Lab 8: Advanced Functions

CSE/IT 107

NMT Department of Computer Science and Engineering

"I think there is a world market for maybe five computers."

— Thomas J. Watson (Founder and Chairman, IBM)

"Judge a man by his questions, rather than his answers."

— Voltaire

"The limits of your language are the limits of your world."

— Ludwig Wittgenstein

---

1 Introduction

Congratulations! In your last three weeks of lab, you have reached the basic knowledge required to work with Python in a useful way. You have learned all the tools required to write basic programs in Python and you have applied them successfully in the programming competition.

This week, we will teach you a bit more about functions and give you a small window to a different kind of programming: functional programming. Here, the word "functional" is not to mean "practical" (although it is very practical!), but that it has to do with a lot of functions.

2 Advanced Functions

2.1 Keyword Arguments

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

```
>>> def celebrate(times=1):
...     i = 1
...     while i <= times:
...         print("Yay!")
...         i += 1
...
>>> celebrate()
Yay!
>>> celebrate(5)
Yay!
Yay!
Yay!
Yay!
Yay!
```

Here, we have a function called `celebrate` which takes the parameter variable `times`. If that parameter is left unspecified in a function call, it will take the *default value* of 1.

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 do not include all of them when calling the function. This is because arguments from a function call are usually matched in order by their position.

```
>>> 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
```

We call the arguments that are non-default arguments *positional arguments*, while the default arguments are called *keyword arguments*.

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
```

- If a function's caller supplies both positional and keyword arguments, all positional parameters must precede all keyword parameters.
- Keyword parameters may occur in any order.
- The function call must supply at least as many parameters as the function has positional arguments.
- If the caller supplies more positional parameters than the function has positional arguments, parameters are matched with keyword arguments according to their position.

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. Let us first look at a good example:

```
def factorial(n):
  if n <= 1:
    return 1
  else
    return n * factorial(n-1)
```

Notice that this is just another way to repeat a statement, similar to a loop. In a loop, we have to specify when to stop repeating something, and we have to do the same thing with recursion; otherwise, it would run forever.

As an example, see this simple *bad* case of recursion:

```
def test():
    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. The case in which the recursion ends 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:

```
def test(x):
    if x > 0:
        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:

```
def sum_lists(x):
    total = 0
    for i in x:
        if type(i) == int:
            total += i
        else:
            total += sum_lists(i)
    return total

data = [1, 2, [3, 4, [5, 6], 7], 8, 9]
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`.

2.3 Summary

Default arguments Function arguments may have a default value, for example:

```
import time
def print_error(message, now = time.time()):
  print("Error: {} at time {}!".format(message, now))

print_error('All wrong')
print_error('All more wrong', now = 1.5)
```

Default arguments must be specified after arguments that do not have defaults.

Recursion Recursion occurs when a function calls itself. There must be a base case to stop the recursion!

3 First-class Functions

In Python, functions are so-called "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, as well as stored in collections. 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.

```
def print_backwards(string):
    print(string[::-1])

def print_half(string):
    print(string[:len(string)//2])

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

# Just use the name of the function!
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!'?
```

This can lead to cool pieces of code where a function takes another function as an argument and uses it in some way. For example, we can use it to write a function that composes two other functions and calls them with a given x. In math, this would be like taking a function $f(x)$ and a function $g(x)$ and composing them to be $h(x) = f(g(x))$. The function below takes f and g and finds the value of h at a x.

```
def h(f, g, x):
  return f(g(x))
```

3.1 Map

A common operation in Python is to call the same function for every element in a collection, such as a list or a tuple and 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))
```

```
Adding this to is
Adding thisis to a
Adding thisisa to test
Final value: thisisatest
```

Note that we need to run `import functools` in order to have access to `reduce`.
Let's take a simpler example:

```
import functools

def add(x, y):
  return x + y

numbers = [1, 3, 4, 6, 7]
s = functools.reduce(add, numbers)
```

`s` is the sum of the values in the list `numbers`. The way `reduce` works is that it takes the first two values, calls the `add` function on them, returns the result, and then takes that result and the next number. In essence, the example above expands to this:

```
s = add(add(add(add(1, 3), 4), 6), 7)
```

This shows that at each step, you get to see an intermediate value: The first call to `add` gets the first two values, 1 and 3, while the second call gets the addition of those, 4 and the next value in the list, 4. The third call to `add` gets the result of that, 8, and the next number, 6. This will go on until the list is exhausted.

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 and 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; we have just used a different syntax to describe them. 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!")
```

