Introduction to Concepts in Functional Programming

CS16: Introduction to Data Structures & Algorithms
Spring 2017
Outline

- Functions
- State
- Functions as building blocks
- Higher order functions
  - Map
  - Reduce
Functional Programming Paradigm

- A style of building the structure and elements of computer programs that treats computation as the evaluation of mathematical functions.
- Programs written in this paradigm rely on smaller methods that do one part of a larger task. The results of these methods are combined using function compositions to accomplish the overall task.
What are functions?

- Takes in an input and always returns an output
- A given input maps to exactly one output
- Examples:
  - In math: $f(x) = x + 1$
  - In python:
    ```python
def f(x):
    return x + 1
```
What is State?

- All stored information to which a program has access to at any given point in time
- How can a program’s state change?
  - **Mutable data** is data that can be changed after creation. Mutating data changes program state.
    - Mutator methods (i.e. setters…)
    - Loop constructs that mutate local variables
  - **Immutable data** is data that cannot be changed after creation. In a language with only immutable data, the program state can never change.
State changes

- In a **stateful** program, the same method could behave differently depending upon the **state** of the program when it was called.
- Let’s look at an example of a **stateful** program.
- Our example is a short program that simulates driving behaviour based on the colour of the stoplight.
light_colour = "RED"

def change_light():
    if light_colour == "RED":
        light_colour = "YELLOW"
    elif light_colour == "YELLOW":
        light_colour = "GREEN"
    elif light_colour == "GREEN":
        light_colour = "RED"

def drive(car):
    if light_colour == "RED":
        car.stop()
    elif light_colour == "YELLOW":
        car.slow_down()
    elif light_colour == "GREEN":
        car.move_forward()
State in Functional Programs

- In pure functional languages, all data is immutable and the program state cannot change.
- What are the implications of this property?
  - Functions are deterministic i.e. the same input will always yield the same output. This makes it easier to re-use functions elsewhere.
  - The order of execution of multiple functions does not affect the final outcome of the program.
  - Programs don’t contain any side effects.
Mutable vs Immutable State

- Consider these two programs

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\begin{align*} a &amp;= f(x) \ b &amp;= g(y) \ \text{return } h(a,b) \end{align*}</td>
<td>\begin{align*} b &amp;= g(y) \ a &amp;= f(x) \ \text{return } h(a,b) \end{align*}</td>
</tr>
</tbody>
</table>

- Will they return the same result if the state is mutable?
- What about when the state is immutable?
Mutable vs Immutable State

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a = f(x))</td>
<td>(b = g(y))</td>
</tr>
<tr>
<td>(b = g(y))</td>
<td>(a = f(x))</td>
</tr>
<tr>
<td>return (h(a,b))</td>
<td>return (h(a,b))</td>
</tr>
</tbody>
</table>

- **Mutable State**: Not guaranteed to give the same results because:
  - The first function call might have changed state.
  - Thus, the second function call might behave differently.
  - This is an example of a *side effect*.

- **Immutable State**: Guaranteed to output the same result for the same inputs!
State and Loops

```python
for i = 0; i < len(L); i++:
    print L[i]
```

The local variable `i` is being mutated!

- If we can’t mutate state - we can’t use our usual `for` and `while` loop constructs!
- Instead, functional languages make use of recursion
State and Loops (cont’d)

- What variables are being mutated in this example?

```python
def max(L):
    max_val = -infinity
    for i in range(0, len(L)):
        if L[i] > max_val:
            max_val = L[i]
    return max_val
```

30 seconds
What variables are being mutated in this example?

How do we write this function without mutation … ?
Replacing Iteration with Recursion

<table>
<thead>
<tr>
<th>Iterative Max</th>
<th>Recursive Max</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>def max(L):</strong></td>
<td><strong>def max(L):</strong></td>
</tr>
<tr>
<td>max_val = -infinity</td>
<td>return max_helper(L, -infinity)</td>
</tr>
<tr>
<td>for i in range(0, len(L)):</td>
<td></td>
</tr>
<tr>
<td>if L[i] &gt; max_val:</td>
<td><strong>def max_helper(L, max_val):</strong></td>
</tr>
<tr>
<td>max_val = L[i]</td>
<td>if len(L) == 0:</td>
</tr>
<tr>
<td>return max_val</td>
<td>return max_val</td>
</tr>
<tr>
<td></td>
<td>if L[0] &gt; max_val:</td>
</tr>
<tr>
<td></td>
<td>return max_helper(L[1:], L[0])</td>
</tr>
<tr>
<td></td>
<td>return max_helper(L[1:], max_val)</td>
</tr>
</tbody>
</table>

The recursive version would work in a pure functional language, the iterative version would not.
First Class Functions

- In the functional paradigm, functions are treated as **first-class citizens** i.e. they can be:
  - Passed as arguments to other functions
  - Returning them as values from other functions
  - Assigning them to variables
  - Storing them in variables just like other data-types

