# Special topic: <br> Functional Programming 

CSI 6: 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:

```
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 $\mathrm{x}=1$; $\mathrm{x}=2$ )
- 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" elif light_color == "GREEN": light_color = "RED"
def drive(car):
if light_color == "RED":
car.stop()
elif light_color == "YELLOW":
car.slow_down()
elif light_color == "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

| Program I | Program 2 |
| :--- | :--- |
| $a=f(x)$ | $b=g(y)$ |
| $b=g(y)$ | $a=f(x)$ |
| return $h(a, b)$ | return $h(a, b)$ |

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


## Mutable vs Immutable State

| Program I | Program 2 |
| :--- | :--- |
| $a=f(x)$ | $b=g(y)$ |
| $b=g(y)$ | $a=f(x)$ |
| return $h(a, b)$ | return $h(a, b)$ |

- 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



- 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 ?

```
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
```


## State and Loops (cont'd)

-What variables are being mutated in this example ?


- How do we write this function without mutation ... ?


## Replacing Iteration with Recursion

| Iterative Max | Recursive Max |
| :---: | :---: |
| ```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``` | ```def max(L): return max_helper(L, L[0]) def max_helper(L, max_val): if len(L) == 0: return max_val if L[0] > max_val: return max_helper(L[1:], L[0]) return max_helper(L[1:], max_val)``` |

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

```
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?

```
def add_one(x):
```

    return \(x+1\)
    - We're binding a function to the identifier add_one
- In python, this is equivalent to



## 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 $\mathbf{x}: \mathbf{x + 1}$
- An example of a function that returns an anonymous function:

```
# 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
```


## Lambda practice

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

## Lambda practice

Write an anonymous function that raises a single argument ' $n$ ' to the $n^{\text {th }}$ 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!

```
print apply_func_to_five(add_one)
>>> 6
```

```
# 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
```


## Using Higher Order Functions

```
# 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\mathrm{ '
>>> 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 is roughly equivalent in functionality to this python function:

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


## Reduce Example

```
# 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

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



## Reduce Example (cont'd)

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

| Math | Python |
| :---: | :---: |
| $((\underline{0}+1)+2)+3)=?$ | reduce $($ add, $[1,2,3], \underline{0})=?$ |
| $\left(\left(\frac{1}{4}+2\right)+3\right)=?$ | reduce $(\operatorname{add},[2,3], \underline{1})=?$ |
| current accumulator |  |

## Reduce Example (cont'd)

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

| Math | Python |
| :---: | :---: |
| $((\underline{0}+1)+2)+3)=?$ | reduce (add, $[1,2,3], \underline{0})=?$ |
| $(\underline{1}+2)+3)=?$ |  |
| $\left(\frac{3}{4}+3\right)=?$ | reduce $($ add, $[2,3], \underline{1})=?$ |
| reduce $\left(\right.$ add, $\left.[3], \frac{3}{4}\right)=?$ |  |
| current accumulator |  |

## Reduce Example (cont'd)

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

| Math | Python |
| :---: | :---: |
| $((\underline{0}+1)+2)+3)=?$ | reduce(add, $[1,2,3], \underline{0})=$ ? |
| $((\underline{1}+2)+3)=?$ | reduce(add, $[2,3], \underline{1})=$ ? |
| $(\underline{3}+3)=?$ | reduce (add, [3], ${ }^{\text {3 }}$ ) $=$ ? |
| 6 |  |
| final accumulator/ return value | final accumulator/ return value |

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

## Reduce Example

```
def multiply(x, y):
    return x * y
reduce(multiply, [1,2,3,4,5], 1)
```

| Math | Python |
| :--- | :--- |
| $((\underline{1} * 1) * 2) * 3) * 4) * 5)=?$ | reduce(multiply, $[1,2,3,4,5], \underline{1})=?$ |
| $((\underline{1} * 2) * 3) * 4) * 5)=?$ |  |
| $((\underline{2} * 3) * 4) * 5)=?$ |  |
| $((\underline{6} * 4) * 5)=?$ |  |
| $(\underline{24} * 5)=?$ | reduce(multiply, $[2,3,4,5], \underline{1})=?$ |
| $\underline{120}$ | reduce(multiply, $[3,4,5], \underline{2})=?$ |
| reduce(multiply, $[4,5], \underline{6})=?$ |  |
| 120 |  |

## 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!

```
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:

| Original Recursive Example | Revised with Higher Order Functions |
| :---: | :---: |
| ```def max(L): return max_helper(L, L[0]) def max_helper(L, max_val): if len(L) == 0: return max_val if L[0] > max_val: return max_helper(L[1:], L[0]) return max_helper(L[1:], max_val)``` | ```def max_of_two(acc, e): if acc > e: return acc return e def max(L): return reduce(max_of_two, L, L[0])``` |

## Using Higher Order Functions

- Can go even further with lambda

| Original Recursive Example | Revised with Higher Order Functions |
| :---: | :---: |
| ```def max(L): return max_helper(L, L[0]) def max_helper(L, max_val): if len(L) == 0: return max_val if L[0] > max_val: return max_helper(L[1:], L[0]) return max_helper(L[1:], max_val)``` | ```def max(L): return reduce( lambda x,y: x if x > y else y, L, L[0])``` |

## 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:
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 $\mathbf{f}(\mathrm{g}(\mathbf{h}(\mathbf{x}))$ ).


## Advantages and Disadvantages of Functional Programming

| Advantages | Disadvantages |
| :---: | :---: |
| - Programs are deterministic. <br> - Code is elegant and concise because of higher order function abstractions. <br> - Easy to run code concurrently because state is immutable. <br> - Easier to test functions-just call on different inputs <br> - Easier to prove functions correct <br> - It's fun! Different way of thinking about problems | - Programs are deterministic. <br> - Potential performance losses because of the amount of garbagecollection that needs to happen when we end up creating new variables as we can't mutate existing ones. <br> - 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 |

## More Higher Order Functions

- There are many more commonly-used higher order functions besides map and reduce.
- filter (f, $\mathbf{x}$ )
- Returns a list of those elements in list ' $\times$ ' that make $f$ true, assuming that $f$ is a function that evaluates to true or false based on the input.
- zipWith (f, $\mathbf{x}, \mathbf{y}):$
- 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 $f(x[i], y[i])$.
- $\boldsymbol{f i n d}(\mathbf{f}, \mathbf{x}):$
- 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 I730
- CSCI I950-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.