Lambda functions can take more than one argument as well:

```
multiply = lambda x, y: x * y
print(multiply(3, 5)) # 15
```

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)]
```

Notice that `key` is just a default argument to the function `sorted`. The default value would be a function that sorts based on the first element.

Lambda functions also allow us to create functions on the fly. In the first-class functions section, we composed two functions f and g and found the value at a point x. However, sometimes we do not want the value at x, we want a function that takes in a parameter x and returns the composition. Let's say, for example, that we want a function $h(x) = sqrt(factorial(x))$. Take a look at the following code:

```
import math

def compose(f, g):
  # Creates another function that is the composition of f and g and returns it
  return lambda x: f(g(x))

# Using math.sin and math.cos as first-class citizens
h = compose(math.sqrt, math.factorial)

# compose returns a function, so we can use h like a function!
print(h(6))
print(h(10))

m = compose(math.sin, math.cos)
print(m(3))
print(m(4))
```

```
26.832815729997478
1904.9409439665053
-0.8360218615377305
-0.6080830096407656
```

If you check the answers, you will find that $\sqrt{6!} = \sqrt{720} = 26.832\cdots$ and similarly for the others.

Continuing the code, we could now compose h and `math.log10` (Logarithm with base 10):

```
logh = compose(math.log10, h)
print(logh(6)) # 1.4286662482156343
```

We can also plug in a lambda function of our choice to the composition function. For example, if we wanted a function $p(x) = \sqrt{x^3}$:

```
p = compose(math.sqrt, lambda x: x**3)
print(p(100)) # 1000.0
print(p(10)) # 31.622776601683793
```

4 Exercises

Boilerplate

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

Exercise 4.1 (palindrome.py).
Write a recursive function that determines whether a string is a palindrome. A palindrome is a word that is the same forwards and backwards, for example "abba".

Exercise 4.2 (fractions2.py).
Using the same tuple representation of fractions as in Lab 6, do the following:

1. Write a function `prompt_fractions()` that implements the following interface that prompts the user for as many fractions as he/she wants to enter:

```
Enter fraction: 10/2
Enter fraction: 5/7
Enter fraction: 3/8
Enter fraction: stop
```

   Use `map` to convert each string, such as "10/2" to a tuple of integers.

   The function should return a list of tuples. In the example, it would return:

```
[(10, 2), (5, 7), (3, 8)]
```

2. Write a function `min_frac` such that calling `functools.reduce(min_frac, list_of_fractions)` will return the smallest value fraction in the list.
3. Write a function `sum_frac` such that calling `functools.reduce(sum_frac, list_of_fractions)` will return a fraction that is the sum of all the fractions in the list.
4. 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. Write a function `reduced` such that calling `filter(reduced, list_of_fractions)` will return a list consisting of only fractions that are reduced to their lowest terms.
6. Write a function `sort_frac` that takes in a list of fractions and uses `sorted` and a lambda `key` function to sort a list of fractions from smallest to largest.

Do not use any built-in fraction functions or libraries.

You should use the following boilerplate code for `fractions2.py`:

```python
import functools

# ADD YOUR FUNCTIONS HERE

def main():
  list_fractions = prompt_fractions()

  format_frac = lambda frac : "{}/{}".format(frac[0], frac[1])

  smallest = functools.reduce(min_frac, list_fractions)
  print("Smallest fraction:", format_frac(smallest))

  sumfrac = functools.reduce(sum_frac, list_fractions)
  print("Sum of fractions:", format_frac(sumfrac))

  # reducedfrac = map() ??? You fill in!
  # print("Reduced fractions:", reducedfrac)

  fracsorted = sort_frac(list_fraction)
  print("Sorted fractions:", fracsorted)

if __name__ == '__main__':
  main()
```

Exercise 4.3 (rpn.py).
Take `rpn_calculator.py` from Lab 6 and rewrite it to use a dictionary of lambda functions to do the actual operation. Operations you should support are $+ - * /$. Also, add a sin and cos operation. For example: 0 sin should give 0, while 1 2 + sin should give 0.1411200080598672 (this is sin(3)).

5 Submitting

You should submit your code as a tarball. It should contain all files used in the exercises for this lab. The submitted file should be named

`cse107_firstname_lastname_lab8.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit