Summary:
KangaScript is a new scripting language for modern development. It was designed by Joey Muller (‘15) with the intent of aiding his senior project. Through time, it has evolved into a much larger project, from a small scripting language into a full-scale, modern programming language. KangaScript is currently in a radioactive state, although prospects for a complete, stable version are in the near future.

What follows is a guide documenting all of the features of KangaScript, including syntax and customs.

Table of Contents:

Intro:
With a KangaScript program, it is possible to interact with users, implement algorithms, and perform computations. This is done by writing segments of text-based code and evaluating it with a standard-conforming KangaScript interpreter.

Environments:
KangaScript programs are evaluated inside of environments. An environment is a representation of the state of the program: a collection of all the values, and their names relating to what the program needs to do. Initially, all KangaScript programs begin in a copy of the global environment. The global environment is mostly blank of anything, except some built-in values, (like `range`, or `print`). The built-ins takes care of commonly used code and structures, and allow the user access to some features that wouldn't be possible without. Some pieces of code (bodies of complex statements, and functions) run in new environments, which restrict their values to themselves, while allowing access to the values of the parent environment. This division of variable access is a principle known as scope.

Types:
Of the values used in a KangaScript program, they can be organized into types. Types allow the programmer to generalize code, and deal with specific data. The types are: null, Boolean, Number, String, Object, Array, and Function

- The default values for variables is blank. blank is not a *type* of value, but is an acceptable *value*. A blank variable indicates that it is defined, but empty and not quite ready to be interacted with. blank is precisely analogous to undefined in JavaScript. It is **not** the same as Python's None, or Java's null, those are similar to the next type.
    - [should be in different section] Functions without return statements return blank by default
- null, although similar, is not precisely the same as blank. null is a data type, unlike blank which is just a placeholder. A null value type can only represent 1 distinct value, also written as null. This fact is unique to only null, as all other data types represent multiple, and in some cases, an infinite range of values. null is used mostly as a default or empty-value. [Side note: Not empty, but a value that's nothing, just like zero is a number, but still represents nothing.] null is precisely analogous to null in JavaScript. It is also analogous None in Python, and null in Java. If you are confused about the existence, or rational behind implementing blank and null distinctly, do not fret! It should not be too troubling in your encounters, and is really just a technical detail. Remember: null is a value, blank is a not-a-value.
- Boolean values resemble the powerful boolean mathematical and computer science notion. Booleans can represent only two distinct values, True or False.
- Numbers store representations of real (as opposed to complex) numerical data. There are an infinite count of possible number values (or within computer memory limits anyway). There is no distinction between the integers and all real number as the case is in Java, in KangaScript they are all classified as Number. Number literals are written

as a sequence of decimal digits 0-9. A single decimal point may be placed in the middle to symbolize non-integers.