```python
def add_one(x):
    return x + 1

def apply_func_to_five(f):
    return f(5)

print apply_func_to_five(add_one)
>>> 6
```
First Class Functions (cont’d)

- What’s actually happening in our definition of the `add_one` function?

```python
def add_one(x):
    return x + 1
```

- We’re binding a function to the identifier `add_one`

- In python, this is equivalent to

```python
add_one = lambda x: x+1
```

- Function identifier
- Python keyword
- Argument passed to the function
- Return value of the function
Anonymous Functions

- Data types such as numbers, strings, booleans etc. don’t need to be bound to a variable. Similarly, neither do functions!
- An **anonymous function** is a function that is not bound to an identifier.
- A python example of an anonymous function is `lambda x: x + 1`
- An example of a function that returns an anonymous function:

```python
# Input: A number k
# Output: A function that increments k by the number passed into it
def increment_by(k):
    return lambda x: x + k
```
## Function Syntax Overview

<table>
<thead>
<tr>
<th>Math</th>
<th>Bound Python Function</th>
<th>Anonymous Python Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) = x + 1 )</td>
<td><code>def f(x): return x + 1</code> or <code>f = lambda x: x + 1</code></td>
<td><code>lambda x: x + 1</code></td>
</tr>
</tbody>
</table>
Higher Order Functions

- A function is a **higher-order function** if it either takes in one or more functions as parameters or returns a function, or both.

- You’ve already seen examples of higher-order functions in the previous slides!

```python
def increment_by(k):
    return lambda x: x + k

print apply_func_to_five(add_one)
```

```python
>>> 6
```

```python
# Input: A number k
# Output: A function that increments k by the number passed into it
def increment_by(k):
    return lambda x: x + k
```
# Input: A number x
# Output: A function that adds the number passed in to x

def add_func(x):
    return lambda y: x + y

# we pass in 1 as the value of ‘x’
>>> add_one = add_func(1)

# add_one holds the function object returned by calling add_func
>>> print add_one
<function <lambda> at 0x123e410>

# ‘5’ is the value of the parameter ‘y’ in the function
# add_one which is lambda y: 1 + y
>>> print add_one(5)
6
Using Higher Order Functions (cont’d)

## Input: A function that takes in a function, and a number
## Output: A function that takes in a number

def satisfy_first_arg(func, x):
    return lambda y: func(x, y)

def add(x, y):
    return x + y

# ‘add’ corresponds to the argument ‘func’ and ‘1’ corresponds to ‘x’
>>> add_one = satisfy_first_arg(add, 1)

# ‘add_one’ represents the function lambda y: add(1, y)
>>> print add_one
<function <lambda> at 0x123e410>

# evaluates and prints lambda 7: add(1,7)
>>> print add_one(7)
8
Map

- **Map** is a higher order function with the following specifications:
  - Inputs
    - f - a function that takes in an element
    - L - a list of elements
  - Output
    - A new list of elements, with f applied to each of the elements of L
Map

map(lambda x: x-2, [11, 9, 24, -5, 34, 4])
Reduce

- **Reduce** is also a higher-order function.
- It *reduces* a list of elements to one element using a binary function to successively combine the elements.
  
  **Inputs**
  
  - f - a binary function
  - L - list of elements
  - acc - accumulator, the parameter that collects the return value

  **Output**
  
  - The value of f sequentially applied and tracked in ‘acc’
Reduce

- Reduce is roughly equivalent in functionality to this Python function:

```python
def reduce(binary_func, elements, acc):
    for element in elements:
        acc = binary_func(acc, element)
    return acc
```
Reduce Example

