Introduction to Functional Programming

Julian Singkham, Joshua Goldshteyn, Devin Illy

In the modern programming world we have Object Orientated Programming (OOP) and Functional Programming (FP). Both have been around for a long time but in this introduction, we will be contrasting FP against OOP.

Here's an example in python of a Pure (stateless) Function:

In [1]:
```
def add(x, y):
    return x + y
```

It's a pure function since it only deals with its inputs. It does add, but since it doesn't return the results, it's useless.

Pure Functions will always produce the same output given the same inputs.

In [2]:
```
print(add(1,3))
print(add(1,3))
print(add(1,3))
```

```
4
4
4
```

Pure functions have no side effects and are expexted to return the same output everytime.

In Imperative Programming Languages such as Javascript, Java, and C#, Side Effects are everywhere.

This makes debugging very difficult because a variable can be changed anywhere in your program. So when you have a bug because a variable is changed to the wrong value at the wrong time, where do you look? Everywhere? That's not good.

Immutability

In Imperative Programming, it is common to have mutate a variable and return the mutated variable in a function. This means it takes the current value of `x` adds `1` to it and put that result back into `x`.

Well, in functional programming, `x = x + 1` is illegal. There are no instance/global variables in FP

Stored values are still called variables because of history but they are constants, i.e. once `x` takes on a value, it's that value for life.

Not to worry though, `x` is can still be a local variable but while it's alive, it can never change.

Here's an example of constant variables in Python

In [3]:
```
def add_one_to_sum(y, z):
    x = 1
    return x + y + z

print(add_one_to_sum(10,5))
```

```
16
```

FP replaces Loops with Recursion

Here's an example of a python for loop. Note - see how each iteration has side effects on the `total` variable

In [4]:
```
total = 0
for num in range(10):
    total += num

print(total)
```

```
45
```

Here is an example of using recursion to solve the same problem. We will need to import the `reduce` function from pythons functools, and will also need to use python's `lambda` function.

Under the hood the reduct function generates a new list then reduces it to a single value and sets total to the value of the function.

It is debatable whether this creates more readable/debuggable code, but it is a "Pure Functional" approach.

In [5]:
```
from functools import reduce

total = reduce(lambda x, y: x + y,range(10))
print(total)
```

```
45
```