FUNCTIONS AS OBJECTS

Brief Recap

- More on Classes
  - Class Variable
  - Private Variable
  - Functions: getattr, setattr
  - Destructors
- Python Iterables
  - Tuples
  - Dictionaries

Menu for Today!

- Functions as First Class Objects
- Functional Programming: map, filter and reduce
- List Comprehensions

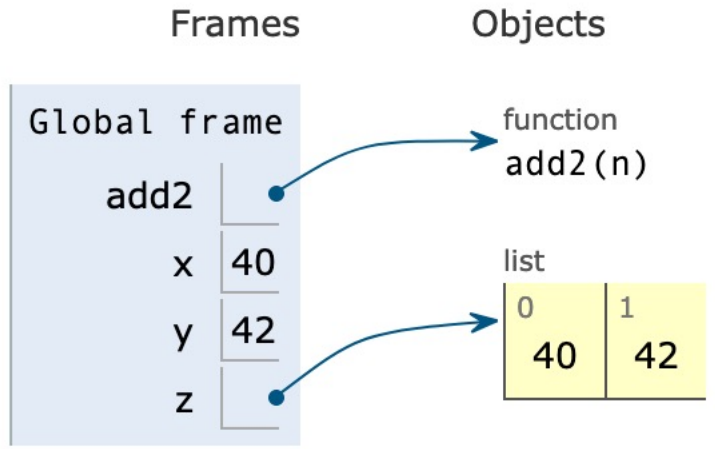
FUNCTIONS AS OBJECTS

Are Functions Objects?

Python 3.6
known limitations

```
1  def add2(n):
2      return(n+2)
3  x = 40
4  y = add2(x)
5  z = [x,y]
```

Edit this code



So far, we have considered functions as code
The picture on the right seems to consider them as data

What Can We Do With Objects?

- Assign an object to a variable
- Return an object as the result from a function / method
- Pass an object as a parameter to a function / method
- Build complex structures (like lists) with the object as an element

Can we do all this with Functions?

Yes, we can!!!

Functions are “first class” objects

Assigning Functions to Variables

```
def add(n,m):
    return (n + m)
myFunc = add
myFunc(2,3)
```

Anonymous Functions

- Python allows us to create objects without giving them names

```
myFunc(2,3 + 5)
```

- How do we create functions without names: lambda

```
lambda n,m: n+m
myFunc1 = lambda n, m: n + m
def myAdd(n,m):
    return(n + m)
add2 = lambda n : myAdd(n,2)
add3 = lambda n : myAdd(n,3)
```

Calling a Function assigned to a Variable

```
def myAdd(n,m):
    return(n + m)
def mySub(n,m):
    return(n-m)
myFunc = myAdd
add2 = lambda n : myFunc(n,2)
myFunc = mySub
x = add2(7)
```

At this point, myFunc points to myAdd

Now myFunc points to mySub

Which myFunc will be called?

Returning A Function

```
def addN (n):
    def tempFn(m):
        return(m+n)
    return tempFn
def addN (N):                          #alternate definition
    return (lambda m: add(m,n))
add2 = addN(2)
add3 = addN(3)
```

Passing Functions as Arguments

- Python allows us to pass functions as arguments to other functions

```
def doTwice(f,n):
    return(f(f(n)))
doTwice(add2, 5)
doTwice(add3, 5)
mult3 = lambda n : n * 3
doTwice(mult3,5)
```

What Functions Take Other Functions as Arguments?

- Remember map from the expression

  ```
  list(map(int,input().split()))
  ```

  - `int` is a function that converts strings to integer
  - So `map` take a function as an argument
- Functions that take other functions as arguments or return functions as results are called **higher order functions**
- Commonly used higher order functions
  - `map`
  - `filter`
  - `reduce`

map(fn, iterable1 [,iterable2, …iterableN])

- First argument is a function (often called an **transformation** function)
- The second argument is an iterable; could have more arguments
- The map function applies the function provided in the first argument to every element of the iterable provided in the second argument
- Returns a `map` object
  - *The `map` object is an iterator that yields items on demands*
  - *You can convert the `map` object to a list using `list( )`*

```
square = lambda x: x * x
a = [1, 2, 3, 4, 5]
b = list(map(square, a))
prod = lambda x,y: x * y
c = list(map(prod, a, a))
```

What is Filtering

```
def isOdd(n):
    return (n % 2 == 1)
def extractOdd(myList):
    oddNumber = []
    for i in myList:
        if (isOdd(i)):
            oddNumber.append(i)
    return(oddNumber)
```

`filter(fn, iterable)`

- First argument is a function that returns a `bool` value (often called a **predicate**)
- The second argument is an iterable
- The map function applies the function provided in the first argument to every element of the iterable provided in the second argument; if the result is True (or *truthy*) the element is added to resulting `filter` object
- Returns a `filter` object
  - *The `filter` object is an iterator that yields items on demands*
  - *You can convert the `filter` object to a list using `list( )`*
- `filter` is an in-built function and often faster than the corresponding loop based implementation

```
def isOdd(n):
    return (n % 2 == 1)
extractOdd = lambda x: filter(isOdd,x)
```

Implementing Accumulator Pattern

- `map` and `filter` work on individual elements of an iterable (list)
- Often, we need to work with multiple elements of a list
  - *Examples include summing a list or find the max of a list*
- Broadly, we want to analyze a recursive structure and combine the results using a combining functions
- This pattern is also called fold, reduce or aggregate

```
def findMax(myList):
    max = None
    for i in myList:
        if (i > max):
            max = i
    return(max)
```

```
def sum(myList):
    sum = 0
    for i in myList:
        sum = sum + i
    return(sum)
```

