Lab 8: Advanced Functions

CSE/IT 107

NMT Computer Science

"All thought is a kind of computation."

— D. Hobbes

"Vague and nebulous is the beginning of all things, but not their end."

— K. Gibran

"It [programming] is the only job I can think of where I get to be both an engineer and artist. There's an incredible, rigorous, technical element to it, which I like because you have to do very precise thinking. On the other hand, it has a wildly creative side where the boundaries of imagination are the only real limitation."

— A. Hertzfeld

---

1 Introduction

2 Advanced Functions

2.1 Default Arguments

Many of the built-in functions we have been using, such as `range` or `input`, accept a variable number of arguments. We can do the same thing with our own funcitions by specifying default values for our arguments:

```
def celebrate(times=1):
    for i in range(times):
        print("Yay!")

print("First call:")
celebrate()
print("Second call:")
celebrate(5)
```

Note what happens for each call of this function. The first time it only prints `"Yay!"` once, since the default argument is `1`, while the second call overwrites the default value and prints it five times.

While you are allowed to have a function that has some arguments with default values and some without, you must always put those with default values after those without any, so that Python will know which arguments go into which values if you don't include all of them when calling the function:

```
>>> def test(a, b=5, c, d=6):
...   print(a, b, c, d)
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
>>> def test(a, c, b=5, d=6):
...   print(a, b, c, d)
...
>>> test(2, 3)
2 5 3 6
>>> test(2, 3, 10)
2 10 3 6
>>> test(2, 3, 10, 100)
2 10 3 100
```

Sometimes we might want to include some default arguments while excluding others. We do this by specifying which of the arguments we wish to pass a value to:

```
>>> def test(a=0, b=0, c=0, d=0):
...   print(a, b, c, d)
...
>>> test(c=10)
0 0 10 0
>>> test(d=5, c=6)
0 0 6 5
```

2.2 Recursion

Recursion is the idea of a function calling itself. This can be useful in a wide variety of situations, but it can also be easy to misuse. As an example, see this simple case of recursion:

```
1 def test():
2     test()
```

If this function is called, it will continue to call itself infinitely until Python prints out an error message and quits. This is because the function calls itself every time it is called – there is no case where the function ends without calling itself again. This case is called a base case, and it is important to include one in every recursive function. Here is a modified version of the above code that allows the initial call to specify a value limiting how many times the recursive call should be made:

```
1 def test(x):
2     if x > 0:
3         test(x - 1)
```

As an example of a more complex way of using recursion to solve a problem, here is an example of a recursive function that will sum a list where each element is either an integer or another list containing integers and lists:

```
1  def sum_lists(x):
2      total = 0
3      for i in x:
4          if type(i) == int:
5              total += i
6          else:
7              total += sum_lists(i)
8      return total
9
10 data = [1, 2, [3, 4, [5, 6], 7], 8, 9]
11 print(sum_lists(data))
```

This program outputs `45`, which is the sum of all the elements of the list, including all of the lists inside it. Notice that the base case in this function is if every element of the list passed to `sum_lists` is an `int`.

3 First-class Functions

In Python, functions are "first-class citizens". This means, essentially that they can be used in the same way as variables: they can be assigned to variables, passed to and returned from functions, and stored in data structures. As an example, here is a short program that stores functions inside of a list, then calls each one of those functions with the same argument.

```
1 def print_backwards(string):
2     print(string[::-1])
3
4 def print_half(string):
5     print(string[:len(string)//2])
6
```

```
def print_question(string):
    print("Why do you want me to print '{}'?".format(string))

print_functions = [print, print_backwards, print_half, print_question]

for func in print_functions:
    func("Hello, World!")
```

```
Hello, World!
!dlroW ,olleH
Hello,
Why do you want me to print 'Hello, World!'?
```

3.1 Map

A common operation in Python is to call the same function for every element in a list, then create a new list with the results. Previously, we would have accomplished this with a `for` loop:

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

data = [5, 23, 76]
new_data = []

for i in data:
    new_data.append(double(i))

print(new_data)
```

```
[10, 46, 152]
```

If we use the `map` function, we can make this code much more simple:

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

data = [5, 23, 76]
data = list(map(double, data))
print(data)
```

```
[10, 46, 152]
```

The `map` function takes in a function and any iterable data type (such as a list, tuple, set, etc.) as arguments. It calls that function for every element of the other value passed, then returns a new data type containing all of those results. This new data type is a special map object, but in most cases we are fine with instantly converting it back to our initial data type (in the above example, a list).

