(Dys)Functional Programming with Python

Chaochih Liu and Paul Hoffman

Does[0]Compute? April 15^{th}, 2016

What is Functional Programming?

The functional programming style aims to reduce mutability of data and promote predictability of code

Functional programming stems from lambda calculus and focuses on mathematical evaluations

It is entirely declarative; we describe what the result should look like, not how to achieve the result

Imperative vs Declarative Programming

Imperative

```
>>> numbers = [1, 2, 3, 4, 5] # Create a list of numbers
>>> doubled = [] # Create an empty list for doubled numbers
>>> for i in numbers: # For each number
...     newNumber = i * 2 # Create a new number equal to the current number times two
...     doubled.append(newNumber) # Add this to the list of doubled numbers
...
>>> doubled # Print our list to the screen
[2, 4, 6, 8, 10]
```

Declarative

```
>>> numbers = [1, 2, 3, 4, 5] # Create a list of numbers
>>> doubled = list(map(lambda num : num * 2, numbers)) # Use the map function and a lambda expression to double our list of numbers
>>> doubled # Print our list to the screen
```

Thinking Functionally

Important concepts to keep in mind

- Functions as first-class objects
- State and mutability
- Side effects
- Recursion as iteration

Functions as First-Class Objects

First class objects can be passed to functions

First class objects can be returned from functions

```
>>> def makeKmer(k): # A function that takes a number `k' as its argument
...     def findKmer(sequence): # Define a function within our first function
...         kmer = [sequence[i:i+k] for i in range(len(sequence) - k + 1)] # Collect a list of kmers of size `k'
...         unique = set(kmer) # Find only the unique kmers
...         return(list(unique)) # Return a list of unique kmers
...     return(findKmer) # Return our inside function
...
>>> twoMer = makeKmer(2) # Create a function called `twoMer' where k is 2
>>> twoMer('ACGTCGTACGCT') # Run twoMer on a sequence and get a list of unique 2mers
['GT', 'TC', 'AC', 'CT', 'GC', 'CG', 'TA']
```

State and Mutability

The state of a program refers to the contents of variables within a program

A program changes state if variables are redefined or changed at any point during the program's execution

When unmanaged, change-of-state can result in unexpected results or loss of data

Change of State

```
>>> numbers = [1, 2, 3, 4, 5]  # Create list of numbers
>>> numbers  # view list contents
[1, 2, 3, 4, 5]
>>> numbers.append(6)  # use append() method to add 6 to end of list
>>> numbers  # view list contents again
[1, 2, 3, 4, 5, 6]
```

No Change in State

```
>>> numbers = [1, 2, 3, 4, 5]  # list of numbers
>>> numbers + 6  # add 6 to numbers
[1, 2, 3, 4, 5, 6]
>>> numbers  # view content of list
[1, 2, 3, 4, 5]
```

Side Effects

A side effect is a state-change or interaction outside of the code's scope

- Modifications to global variables
- Change its own arguments
- Exception raising
- File I/O

Order of evaluation matters when side effects are present, and debugging side effects involves knowing the context and history of the code

Recursion as Iteration

```
>>> def fib_loop(num_times): # Get the nth Fibonacci number using a for-loop
...     n0 = 0 # The first Fibonacci number
...     n1 = 1 # The second Fibonacci number
...     if num_times <= 0: # If we get a number less than or equal to zero
...         return 0 # Return 0
...     elif num_times == 1: # If we get one
...         return 1 # Return 1
...     else: # Otherwise
...         for iteration in range(num_times - 1): # For every Fibonacci number until the one we want
...             fib = n0 + n1 # Get the next Fibonacci number
...             n0 = n1 # Move the second number into the first slot
...             n1 = fib # Move the Fibonacci number into the second slot
...         return fib # Return our Fibonacci number
...
```

Recursion as Iteration

```
>>> def fib_rec(num_times): # Get the nth Fibonacci number using recursion
...     if num_times == 0: # If n equals 0
...         return 0 # Return 0
...     elif num_times == 1: # If n equals 1
...         return 1 # Return 1
...     else: # Reduction step
...         return fib_rec(num_times-1) + fib_rec(num_times-2) # Get the Fibonacci for n-1 and n-2
...
```

Functional Programming in Python

Functional programming breaks problems down into small parts

- Creating functions
- lambda expressions

Python's evaluation strategies pose unique challenges to functional programming

- Call-by-Sharing
- Call-by-Need

Iteration can be accompished through means other than for-loops

- Recursion
- map

What is a Function?

