

Note: In order for this lab (and all of your future assignments) to work properly, you need to download and import the `cs115.py` file from Canvas. Once you've downloaded it into your CS 115 workspace folder, you can import it by typing the following block of code at the top of your Python source file:

```
from cs115 import [what you need]
```

If you need only the `map` function, type this:

```
from cs115 import map
```

If you need multiple functions, say `map` and `reduce`, type this:

```
from cs115 import map, reduce
```

Lab1! - Google's "Secret"

In this problem you will write a few interesting Python functions using Python's `map` and `reduce` functions. These functions are at the heart of the software in Google. You can Google "mapreduce" to see evidence of this.

Please be sure to include a docstring for every function that you write.

```
map and reduce
```

Consider a very simple `dbl` function:

```
def dbl(x):
    """Returns twice its input x

    input x: a number (int or float)"""
    return 2 * x
```

Now, take a look at this example:

```
>>> mylist = range(1, 5)

>>> mylist

[1, 2, 3, 4]

>>> newlist = map(dbl, mylist)

>>> newlist
```

```
[2, 4, 6, 8]
```

```
>>> mylist
```

```
[1, 2, 3, 4]
```

Notice that the `map` function (which is built in to Python) took two inputs: The first is the name of a function (in this case `dbl`) and the second is a list (in this case the list `[1, 2, 3, 4]` that we created with `range`). The result of `map(dbl, mylist)` was a new list which has the same number of elements as the original list but every element got doubled! What actually happens here is that `map` causes each element in `mylist` to get "plopped" in to the `dbl` function. The `dbl` function then returns a new number (the double of its input) and that new number goes into the new list which `map` then returns to us.

Recall that the built-in `sum` function takes a list of numbers as input and returns the sum of the numbers in the list. For example:

```
>>> sum(range(1, 5))
```

```
10
```

Remember that `range` generates a list that does not include the endpoint, so `range(1, 5)` returns the list `[1, 2, 3, 4]`.

Here is a function that takes as input an integer `n` and returns the sum $0 + 2 + 4 + \dots + 2n$.

```
def doublesum(n):
    """Returns the sum 0 + 2 + ... + 2n"""
    list1 = range(1, n+1)
    list2 = map(dbl, list1)
    answer = sum(list2)
    return answer
```

Of course, this could also have been written this way:

```
def doublesum1(n):
    return sum(map(dbl, range(1, n+1)))
```

And then we could have been sneaky and factored out the 2 to compute $2(0 + 1 + \dots + n)$ and written it this way:

```
def doublesum2(n):
```

```
return 2 * sum(range(1, n+1))
```

Your first task: Going natural

First, write a very short and simple function called `inverse(n)` that takes a number `n` as input and returns its reciprocal. This function should always return a floating point number, even if the input is an integer. For example:

```
>>> inverse(3)

0.33333333333333331
```

Notice that Python has a `1` at the end of that reciprocal. Soon, we'll talk more about why Python erred here.

Next, write a function called `e(n)` that approximates the mathematical value `e` using a Taylor expansion. You may recall that `e` can be expressed as the sum $1 + 1/1! + 1/2! + 1/3! + \dots$. We'll approximate `e` by adding up just the first `n` terms of this sequence (after the leading 1) where `n` is some positive integer provided by the user. For example:

```
>>> e(1)

2

>>> e(2)

2.5

>>> e(3)

2.6666666666666666

>>> e(10)

2.718281801146385
```

To this end, you will need to use the `factorial` function in the `math` module. You'll also need to use your `inverse` function and `map` (possibly more than once)!

Finally, write a function called `error(n)` that returns the absolute value of the difference between the "actual" value of `e` (you can get this using `math.e`) and the approximation in your `e(n)` function assuming that `n` terms (beyond the leading 1) are used. The absolute value function is built-in to Python and is called `abs`. Here are some examples of `error(n)` in action:

```
>>> error(1)

0.71828182845904509
```

```
>>> error(2)
```

```
0.21828182845904509
```

```
>>> error(3)
```

```
0.051615161792378128
```

```
>>> error(10)
```

```
2.7312660133560485e-08
```

The last error is on the order of 10^{-8} !