CMSC330: Higher-Order Functions (in Python)

Chris Kauffman

Last Updated:
Tue Sep 5 09:22:52 AM EDT 2023

Logistics

Assignments

- Lecture Quiz 1 now closed
- Project 0 "Setup" Ongoing
- Project 1 "Intro Python" Due Sun 10-Sep

Reading

Python Functional Programming HOWTO : Focus on `map()`, `filter()`, `reduce()` and `lambda()` expressions

Goals

- Wrap up Python Basics
- Discuss first-class functions, relation to nested scope
- Uses for higher-order function which take function args
- Big-4 higher-order funcs
- Lambda expressions for anonymous func args

First-Class Functions which Act as Values

- Like many PLs, Python supports treating functions as values
- Referred to as **First-Class Functions** though this term often carries additional obligations (some of which Python fulfills)

```
# function_value.py:
def double_it(x):                 # define function
  return 2*x
print(double_it)                  # show a printed rep of func
# <function double_it at 0x7fb221d984a0>

a = double_it(5)                  # call function
print(a)
# 10

di = double_it                    # alias for double_it()
print(di)
# <function double_it at 0x7fb221d984a0>

b = di(7)                         # call di() -> double_it()
print(b)
# 14
```

Higher-Order Functions: Function Parameters / Returns

- **Higher-Order Functions** accept function arguments or return functions (or both)
- Function args are useful to tailor semi-complex behavior: rather than trying to implement all options internally, HO func accepts behavior as an argument

```python
# function_args.py:
def scale_list(func, alist):
  for i in range(len(alist)):
    alist[i] = func(alist[i])

def double_it(x):
  return 2*x

def halve_it(x):
  return x/2

import math
def log2_it(x):
  return math.log2(x)

l1 = [10, 20, 30, 40]

l2 = l1.copy()
scale_list(double_it, l2)
print(l2)  # [20, 40, 60, 80]

l3 = l1.copy()
scale_list(halve_it, l3)
scale_list(log2_it,  l3)
print(l3)  # [2.32, 3.32, 3.90, 4.32]
```

Exercise: apply2 and apply_all

Write the following two higher order functions, 1-4 lines each

```
def apply2(func1, func2, data):
  # WRITE ME!
```

```
apply2(double_it, halve_it, 10)
# (20, 5)

apply2(log2_it, double_it, 32)
# (5, 64)
```

```
def apply_all(func_list, data):
  # WRITE ME!
```

```
flist = [double_it, halve_it, log2_it]
apply_all(flist, 10)
# [20, 5.0, 3.32]
```

Answers: apply2 and apply_all

Write the following two higher order functions, 1-4 lines each

```
# apply_exercise.py:
def apply2(func1, func2, data):
  data1 = func1(data)
  data2 = func2(data)
  # return (func1(data),func2(data))
  return (data1,data2)

def apply_all(func_list, data):
  data_list = []
  for func in func_list:
    data_list.append(func(data))
  return data_list

```

Standard Higher-Order Functions

- Several Higher-Order Functions appear widely in computing
- Worth knowing about as their own entity, will appear in Python, OCaml, Racket, and others
- Each function works with a **Data Structure (DS) like a List**

The 4 Recurring Higher Order Funcs

Map
: Create a new DS with function applied to each element, same shape of DS with new elements

Filter
: Create a new DS with only elements of that return `True` from a function; converts a DS to a (probably) smaller DS

Reduce
: Repeatedly apply function to an element of DS and a current value; transforms DS to a single value, generalizes "summing" a list

Iterate
: Execute a function on each element of DS for side-effects (e.g. `print()`) only; discards return values

Aside: Python Iterators and `list()` Coercion

- Python supports **generators / iterators**, an efficient means of providing large collections of items WITHOUT storing them in memory
- Central idea: Generator asked for *next* item, returns item or indicates none left in which case iteration terminates
- Used with the `for a in X:` syntax where `X` is iterable
- Lists, Dictionaries, Sets are all iterable in Python
- `range()` is a generator, can be coerced to a list

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Higher-order functions like `map()` work on an iterator, produce a new iterator
- Will coerce results to a `list()` to see the results

map(func,data)

