Special topic: Functional Programming

CS16: Introduction to Data Structures & Algorithms
Summer 2021
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…)
    - Changing local variables (i.e. `int x = 1; x = 2`)
  - **Immutable data** is data that cannot be changed after creation. In a program with only immutable data, there’s no state that needs to be tracked
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 behavior based on the color of the stoplight.
light_color = "RED"

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

def drive(car):
    if light_color == "RED":
        car.stop()
    elif light_color == "YELLOW":
        car.slow_down()
    else:
        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>$a = f(x)$</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>

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

- **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!

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

Program 1

```python
a = f(x)
b = g(y)
return h(a,b)
```

Program 2

```python
b = g(y)
a = f(x)
return h(a,b)
```
State and Loops

for int 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)

- Which variables are being mutated in this example?

```python
def max(L):
    max_val = L[0]
    for i in range(0, len(L)):
        if L[i] > max_val:
            max_val = L[i]
    return max_val
```
What variables are being mutated in this example?

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

- `i` is being mutated!
- `max_val` is being mutated!

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 = L[0]</td>
<td>return max_helper(L, L[0])</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>if len(L) == 0:</td>
</tr>
<tr>
<td>max_val = L[i]</td>
<td>return max_val</td>
</tr>
<tr>
<td>return max_val</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 never mutates state!
First Class Functions

- In the functional paradigm, functions are treated **just like other values**! They can be:
  - Passed as arguments to other functions
  - Returning them as values from other functions
  - 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
```
Write an anonymous function that raises a single argument 'n' to the $n^{th}$ power
Write an anonymous function that raises a single argument 'n' to the $n^{th}$ power

**Solution:**

`lambda n: n**n`
Higher Order Functions

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

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

```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

print apply_func_to_five(add_one)  # Output: 6
```
# 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
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 example

```
list(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
```
# in Python, need to import reduce
from functools import reduce

# 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

\[ \text{add} = \lambda x, y: x + y \]
\[ \text{reduce}(\text{add}, [1,2,3], 0) \]

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

- current accumulator

- 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 ) = ? \</td>
<td>reduce(add, [1, 2, 3], 0) = ?</td>
</tr>
<tr>
<td>( (1 + 2) + 3 ) = ? \</td>
<td>reduce(add, [2, 3], 1) = ?</td>
</tr>
</tbody>
</table>

**current accumulator**
Reduce Example (cont’d)

```
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>((1 + 2) + 3) = ?</td>
<td>reduce(add, [2,3], 1) = ?</td>
</tr>
<tr>
<td>(3 + 3) = ?</td>
<td>reduce(add, [3], 3) = ?</td>
</tr>
</tbody>
</table>

---

**Current accumulator**
Reduce Example (cont’d)

\[
\text{add} = \lambda x, y: x + y \\
\text{reduce}(\text{add}, [1,2,3], 0)
\]

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

final accumulator/return value

final accumulator/return value
def multiply(x, y):
    return x * y
reduce(multiply, [1,2,3,4,5], 1)
# Reduce Example

```python
def multiply(x, y):
    return 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>(\text{reduce}(\text{multiply}, [1,2,3,4,5], 1)) = ?</td>
</tr>
<tr>
<td>((1*2)*3)*4)*5) = ?</td>
<td>(\text{reduce}(\text{multiply}, [2,3,4,5], 1)) = ?</td>
</tr>
<tr>
<td>((2*3)*4)*5) = ?</td>
<td>(\text{reduce}(\text{multiply}, [3,4,5], 2)) = ?</td>
</tr>
<tr>
<td>((6*4)*5) = ?</td>
<td>(\text{reduce}(\text{multiply}, [4,5], 6)) = ?</td>
</tr>
<tr>
<td>((24*5)) = ?</td>
<td>(\text{reduce}(\text{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]])
```
Using Higher Order Functions

- 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><strong>def max(L):</strong></td>
<td><strong>def max_of_two(acc, e):</strong></td>
</tr>
<tr>
<td>return max_helper(L, L[0])</td>
<td>if acc &gt; e:</td>
</tr>
<tr>
<td><strong>def max_helper(L, max_val):</strong></td>
<td>return acc</td>
</tr>
<tr>
<td>if len(L) == 0:</td>
<td>return e</td>
</tr>
<tr>
<td>return max_val</td>
<td><strong>def max(L):</strong></td>
</tr>
<tr>
<td>if L[0] &gt; max_val:</td>
<td>return reduce(max_of_two, L, L[0])</td>
</tr>
<tr>
<td>return max_helper(L[1:], L[0])</td>
<td></td>
</tr>
<tr>
<td>return max_helper(L[1:], max_val)</td>
<td></td>
</tr>
</tbody>
</table>
Using Higher Order Functions

- Can go even further with lambda

<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></td>
<td><code>def max(L):</code></td>
</tr>
<tr>
<td>return max_helper(L, L[0])</td>
<td>return reduce(</td>
</tr>
<tr>
<td></td>
<td>lambda x,y: x if x &gt; y else y,</td>
</tr>
<tr>
<td><code>def max_helper(L, max_val):</code></td>
<td>L,</td>
</tr>
<tr>
<td>if len(L) == 0:</td>
<td>L[0])</td>
</tr>
<tr>
<td>return max_val</td>
<td></td>
</tr>
<tr>
<td>if L[0] &gt; max_val:</td>
<td></td>
</tr>
<tr>
<td>return max_helper(L[1:], L[0])</td>
<td>return max_helper(L[1:], max_val)</td>
</tr>
<tr>
<td>return max_helper(L[1:], max_val)</td>
<td></td>
</tr>
</tbody>
</table>
Reduce practice

Remove odd numbers from an input list of numbers using **reduce**.
Reduce practice

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

**Solution:**

```python
reduce(lambda acc, x: acc+[x] if x%2 == 0 else acc, L, [])
```
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

<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>• Easy to run code concurrently because state is immutable.</td>
<td>• The real world is stateful! Writing to a file changes its state. Functional programming languages have ways of handling this, but doing so elegantly + efficiently is an open research issue.</td>
</tr>
<tr>
<td>• Easier to test functions—just call on different inputs</td>
<td></td>
</tr>
<tr>
<td>• Easier to prove functions correct</td>
<td></td>
</tr>
<tr>
<td>• It’s fun! Different way of thinking about problems</td>
<td></td>
</tr>
</tbody>
</table>
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.
Further Exploration

- Examples of functional languages include Haskell, Racket, Common Lisp, Closure, OCaml
  - (and Pyret!)
- Classes at Brown using FP:
  - CSCI 1260
  - CSCI 1730
  - CSCI 1950-Y
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 values that can be passed around and returned

- 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.