Python Programming

By

Subhadip Mukherjee

Dept. of Computer Science and BCA,
Kharagpur College,
Kharagpur, India

LECTURE 4

For CBCS 4th Semester

FUNCTIONAL PROGRAMMING TOOLS

Last time, we covered function concepts in depth. We also mentioned that Python allows for the use of a special kind of function, a *lambda* function.

Lambda functions are small, anonymous functions based on the lambda abstractions that appear in many functional languages.

As stated before, Python can support many different programming paradigms including functional programming.

Right now, we'll take a look at some of the handy functional tools provided by Python.

LAMBDA FUNCTIONS

Lambda functions within Python.

- Use the keyword *lambda* instead of *def*.
- Can be used wherever function objects are used.
- Restricted to one expression.
- Typically used with functional programming tools.

```
>>> def f(x):
...     return x**2
...
>>> print f(8)
64
>>> g = lambda x: x**2
>>> print g(8)
64
```

FUNCTIONAL PROGRAMMING TOOLS

Filter

- filter(*function*, *sequence*) filters items from sequence for which function(*item*) is true.
- Returns a string or tuple if sequence is one of those types, otherwise result is a list.

```
def even(x):
    if x % 2 == 0:
        return True
    else:
        return False

print(filter(even, range(0,30)))
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

FUNCTIONAL PROGRAMMING TOOLS

Map

- map(*function, sequence*) applies function to each item in sequence and returns the results as a list.
- Multiple arguments can be provided if the function supports it.

```
def square(x):
    return x**2

print(map(square, range(0,11)))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

FUNCTIONAL PROGRAMMING TOOLS

Map

- map(*function, sequence*) applies function to each item in sequence and returns the results as a list.
- Multiple arguments can be provided if the function supports it.

```
def expo(x, y):
    return x**y

print(map(expo, range(0,5), range(0,5)))
```

```
[1, 1, 4, 27, 256]
```

FUNCTIONAL PROGRAMMING TOOLS

Reduce

- reduce(*function, sequence*) returns a single value computed as the result of performing *function* on the first two items, then on the result with the next item, etc.
- There's an optional third argument which is the starting value.

```
def fact(x, y):
    return x*y

print(reduce(fact, range(1,5)))
```

```
24
```

FUNCTIONAL PROGRAMMING TOOLS

We can combine lambda abstractions with functional programming tools. This is especially useful when our function is small – we can avoid the overhead of creating a function definition for it by essentially defining it in-line.

```
>>> print(map(lambda x: x**2, range(0,11)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

MORE DATA STRUCTURES

- Lists
  - Slicing
  - Stacks and Queues
- Tuples
- Sets and Frozensets
- Dictionaries
- How to choose a data structure.
- Collections
  - Deques and OrderedDicts

WHEN TO USE LISTS

- When you need a non-homogeneous collection of elements.
- When you need to ability to order your elements.
- When you need the ability to modify or add to the collection.
- When you don't require elements to be indexed by a custom value.
- When you need a stack or a queue.
- When your elements are not necessarily unique.

CREATING LISTS

To create a list in Python, we can use bracket notation to either create an empty list or an initialized list.

```
mylist1 = [] # Creates an empty list
mylist2 = [expression1, expression2, ...]
mylist3 = [expression for variable in sequence]
```

The first two are referred to as *list displays*, where the last example is a *list comprehension*.

CREATING LISTS

We can also use the built-in list constructor to create a new list.

```
mylist1 = list()
mylist2 = list(sequence)
mylist3 = list(expression for variable in sequence)
```

The sequence argument in the second example can be any kind of sequence object or iterable. If another list is passed in, this will create a copy of the argument list.

CREATING LISTS

Note that you cannot create a new list through assignment.

```
# mylist1 and mylist2 point to the same list
mylist1 = mylist2 = []

# mylist3 and mylist4 point to the same list
mylist3 = []
mylist4 = mylist3

mylist5 = []; mylist6 = [] # different lists
```

ACCESSING LIST ELEMENTS

If the index of the desired element is known, you can simply use bracket notation to index into the list.

```
>>> mylist = [34,67,45,29]
>>> mylist[2]
45
```

If the index is not known, use the index() method to find the first index of an item. An exception will be raised if the item cannot be found.