- Strings represent textual data. Like Numbers, there are an infinite count of possible string values in KangaScript (bounded only by physical limits). A string is a sequence of characters. String literals are bounded by a single quotation mark (“) or (‘). Either can be used in a program, but the same mark must bound a string. Strings may contain alphanumeric characters, special symbols, and special characters. The backslash (\) is the escape character, and it used to help signify characters that wouldn’t be typeable otherwise. For example, to write a double quote (“) inside of a string literal bounded by double quotes, it must be prefaced with the backslash. Return, and tab are also stringable because of the backslash. This shouldn’t be so difficult, so in short, all Javascript strings are also valid in KangaScript.
    - For example: “He said: /”That’s my language/” ” Note how the single quote didn’t have to be escaped, since the string literal was bounded with just double-quotes
- Arrays are a complex data structure, capable of holding a 1D collections of values. Perhaps KangaScript Arrays would be better known as lists, but for now, it is as I say it is. KS Arrays are mutable and may change size. They can hold a list of values of any type; It is not necessary to store uniform types. Array literals are written as a square-bracket set, bounding comma-separated values. The values may be either literals or variables. To refer to the value of an array at an index, it is notated by the array, followed by a square bracket with the index inside.
    - Literal examples: [5, ‘hi’, ‘friends’] or [true, false, true, ‘true’, ‘64’, 64] or [] # empty array
    - Reference examples: list[index] or list[0] or [1, 2, 3][0]
- Objects are also a complex data structure, comparable to a 2D array, associative array, dictionary, or hash-table. Objects are written as a curly-bracket set, bounding comma separated key-value pairs. A key value-pair is a string-literal with a colon, and then an expression. The string-literal is the object’s attribute or property, and the expression assigns the value to the attribute. An object literal declaration is the same as the JSON object. To refer to an attribute of an object, one of two notations can be used. The first is array notation (because it’s an associative array afterall), ex. obj[‘index’] or obj[5]. The second is dot-notation, the object with a dot and then the attribute name. Ex: obj.apple. However, dot notation can only be used to refer to object attributes, when the attributes are strings and valid identifiers.
- A Function is a type of value representing pieces of code. It is very much the same as the Function element <link here> discussed in later sections. KangaScript is multi-paradigmal, functional being one of them. Functions are first-class objects and can be manipulated and referenced in ways similar to other types of values. You will learn more about it below, but when a function is defined a new Function value is declared in the environment with the same name as the function, and value representing the function’s internals. Unnamed functions are also creatable, written exactly the same as named functions, but without the identifier-name. These

anonymous functions are useful for dealing with functions as more than callable-things, and somewhat higher-level concept.

- Equivalence notions between datatypes

Code:

What makes up a KS program?:
A KangaScript program is composed of a series of statements and function definitions. A statement is a section of code that either does something to alter the program state, or changes the interpretation of other code. Because of this, statements are grouped into two ways: simple and complex, their purposes aligning with the prior goals respectively. A function is a group of statements, put together with a name, so to make it easy to generalize and reuse certain code. The next section details the jist of statements and functions.

Comments:

A comment is a piece of text in a program, but is not code. It is ignored by the interpreter and it left by the creator for a variety of reasons. To help the coder code, leave attribution or reminders, or to explain confusing/ambiguous code are a few of the reasons for comments, and are essential to writing “good” code. Comments begin with the octothorpe symbol (#). KangaScript has what are known as 1-line ‘end of line’ comments, meaning the text of the comment is composed from the start, until the end of line. After that, (unless another comment comes), the interpreter continues evaluating code. Comments can be on the same lines as code, but only at the end, there is no way to stop them. Below are a few examples of comments. Throughout this documentation, comments will commonly be used in code examples, so it is important to recognize that while reading they are not interfering with code evaluation.

- # Hey, I’m a comment!
- {some code} # comments 4 da win
- print var # display to user the variable

Simple Statements:

Simple statements are typically, short and 1-line commands. There are 3 kinds: control_flow, import, and expression.

- Control flow statements alter the logic of a program, and work with compound statements. They are: continue, break, pass, and return. Because they co-operate with compound statements (except pass) they will be explained in exact detail where they match. pass is an empty statement. It literally does nothing to the program except exists. Why? you might ask: pass can be useful as a placeholder, reminding you to implement stuff inside of a compound statement later. It could also be used for explicitly writing that this piece of code is empty, or style of looks. It is written one-line by the pass keyword.
- An import statement informs the interpreter about code existing outside of the current program it is evaluating. The outside code is known as modules. Using an import statement and dividing code into modules is a specific design technique, and often makes a coding project more manageable. Grouping like-code is a practical way to organize and later find code.
    - It is easy to indicate a program as a module, the file name will become the module name, and the module it must be in the same directory as the program that will import it. If you wish, the imported file may also exist in a sub-folder of the directory.
    - The import statement has the following syntax:
        - import moduleName # a single module
        - import moduleA, moduleB # multiple modules
        - import dir1.dir2.moduleC # moduleC inside dir1 inside dir2
        - import dir3.moduleD, moduleE # moduleD in ./dir3 and moduleE in ./
    - When an import statement is evaluated, the indicated module will be run, and can later be referenced (and values defined while executing) through a same name value in the global environment, ex:
        - moduleA.ks
        - # some value foo
        -
        - moduleB.ks
        - import moduleA
        - # use value moduleA.foo
    -
- Lastly, and perhaps most important, are the expression statements. Expressions are the values and computations of programs, and include the results of function calls. Because there are ***MANY*** different types of expression statements, I'll keep this section short and detail them later. For now, it will suffice to know what expressions are, to continuing to complex statements.

