



**A Comparison of Python & C**  
**In the context of Names, Bindings and Scope**



**COEN171**  
**Explore A Programming Language Project**  
**28 November 2016**  
Steven Booth Nick Goodpaster

Our project will compare the programming languages Python and C in three categories: Names, Bindings and Scope. We will give an explanation of each category's relevance to programming languages, and then dive deeper into the relationship between C and Python with respect to these criteria, assessing each language in the process using the language criteria.

## Names

Names are any words used to help describe a program's functionality. Names mainly consist of identifiers and keywords.

### Identifiers/Keywords:

In programming, identifiers refer to names given to any variables, objects, functions, structures, etc. Both C and Python have certain restrictions for their identifiers.

In C, identifiers can contain any digits, characters, or symbols, but should start with only a character or underscore, though the latter is discouraged. Identifiers in C can be of any length, however, only 31 characters are observed by the compiler and are case sensitive. Keywords in C cannot be used for identifiers; the keywords in C are as follows:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
continue	for	signed	void
do	if	static	while
default	goto	sizeof	volatile
const	float	short	unsigned

In Python, Identifiers cannot contain symbols and cannot start with a digit. They can also be of any length. Identifiers are also case sensitive in Python. Keywords in Python, like C, can also not be used for identifiers; the keywords in Python are as follows:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

## Names in Python:

In Python, it is better to think of variables as names. Unlike C, names in Python do not have an associated type. So, we cannot even declare or initialize a variable without first defining it. In python, a name is simply a way of referencing an object. In this sense, the names, or variables, in Python are just references to objects and you can reuse the same name for any object type without having to worry about its associated type.

In Python:

```
z = 5
z = "hello"
print(z)
```

In C:

```
int i = 10;
i = "string"; //Error
i = 5; //OK
i = (int) 5.3; //OK
```

## **Names Analysis:**

Both readability and writeability for Python and C's naming conventions are very similar.

**Readability:** Both languages are case-sensitive, which helps to improve their readability. Each language has over 30 keywords to help improve its readability.

**Writeability:** Both require the first character of an identifier to be a specific character, which restricts naming and thus negatively impacts writeability.

**Cost:** Since you don't need to worry about binding of a name to a type in Python. The programmer does not necessarily need to know much about types to assign a name properly in Python, where in C you need to explicitly declare the variable type in order to assign it to a value.

# Binding

As far as name bindings goes, C makes use of static, stack dynamic, and explicit heap dynamic binding, while Python, as a dynamic language, strictly uses implicit heap dynamic binding.

In C, variables are marked static with the static keyword and memory is allocated for them at compile time. Static variables are those that have a lifetime of the entire duration of the program. C uses stack dynamic binding for local variables in functions. These variables are allocated on the stack when the function is called and deallocated when the function returns. This allows for recursion and are easier to manage than heap dynamic ones.

## Name Binding in Python

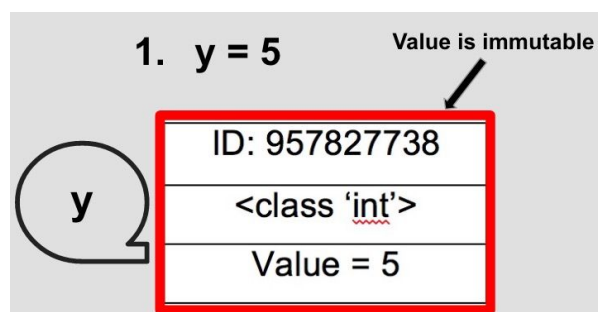
Dynamic languages, like Python, allocate space for values automatically as needed, a process known as Implicit Heap Dynamic Binding.

