **Easier to Understand:** Because functions are predictable and don't have hidden side effects, it's easier to know what they'll do.
- **Easier to Test:** You can test each gadget (function) in isolation, knowing it will always behave the same way.

- **Safer for Multiple Hands:** If you have multiple people making parts of the salad at the same time, they won't accidentally interfere with each other's work or change shared ingredients because everyone is always creating *new* versions of things, not altering existing ones. This is great for modern computers with many "cores" working simultaneously.

In short: Functional Programming is like baking using only predictable, single-purpose kitchen gadgets (pure functions) that never change anything around them, and every step produces a new, fresh version of your ingredients (immutability), rather than modifying the original ones.