```python
# binary function 'add'
add = lambda x, y: x + y

# use 'reduce' to sum a list of numbers
>>> print reduce(add, [1,2,3], 0)
6
```
Reduce Example

```python
add = lambda x, y: x + y
reduce(add, [1,2,3], 0)
```

<table>
<thead>
<tr>
<th>Math</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>(((0 + 1) + 2) + 3) = ?</td>
<td>reduce(add, [1,2,3], 0) = ?</td>
</tr>
<tr>
<td>current accumulator</td>
<td>current accumulator</td>
</tr>
</tbody>
</table>
Reduce Example (cont’d)

Math

\[
((0 + 1) + 2) + 3) = ?  \\
((1 + 2) + 3) = ?
\]

Python

\[
\text{reduce(add, [1,2,3], 0)} = ?  \\
\text{reduce(add, [2,3], 1)} = ?
\]

add = lambda x, y: x + y
reduce(add, [1,2,3], 0)
Reduce Example (cont’d)

\[
\begin{align*}
\text{add} &= \lambda x, y: x + y \\
\text{reduce}(\text{add}, [1,2,3], 0) &= 6
\end{align*}
\]

<table>
<thead>
<tr>
<th>Math</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>((0 + 1) + 2 + 3) = ?</td>
<td>\text{reduce}(\text{add}, [1,2,3], 0) = ?</td>
</tr>
<tr>
<td>((1 + 2) + 3) = ?</td>
<td>\text{reduce}(\text{add}, [2,3], 1) = ?</td>
</tr>
<tr>
<td>(2 + 3) = ?</td>
<td>\text{reduce}(\text{add}, [3], 2) = ?</td>
</tr>
</tbody>
</table>

current accumulator
Reduce Example (cont’d)

```python
add = lambda x, y: x + y
reduce(add, [1,2,3], 0)
```

<table>
<thead>
<tr>
<th>Math</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>$((0 + 1) + 2) + 3) = \text{?}$</td>
<td>$\text{reduce}(add, [1,2,3], 0) = \text{?}$</td>
</tr>
<tr>
<td>$(1 + 2) + 3) = \text{?}$</td>
<td>$\text{reduce}(add, [2,3], 1) = \text{?}$</td>
</tr>
<tr>
<td>$(3 + 3) = \text{?}$</td>
<td>$\text{reduce}(add, [3], 3) = \text{?}$</td>
</tr>
<tr>
<td>$6$</td>
<td>$6$</td>
</tr>
</tbody>
</table>

final accumulator/return value
Reduce Activity

```python
multiply = lambda x, y: x*y
reduce(multiply, [1,2,3,4,5], 1)
```

1.5 mins
# Reduce Activity

```python
multiply = lambda x, y: x*y
reduce(multiply, [1,2,3,4,5], 1)
```

<table>
<thead>
<tr>
<th>Math</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>(((1*1)*2)*3)*4)*5) = ?</td>
<td>reduce(multiply, [1,2,3,4,5], 1) = ?</td>
</tr>
<tr>
<td>(((1*2)*3)*4)*5) = ?</td>
<td>reduce(multiply, [2,3,4,5], 1) = ?</td>
</tr>
<tr>
<td>(((2*3)*4)*5) = ?</td>
<td>reduce(multiply, [3,4,5], 2) = ?</td>
</tr>
<tr>
<td>(((6*4)*5)) = ?</td>
<td>reduce(multiply, [4,5], 6) = ?</td>
</tr>
<tr>
<td>((24*5)) = ?</td>
<td>reduce(multiply, [5], 24) = ?</td>
</tr>
<tr>
<td>120</td>
<td>120</td>
</tr>
</tbody>
</table>
Reduce

- The accumulator doesn’t always have to be an integer and `reduce` doesn’t have to necessarily reduce the list to a single number.
- Another example is removing consecutive duplicates from a list.
  - The accumulator is also a list!