Below is a description of how name binding occurs in Python:

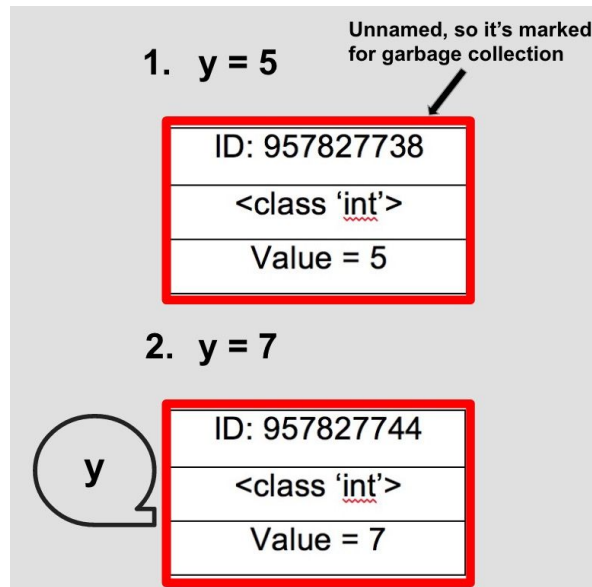
1. A variable is defined.

```
y = 5
```

2. The system will create an instance of this variable that will include: the ID, the object type, and the value, which is immutable for primitive types, meaning it cannot be modified.



3. We can change name reference by redefining the name to a different value. This will only change the name reference and will not change the previous value of the variable. Instead, the previous definition is still in memory and is marked for garbage collection.



## Static Name Binding in Python

Unlike C, Python doesn't make use of a keyword to create static member variables, so we have to get clever. To mimic a static binding in Python, we will, like we did in name binding, bind the value to an object rather than to a name by creating our own object. The object itself will then be bound to the name so that no matter how many instances of the object we create, the object definition itself will still be bound to the original object values.

1. First we will create an object and define a variable `x` inside of it:

```
class cleverClass:  
    x = 10
```

2. Next we will create an instance of the `cleverClass` object and modify the member variable of the instance to equal 20.

```
instance = cleverClass()  
instance.x = 20
```

3. Now, if we were to print the instance value of `x` and the object value of `x`, we will get the results 20 and 10 respectively. This is because the value 20 is bound to the object referred to by the name 'instance' while the value 10 is bound to the object `cleverClass()`, which was not modified.

```
print(instance.x)  
print(cleverClass.x)
```

**OUTPUT:**

20  
10

So, in order to mock a static member variable in Python, we define it within the object definition. This way, no matter when we refer to cleverClass.x in the program, the same value of x will be referenced.

### **Bindings Conclusion**

**Readability:** C is more readable because it is clearer to see exactly how much memory is being allocated, because either the type (static, stack dynamic) or actual memory space needed (explicit heap dynamic) are explicitly declared.

**Writeability:** Python is easier to write because neither the memory space nor even the type of the variable need to be explicitly declared.

**Reliability:** Python is more reliable because there is less chance of a memory leak because it is a dynamic language and has its own built in memory management system with a garbage collector for objects left unnamed at the end of a program.

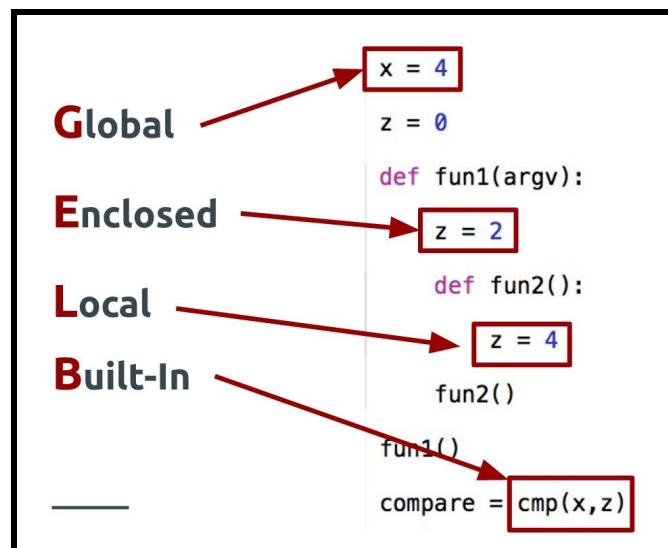
# Scope

In programming, scope refers to the part of code where a name binding is valid, or where a name can be used to refer to its object. Both Python and C are statically scoped languages.

