National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester II, 2017/2018

Mission 3
More Than Thrice

Release date: 07 February 2018
Due: 13 February 2018, 23:59

Required Files

- mission03-template.py

Background

One of the things that makes Python different from other common programming languages is the ability to operate with *higher-order* functions, namely, functions that manipulate and generate other functions.

The function `square` may be applied to a `float` (or `int`) and will return another `float` (or `int`). We indicate this with the notation:

$$\texttt{sqr} : \texttt{float}|\texttt{int} \rightarrow \texttt{float}|\texttt{int}$$

If `f` and `g` are functions of type `float`|`int` → `float`|`int`, then we may *compose* them:

```
def compose(f, g):
    return lambda x: f(g(x))
```

For example, `compose(sq, log)` is a function of type `float` → `float` that returns the square of the logarithm of its argument, while `compose(log, sq)` returns the logarithm of the square of its argument:

```
>>> from math import *
>>> def sq(x): return x**2
>>> sq(log(2))
0.4804530139182014
>>> compose(sq, log)(2)
0.4804530139182014
>>> log(sq(2))
1.3862943611198906
>>> compose(log,sq)(2)
1.3862943611198906
```

As we have used it above, the function `compose` takes as arguments two functions of type $F = \texttt{float}|\texttt{int} \rightarrow \texttt{float}|\texttt{int}$, and returns another such function. We indicate this with the notation:

$$\texttt{compose} : (F, F) \rightarrow F$$

Just as squaring a number multiplies the number by itself, `thrice` of a function composes the function three times. That is, `thrice(f)(n)` will return the same result as `f(f(f(n)))`:

```
>>> def thrice(f): return compose(compose(f,f),f)
>>> thrice(sq)(3)
6561
>>> sq(sq(sq(3)))
6561
```

As used above, `thrice` is of type $(F \to F)$. That is, it takes as input a function of type F and returns the same kind of function. But `thrice` will actually work for other kinds of input functions. It is enough for the input function F to have a type of the form $T \to T$ (instead of $F = \texttt{float}|\texttt{int} \to \texttt{float}|\texttt{int}$), where T may be any type. So more generally, we can write

$$\texttt{thrice} : (T \to T) \to (T \to T)$$

Composition, like multiplication, may be iterated. Consider the following:

$$\texttt{repeated} : ((T \to T), \texttt{int}) \to (T \to T)$$

Example:

```
>>> def identity(x): return x
>>> def repeated(f, n):
        if n == 0:
            return identity
        else:
            return compose(f, repeated(f, n-1))

>>> repeated(sin, 5)(3.1)
0.041532801333692235
>>> sin(sin(sin(sin(sin(3.1)))))
0.041532801333692235
```

Task 1: Thrice (5 marks)

(a) The type of `thrice` is of the form $(T' \to T')$ (where T' happens to equal $(T \to T)$), so we can legitimately use `thrice` as an input to `thrice`! For what value of `n` will `thrice(thrice(f))(0)` return the same value[1] as `repeated(f, n)(0)`?

(b) See if you can now predict what will happen when the following expressions are evaluated. Briefly explain what goes on in each case.

Note: Function `add1` is defined as follows:

```
def add1(x): return x + 1
```

(i) `thrice(thrice)(add1)(6)`

[1]"Sameness" of function values is a sticky issue which we don't want to get into here. We can avoid it by assuming that `f` is bound to a value of type F, so evaluation of `thrice(thrice(f))(0)` will return a number.

(ii) `thrice(thrice)(identity)(compose)`

(iii) `thrice(thrice)(sq)(1)`

(iv) `thrice(thrice)(sq)(2)`.

Task 2: Combine them together! (5 marks)

Higher order functions can be used to implement other functions as well. Consider the following higher order function called `combine`:

```
def combine(f, op ,n):
    result = f(0)
    for i in range(n):
        result = op(result, f(i))
    return result
```

(a) Let's define the smiley_sum $S(t)$ as follows:

$S(1) = 1$
$S(2) = 4 + 1 + 4 = 9$
$S(3) = 9 + 4 + 1 + 4 + 9 = 27$
$S(4) = 16 + 9 + 4 + 1 + 4 + 9 + 16 = 59$
$S(5) = 25 + 16 + 9 + 4 + 1 + 4 + 9 + 16 + 25 = 109$

If we look closer, we can actually define `smiley_sum` in terms of `combine`!

```
def smiley_sum(t):
    def f(x):
        ...

    def op(x, y):
        ...

    n = ...

    # Do not modify this return statement
    return combine(f, op, n)
```

Fill in the appropriate implementations for `f` and `op`. You are reminded to test your code.

Reminder: You are not allowed to modify the `return` statement!

(b) Your friend who had attended the lecture on higher order function challenges you to define a function that computes the n-th Fibonacci number using the function `combine`

Recall the definition for Fibonacci numbers:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

This is his challenge:

```
def new_fib(n):
    def f(x):
        ...

    def op(x, y):
        ...

    return combine(f, op, n+1)
```

Are you able to answer his challenge? If yes, provide a working implementation. If no, explain why.