ADVANCED FUNCTIONAL THINKING

DR SURBHI SARASWAT
SCHOOL OF COMPUTER SCIENCE

DECLARATIVE VS IMPERATIVE PROGRAMMING

- Imperative programming:
  - Algorithms as a sequence of steps
  - Often used synonymously: procedural programming
    - emphasizing the concept of using procedure calls (functions) to structure the program in a modular fashion
  - OOP is usually considered a subcategory

In this, programs specify how it is to be done
It simply describes the control flow of computation.
In imperative programming, the programmer is responsible for optimizing the code for performance.
In imperative programming, variables can be mutable.
OOPs is considered a subcategory.

- Declarative programming:
  - Program describes logic rather than control flow
  - Program describes “what” rather than “how”
  - Aims for correspondence with mathematical logic
  - FP is usually considered a subcategory

In this, programs specify what is to be done.
It simply expresses the logic of computation.
In declarative programming, the system optimizes the code based on the rules and constraints specified by the programmer.
In declarative programming, variables are typically immutable.
FP is usually considered a subcategory

FUNCTIONAL PROGRAMMING

- C language
    - No classes/objects
    - Imperative (its instructions are shaped as commands like do_things())
    - Procedural
        - Void do_things(){ - - - - }
- Java Language
    - Object oriented
        - Instructions inside their methods are still commands

FUNCTIONAL PROGRAMMING

- Functional languages
    - We have expressions
    - No concatenations of commands
        - Eg. do_thing1(), do_thing2();
    - It is a composition of functions
        - print(sum(2, exp(1,2)))

FUNCTIONS

- Single valued
- Maps between sets
- A function called with the same parameters twice should result in the same output.
- A procedure can be a function, Eg.
  - def square(x): return x*x
    - The function takes integer input and returns integer o/p
  - Mathematically, $f : \mathbb{Z} \to \mathbb{Z} \text{ such that } x \mapsto x^2$ where $\mapsto$ refers as “Maps to”

$f : x \mapsto x^2$ or $f : f(x) = x^2$

FUNCTIONS

- Example, let us define a function that returns the sum of the squares of two numbers.
- def sum_of_squares(a,b):
  return (a**2) + (b**2)
  - Or
- def sum(a,b):
  return a + b

```
def square(x):
    return x*x
def sum_of_squares(a,b):
    x= square(a)
    y= square(b)
    return sum(x,y)
```

FUNCTIONS

- To convert to functional instead of concatenating commands, let us use the composition of functions.

- def sum(a,b):

  return a + b

  def square(x):

  return x*x

  def sum_of_squares(a,b):

  return sum(square(a) + square(b)

- But the return statement is still a command.
- For this, we will use Anonymous functions (without names)
  - $(\text{parameters}) \mapsto f(\text{parameters})$
  - $\text{sum\_of\_squares} = (\text{a,b}) \mapsto \text{sum}(\text{square}(\text{a}), \text{square}(\text{b}))$

FUNCTIONS

- $\mathrm{sum\_of\_squares} = (\mathrm{a,b}) \mapsto \mathrm{sum}(\mathrm{square}(\mathrm{a}), \mathrm{square}(\mathrm{b}))$
- Note that, " = " is same as the assignment operator that was used to assign values to variables.
- This illustrates that anonymous functions can be assigned to variables.

- Just as numbers can be input and output of a function, similarly anonymous functions can be input and output of other functions.
- Eg. $f : f(x \mapsto x^2)$

- Eg. $foo = (y) \mapsto (x \mapsto x * y)$
  - it takes a number 'y' and returns a function f such that f(x) = x*y
  - $foo(2) = (x \mapsto x * 2)$
  - $(foo(2))(5) = (5 * 2) = 10$

FUNCTIONS

- Several languages allow us to use this concept of functional paradigm.
- In python we use, lambda a, b : expression
- Or in JavaScript we can use (a, b) => expression