## Static Scoping

In static scoping (also referred to as lexical scoping), names are bound to their local environment. A name's scope is its current block of code, and is determined at compile time. If the block of code in which the name resides is a function, the name's scope is then extended to any contained functions, unless any of those contained functions redefines the name.

In Python, to resolve the scope of a name, the LEGB rule is followed. LEGB stands for Local, Enclosed, Global, and Built-in, and refers to the four types of scopes for names in Python. A name has local scope if it is located within the current function or class method. A name has enclosed scope if it is located inside of an enclosing function. If a variable `x` is in a function that also includes a nested function definition, the variable `x` has enclosed scope within the nested function, and can only be referred to using the `nonlocal` keyword. A name has global scope if it is defined outside of all functions, and can only be referred to using the `global` keyword. Finally, a name has built-in scope if it is a special name that Python has reserved for itself.





## Scoping Keywords in Python

In C, one can reference a global or nonlocal variable within a nested function so long as it is not redeclared in the local scope. However, in Python, the keywords `global` and `nonlocal` are required for referencing a name outside of the local scope. This is necessary for Python because there are no name declarations and everything is just a reference, so if you wanted to redefine a variable, it would be difficult to tell whether you were intending to overwrite the nonlocal name or make a new local name with reference to another object.

Below is the way variables are typically, statically scoped in Python without any keywords:

```
z = 0

def fun1():
    z = 2
    def fun2():
        z = 4
        print("fun2: ", z)
    fun2()
    print("fun1: ", z)
fun1()
print("global: ", z)
```

**OUTPUT:**

```
>> fun2: 4
>> fun1: 2
>> global: 0
```

Below is an example using the `global` keyword:

**'global' keyword**

```
z = 0
def fun1():
    z = 2
    def fun2():
        global z
        z = 4
        print("fun2: ", z)
    fun2()
    print("fun1: ", z)
fun1()
print("global: ", z)
```

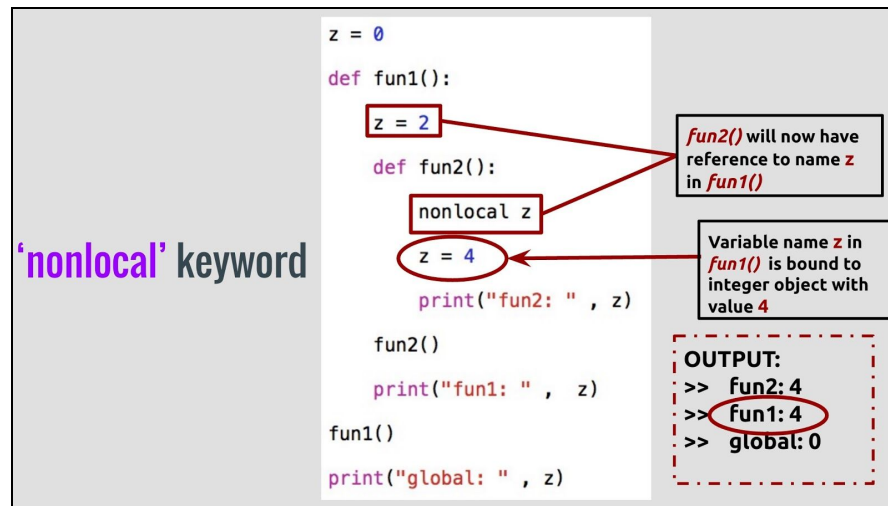
**fun2() will now have reference to global name z**

**global variable name z is bound to integer object with value 4**

**OUTPUT:**

```
>> fun2: 4
>> fun1: 2
>> global: 4
```

Notice how the 'global' keyword changed the reference of name z inside of fun1() to the global name z. A similar method is used to determine the scope resolution for the local name z, but in this case the name z in fun1() has reference to its enclosing function's (fun2()'s) name z. Below is an example using the nonlocal keyword:



### Scoping Conclusion:

**Readability:** Python's global and nonlocal keywords remove ambiguity when determining which variable is being used in the subprogram. Without these keywords, scope resolution within subprograms in C becomes less apparent, because you have to trace back through the enclosing functions to find the variable being referenced.

