CS2023 - Module 2

Introduction to Functional Programming
Higher-Order Functions

Dr. Fred Annexstein

Part 1 - Functional Programming

- Python allows for multiple approaches to programming
- Functional programming FP style of programming is an important discipline
- FP avoids side-effects - avoids impacting state changes
- Print statements and other I/O are examples of programming with side effects
- FP significantly differs from Object-oriented programming
- **FP** decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output.
- Well-known purely functional languages include the ML, OCaml, and Haskell.

Pure Functions vs. Non-Pure Functions

With every function we need to pay attention to the
domain (the values for which the function is defined) and the range (the values that are potentially returned)!

A Pure function - ONLY produces a return value (with no side effects) and always evaluates to the same result, given the same argument value(s).

A Non-Pure function - may produce some side effects (like printing to the screen) and may not always evaluate to the same result, given the same argument values.

Advantages of FP

There are theoretical and practical advantages to the functional style of programming:

- Formal provability -A theoretical benefit is that it's easier to construct a mathematical proof that a functional program is correct. This is different from testing a program on numerous inputs and concluding that its output is usually correct,
- Modularity and Ease of debugging and testing.- Functional Programs are more modular and programs composed of small functions are easier to read and to check for errors.
- Composability - Functional programs are often created by arranging existing functions in a new configuration and writing a few functions specialized for the current task.
-

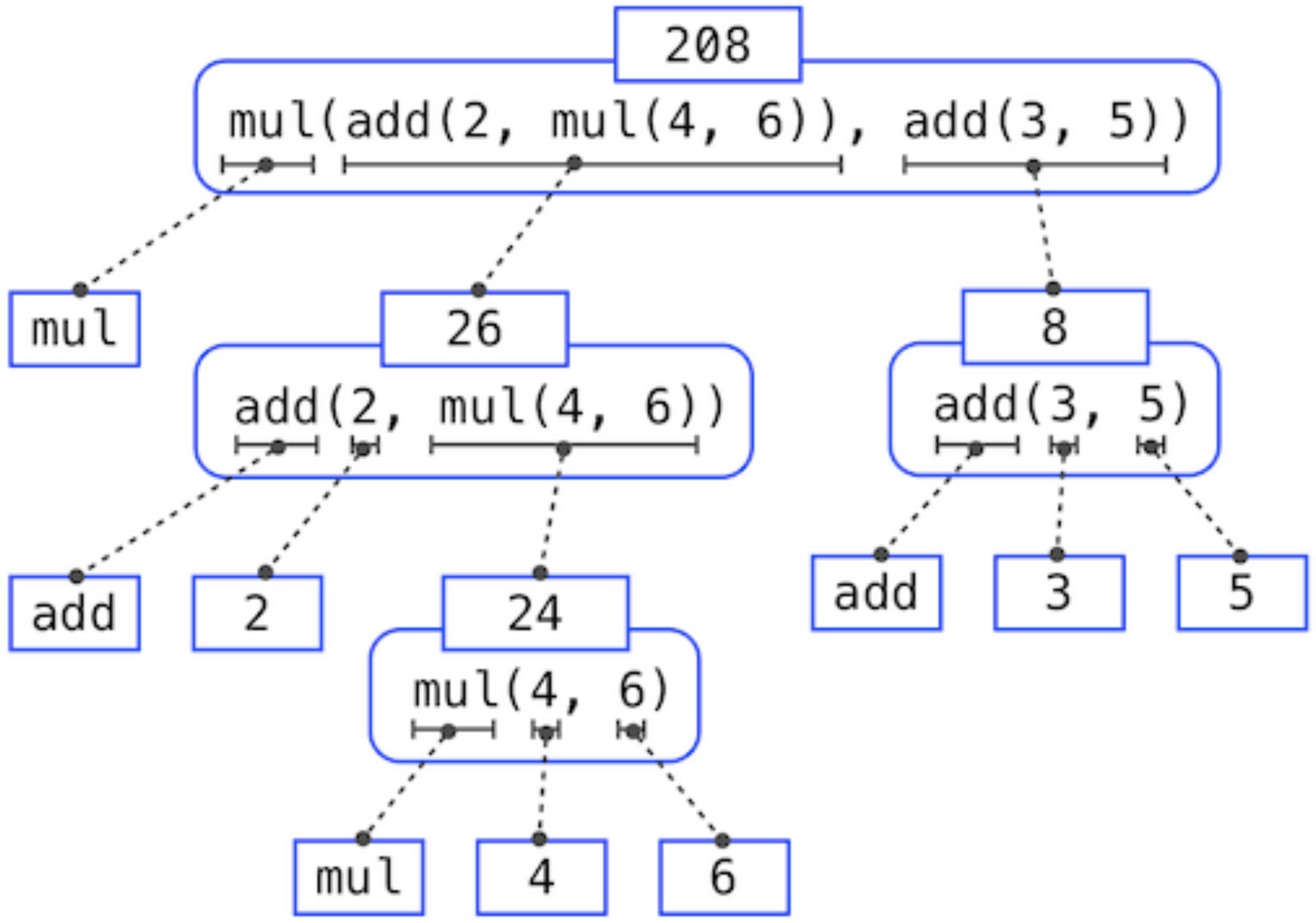
Part 2 - Higher-Order Functions

Python Primitive Expressions
Expressions describe a computation and evaluates to a value.
A primitive expression is a single evaluation step: you either look up the value of a name or take the literal value. For example, numbers, function names and strings are all primitive expressions.

