Computer Programming I

Binnur Kurt, PhD
BAU Bahçeşehir University
binnur.kurt@rc.bau.edu.tr

MODULE 8

FUNCTIONAL PROGRAMMING

Function-Call Stack

- To understand how Python performs function calls, consider a data structure known as a stack.
  - Stacks are known as last-in, first-out (LIFO) data structures
  - The last item pushed onto the stack is the first item popped from the stack.
- The function-call stack supports the function call/return mechanism.
  - 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.

Local Variables and Stack Frames

- Most functions have one or more parameters and possibly local variables that need to:
  - exist while the function is executing,
  - remain active if the function makes calls to other functions, and
  - "go away" when the function returns to its caller.
- A called function's stack frame is the perfect place to reserve memory for the function's local variables.
  - That stack frame is pushed when the function is called and exists while the function is executing.
  - When that function returns, it no longer needs its local variables, so its stack frame is popped from the stack, and its local variables no longer exist.

Functional-Style Programming

- Like other popular languages, such as Java and C#, Python is not a purely functional language.
- Rather, it offers “functional-style” features that help you write code which is
  - less likely to contain errors,
  - more concise and easier to read, debug and modify.
- Functional-style programs also can be easier to parallelize to get better performance on today’s multi-core processors

What vs. How

- As the tasks you perform get more complicated, your code can become harder to read, debug and modify, and more likely to contain errors.
- Specifying how the code works can become complex.
- Functional-style 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

What vs. How

- Consider the for statement in many other programming languages.
- Typically, you must specify all the details of counter-controlled iteration:
  - a control variable,
  - its initial value,
  - how to increment it and a loop-continuation condition that uses the control variable to determine whether to continue iterating.
- This style of iteration is known as ***external iteration*** and is error-prone.
  - you might provide an incorrect initializer, increment or loop-continuation condition.
  - External iteration mutates (that is, modifies) the control variable, and the for statement's suite often mutates other variables as well.
- Every time you modify variables you could introduce errors.

What vs. How

- Functional-style programming emphasizes immutability.
  - It avoids operations that modify variables' values.
- Python's **`for`** statement and **`range`** function hide most counter-controlled iteration details.
  - You specify what values **`range`** should produce and the variable that should receive each value as it's produced.
  - Function **`range`** knows how to produce those values.
  - The **`for`** statement knows how to get each value from **`range`** and how to stop iterating when there are no more values.
- Specifying what, but not how, is an important aspect of ***internal iteration***—a key functional-style programming concept.
  - The Python built-in functions **`sum`**, **`min`** and **`max`** each use ***internal iteration***

Pure Functions

> In pure functional programming language you focus on writing pure functions.
> A pure function's result depends only on the argument(s) you pass to it.
> A pure function always produces the same result given argument (or arguments).

Pure Functions

```
In [1]: values = [4, 8, 15, 16, 23, 42]

In [2]: sum(values)
Out[2]: 108

In [3]: sum(values)
Out[3]: 108

In [4]: sum(values)
Out[4]: 108
```

Filter, Map and Reduce

```
In [5]: numbers = [4, 8, 15, 16, 23, 42]

In [6]: def is_odd(number):
            return number % 2 == 1

In [7]: list(filter(is_odd, numbers))

Out[7]: [15, 23]
```

Filter, Map and Reduce

```
In [9]: [number for number in numbers if is_odd(number)]

Out[9]: [15, 23]
```

Using a lambda Rather than a Function

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

```
In [10]: list(filter(lambda number : number % 2 == 1, numbers))
Out[10]: [15, 23]
```

Mapping a Sequence's Values to New Values

```
In [11]: list(map(lambda x : x * x, filter(lambda number : number % 2 == 1, numbers)))
Out[11]: [225, 529]
```

Mapping a Sequence's Values to New Values

```
In [12]: odd_numbers = filter(lambda number : number % 2 == 1, numbers)

In [13]: squared_numbers = map(lambda x : x * x, odd_numbers)

In [15]: list(squared_numbers)

Out[15]: [225, 529]
```

Reduction: Totaling the Elements of a Sequence with sum

```
In [58]: numbers = [4, 8, 15, 16, 23, 42]

In [59]: odd_numbers = list(filter(lambda number : number % 2 == 1, numbers))

In [60]: squared_numbers = list(map(lambda x : x * x, odd_numbers))

In [61]: from functools import reduce

In [62]: total = reduce(lambda s,x : s+x ,squared_numbers , 0 )

In [63]: total

Out[63]: 754
```