LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 2: FUNCTIONAL PROGRAMMING

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

2. FUNCTIONAL PROGRAMMING

BY FRED ANNEXSTEIN

CHAPTER 2 OVERVIEW

In this chapter we will be covering an introduction to the practice of functional programming. Functional programming gives you capabilities to use abstractions and compositions. Abstractions allow you to handle complex data in an intuitive way, and compositions allow you to develop software in a top-down or bottom-up fashion to meet your needs. We cover the basics of defining or constructing functions and the role of higher-order functions in software development.

CHAPTER 2 OBJECTIVES

By the end of this chapter, you will be able to….

- Identify the role of functional programming in the software development process.
- Justify the concept of pure functions and function objects in Python.
- Understand the function call stack and scope rules used in Python.

- Understand the syntax of call expressions and lambda expressions
- Understand and apply the concept of higher-order functions.
- Formulate a dispatch-function employing higher-order functions.
- Gain experience with programming in a functional style.

FUNCTIONAL PROGRAMMING

Python allows for multiple approaches to programming. The functional-programming (FP) style of programming is an important discipline to master. FP significantly differs from Object-oriented programming (OOP) which we will cover in Chapter 6. FP prevents many errors from creeping into code by avoiding coding side-effects. Such side-effects often result in unexpected state changes.

As the computational tasks you perform get more complicated, your code can become harder to read, debug and

modify, and therefore more likely to contain errors. Specifying how you want the code to work can become complex. A functional-style of programming lets you simply say what you want to do. It hides many details of how to perform each task. Typically, library code handles the how for you. As you'll see, this can eliminate many errors.

Simply stated, FP style allows you to decompose a problem into a set of functions. Ideally, the functions you write will only take inputs and produce outputs, and don't have any internal state that may impact the rest of the computation. Python works well with this style of programming, however it is not strictly a purely functional programming language. Well-known purely functional languages include the ML, OCaml, and Haskell.

PURE FUNCTIONS VS. NON-PURE FUNCTIONS

With every function you write you need to pay attention to the domain, that is the values for which the function is defined. And

also the range, that is the values that are potentially returned by the function.

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, such as printing to the screen, and may not always evaluate to the same result, given the same argument values.

There are theoretical and practical advantages to using only pure functions when practicing the functional style of programming. For example it is easier to construct a mathematical proof that a functional program is correct. Proofs of correctness are different from testing a program on numerous inputs and concluding that its output is usually correct. Functional programs typically are more modular, that is programs are composed of small functions are easier to read and to check for errors. Functional programs are often created by arranging existing functions

in a new configuration and writing a few functions specialized for the current task.

EXPRESSIONS

We begin with python expressions, which 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 and strings are all primitive expressions. Call expressions are simply expressions that involve a call to some function using pair of parentheses. Call expressions call some function with arguments and return some value from them.

EVALUATION OF A CALL EXPRESSION

The evaluation of a function call expression proceeds as follows:

1. First python evaluates the operator and all the operands.

2. Next python applies the function which is the value of the operator to the arguments which are the values of the operands.

FUNCTION CALL STACK

To understand how Python performs function calls, consider a collection of data known as a stack. Stacks can be thought of like a pile of dishes. When you add a dish to the pile, you place it on the top. Similarly, when you remove a dish from the pile, you take it from the top. Stacks are known as last-in, first-out (LIFO) data structures— the last item pushed or placed) onto the stack is the first item popped or removed from the stack.

Similarly, the function-call stack supports the function call/return mechanism by pushing and popping values local to a function. Eventually, each function must return program control to the point at which it was called. For each function call, the interpreter pushes an entry called a stack frame (or an activation record) onto the stack. This entry contains the return location that the called function needs so it can return

control to its caller. When the function finishes executing, the interpreter pops the function's stack frame, and control transfers to the return location that was popped.

ENVIRONMENT FRAMES

An environment is similar to a namespace, which 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 throw an exception called a name error - since the name is not yet defined.

An environment frame is like a box that contains a set of bindings from variables names to values. An environment frame can "extend" another frame; that is, a 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 may occur in multiple frames and are resolved to a value by starting in the current extension frame and following 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 we have a name error! Remember that the global environment is the only environment that extends nothing, and we therefore stop there.

HOW INTERPRETERS USE ENVIRONMENT FRAMES

1. Interpreters start off with the frame labeled global environment. This box starts out with bindings for all the built-in functions like +, abs, max, etc.

2. Interpreters have an active frame which is set initially to the global environment.

3. Interpreters evaluate each expression, one by one, by evaluating primitive expressions and resolving name bindings.

4. Each function call extends the active environment with a new frame.

When a function is called python first evaluates all the arguments as normal. Then, a new frame box is created as the new current frame. All the parameters are put into the frame box pointing to the argument values previously evaluated. Python then evaluates the body of the function in the current active frame. Once done, the active frame goes back to the frame from which the function call was made.