```
>>> mylist = [34,67,45,29]
>>> mylist.index(67)
1
```

SLICING AND SLIDING

- The length of the list is accessible through `len(mylist)`.
- Slicing is an extended version of the indexing operator and can be used to grab sublists.

```
mylist[start:end] # items start to end-1
mylist[start:]    # items start to end of the array
mylist[:end]      # items from beginning to end-1
mylist[:]         # a copy of the whole array
```

- You may also provide a step argument with any of the slicing constructions above.

```
mylist[start:end:step] # start to end-1, by step
```

SLICING AND SLIDING

- The start or end arguments may be negative numbers, indicating a count from the end of the array rather than the beginning. This applies to the indexing operator.

```
mylist[-1]      # last item in the array
mylist[-2:]     # last two items in the array
mylist[:-2]     # everything except the last two items
```

- Some examples:

```
mylist = [34, 56, 29, 73, 19, 62]
mylist[-2]          # yields 19
mylist[-4::2]       # yields [29, 19]
```

INSERTING/REMOVING ELEMENTS

- To add an element to an existing list, use the append() method.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.append(47)
>>> mylist
[34, 56, 29, 73, 19, 62, 47]
```

- Use the extend() method to add all of the items from another list.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.extend([47,81])
>>> mylist
[34, 56, 29, 73, 19, 62, 47, 81]
```

INSERTING/REMOVING ELEMENTS

- Use the insert(*pos, item*) method to insert an item at the given position. You may also use negative indexing to indicate the position.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.insert(2,47)
>>> mylist
[34, 56, 47, 29, 73, 19, 62]
```

- Use the remove() method to remove the first occurrence of a given item. An exception will be raised if there is no matching item in the list.

```
>>> mylist = [34, 56, 29, 73, 19, 62]
>>> mylist.remove(29)
>>> mylist
[34, 56, 73, 19, 62]
```

LISTS AS STACKS

- You can use lists as a quick stack data structure.
- The append() and pop() methods implement a LIFO structure.
- The pop(*index*) method will remove and return the item at the specified index. If no index is specified, the last item is popped from the list.

```
>>> stack = [34, 56, 29, 73, 19, 62]
>>> stack.append(47)
>>> stack
[34, 56, 29, 73, 19, 62, 47]
>>> stack.pop()
47
>>> stack
[34, 56, 29, 73, 19, 62]
```

LISTS AS QUEUES

- Lists *can* be used as queues natively since insert() and pop() both support indexing. However, while appending and popping from a list are fast, inserting and popping from the beginning of the list are slow (especially with large lists. Why is this?).
- Use the special *deque* object from the *collections* module.

```
>>> from collections import deque
>>> queue = deque([35, 19, 67])
>>> queue.append(42)
>>> queue.append(23)
>>> queue.popleft()
35
>>> queue.popleft()
19
>>> queue
deque([67, 42, 23])
```

OTHER OPERATIONS

- The count(x) method will give you the number of occurrences of item x within the list.

```
>>> mylist = ['a', 'b', 'c', 'd', 'a', 'f', 'c']
>>> mylist.count('a')
2
```

- The sort() and reverse() methods sort and reverse the list in place. The sorted(mylist) and reversed(mylist) built-in functions will return a sorted and reversed copy of the list, respectively.

```
>>> mylist = [5, 2, 3, 4, 1]
>>> mylist.sort()
>>> mylist
[1, 2, 3, 4, 5]
>>> mylist.reverse()
>>> mylist
[5, 4, 3, 2, 1]
```

CUSTOM SORTING

- Both the sorted() built-in function and the sort() method of lists accept some optional arguments.

```
sorted(iterable[, cmp[, key[, reverse]]])
```

- The *cmp* argument specifies a custom comparison function of two arguments which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument. The default value is None.
- The *key* argument specifies a function of one argument that is used to extract a comparison key from each list element. The default value is None.
- The *reverse* argument is a Boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

CUSTOM SORTING

```
>>> mylist = ['b', 'A', 'D', 'c']
>>> mylist.sort(cmp = lambda x,y: cmp(x.lower(), y.lower()))
>>> mylist
['A', 'b', 'c', 'D']
```

Alternatively,

```
>>> mylist = ['b', 'A', 'D', 'c']
>>> mylist.sort(key = str.lower)
>>> mylist
['A', 'b', 'c', 'D']
```

str.lower() is a built-in string method.

Thank You