course: CSC 135

instructor: Ted Krovetz

related_notes: 2022-02-08

Functional Programming

W06.2 | Tuesday, February 8, 2022 | 09:01 AM

Notes

1. 1900s key figures
2. Both started thinking
    1. What is computational
    2. How can machines compute
        1. Alonzo Church
            1. Developed Lambda Calculus
                1. Computational as Mathematical Functions
        2. Alan Turing
            1. Invented the Turing Machine
                1. An algorithm as "loop-base"
3. Functional Computation/Languages
    1. Reduce bugs
    2. Immutable states/variables
    3. Models computation as evaluation of mathematical functions
    4. No mutation (variables never change)
    5. No global state; as a result, no side-effect in a function
    6. No Loops, so there is rich library to do loop-like things and recursion for the rest
4. Two First Functional Language
    1. Fortran - Loops, ect.
    2. Lisp - Functional

Legacy

1. Higher Order Functions
    1. Is a function that can another function as parameters
2. Lambda Function
    1. An anonymous function: a function without name (Foo, Bar)

3. Library functions that take function parameters
4. Constructions that do loop-like things
    1. For example in Java the "for-each": `for(in s: list){ <code> }`
    2. Versus the mutable version of the for-loop `for(int i = 0; i < 10; i++`
5. Immutable data structure

Higher Order Functions

Because it's a dynamic language it's up the the caller to determine which function is called

```
# f is the name of the function
# x is the int we want as base
# n is the number of times
# Example multi(f, 1, 4)
# >> f(f(f(1)))
def multi(f):
        acc = x
        for i in range(n):
                acc = f(acc)
        return acc
```

If we wanted to add one... We'll need to make a function that does so

```
def add_1(x):
        return x+1

multi(add1(1), 0, 10)
# >> 10
```

Lambda Function:

Lambdas in Python are restrictive and doesn't allow multiple lines

It's useful when you need a "one-off" function; however, if you need to use that function multiple times the normal function creation, `def function_name()`, would be optimal

```
# if we want add_1 as a lambda
lambda x: x + 1
```

So instead of doing

```
multi(add1(1), 0, 10)
```

You can do this instead

```
multi(lambda x: x + 1, 0, 10)
```

Higher Order Library Functions (library functions that take functions)

```
sort(list)

# The functional version
# Takes the old list and creates a new list

list_02 = sorted(list)

# Python has a feature where one may name the paramters
# Key is a named paramter and f is a function where it returns an int.
# Items sorted based on int

list_03 = sorted(list, key = f)

int_array = [1, 2, 3, 4] # we want this to become [2, 4, 1, 3]
int_array_02 = sorted(int_array, key = lambda x: x % 2)
```

Map

Maps takes an object and returns an object

we want this

```
integer_list = [1, 2, 3]
map(f,l) -> [f(1), f(2), f(3)]
```

```
def myMap(f, l):
        new_list = [ ]
        for element in l:
```

```
            new_list.append(f(x))
        return new_list
```

if we want the have the returned value to function as a list: `list(map(f, l))`

Filter

Filter takes an object (anything) and returns a Boolean

list = [1, 2, 3, 4]

filter(f, l) → [list of elements making f True]

```
filter(lambda x: x % 2 == 0, the_list)
```

The loop version of filter

```
def myFilter(f, the_list):
        new_list = [ ]
        for x in list:
                if f(x):
                        new_list.append(x)
        return new_list
```

Reduce

Reduce is a method to take a list and "summarize" it

Is called like so `reduce(f, the_list)`

For example

```
the_list = [1, 2, 3]
reduce(lambda x,y: x+y, the_list)
# Produces 6
# Because 1 + 2 = 3 and then we have the last elment (3)
# 3 + 3 = 6
```

If we were to create this via a loop

```
def myReduce(f, the_list)
        acc = None
        for x in the_list:
                if acc == None:
                        acc = x
                else:
                        acc = f(acc, x)
        return acc
```

List Comprehension

The Syntax

```
newlist = [_expression_ for _item_ in _iterable_ if _condition_ == True]
```

```
list01 = [1,2,3]
list02 = [2*x for x in list] # produces [2,4,6]
# {x x E and x mod 2 == 0}
```