National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2022/2023

Mission 3
More Than Thrice

Release date: 31st August 2022
Due: 6th September 2022, 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.

In this mission, we will be dealing with two primary types of functions:

F-num is a function that takes in a **Number** (either a `float` or `int`) and returns another **Number** (`float` or `int`). For example, the function `sq`—which returns the square of input number—is an **F-num**. We indicate this with the notation:

`sq`(**Number**) $\rightarrow$ **Number**

F-generic is a function that takes in any generic input type, and returns an output of the *same* type. More elaboration on this will be provided later.

Examples

If both `f` and `g` are functions of type **F-num**(**Number**) → **Number**, then we may *compose* them to return a new function:

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

Following the definition above, the function `compose` takes as arguments two **F-num** functions, and returns another **F-num** function. We indicate this with the notation:

`compose`(**F-num**, **F-num**) $\rightarrow$ **F-num**

For example, `compose(sq, log)` is an **F-num** that returns the square of the logarithm of its argument, while `compose(log, sq)` is a different **F-num** function that returns the logarithm of the square of its argument:

```
>>> from math import *
>>> sq = lambda x: x**2

>>> sq(log(2))          # square of the logarithm of 2
0.4804530139182014
>>> log_then_square = compose(sq, log)
>>> log_then_square(2) # square of the logarithm of 2
0.4804530139182014

>>> log(sq(2))          # logarithm of the square of 2
1.3862943611198906
>>> square_then_log = compose(log, sq)
>>> square_then_log(2) # logarithm of the square of 2
1.3862943611198906
```

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, we observer that `thrice` takes as input a **F-num** and returns a new **F-num**. That is, `thrice`(**F-num**) $\rightarrow$ **F-num**. But `thrice` will actually work for other kinds of input functions. In fact, it is enough for the input function to be of the form **F-generic**, `<function>`(T) $\rightarrow T$ (where T is some type). So more generally, we can write:

`thrice`(**F-generic**) $\rightarrow$ **F-generic**

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

`repeated`(**F-generic**, `int`) $\rightarrow$ **F-generic**

Example:

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

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

This mission consists of **two** tasks.

Task 1: Thrice (4 marks)

(a) The form `thrice`(**F-generic**) $\rightarrow$ **F-generic** means that `thrice` itself is an **F-generic**. Therefore, we can legitimately use `thrice` as an input to `thrice`! What value of `n` will `repeated(f, n)(0)` return the same value[1] as `thrice(thrice(f))(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)`

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

(iii) `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):
        ... # <--- your code here

    def op(x, y):
        ... # <--- your code here

    n = ... # <--- your code here

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

[1] "Sameness" of function values is a sticky issue which we don't want to get into at this point. We can avoid it by assuming that `f` is of type **F-num**, so evaluation of `thrice(thrice(f))(0)` will return a **Number**.

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

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

(b) Your friend who attended the lecture on higher order functions 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):
        ... # <--- your code here

    def op(x, y):
        ... # <--- your code here

    # Do not modify this return statement
    return combine(f, op, n+1)
```

Fill in the appropriate implementations for `f` and `op` **only**. Are you able to answer his challenge? If yes, provide a working implementation. If no, explain why.

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