- Creates a new DS (list) with each element "transformed" by applying func()
- New DS is distinct and separate from old, return vals of function populate new DS

```
# map_demo.py:
def add_one(x):
  return x+1

nums0  = [10,20,30,40]
nums1  = map(add_one,nums0)
nums1l = list(map(add_one,nums0))

print(f"nums0:  {nums0}")  # nums0:  [10, 20, 30, 40]
print(f"nums1:  {nums1}")  # nums1:  <map object at 0x7f597bd67d00>
print(f"nums1l: {nums1l}") # nums1l: [11, 21, 31, 41]

def upcase(x):
  return x.upper()

strsm  = ["cat","Dog","pIg"]
strsu  = map(upcase, strsm)
strsul = list(map(upcase, strsm))

print(f"strsm:  {strsm}")  # strsm:  ['cat', 'Dog', 'pIg']
print(f"strsu:  {strsu}")  # strsu:  <map object at 0x7f597bd66620>
print(f"strsul: {strsul}") # strsul: ['CAT', 'DOG', 'PIG']
```

A Code Pattern for HOFs

You should have noticed the following pattern

```
def smallfunc1(arg):
  ...

def smallfunc2(arg):
  ...

def hofunc(func_arg, othe_args):
  ...
  ...
  ...

hofunc(smallfunc1, ...)
hofunc(smallfunc2, ...)
```

- Higher-order functions may be modest in length or quite long
- The small functions that become arguments are often one-liners
- It would be nice if one could avoid the need to `def`-ine the small functions . . . .

Lambda Expressions: Anonymous Function Creation

- **Lambda Expression** or just **Lambda**: a syntax to create a function body without naming the function
- Sometimes referred to as anonymous functions
- Often part of what's meant by "first-order functions" in PLs

```
# lambda_demo.py:
def double_it1(x):                   # standard func binding
  return 2*x

double_it2 = lambda x: 2*x           # lambda binding
#  NAME      LAMBDA EXPRESSION

alist = [1,2,3,4,5]

print(list(map(double_it1, alist)))    # call w/ standard func
# [2, 4, 6, 8, 10]

print(list(map(double_it2, alist)))    # call w/ lambda func
# [2, 4, 6, 8, 10]

print(list(map(lambda y: 2*y, alist))) # call w/ lambda directly
# [2, 4, 6, 8, 10]

print(list(map(lambda x: x+1, alist))) # call w/ different lambda
# [2, 3, 4, 5, 6]
```

Lambdas in Python

- Python has limited support for functional programming so doesn't endow Lambdas with much power
  - Can accept multiple arguments but...
  - Single line only, no use of conditionals / loops
  - Single expression only which is its return
- Partly the lack of support stems from Guido's preference for other styles

  > *About 12 years ago, Python aquired lambda, reduce(), filter() and map(), courtesy of (I believe) a Lisp hacker who missed them and submitted working patches. But, despite of the PR value, I think these features should be cut from Python 3000.*
  > *– Guido van Rossum, "The fate of reduce() in Python 3000", March 10, 2005*

- Functional languages like OCaml and Racket will have richer support for Lambdas and related **lexical closures**

Filter

Create a smaller DS (list) containing only elements that return True from filter function

```
# filter_demo.py:
words = ["apple","banana","apricot","grape","artichoke"]

awords = list(filter(lambda x: x[0]=="a", words))
print(awords)         # ['apple', 'apricot', 'artichoke']

short_words = list(filter(lambda x: len(x) <= 5, words))
print(short_words)    # ['apple', 'grape']

long_words = list(filter(lambda x: len(x) > 5, words))
print(long_words)     # ['banana', 'apricot', 'artichoke']

all_words = list(filter(lambda x: 5.5, words))
print(all_words)      # entire list due to 5.5 being truthy
```

Reduce

- Generalizes "summing a list": initial value 0, add each item
- Reduce allows operations other than "add" and other initial values than "0" so that
- Create a single value from a DS of elements by repeatedly applying an operation beginning with an initial value
- `reduce()` requires an `import` from `functools` as it was dropped funcs automatically available
- Reductions come up elsewhere in computing and are worth noting