Complex Statements:

Complex statements essentially change the interpretation of other code. They are typically multi-line groups, with a header and body of other program elements. The code of the header of the compound statement, and the context (environment) in which it is run determines the way the inner elements are evaluated. The control flow simple statements provide a way to communicate with the outer complex statements from within the body, and change how the body is evaluate. These mechanisms provide much of the logic of program, and can prove useful and optimal when used appropriately. There are 3 types of compound statements: two loops: for and while, and a switch: the if group

- The for loop allows you to iterate code over elements in an Array. It will, (for each element in the Array expression), repeat the body with some identifier set to the element (to generalize statements to the code). It is written (typically multi-line), as the for keyword, an identifier, the in keyword, and then an expression representing an Array. The body begins after a colon (:), and ends with the endfor keyword.
    - for i in range(5):
    - # body here
    - endfor
    -
    - for e in list
    - # body here
    - endfor
- The while loop allows you to iterate code as long as a condition is met. This makes it more useful than for-loops in cases where code is interacting with variables outside the loop, or will run an undetermined amount of times. But to note: for-loops and while-loops are essentially equivalent. All programs that a written with a for-loop can be rewritten with exactly the same meaning with while loops, and vice versa. After every run of the body, the condition will be re-evaluated, and if it proves true it will execute the body (again). It is written (typically multi-line), as the while keyword, and then a conditional expression to evaluate. The body begins after a colon (:), and ends with the endwhile keyword.
    - while i < 42:
    - # body here
    - endwhile
    -
    - while happy == “no”:
    - # body here
    - endwhile
    -
    - while true:
    - # body here
    - # btw, condition is ALWAYS true so infinite loop
    - endwhile

- The break and continue simple statements may be used within loops. Neither have meaning in an if-group, and raise errors.
    - The break statement stops a loop from executing the rest of it's body immediately. Code will continue interpreting following the end of the loop.
    - The continue statement stops a loop from executing where it is, but then it will restart from the beginning and 'continue' it's progress. This means different things in the loops. In a while-loop, the interpreter will exit the body, look at the top to re-evaluating the condition, and determine whether to run the body again. In a for-loop, the interpreter exits the body, and continues with the next element in the Array.

```
for num in range(10):
    # something smart to say
    # finish at this element
    break
endfor

while true:
    # would be infinite loop, but…
    # break goes outside
    break
endwhile

for num in range(10):
    # wanna stop current element
    continue
    # something cool here!
endfor

while i<20:
    # do something
    # maybe wanna skip?
    # continue
    # something else that won't always be executed
endwhile
```

- The if group allows you to make decisions on what pieces of code to evaluate based off a condition. It comes in four flavors: if, if-otherwise, if-elif, if-elif-otherwise. The basic if-statement lets you consider one condition and one code block. An elif also lets you consider one condition and one code block, but only if the previous if-statements tested false. Actually an if-group can have multiple elifs, each evaluated after the other until one is true, and its code block is evaluated, or until the end is reached. To be clear, with if and elifs, only one code block is evaluated (if any at all), the first one where the condition is true. In this sense, the elif-statement and if-statement are essentially equivalent, the only difference being that if comes first, and others have.