Call Expressions
Call expressions are expressions that involve a call to some function. Call expressions are just another type of expression which are called compound expressions. The point is that they are calling some function with arguments and returning some value from them.

Evaluation of a function call expression:
1. Evaluate the operator and all the operands.
2. Apply the function, (the value of the operator), to the arguments, (the values of the operands).

Expression Trees And Environments



Remember that an expression describes a computation — and evaluates to a value. An expression tree helps us to visualize the break down an expression into smaller "subexpressions" and to understand how they combine with one another to return a single value.

An environment is a set of bindings of variable names to values. Every time we encounter a variable name, we look up its value in the current environment. Note that we only lookup the value when we see/read the variable, and so we'll lookup the value of x only after we've already defined x. Otherwise this will be an name error - as name is not yet defined.

Evaluating Expressions and Environments

An environment frame is a theoretical box that contains all the bindings from variables to values. An environment frame can “extend” another frame; that is, this frame can see all bindings of the frame it extends. We represent this by drawing an arrow from an environment frame to the frame it is extending.

Names (which may occur in multiple frames) are resolved by starting in the current extension frame and follows the arrows back to the extended frame. If the name cannot be resolved by reaching the global environment frame, then it can’t be resolved, and thus it’s a name error! Remember that the global environment is the only environment that extends nothing, and we therefore stop there.

Here is how you (as the interpreter) go about using environment frames:
1. Start with the frame labeled global environment. This box starts out with bindings for all the built-in functions like +, abs, max, etc.
2. Set the current frame to be the global environment. The current frame is just the environment that you (as the interpreter) are in at the current moment. You always start off an interpreter in the global environment.
3. Evaluate each of your expressions, one by one, by evaluating primitive expressions and resolving name bindings.
4. Each function calls extend the current environment with a new frame.

Function call expressions

In a function call first evaluate all the arguments as normal. Then, create a new box. Set this new box as the new current frame, and add all the parameters into the box, and have them point to the argument values you evaluated. Evaluate the body of the function in the current frame. Once done, go back to the frame from which you made the function call.

Higher-Order Functions

A Higher Order Function is a function which takes other functions as arguments or returns a function (or both).

Functions as arguments:
Suppose we'd like evaluate a function on each natural number from 1 to n and print the result as we go. (For example, you want to square each number or take the squareroot of every number up to n.)

It is nice to generalize functions of this form into something more convenient? When we pass in the number n we could also pass in the particular function. To do that, we define a higher order function. The function takes in the function you want to apply to each element as an argument, and applies it to each of the n natural numbers starting from 1.

Higher-order functions that return functions

Functions as return values
Sometimes we wish to write a function so that given some set of arguments returns a function that will do something with those arguments when it is called. The key here is that your function is supposed to return a function and not evaluate anything yet.

A classic example of this is a “dispatch function”, which takes several arguments and links a function to each one through a “registration” process. The registration will be a higher-order function that returns the dispatch function, as the example shows.

A Dispatch Function

A dispatch function gives a programmer the flexibility to use a single named function to operate differently based on single special argument, which we call an option. Each option is associated with a different function used to process the values(s). The following simple example shows how a dispatch function can be created with 2 options (associated with 2 functions). This dispatch function is created using a higher-order function that registers the associations and returns the dispatch function. (see also Q5 in Lab 2)

```python
def dispatch_function(option1, f1, option2, f2):
    """Takes in two options and two functions. Returns a function that takes in
    an option and value and calls either f1 or f2 depending on the given option.

    >>> func_d = dispatch_function('c to f', c_to_f, 'f to c', f_to_c)
    >>> func_d('c to f', 0)
    32.0
    >>> func_d('f to c', 68)
    20.0
    >>> func_d('blabl', 2)
    AssertionError
    """
    def func(option, value):
        assert option == option1 or option == option2
        if option == option1:
            return f1(value)
        else:
            return f2(value)
    return func
```

Module #2 - Deliverable #1

Download the the folder for the Lab 2 on Higher Order Functions.

Extract the files in the folder if needed and open the index.html file in you browser. The Lab sheet should be formatted with a Red Banner on top. Turn in a lab2.py with your answers to the Required Questions.