```python
def compress(acc, e):
    if acc[len(acc)-1] != e:
        return acc + [e]
    return acc

def remove_consecutive_dups(L):
    return reduce(compress, L, [L[0]])
```
With higher-order functions, we can make our programs much more concise!

Let’s look at the recursive `max` function we defined earlier:

<table>
<thead>
<tr>
<th>Original Recursive Example</th>
<th>Revised with Higher Order Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def max(L):</code> <code>return max_helper(L, -infinity)</code></td>
<td><code>def max_of_two(acc, e):</code> <code>if acc &gt; e:</code> <code>return acc</code> <code>return e</code></td>
</tr>
<tr>
<td><code>def max_helper(L, max_val):</code> <code>if len(L) == 0:</code> <code>return max_val</code> <code>if L[0] &gt; max_val:</code> <code>return max_helper(L[1:], L[0])</code> <code>return max_helper(L[1:], max_val)</code></td>
<td><code>def max(L):</code> <code>return reduce(max_of_two, L, -infinity)</code></td>
</tr>
</tbody>
</table>
Building Functional Programs

- With the power to use higher-order functions and using functions as first-class citizens, we can now build programs in a functional style!
- A functional program can be thought of as one giant function composition such as $f(g(h(x)))$. 
Advantages and Disadvantages of Functional Programming

1 min
## Advantages and Disadvantages of Functional Programming

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Programs are deterministic.</td>
<td>• Programs are deterministic.</td>
</tr>
<tr>
<td>• Code is elegant and concise because of higher order function abstractions.</td>
<td>• Potential performance losses because of the amount of garbage-collection that needs to happen when we end up creating new variables as we can’t mutate existing ones.</td>
</tr>
<tr>
<td>• It’s easy to run code concurrently because state is immutable.</td>
<td>• File I/O is difficult because it typically requires interaction with state.</td>
</tr>
<tr>
<td>• Learning functional programming teaches you as a programmer to think about problems very differently than imperative programming which makes you a better problem-solver!</td>
<td>• Programmers who’re used to imperative programming could find this paradigm harder to grasp.</td>
</tr>
</tbody>
</table>
Functional Programming Practice

6 mins
Problem # 1

Write an anonymous function that raises a single argument \( n \) to the \( n^{th} \) power
Problem #1

Write an anonymous function that raises a single argument 'n' to the n\textsuperscript{th} power

\textbf{Solution:}

\texttt{lambda n: n**n}
Problem #2

Write a line of code that applies the function you wrote in problem 1 to an input list.
Problem #2

Write a line of code that applies the function you wrote in problem 1 to an input list.

Solution:

```python
map(lambda n: n**n, input_list)
```
Problem #3

Write an anonymous function that takes in a single argument ’n’ and returns a function that consumes no arguments and returns n.
Problem #3

Write an anonymous function that takes in a single argument 'n' and returns a function that consumes no arguments and returns n.

Solution:

\[
\text{lambda n: lambda: n}
\]
Problem #4

Write a line of code that applies the function you wrote in problem #3 to an input list. This should give you a list of functions.
Then write a line of code that takes in the list of functions and outputs the original list again.
Problem #4

Write a line of code that applies the function you wrote in problem #3 to an input list. This should give you a list of functions. Then write a line of code that takes in the list of functions and outputs the original list again.

Solution:

```python
function_list = map(lambda n: lambda: n, input_list)
map(lambda f: f(), function_list)
```
Problem #5

Remove odd numbers from an input list of numbers using `reduce`. 
Problem #5

Remove odd numbers from an input list of numbers using `reduce`.

**Solution:**

```
reduce(lambda acc, x: acc+[x] if x%2 == 0 else acc, L, [])
```
More Higher Order Functions

- There are many more commonly-used higher order functions besides `map` and `reduce`.
- **filter(f,x)**
  - Returns a list of those elements in list ‘x’ that make f true, assuming that f is a function that evaluates to true or false based on the input.
- **zipWith(f,x,y):**
  - Takes in 2 lists of the same length and passes elements at the same index into a binary function ‘f’ to return a single list that contains elements of the form f(x[i], y[i]).
- **find(f,x):**
  - Returns the first element of list ‘x’ for which ‘f’ evaluates to true.
Main Takeaways

- Functional Programming is a way of structuring programs using mathematical functions that take in inputs and return an output.

- Functions written in this paradigm:
  - Don’t mutate any data (stateless)
  - Are deterministic
  - Are first-class citizens

- The functional approach allows us to write programs very concisely using higher-order function abstractions.

- Testing and debugging is easier when the overall program is split into functions that are used as a big function composition.
Further Reading

- Examples of functional languages include Haskell, Common Lisp, Closure, OCaml
- Learning the functional paradigm through Haskell: [http://learnyouahaskell.com/](http://learnyouahaskell.com/)
Questions?

I code on a boat in international waters to avoid states.