Reduce Examples

```
# reduce_demo.py:
from functools import reduce    # reduce() not in default imports

nums = [10,20,30,40]            # some date to operat on

asum0 = reduce(lambda cur,x: x+cur, nums, 0)   # sum starting at 0
print(asum0)       # 100

asum13 = reduce(lambda cur,x: x+cur, nums, 13) # sum starting at 13
print(asum13)      # 113

asum_def = reduce(lambda cur,x: x+cur, nums)   # default to sum list only
print(asum_def)    # 100

aprod1 = reduce(lambda cur,x: x*cur, nums, 1)  # product of list, init 1
print(aprod1)      # 240000

aprod_def = reduce(lambda cur,x: x*cur, nums)  # product of list only
print(aprod_def)   # 240000

astr = reduce(lambda cur,x: cur+str(x)+" ", nums, "") # string concat
print(astr)        # "10 20 30 40 "

amax = reduce(lambda cur,x: x if x>cur else cur, nums) # reduce via max
print(amax)        # 40

amax2 = reduce(max, nums)                      # max() func used directly
print(amax)        # 40

print(max(nums))                               # pythonic style
```

Iter

- Iterate over a DS (list) and apply a function solely for side effects (e.g. printing, writing to file, logging, etc.)
- Being an imperative language, Iter is not available in standard Python as it is more canonical to use a `for` loop
- Available in via the `more_itertools` package as `side_effect`
- Additionally requires use of the `consume()` function to evaluate all iterations

```
from more_itertools import *

words = ["apple","banana","apricot","grape","artichoke"]
consume(side_effect(lambda x: print(x),words))
# prints all words

alist=[]                                # empty list
consume(side_effect(lambda x: alist.append(x),words))
# iterate over words appending to alist

print(alist)                            # copy of words[]
```

Python List Comprehensions

- Python has other mechanisms that are more canonical than Map/Reduce/Filter
- **List comprehensions** are a semi-complex syntax to create lists and are often used in place of Map / Filter
- Worth knowing about but NOT a subject of further discussion in CMSC330

```
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> [2*x for x in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

>>> [x for x in range(20) if x%2==0]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

#  transform         iterable   filter
>>> [3*x+1 for x in range(20) if x%2==0]
[1, 7, 13, 19, 25, 31, 37, 43, 49, 55]

>>> words = ["apple","banana","apricot","grape","artichoke"]
>>> [x for x in words if x[0]=="a"]
['apple', 'apricot', 'artichoke']
```

Python's sort() w/ First-Class Functions

- One common place you will see functions passed as arguments is in Sorting functions
- The **comparison / comparator** function is what is used to compare elements and determine sorting order as used in Java, C, OCaml, Racket, and most other PLs
- Python has a limited version of this, a "key" parameter that allows transformation of values in the list
- Will revisit first-class funcs in OCaml / Racket to see this

```python
# sort_demo.py:
nums = [23426, -16781, 9963, 10870, 677,
        -21218, 22541, 11610, 24488, -24855]

nums.sort()                     # sort the list
print(nums)                     # w/ standard order
# [-24855, -21218, -16781, 677, 9963, 10870, 11610, 22541, 23426, 24488]
nums.sort(key=abs)              # sort by absolute
print(nums)                     # value via abs()
# [677, 9963, 10870, 11610, -16781, -21218, 22541, 23426, 24488, -24855]
nums.sort(key=lambda x: -x)     # sort in reverse
print(nums)                     # via a lambda
# [24488, 23426, 22541, 11610, 10870, 9963, 677, -16781, -21218, -24855]
```

Nested Functions and Scope in Python

```
1  # nested_scope.py:
2  def outer_func(oarg):
3    # oloc = "q"
4
5    def inner_func1(iarg):
6      iloc = "j"
7      print(f"inner_func1():")
8      print(f"  iloc:{iloc} iarg:{iarg}")
9      print(f"  oloc:{oloc} oarg:{oarg}")
10     return 1
11
12   def inner_func2(iarg):
13     iloc = "k"
14     print(f"inner_func2():")
15     print(f"  iloc:{iloc} iarg:{iarg}")
16     print(f"  oloc:{oloc} oarg:{oarg}")
17     return 2
18
19   oloc = "q"
20   r1 = inner_func1("x")
21   oloc = "u"
22   r2 = inner_func2("y")
23   # print(iloc)  # error
24   return r1+r2
25
26
27 r = outer_func("a")
28 print(r)
```

- Python supports nested functions with more/less expected behavior of **scoping**
- Scope: where variable / symbol is visible and can be used
- Inner functions have access to outer function variables
  - `inner_func1()` can "see" `oarg` and `oloc` from the outer scope
  - Likewise for `inner_func2()`
- Outer scope cannot "see" inner variables: line 23 error