# Module 3 Lab: Timing Functions

## Reminder

Reminder - wait till your lab section to begin this assignment. You'll earn lab participation by working on these problems with a partner during lab.

## Part 1 - Timing Functions

Start by writing a basic function `time_function(func, args)` that returns the number of seconds to run `func` with `args`.

`TimeFunctions.py` contains some trials that should have a running time ratio of ~10x to help test your function.

### Starter code

```
if __name__ == '__main__':
    def test_func(L):
        for item in L:
            item *= 2

    L1 = [i for i in range(10**5)]
    t1 = time_function(test_func, L1)

    L2 = [i for i in range(10**6)]
    t2 = time_function(test_func, L2)

    print("t(L1) = {:.3g} ms".format(t1*1000))
    print("t(L2) = {:.3g} ms".format(t2*1000))
```

### Expected behavior in terminal:

```
$ python3 TimeFunctions.py
t(L1) = 4.99 ms
t(L2) = 51.9 ms
```

## Part 2 - Better Results

Running your function one time leaves a lot of room for noise - other processes are vying for resources on your computer. We can get a better reading by running the function multiple times and returning just the **minimum**. Note that we should use the minimum time from multiple trials, not the average, as that is probably the trial with the least amount of noise.

Modify `time_function` to take an extra parameter, `n_trials`, which defaults to 10. Return the minimum time from 10 trials.

Note - python let's you use "infinity" with `float('inf')`. This may be helpful here.

## Part 3 - More flexible functions

Your method as-is probably can only time functions that take a single argument. This is an issue if you want to handle functions with an arbitrary number of parameters. Say we have two addition methods:

```
def add_2_nums(p1, p2):
    return p1 + p2

def add_3_nums(p1, p2, p3):
    return p1+p2+p3
```

If we try to time these functions, we'll get an error:

```
>>> time_function(add_2_nums, 3, 4, n_trials=10)
...
TypeError: time_function() got multiple values for argument 'n_trials'
```

Python thinks `4` is supposed to be the number of trials, rather than a second argument.

We can use **tuple unpacking** to handle an arbitrary number of arguments instead. We'll pack all of our arguments into a tuple, then unpack that tuple when we call the function we want to time:

```
>>> def foo(f, args):
...     return f(*args) # The * unpacks "args" into separate arguments
...
>>> foo(add_2_nums, (3, 4))
7
>>> foo(add_3_nums, (3, 4, 5))
12
```

Add a function `time_function_flexible()` that operates as above, but it's middle parameter should be a tuple of arguments:

- `f` - the function to be executed
- `args` - a tuple of an arbitrary number of arguments to be passed to `f`
- `n_trials` - the number of trials you want to run

## Submitting

At a minimum, submit the following files:

- `TimeFunctions.py`

Students must submit by the due date (typically, Friday at 11:59 pm EST) to receive credit.