**Writeability:** Python requires the keyword global to use a global variable in a function, while C does not. C can access global variables so long as the variable isn't redefined in the subprogram.

**Reliability:** Using static scoping improves reliability because there is less ambiguity as to which variable is being referenced, making the code easier to modify.

## Our Algorithm: k-Nearest Neighbors

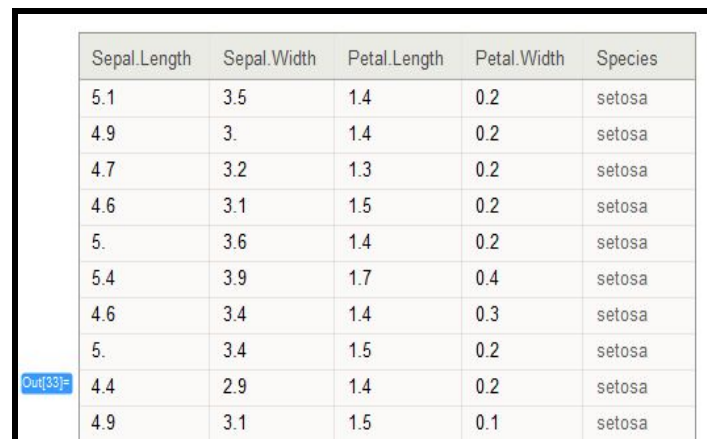
**Background:** k-Nearest Neighbors is a machine learning algorithm. It is instance-based meaning it relies on a data set to come to its prediction. kNN is known as a “lazy” learning algorithm because it doesn’t create a prediction model until one is needed. It is also a competitive algorithm because it relies on competition between data values to come to a prediction.

### How it works:

1. Load the data set.
2. Calculate the distance between the test instance and every instance of training data.
3. Find the test instance’s “nearest neighbors”.
4. Use these neighbors to vote on the most common neighbor identifier.
5. Make an identifier prediction based on the identifier votes.

### Our Data Set

We used a commonly used data set for machine learning testing called the iris data set. The set is split into 4 attributes: petal width, petal length, sepal width, sepal length. It is best to think of these attributes as variable dimensions on a graph. We also had 3 identifiers, which were the types of flowers: virginica, versicolor, and setosa. We had 50 data instances for each of the identifiers for a total of 150 data values. The csv file that we used followed the following format:



The image shows a screenshot of a Jupyter Notebook cell. On the left, there is a blue label 'Out[33]:'. To its right is a table with 5 columns: 'Sepal.Length', 'Sepal.Width', 'Petal.Length', 'Petal.Width', and 'Species'. The table contains 10 rows of data, all of which are 'setosa' species. The values for the first four columns range from 4.4 to 5.4 for Sepal.Length, 2.9 to 3.6 for Sepal.Width, 1.3 to 1.7 for Petal.Length, and 0.1 to 0.4 for Petal.Width.

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa

First our programs partition the data into training and testing data sets. The python

program uses the csv reader library for reading and parsing the csv file. It splits the training and testing data by a set partition split which divides the data instances into the training and testing data sets, making use of the random() function from the random library which returns a float value between 0.0 and 1.0. The python program makes this code look much cleaner because we can access regions of the csv file using subscripting, while the C program had to read the files as text files and stepping through each line. For the C implementation, we load in the two data sets as arguments in the command line, while in the Python implementation we actually split the data each time the program is run. The two csv partition functions are below:

### Python loadDataSet() Implementation

```
"""
Function to load the data set from the irisdata file and split the data into
training and testing data records.
"""

def loadDataSet(filename, split, trainingDataSet = [], testingDataSet = []):
    with open(filename, 'r') as csvfile:
        #Reads lines of the csv file using the built in csv reader
        lines = csv.reader(csvfile)
        #makes a list of the lines in the csv file
        data_csv = list(lines)
        for x in range(len(data_csv)):
            #for each of the attributes in the csv file, the values are cast
            #to floats
            for y in range(5):
                data_csv[x][y] = float(data_csv[x][y])
            #creates a random number between 0 and 1.0 and appends value
            #to training or testing data set with a weighted split
            if random.random() < split:
                trainingDataSet.append(data_csv[x])
            else:
                testingDataSet.append(data_csv[x])
```

## C loadDataSet() Implementation

```
/*
 * Function to parse the csv files for the training and testing
 * data sets into the testingData and trainingData arrays.
 */
void loadDataSet(char *filename, float **data, char **ID){
    FILE* file = fopen(filename, "r");
    char line[40];
    char *instance = (char *) malloc(MAX * sizeof(char));
    int i, j, rowcount = 0, instancecount = 0;
    float instanceData;
    while(fgets(line, sizeof(line), file)){
        i = 0;
        instancecount = 0;
        while(i < sizeof(line)){
            j = 0;
            if(!isalpha(line[i])){
                do{
                    instance[j] = line[i];
                    j++;
                    i++;
                }while(line[i] != ',');
                instanceData = atof(instance);
                data[rowcount][instancecount] = instanceData;
                instance[0] = '\0';
                instancecount++;
            } else if(isalpha(line[i])){
                do{
                    instance[j] = line[i];
                    j++;
                    i++;
                }while(isalpha(line[i]));
                instance[j++] = '\0';
                strcpy(ID[rowcount], instance);
                break;
            }
            i++;
        }
        rowcount ++;
    }
    free(instance);
    fclose(file);
}
```

Both the C and the Python programs make use of a distance formula function to compute the distance between a testing data instance and a training data instance, the code for these are pretty similar and shown below:

### Python distanceFormula() Implementation

```
"""
Function to compute the distance between a testingInstance and trainingData.

Returns the distance between the testing attribute and the training attribute
"""

def distanceFormula(testingInstance, trainingData, length):
    distance = 0
    for x in range(length):
        distance += pow((float(testingInstance[x])) - (float(trainingData[x])), 2)
    return math.sqrt(distance)
```

### C distanceFormula() Implementation

```
/*
 * Euclidian distance formula, calculates distance between
 * flower attributes.
 *
 * Returns the distance between the test instance and the training
 * instance.
 */
float distanceFormula(float **train, float *testInstance, int instance){
    float distance = 0;
    for(int i = 0; i < ATTRIBUTES; i++){
        distance += pow(testInstance[i] - train[instance][i], 2);
    }
    return sqrt(distance);
}
```



Next, each program has a function to compute the distances between a single test instance and all of the training data instances. These distances are recorded and stored in an array of distances. This array of distances is then sorted by distance. In C, we created a structure to hold a 'neighbor' type and it contains the distance and the identifier of the neighbor (type of flower). Then, in the C implementation, we sorted this array of neighbors by their distance member variable. In both implementations the next step was to make a smaller subset of the distances array that contained just the k-nearest neighbors, where k is the number of neighbors selected for the given program execution. This functionality is shown below:

### Python getNN() Implementation

```

"""
Function to compute the k-Nearest Neighbors. Takes in a test instance and
compares the distance between its attributes and the attributes of all the
instance values in the training data. Takes the closest k neighbors and returns
an array containing the k-Nearest Neighbors neighbors.

Returns the array of nearest neighbors.
"""
def getNN(trainingDataSet, testInstance, k):
    print('\n')
    #creates an array to hold distances for neighbors
    distances = []
    #sets the length to be the number of attributes, so the number of columns
    #in the test instance minus the identifier for the type of flower
    length = len(testInstance) - 1
    for x in range(len(trainingDataSet)):
        #computes distance between training instance and testing instance
        distance = distanceFormula(testInstance, trainingDataSet[x], length)
        #appends to distances array
        distances.append((trainingDataSet[x], distance))
    #sorts the distances
    distances.sort(key=operator.itemgetter(1))
    neighbors= []
    #appends the k-Nearest neighbors to the neighbors array
    for x in range(k):
        neighbors.append(distances[x][0])
        print('Neighbor #' + repr(x + 1) + ': ' + repr(neighbors[x][-1]))
    return neighbors