SCOPE RULES AND GLOBAL VARIABLES

A function may have a local variable name and it has local scope. A local variable name can be used only inside the function that defines it. The local variable name is only in scope only from its definition to the end of the function's block. The name goes out of scope when the function returns to its caller.

Names defined outside any function have global scope. Variables with global scope are known as global variables.

Identifiers with global scope can be used in a interactive session anywhere after they are defined.Y ou can access a global variable's

value inside a function, however, by default, you cannot modify a global variable in a function—since whenever you first assign a value to a variable within a function block, Python creates a new local variable.

To modify a global variable in a function's block, you must use a global statement to declare that the variable is defined in the global scope and not within the local function.

```
def modify_global_var():
    global x
    x = ‘local x'
    print('x printed from modify_global:', x)

>>> modify_global()
x printed from modify_global: local x

>>> x = ‘global x’
>>> modify_global()
x printed from modify_global: local x
```

HIGHER ORDER FUNCTIONS

A Higher Order Function is a function which takes other functions as arguments or returns a function (or both). Suppose we'd like to evaluate one function on each natural number from 1 to n and print the result as we

go along. For example, say we want to square every number from 1 up to n. We can generalize functions of this form into something more convenient. When we pass in the number n as an argument we could also pass in the particular function at the same time. 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.

```
def square(x):
    return x * x

def apply(func, n):
    result = func(1)
    for i in range (1,n+1):
        result = func(i)
        print(result, end = " ")

>>> apply(square,10)
1 4 9 16 25 36 49 64 81 100
```

Python functions can also return functions. 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 (higher order) function is designed

to return a function and not evaluate anything yet.

LAMBDA EXPRESSIONS

For simple functions it is common practice in Python to use a lambda expression (or simply a lambda) to define the function inline where it's needed— typically as it's passed to another function.

A lambda expression is an anonymous function—that is, a function without a name.

```
>>> apply(lambda x: x*x , 5)
1 4 9 16 25
```

Here we pass to the function apply a lambda expression, rather than the function name square.

A lambda begins with the lambda keyword followed by a comma-separated parameter list, a colon (:) and an expression. In this case, the parameter list has one parameter named x. A lambda implicitly returns its expression's value, which in this case is the square of x. Note that any number of arguments and any simple expression

involving those arguments may be used within the lambda expression.

DISPATCH FUNCTIONS

A dispatch function is a classic example of a higher order function that returns a function. Dispatch functions takes several arguments and links a function to each one through a registration process. 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 each associated with a unique functions. The dispatch function is designed to register each of the option associations and returns a single function with two arguments, one for the option and one for the value the option function should be applied to. In the following code note how a function is defined and then returned in the dispatch function.

```
def f_to_c(fahrenheit):
    return (fahrenheit - 32) * 5 / 9
def c_to_f(celsius):
    return (celsius * 9) / 5 + 32

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
```

The next example goes a step further by showing how a dispatch function can be created by using a separate registration function in which 2 or more options can each be associated with a unique functions. The dispatch function then defines a collection of functions and calls the registration function

to make the option associations and return the single function.

In the following example, we create a list (a tuple of pairs) called an arglist which we will use in a for loop to repeat the execution of our dispatch function mydispatch.

```
def register(status1, f1, status2, f2):
  def helper(status,arg):
    assert status == status1 or
      status == status2, "status Not Registered"
    if status == status1:
      return f1(arg)
    elif status == status2:
      return f2(arg)
  return helper

def mydispatch():
  def say_hello(name):
    return "Aloha " + name
  def say_bye(name):
    return "Ciao " + name
  dispatch = register("Say Hi", say_hello,
"Say Bye", say_bye)
  return dispatch


arglist = (("Say Hi", "Alonzo"),
            ("Say Bye", "Maria"),
            ("No,no, register", "Bob"))

for a in arglist:
   print(mydispatch()(*a))


Output:
Aloha Alonzo
Ciao Maria
```

```
Traceback (most recent call last):
AssertionError: status Not Registered
```

There are several functional programming ideas in the code example above that we will review. The function *mydispatch* is a function that returns a function when called. The expression *mydispatch()* is a call expression that returns that function. Calling the function *mydispatch()(a)* with the single argument a will not work and generates an error, since a is a single object (a tuple).

Calling *mydispatch()(*a)* with the * notation will first unpack the tuple a into two arguments and thus matches the number of arguments expected and will thus cause no error. The functions *say_hello* and *say_bye* defined within *mydispatch* are function objects that can be passed as arguments to register just like any other object.

CHAPTER 2 EXERCISES

Homework #2: 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 worksheet should be formatted with a Red

Banner on top. Turn in a file lab2.py with your answers to the Required Questions