Functional Programming

Introduction

- Functional programming decomposes a problem into a set of functions
- only take inputs and produce outputs, and don’t have any internal state that affects the output produced for a given input.
- FP defines computation using expressions and evaluation—often encapsulated in function definitions.
- It avoids the complexity of state change and mutable objects.
- This tends create programs more succinct and expressive.
- Based on mathematical functions.

Introduction

- Python has numerous functional programming features.
- It is not a purely functional programming language.
- **Exploratory Data Analysis** (**EDA**) algorithms are good examples of FP.
- FPL are designed to handle symbolic computation and list processing applications.
- E.g - Lisp, Python, Erlang, Haskell, Clojure, etc
- FP is considered opposite of OOP.
- Objects contains internal state + collection of method calls - let modify state, FP wants to avoid state changes.

Introduction

- Functional style discourages functions with side effects
- Side effect - modify internal state or make other changes that aren’t visible in the function’s return value.
- Functions that have no side effects at all are called **pure functional**.
- Avoiding side effects means not using data structures that get updated as a program runs.
- Every function’s output must only depend on its input.

Imperative vs Programming Functional

- Distinguishing feature between Imperative and Function programming is the concept of state.
- *Imperative* programming is a programming paradigm that uses statements that change a program's state.
- In IP state of the computation is reflected by the values of the variables in the various namespaces.
- Value of the variables establish the state of a computation.
- Each kind of statement makes a well-defined change to the state.
- A language is imperative because each statement is a command, which changes the state in some way.

Imperative vs Programming Functional

- In FP, state replace with a simpler notion of evaluating functions.
- Each function evaluation creates a new object from existing objects.
- FP is a composition of a function:
  - We can design lower level functions - easy to understand.
  - Higher-level compositions - easier to visualize.
- FP are relatively succinct, expressive, and efficient when compared to imperative.

Procedural

- Line by line
- Heavy use of statements
- Heavy use of expression
- Long function

Functional

- Little use of statements
- Heavy use of expression
- Single line of function

Recursion

- Recursion is elegant
- Difficult to follow
- Limited by Python max recursion depth
- E.g. notebook

Two Groups

- **Pure FL** – support only the functional paradigms. For example – Haskell.
- **Impure FL** – support the functional paradigms and imperative style programming. For example – LISP.

Characteristics

1. Concept of mathematical functions - conditional expressions.
2. Limit the use of for loops - directly use the functions and functional calls
3. Good support for recursion
4. Avoid state representation
5. Data are immutable
6. First class functions
7. High-order functions and lazy evaluation
8. Like OOP, support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.
9. Pure functions - there should not be any side-effects in them

Advantages

1. Bugs-Free Code
    - does not support state
    - no side-effect results
    - can write error-free codes
2. Efficient Parallel Programming .
3. Codes support easy reusability and testability.
4. Efficiency − independent, run concurrently
5. Supports nested functions
6. Lazy evaluation

Disadvantages

1. Requires a large memory space.
2. As it does not have state, need to create new objects every time
3. Heavy use of recursion - stack is very much finite
4. Much garbage will be generated in functional programming due to the concept of immutability
5. Using only immutable values can potentially lead to performance problems, including RAM use and speed.

Functional Programming	OOP
Uses Immutable data	Uses Mutable data
Follows Declarative Programming Model	Follows Imperative Programming Model
Focus is on: “What you are doing”	Focus is on “How you are doing”
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side effects.
Flow control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements
It uses "Recursion" concept to iterate Collection data	It uses "Loop" concept to iterate Collection Data. e.g. for-each loop in Java
Execution order of statements is not so important	Execution order of statements is very important
Supports both "Abstraction over Data" and "Abstraction over Behavior"	Supports only "Abstraction over Data"

Referential Transparency in Math

- Referential transparency is the property of expressions that can be replaced by other expressions having the same value without changing the result in any way.
  - x = 2 + (3 * 4)
- replace the subexpression (3 * 4) with any other expression having the same value without changing the result (the value of x).
  - x = 2 + 12

Referential Transparency in Programming

- Referential transparency applies to programs.
- As programs are composed of subprograms, which are programs themselves, it applies to those subprograms, too.
- Subprograms may be represented, among other things, by methods.
- That means method can be referentially transparent - call to method is replaced by its return value.

Referential Transparency in Programming

```
int add(int a, int b) {
    return a + b
}

int mult(int a, int b) {
    return a * b;
}

int x = add(2, mult(3, 4));
```

Referential Transparency

- R*eferential transparency* is defined as the fact that an expression, in a program, may be replaced by its value (or anything having the same value) without changing the result of the program.
- A function is said to be referentially transparent if its invocation can be substituted by the return value in a program without impacting the program.
- This implies that methods should always return the same value for a given argument, without having any other effect.
- Referential transparency referred to a function, indicates that we can determine the result of applying that function only by looking at the values of its arguments.
- Such functions are called pure functions.
- An expression that is not referentially transparent is called referentially opaque.

Referential Transparency

```
counter = 0

def foo(x):
  global counter

  counter += 1
  return x + counter
```

is not referentially transparent, in fact calling

```
foo(x) + foo(x)
```

and

```
2 * foo(x)
```

Referential Transparency

- Modularization
- Ease of debugging - Functions are isolated, they only depend on their input and their output, so they are very easy to debug.
- Parallelization - Functions calls are independent.
- Concurrence - With no shared data, concurrence gets a lot simpler.
- Idempotence - Same results regardless how many times function is called.