```

## C getDistances(), sortedDistances(), & getNN() Implementation

```

/*
 * Calculates distance between testing instance and every
 * training value.
 *
 * Returns an unsorted array of distanced betearn the training instance
 * and all of the training instances.
 */
float* getDistances(float **train, float *testInstance){
    //printf("%f, %f\n", testInstance[0], testInstance[1]);
    float *distances = (float *) malloc(sizeof(int) * TRAIN);
    for(int i = 0 ; i < TRAIN; i ++){
        float distance = distanceFormula(train, testInstance, i);
        distances[i] = distance;
    }
    return distances;
}

/*
 * Pairs the distance and the ID to create a neighbor type for each
 * of the distances. Sorts this array of neighbors to be used for
 * finding the k-Nearest Neighbors.
 *
 * Returns a pointer to an array containing the neighbors of the test
 * instance sorted by distance.
 */
neighbor* sortDistances(float *distances, char **trainingID){
    neighbor* neighbors = (neighbor *) malloc(TRAIN * sizeof(neighbor));
    neighbor* temp = (neighbor *) malloc(sizeof(neighbor));

    //Loading distances and trainingID's into array of neighbors
    for(int i = 0; i < TRAIN; i ++){
        neighbors[i].distance = distances[i];
        neighbors[i].ID = malloc(10 * sizeof(char));
        strcpy(neighbors[i].ID, trainingID[i]);
    }

    //Bubble sort array of neighbors by distance
    for(int i = 0; i < (TRAIN - 1); i++){
        for(int j = 0; j < (TRAIN - i - 1); j++){
            if(neighbors[j].distance > neighbors[j + 1].distance){
                *temp = neighbors[j];
                neighbors[j] = neighbors[j + 1];
                neighbors[j + 1] = *temp;
            }
        }
    }
    return neighbors;
}

/*
 * Uses the sorted neighbor array to create a subarray containing
 * only the k-Nearest Neighbors of the test instance.
 */
neighbor* getNN(neighbor *sortedNeighbors, int k){
    printf("\n\n");
    neighbor* NN = (neighbor *) malloc (k * sizeof(neighbor));
    for(int i = 0; i < k; i ++){
        NN[i] = sortedNeighbors[i];
        printf("Neighbor #%d = '%s'\n", (i+1), NN[i].ID);
    }
    return NN;
}

```



Finally, once the nearest neighbors are found, both programs then make use of a get vote function to count the amount of neighbors that are of certain identifiers. The neighbors cast a “vote” with their identifier. These vote counts are then passed into a getMax function which returns a string representation of the identifier that has been voted to be the prediction. The code for this voting functionality is shown below:

### Python getVote() & getMax() Implementation

```
"""
Function to get the votes from the nearest neighbors to decide the predicted
type of the testing instance. Will use the getMax function to generate a
prediction.

Returns string representation of prediction to be added to the predictions
array in the main.
"""
def getVotes(neighbors):
    setosaVotes = 0
    virginicaVotes = 0
    versicolorVotes = 0
    for x in range(len(neighbors)):
        if neighbors[x][5] == 'setosa':
            setosaVotes += 1
        if neighbors[x][5] == 'virginica':
            virginicaVotes += 1
        if neighbors[x][5] == 'versicolor':
            versicolorVotes += 1
    prediction = getMax(setosaVotes, virginicaVotes, versicolorVotes)
    return prediction

"""
Function to get the maximum value of the three votes and returns the string
representation of the most voted flower type from the neighbors array.

Returns the prediction in the form of a string.
"""
def getMax(setosaVotes, virginicaVotes, versicolorVotes):
    if((setosaVotes > virginicaVotes) & (setosaVotes > versicolorVotes)):
        prediction = 'setosa'
    if((virginicaVotes > setosaVotes) & (virginicaVotes > versicolorVotes)):
        prediction = 'virginica'
    if((versicolorVotes > setosaVotes) & (versicolorVotes > virginicaVotes)):
        prediction = 'versicolor'
    return prediction
```

