Functions in Python

A familiar function

```
import numpy as np

def manhattanDistance(coord1, coord2):
        dist = 0
        errorstring = "Coordinate dimension mismatch."
        if len(coord1)==len(coord2):
                for i in range(len(coord1)):
                        dist+=np.abs(coord1[i]-coord2[i])
                return dist
        else:
                raise runtimeError(errorstring)
```

A familiar function

```
def manhattanDistance(coord1, coord2):
    ...
    return dist
```

Let's focus on one part of that code snippet

- Defines a function called `manhattanDistance`, as well as its arguments
- Functions are extremely powerful!

Why Use Functions?

In Python (and most languages), we can just write code as a sequence of commands, and everything works just fine.

```
import numpy as np

myCoord1 = [10,20,30]
myCoord2 = [1,2,3]
dist = 0

dist+=np.abs(myCoord1[0]-myCoord2[0])
dist+=np.abs(myCoord1[1]-myCoord2[1])
dist+=np.abs(myCoord1[2]-myCoord2[2])
```

Why Use Functions?

What might go wrong with the code on the previous slide?

- What if I want to use a new set of coordinates with 4 dimensions?
- What if I don't notice that my coordinates do not have the same number of dimensions?
- What if I want to run that code as part of another program in a different file?
- What if I want to use Euclidean Distance in the future?

Nature of Functions

- Allow for reuse of code
    - Can import functions into other scripts!
- Help us to organize our code
- Are limited in **scope** (more on that soon!)
- Allow us to quickly make broad changes

Starting to Write Functions

```
def myFunction(arguments_go_here):
```

First, we need to use the `def` statement to declare our function.

Later, after we have completed the code that runs inside of the function, we write our return statement:

```
        return objects_to_be_returned
```

Arguments, Scope

Arguments are **variables** that we provide to our function. These variables receive special names (assigned by us), so that our function will work, no matter what specific variables we provide to them.

Arguments to a function are **local** in scope, meaning that these special function names **do not exist outside of our function**

Exercise

Write a function that returns the product of two numbers (note: the product of x and y is $x \times y$). Name the function `product`.

- What is the result of `product(2,5)`?
- What is the result of `product(2.71828,5)`?
- What is the result of `product("Howdy!",3)`?

Observations

When we use the function `product`, we are able to use a string as one argument. Why?

- Python is able to determine that the multiplier function `string * y` means that we want to repeat a string y times.

Exercise, Part 2!

Write a function that ONLY utilizes your `product` function to calculate the area of a circle with radius `r` (note: area is calculated as πr^2). Call that function `areaCircle`

- What is the result of `areaCircle(2)`?
- What is the result of `areaCircle(2.71828)`?
- What is the result of `areaCircle("Howdy!")`?

Observations

The function `areaCircle` can be created by utilizing our `product` function:

```
def areaCircle(r):
    r2 = product(r,r)
    return product(r2, 3.1415)
```

or, even more succinctly,

```
def areaCircle(r):
    return product(product(r,r), 3.1415)
```

Observations

- We can use functions inside of functions
- Use small functions to build part of a whole
- We can even use functions **recursively**

Recursive Functions

Recursive Functions

Try writing a function to calculate Factorials.

$$0! = 1$$

$$1! = 1$$

$$2! = 2 \times 1$$

$$3! = 3 \times 2 \times 1$$

$$n! = n \times n - 1 \times n - 2 \times ... \times 1$$

How can we write a function to determine an arbitrary Factorial?

Recursive Functions

```
def factorial(n):
    if n==0:
        return 1
    elif n==1:
        return 1
    else:
        return n*factorial(n-1)
```

This function is **recursive** because it calls *itself* in order to complete its own execution.

Functions are the backbone of programming

We will use functions EVERY DAY as programmers, and they will save us a LOT of time as we move through more advanced topics.

Lab Time!