A group of statements that can be run more than once

- Maximizes code reuse and reduces maintenance effort
- Provide a way to split tasks into pieces

Function Design Concepts

Principles to keep in mind when designing functions:

- Use arguments for inputs and return for outputs
- Do not rely on global variables
- Don’t change the values of arguments
- Abstract away from hard values to the **type** of data coming in and going out

Python Quirks: Call-by-Sharing

```
>>> def myAppend(myList, myVal): # Create a function that accepts a list and a value
...     myList.append(myVal) # Append the value to the list
...
>>> numbers = [1, 2, 3, 4] # Create a list of numbers
>>> myAppend(numbers, 5) # Run myAppend on the list of numbers with the number 5
>>> numbers # Side effect!
[1, 2, 3, 4, 5]
```

How can we make functions?

Functions can be made in two ways

1. def
2. lambda

Def: Traditional Function Definitions

Function definitons consist of the following parts:

- def keyword
- argument list
- computational statements
- *optional* return statement with list of objects to be returned

```
>>> def myFunction(arg1, arg2):
        # do something one
        # do something two
        return(result1, result2)
```

Lambda Expressions: *Anonymous* Functions

It is a specialized function where you get one statement to do anything you want and the value of that will be returned to the user.

Basic form

Lambda expressions consist of the following parts:

- lambda keyword
- argument list
- **single** statement whose value gets returned

```
lambda argument1, argument2,... argumentN : expression using arguments
```

Lambda example

```
>>> myLambda = lambda x : x + 10  # add 10 to x
>>> myLambda(10)  # run myLambda for x = 10
20
>>> myLambda(15)  # run myLambda for x = 15
25
```

Comparisons of Lambda and def

Lambda	def
Optional naming	Always names new function
Always one liner	Code unlimited number of lines
Always returns result	Return result is optional

Map

A way to apply a function to items in an iterable

map expects a function to be passed in and applied

In Python 3, elements are generated one at a time as they are needed (*lazy* evaluation)

Map Example

for-loop

```
>>> counters = [1, 2, 3, 4]  # list of numbers 1 through 4
>>>
>>> updated = []  # empty list to store results
>>> for x in counters:  # for every number in list
        updated.append(x + 10)  # Add 10 to each item and append to empty list
>>> updated  # view list
[11, 12, 13, 14]
```

map function

```
>>> inc = lambda x : x + 10  # add 10 to every x
>>> list(map(inc, counters))    # Collect results
[11, 12, 13, 14]
```

Python Quirks: Call-by-Need

```
>>> myRange = range(3, 8) # Create a range from 3 to 8 (remember how Python counts!)
>>> myRange # What kind of object do we get with the range function
range(3, 8)
>>> for i in myRange: # Use a for-loop to get the values of the range object
...     print(i)
...
3
4
5
6
7
```

Utilizing Map

Pay attention to input arguments

- Ensure your data is in an iterable form
- Try to limit to one input at a time

Make your own functions

- Design to be run on a subset of data
- If possible, use one argument

If you need multiple arguments, look at the itertools module

Coding Challenge

Take the following code and rewrite it to follow the functional programming paradigm

A fully commented version is available on the GitHub

```
#!/usr/bin/env python3

import utilities

sequences = utilities.setup()

ATcontent = []

for seq in sequences:
    thisAT = 0
    for base in seq:
            if base == 'A' or base == 'T':
                    thisAT += 1
    thisAT = round(thisAT / len(seq) * 100.0, 2)
```