### C getVote() & getMax() Implementation

```

/*
 * Each neighbor will vote for it's ID type, these votes will
 * then be passed into the getMax function to find the majority
 * ID of the neighbors.
 *
 * Returns the prediction for the test instance, which is determined
 * by the identifier with the most number of votes.
 */
char* getVote(neighbor *NN, int k){
    int setosa = 0, virginica = 0, versicolor = 0;
    for(int i = 0; i < k; i ++){
        if(strcmp("setosa", NN[i].ID) == 0){
            setosa++;
        } else if(strcmp("virginica", NN[i].ID) == 0){
            virginica++;
        } else if(strcmp("versicolor", NN[i].ID) == 0){
            versicolor++;
        }
    }
    return getMax(setosa, virginica, versicolor);
}

/*
 * Returns the prediction, which is the ID with the most
 * votes.
 */
char* getMax(int setosa, int virginica, int versicolor){
    char* prediction = malloc(MAX * sizeof(char));
    if(setosa > virginica && setosa > versicolor){
        strcpy(prediction, "setosa");
    } else if(virginica > setosa && virginica > versicolor){
        strcpy(prediction, "virginica");
    } else if(versicolor > setosa && versicolor > virginica){
        strcpy(prediction, "versicolor");
    } else{
        strcpy(prediction, "still cannot be determined");
    }
    return prediction;
}

```

Each program also makes use of a get accuracy function that computes the accuracy of the system by comparing the identifiers stored in the predictions array and the testing data identifiers array (in C) or the testing data array (Python). The function returns the accuracy of the programs last execution as a float.

### Python getAccuracy() Implementation

```
"""
Function to compute the accuracy of the algorithm by comparing the correct
flower types from the testing data and the predictions computed in our
algorithm.

Returns the value of the accuracy as a float to be printed int the main
function.
"""
def getAccuracy(testSet, predictions):
    correct = 0
    for x in range(len(testSet)):
        if testSet[x][5] == predictions[x]:
            correct += 1
    return (correct/float(len(testSet))) * 100
```

### C getAccuracy() Implementation

```
/*
 * Function to calculate the accuracy of the alogorithm by evaluating
 * how many of the testingDataID's were predicted correctly.
 */

float getAccuracy(char **predictions, char **testingID){
    float correct = 0;
    for(int i = 0; i < TEST; i ++){
        if(strcmp(testingID[i], predictions[i]) == 0){
            correct++;
        }
    }
    return (100 * (correct/30));
}
```

The main for both of these programs is pretty similar (though the Python code looks a little cleaner because the memory allocation does not need to be explicitly declared). The main will load the training and testing data and run a loop that will for each test instance compute the distances, nearest neighbors, and prediction and load the prediction into the predictions array. The program will print the three nearest neighbors to the test instance as well as its predicted and actual types. Once this loop terminates, the function then calculates and prints the accuracy of the algorithm. The code for main functions are below:

### Python main() Implementation

```
"""
Main function definition.
"""
def main():
    #prepare data
    trainingDataSet = []
    testingDataSet = []
    split = 0.8
    loadDataSet('iris.csv', split, trainingDataSet, testingDataSet)
    #generate predictions
    predictions=[]
    k = 3
    for x in range(len(testingDataSet)):
        neighbors = getNN(trainingDataSet, testingDataSet[x], k)
        result = getVotes(neighbors)
        predictions.append(result)
        print('Predicted=' + repr(result) + ', Actual=' + repr(testingDataSet[x][-1]))
        print('-----')
    accuracy = getAccuracy(testingDataSet, predictions)
    print('\n\nAlgorithm Accuracy: ' + repr(accuracy) + '%')

#main function call
main()
```