- Lambda functions

$$(a, b) \mapsto sum(square(a), square(b))$$

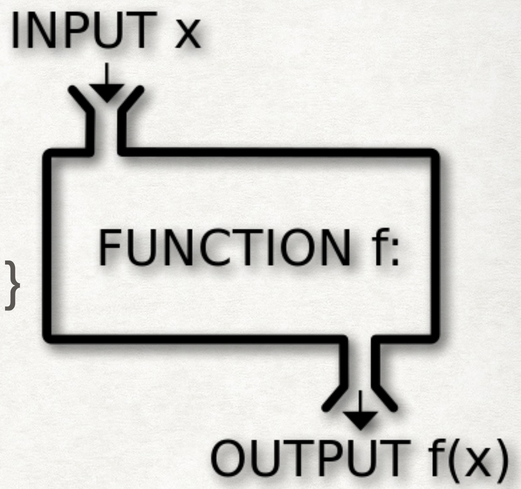
$\lambda(a, b).sum(square(a), square(b))$ equivalant to: $\mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Z}$

$\lambda(a).(\lambda(b).sum(square(a), square(b))$ equivalant to: $\mathbb{Z} \mapsto (\mathbb{Z} \mapsto \mathbb{Z})$

PURE FUNCTIONS

- In computer programming, a pure function is a function (a block of code) that has the following properties:
  - It **always returns the same result if the same arguments are passed**. It does not depend on any state or data change during a program's execution. Rather, it only depends on its input arguments.
  - Also, a pure function does not produce any observable side effects such as network requests or data mutation, etc.
- Thus a pure function is a computational analogue of a mathematical function.
- Since pure functions have identical return values for identical arguments, they are well suited to unit testing.

PURE FUNCTIONS



- Let's see the below JavaScript Functions:
  - `function calculateGST( productPrice ) { return productPrice * 0.05;}`
  - ```
    var tax = 20;
    function calculateGST( productPrice ) { return productPrice * (tax / 100) + productPrice;  }
    ```

- The first function will always return the same result if we pass the same product price. In other words, its output doesn't get affected by any other values/state changes. So we can call the "calculate GST" function a Pure Function.

- The Second function is not a pure function as the output is dependent on an external variable "tax". So if the tax value is updated somehow, then we will get a different output though we pass the same productPrice as a parameter to the function.
    ```

PURE FUNCTIONS

- It should **not modify any non-local state**. It means the function should not manipulate anything other than the data stored in the local variables declared within the function.
- The function should **not have any side effects**, such as reassigning non-local variables, mutating the state of any part of code that is not inside the function or calling any non-pure functions inside it.
- If a pure function calls a pure function, this isn't a side effect and the calling function is still considered pure. (Example: using Math.max() inside a function)
- Below are some side effects (but not limited to) that a function should not produce in order to be considered a pure function –
  - Making an HTTP request
  - Mutating data
  - Printing to a screen or console
  - Math.random()
  - Getting the current time

```
object GFG
{
    def main(args: Array[String])
    {
        def square(a:Int) = {
            var b:Int = a * a;
            println("Square of the number is " + b);
            println("Number is " + a);
        }
        square(4);
    }
}
```

WHY PURE FUNCTIONS

- Pure functions make the code more **readable**, **testable**, and **performant.**
- The application that uses pure functions is easier to reason about and construct.

A function is called pure if

1. Same arguments $\Longrightarrow$ same return value ($x = y \implies f(x) = f(y)$)
2. The evaluation has no side effects (no change in non-local variables, ...)

Which of the following functions are pure?

```
1  def f1(x):
2      return x**2
3
4
5  def f2(x):
6      print(x)
7      return x**2
8
9
10 global y = 0
11
12
13 def f3(x):
14     y += 1
15     return x + y
```

```
18 def f4():
19     return int(input()) + 1
20
21
22 def f5(lst: List):
23     lst[0] = 3
24     return lst
```

Answer: `f1` is pure; `f2`, `f3`, `f5` violate rule 2; `f4`, `f3` violate rule 1.