3.2 Reduce

`reduce` is a similar function to `map`, but rather than create a new list with the modified values, `reduce` should be used to simplify a list into a single value. Similarly to `map`, `reduce` takes in a

function and an iterable data type. The difference is that, rather than taking and returning a single value, the function passed to `reduce` takes in two arguments and returns their combination. As an example, here is an example of using `reduce` to join a list of strings into a single larger string.

```
import functools

def combine(glob, new):
    print("Adding {} to {}.".format(new, glob))
    return glob + new

data = ['this', 'is', 'a', 'test']
data = functools.reduce(combine, data)
print("Final value: {}".format(data))
```

Note that we need to run `import functools` in order to have access to `reduce`.

3.3 Filter

`filter` is a function that allows easy removal of some elements from a list. It works similarly to `map`, but the function used needs to return either `True` or `False`. If it returns `True` then the given element is included in the final list. If `False` is returned, then it is not. As an example, here is a short program that removes all odd elements from a list:

```
def is_even(x):
    return x % 2 == 0

data = [5, 3, 34, 36, 38, 1, 0, 0, 2]
data = list(filter(is_even, data))
print(data)
```

```
[34, 36, 38, 0, 0, 2]
```

3.4 Lambda Functions

Lambda functions are anonymous functions that are created while the program is running that are not assigned to a name like normal functions. They can be used very similarly to normal functions if assigned to a variable:

```
>>> def f(x):
...     return x + 1
...
>>> g = lambda x: x + 1
>>> f(5)
6
>>> g(5)
6
```

In the above case, `f` and `g` are functionally equivalent. Both are functions that take in a single value and return that value increased by one. The primary difference is in syntax: note that the lambda function does not have a return statement: it will simply return the value computed after the colon.

Lambda functions are commonly used in conjunction with `map`, `reduce`, and `filter` as shown below:

```
data = [5, 3, 34, 36, 38, 1, 0, 0, 2]
data = list(filter(lambda x: x % 2 == 0, data))
print(data)
```

```
[34, 36, 38, 0, 0, 2]
```

This program works just like the example shown in the section on `filter`, but it does not require us to declare the separate `is_even` function. Instead, we simply define a lambda inside the call to `filter`, accomplishing the same thing.

We can similarly use lambda functions to shorten our initial example of first-class functions:

```
print_functions = [print,
    lambda string: print(string[::-1]),
    lambda string: print(string[:len(string)//2]),
    lambda string: print("Why do you want me to print '{}'?".format(string))]

for func in print_functions:
    func("Hello, World!")
```

One extra use for lambda functions is to modify default sorting behavior. For example, if we have a list of tuples and try to sort them, Python will sort them based on their first element, then second if the first elements are the same, and so on:

```
>>> data = [(5, 2, 4), (6, 3, 2), (4, 4, 4), (3, 3, 3), (5, 3, 10)]
>>> sorted(data)
[(3, 3, 3), (4, 4, 4), (5, 2, 4), (5, 3, 10), (6, 3, 2)]
```

However, what if we want to sort these tuples based on their last element? To do this, we define a special `key` value for `sorted`. Each value of the tuple will be passed to this function, and then the return values will be used when sorting in place of the actual value of the tuple:

```
>>> data = [(5, 2, 4), (6, 3, 2), (4, 4, 4), (3, 3, 3), (5, 3, 10)]
>>> sorted(data, key=lambda x: x[-1])
[(6, 3, 2), (3, 3, 3), (5, 2, 4), (4, 4, 4), (5, 3, 10)]
```

4 Exercises

Boilerplate

Remember that this lab *must* use the boilerplate syntax introduced in Lab 5, including the review exercises.

fractions2.py Using the same tuple representation of fractions as in Lab 6, do the following:

1. Write a function `min_frac` such that calling `functools.reduce(min_frac, list_of_fractions)` will return the smallest value fraction in the list.
2. Write a function `sum_frac` such that calling `functools.reduce(sum_frac, list_of_fractions)` will return the sum of all the fractions in the list.
3. Write a function `reduce_frac` that takes in a fraction and returns that fraction reduced to its lowest terms. Use this function with `map` to reduce a list of fractions.

5 Submitting

We will be adding more exercises later. We have just not had the time to finish them. You will get an email about them.

Files to submit:

- fractions2.py (Section 4)

You may submit your code as either a tarball (instructions below) or as a .zip file. Either one should contain all files used in the exercises for this lab. The submitted file should be named either `cse107_firstname_lastname_lab8.zip` or `cse107_firstname_lastname_lab8.tar.gz` depending on which method you used.

For Windows, use a tool you like to create a `.zip` file. The TCC computers should have `7z` installed. For Linux, look at lab 1 for instructions on how to create a tarball or use the "Archive Manager" graphical tool.

Upload your tarball or .zip file to Canvas.