## C main() implementation

```
//Main function
int main(int argc, char* argv[]){
    int k = 3;

    //Allocates memory for testing data set
    float **testingData = (float **) malloc(TEST * sizeof(float *));
    for(int i = 0; i < TEST; i++){
        testingData[i] = (float *)malloc(ATTRIBUTES * sizeof(float));
    }
    char **testingDataID = malloc(TEST * sizeof(char*));
    for(int i = 0; i < TEST; i++){
        testingDataID[i] = (char *) malloc(MAX * sizeof(char));
    }

    //Allocates memory for training data set
    float **trainingData = (float **) malloc(TRAIN * sizeof(float *));
    for(int i = 0; i < TRAIN; i++){
        trainingData[i] = (float *)malloc(ATTRIBUTES * sizeof(float));
    }
    char **trainingDataID = malloc(TRAIN * sizeof(char*));
    for(int i = 0; i < TRAIN; i++){
        trainingDataID[i] = (char *) malloc(MAX * sizeof(char));
    }

    //Allocates memory to hold predictions
    char **predictions = malloc(TEST * sizeof(char *));
    for(int i = 0; i < TEST; i++){
        predictions[i] = (char *) malloc(MAX * sizeof(char));
    }

    //Loads the data sets from input files into data arrays
    loadDataSet(argv[1], trainingData, trainingDataID);
    loadDataSet(argv[2], testingData, testingDataID);

    //calculates a prediction for each data sample in testingData
    float *distances;
    neighbor *neighbors;
    neighbor *NN;

    for(int j = 0; j < TEST; j++){
        distances = getDistances(trainingData, testingData[j]);
        neighbors = sortDistances(distances, trainingDataID);
        NN = getNN(neighbors, k);
        predictions[j] = getVote(NN, k);
        printf("Prediction: '%s', Actual: '%s'\n", predictions[j], testingDataID[j]);
        printf("-----");
        free(neighbors);
        free(distances);
        free(NN);
    }

    printf("\n\nAlgorithm Accuracy: %.2f%%\n", getAccuracy(predictions, testingDataID));
    free(predictions);
    free(testingData);
    free(testingDataID);
    free(trainingData);
    free(trainingDataID);
    return 0;
}
```

Here is a piece of both of the program outputs. As you can see, the outputs are pretty much the same for both implementation.

## Python

```
Neighbor #1: 'virginica'
Neighbor #2: 'virginica'
Neighbor #3: 'virginica'
Predicted='virginica', Actual='virginica'
-----

Neighbor #1: 'virginica'
Neighbor #2: 'virginica'
Neighbor #3: 'virginica'
Predicted='virginica', Actual='virginica'
-----

Algorithm Accuracy: 100.0%
```

## C

```
Neighbor #1 = 'virginica'
Neighbor #2 = 'virginica'
Neighbor #3 = 'virginica'
Prediction: 'virginica', Actual: 'virginica'
-----

Neighbor #1 = 'virginica'
Neighbor #2 = 'virginica'
Neighbor #3 = 'virginica'
Prediction: 'virginica', Actual: 'virginica'
-----

Algorithm Accuracy: 96.67%
```

## Conclusion:

As far as the programming languages' application in the real world, we came to the conclusion that the most apparent distinction between Python and C comes to memory management. As a dynamic language, Python has built-in memory management (implicit heap dynamic binding) which makes it a very efficient language for constructing applications quickly – because the programmer doesn't have to worry about how the memory is handled. This makes it easier to throw together a program to test the logic or the functionality before worrying too much about the internal memory management and program efficiency. C, on the other hand, allows for explicit management of memory, which is essential to making high end programs run efficiently. This is why C is preferred for operating systems and programs that require precise memory management. In the case of

our algorithm, we noticed that the code in the Python algorithm was easier to write, required less attention to memory or type when defining new variables. The C code was less intuitive as far as its memory allocation because we had to not only initialize and allocate space for the variables that we wanted to pass by reference, but we also had to make sure that we freed this memory when the program execution was complete or the instance variables were done being executed on.