Note, however it is invalid syntax to substitute if for elif. You cannot have more than one if-keyword for an if-group (unless you're nesting the statements which is fine). Also, it is important to note that a bunch of if-groups is not necessarily equivalent as one if-group with many elifs. The thing is that a bunch of if-groups will evaluate all code-blocks where the condition is true, whereas a single if-group will only evaluate the code block of the first true condition. The otherwise clause can be used in an if-group as a default code block. In case none of the conditions in if/elif statements evaluate true, the body of the optional otherwise statement will be evaluated. All if-groups end with the endif keyword. The if:-elif:-otherwise: structure of the if-group mirrors the if:-elif:-else: of Python and if{}-else if{}-else{} of Java and JavaScript.

```
# short if
if i<20:
    # do something
endif

# if-elif
if i<20:
    # do A
elif i<10:
    # do B
endif

# if-otherwise
if i<20:
    # do A
otherwise:
    # do C
endif

# if-elif-otherwise
if i<20:
    # do A
elif i<10:
    # do B
otherwise:
    # do C
endif

# the difference between multiple if-groups and elifs

# multiple if-groups
if condition_A:
    # body_A
```

```
    endif

    if condition_B:
        # body_B
    endif

    if condition_C:
        # body_C
        # because body_C is in an independent if-group,
        # it will be evaluated if and only if condition_C is true
    endif

    # many elifs
    if condition_D:
        # body_D
    elif condition_E:
        # body_E
    elif condition_F:
        # body_F
        # because body_F is part of an if-elif group
        # and is the last clause
        # it will be evaluated if and only if
        #    condition_D and condition_E are false, and condition_F is true
    endif

```

Functions:

A function is a group of statements with a name, but is more so a group of anything with a name, including functions. A function definitions looks like so:

```
function myName:
    # statements go here
endfunction
```

Expressions:

As stated earlier, expression statements are perhaps the most important type of simple statement. They are composed only of an expression (which need not exist in an expression statement btw), expressions being the values and computations of a program.
Expressions come in many forms, and are sometimes composed by combinations of other expressions. The expressions are literals, identifiers, function calls, and valid alterations and combinations of them via operators.

- Literals are expressions that explicitly represent a value of one of the KangaScript data types. For information about the data types and how they are written, refer to their section above.
- Identifiers/variables allow the programmer to save and name values. To refer to the value of a variable, one just needs to write out its name. Storing is done with the Assign_Equals operator. An identifier must fit a few criteria: They are made of one or more characters, beginning with an alphabetic letter (A-Z or a-z) or underscore (_). Additional characters may be alphanumeric (A-Z, a-z, or 0-9) or underscores. Also, identifiers cannot be the name of an existing KangaScript keyword. KangaScript identifiers are case-sensitive (like Java, JavaScript, and Python, unlike Visual Basic).
    - # valid examples
    - rock = 64
    - _robot = “Hello, Earth”
    - f1f4 = 1234
    - _____ = “qwerty” # valid, but not recommended. very confusing!
    -
    - # invalid examples
    - 4 = 8
    - 0ctpus = “Swim, Swim!”
- The ‘this’ keyword is a special type of identifier. It is an object and refers to the current environment the code using the identifier is running in. this cannot be overwritten, but its member attributes can. When used in the global environment, this will substitute for the env, and it’s member attributes are the other variables in the environment. When in the body of a complex statement inside the global environment, this is a merge of the looping and global environments. When in the body of a function, this refers to the function value
- Function calls
- Parentheses and PEMDAS
- Operators
    - Binary
        - Comparison
            - ==
            - > // GT
            - >=
            - < // LT

            - <=
            - in
            - has
        - Assignment
            - =
            - variation: _=
                - .=
                - +=
                - -=
                - *=
                - /=
                - %=
                - ^=
                - &=
                - |=
        - Manipulation
            - .
            - +
            - -
            - *
            - /
            - %
            - ^
            - and
            - or
    - Unary
        - not
        - =!
    - Array
        - arr[index]
        - arr[:], arr[s:], arr[:e], arr[s:e]
        - arr[::], arr[s::], arr[:e:], arr[s:e:], arr[::st], arr[s::st], arr[:e:st], arr[s:e:st]

References:

http://stackoverflow.com/questions/359494/does-it-matter-which-equals-operator-vs-i-use-in-javascript-comparisons
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String
http://saladwithsteve.com/2008/02/javascript-undefined-vs-null.html
http://en.wikipedia.org/wiki/Boolean_data_type