`reduce(fn, iterable, initialiser)`

- Defined in functools library

```
import functools
```

- Arguments:
  - *First argument is a function of two variable*
  - *The second argument is an iterable*
  - *The third (optional) argument is an initializer*
- Semantics:
  - *Apply a function to the first two items in an iterable and generate a partial result.*
  - *Use that partial result, together with the third item in the iterable, to generate another partial result.*
  - *Repeat the process until the iterable is exhausted and then return a single cumulative value.*
  - *If you supply a value to* `initializer`*, then* `reduce()` *will feed it to the first call of function as its first argument.*

reduce(fn, iterable, initialiser)

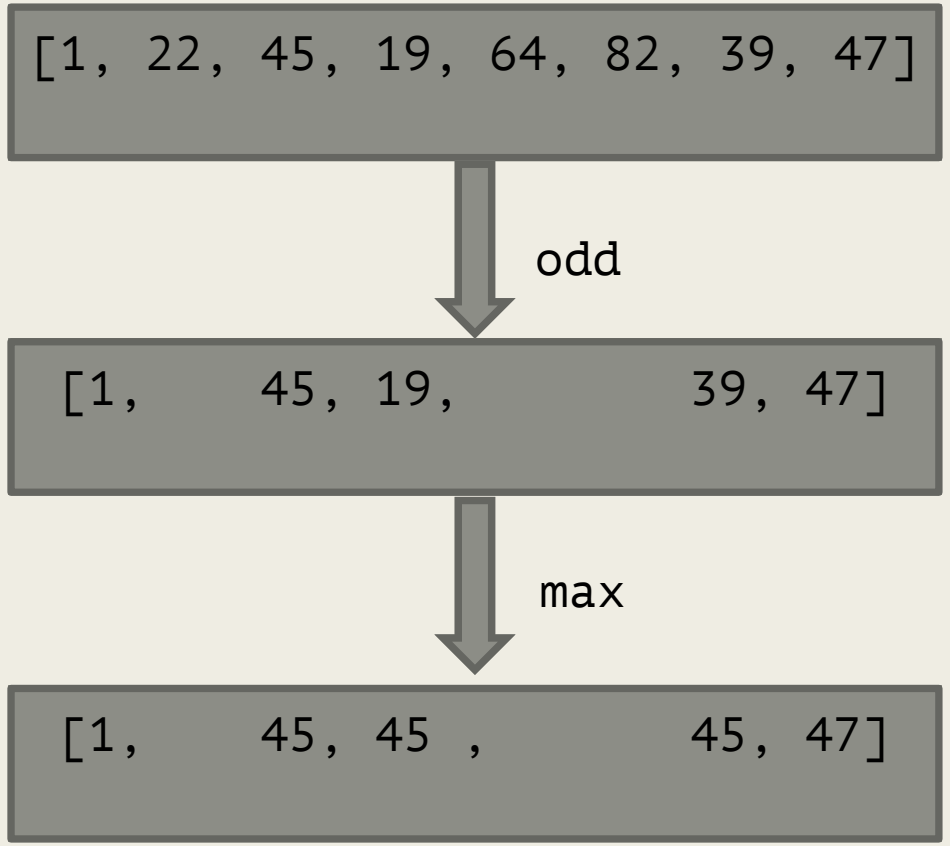
```
def reduce(function, iterable, initializer=None):
    it = iter(iterable) #contruct an iterator
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

Using reduce

```
mySum = reduce((lambda x,y: x+y), myList)
def max(x,y):
    if x > y:
        return x
    else:
        return y
maxValue = reduce(max, myList)
```

Recall Iteration Patterns (Lecture 5)

- Find the max odd number is a list of positive integers
  - *Filter out odd numbers using enumerations*
  - *Keep track of the max number so far using an accumulator variable*
- We had implemented this using loops and if statements
- A cleaner method is using map-reduce



Some Basic Infrastructure

```
def isOdd(n):
    return (n % 2 == 1)

def isEven(n):
    return (n % 2 == 0)

def add(m,n):
    return (m+n)
```

```
def max(m,n):
    if n == None:
        return m
    elif m == None:
        return n
    elif m >= n:
        return m
    else:
        return n
```

Putting It All Together

```
inputList = list(map(int, input().split()))
oddMax = reduce(max, filter(isOdd, inputList),None)
```

- This seems like a lot of work for a simple task
- Why did we do all this work?
- We solved multiple problems at once

```
isEven = lambda n: n % 2 == 0
sumEven = reduce(add, filter(isEven, inputList), 0)
```

- Bonus: this implementation is faster than using for loops!!

List Comprehension

- Coding using `map`, `reduce` and `filter` is typical of a style of programming called **functional programming**
- Python provides another (sometimes) faster form of iterating through lists called list comprehension
- We often encounter the expression `list(map(myFunc, myList))`
- Python has a recommended syntax for this: List comprehension

```
[myFunc(x) for x in myList]
```

- What about the expression `list(map(myFunc,filter(myFunc1, myList)))`

```
[myFunc(x) for x in myList if myFunc1(x)]
```

Some Examples

```
myMatrix = [[1,2,3],[4,5,6],[7,8,9]]
myRowSum = [sum(x) for x in myMatrix]
def transpose(matrix):
    return [[row[i] for row in matrix]
            for i in range(len(matrix[0]))]
myColSum = [sum(x) for x in transpose(myMatrix)]
```