Cygni Primer

He Yanjie

November 2016

Contents

Chapter 1

Introduction

1.1 What is Cygni

Cygni is a scripting language, implemented in C#. I designed Cygni in my spare time, and I love it very much. I hope you will like it!

1.2 What can Cygni do

Cygni supports both procedure-oriented and object-oriented programming. It has simple syntax, you will learn it quickly if you have programming experience before. It interacts with C# pretty well, therefore you can easily wrap C# methods and classes, then import them into Cygni. Or you can do it the other way round.

1.3 The inspiration of Cygni

The design of Cygni is inspired by some scripting language, such as Python, Lua, etc. Thanks to these great languages!

1.4 Any Questions?

If you have any questions, please feel free to discuss with me. My github: https://github.com/JasonHe0727

Chapter 2

Core Cygni

Ok, if you have some programming experience before, I think maybe you don't have to go through all of the contents. Some small examples may be more helpful. Let's begin.

2.1 Your First Cygni Program

Here is a simple script which writes message to the screen.

2.1.1 The Code

```
print("Hello Cygni!")
```

2.1.2 Running the Program

Input the previous code in the interpreter, then you will see the input "Hello Cygni!" in the screen.

2.2 Variables

The variables in Cygni are dynamic, which means they can be any type, such as number, boolean, string, etc.

You declare variables in Cygni by simply assigning values to them:

```
a = 12
b = true
```

You may assign one value to several variables:

```
a = b = 89.32
```

Name	CTS Type	Description	Range(Approximate)
number	System.Double	64-bit, double precision floating point	$\pm 4.9 \times 10^{-324} \sim 1.7 \times 10^{308}$
boolean	System.Boolean	Represents true or false	true or false
string	System.String	Unicode character string	

Table 2.1: Predefined Types

2.3 Variable Scope

The scope of a variable is the region of code from which variable can be accessed. In Cygni, the scope is determined by the following rules:

- A class, which contains fields, has its own scope.
- A function, which contains local variables, has its own scope.
- If a variable is defined in the program, but not defined in a class or a scope, then it is a global variable, which stays in the global scope.
- Built-in variables are in the built-in scope, which is immutable in the runtime.

The rule of finding variables is: local -¿ field -¿ global -¿ built-in

2.4 Types

See table 2.4.

2.4.1 The number type

The number type is actually the double type in C#.

2.4.2 The boolean type

The boolean type can only be true or false.

2.4.3 The string type

Literals of the string type should be enclosed by ” or ””. Backslashes represents escape sequence. If you write symbol @ at the beginning of the sentence, the backslashes will lose its meanning.

```
s1 = "a string"
s2 = 'another string'
s3 = @'c:\test.txt'
# If literals start with @,
# the backslashes will lose its meanning
s4 = 'c:\\test.txt' # The 's4' equals to 's3';
```

2.5 Flow Control

2.5.1 Conditional Statement: The if Statement

The syntax of 'if' statement is as followings:

```
if condition {
        # Do something
} else {
        # Do something
}
```

The final 'else' statement is not a must. If you have several branches, it is convenient to use 'elif' statement for represent them.

```
if condition {
        # Branch One
} elif {
        # Branch Two
} elif {
        # Branch Three
}
```

Here is an 'if' statement example:

```
if x > 10 {
        print('x is greater than 10')
} else {
        print('x is not greater than 10')
}

if y == 5 {
        print(5)
}
elif y == 10 {
        print(10)
```

```
}
elif y == 20 {
        print(20)
}
else {
        print('y is not 5, 10 and 20.')
}
```

2.5.2 Loops

The for loop

Cygni for loop provides a mechanism for iterating from a given start to a given end. The syntax is

```
for iterator = start, end {
        # Do something
}
```

The iterator will increase one in performing one iteration. If you want to define the step rather than one, you may use the following syntax:

```
for iterator = start, end, step {
        # Do something
}
```

If the step is negative, the negative step will be added to the iterator in every iteration. Note that the step can't be zero. Namely, the step is a non-zero integer. If the step is not an integer, it will be forced into an integer. Another important point is that you should not change the value of the iterator inside the loop.

The while loop

The syntax of while is

```
while condition {
        # Do something
}
```

Unlike the for loop, the while loop are able to repeat a block of statements for a number of times that is not known before the loop begins. The condition only taks a boolean value.

The foreach loop

The foreach loop enables you to iterate through each item in a collection(If you don't know what is collection, don't worry. We will talk about it later). There are some built-in collections, such as lists, dictionaries, etc. The syntax is

```
foreach item in collection {
        # Do something
}
```

Note that you should not change value inside the loop.

2.5.3 Loop Examples

The followings are three examples using different statements to print 1 to 9 in the console.

```
for i = 0, 10 {
        print(i)
}


i = 0
while i < 10 {
        print(i)
        i = i + 1
}

foreach i in range(0,10) {
        print(i)
}
```

2.5.4 Jump Statements

There are three jump statement in Cygni: break, continue, and return. 'break' can jump out from the current loop, 'continue' can start a new round in the loop. 'return' is used to return value from a function.

The break Statement

If the break statement occurs in a loop, the current loop will stop.

The continue Statement

The continue statement is similar to break. Nevertheless, the execution of the current loop will restart at the beginning of the next iteration of the loop, rather than stop.

The return Statement

The return statement is used to exit a function, returning control to the caller of the function. If the function don't return value, you may just write 'return null' instead.

2.6 Console I/O

There are two functions used to read and write in the console in most of the cases: print and input.

```
print('Always nice to see you, Waston.')
```

2.7 Using Comments

The line comments start with symbol #.

```
# This is a line comment
```

2.8 Rules for Identifiers

Identifiers are the names you give to variables, functions, structs, classes, and so on. Identifiers are case senstive. Here are the rules for identifiers you can use in Cygni:

- They can contain letter , underscore or numeric characters. However, they must begin with a letter or underscore.
- The Cygni keywords can't be used as identifiers.

The following list displays the Cygni reserved keywords.

- if
- else

- elif
- for
- while
- foreach
- in
- break
- continue
- return
- def
- class
- true
- false
- null

Chapter 3

Functions

3.1 Declaring Functions

In Cygni, the definition of a function consists of def keyword, followed by the function name, followed by a list of input arguments enclosed in parentheses, followed by the body of the function enclosed in curly braces:

```
def FunctionName([parameters]) {
        # Function Body
}
```

The function will return the last value of the function body without the return statement. If you write a function without "return" statement, the execution of the function will return null value.

In Cygni, you can wrap C# methods as Cygni functions, which are called the native functions. The usage of native functions is the same as Cygni functions.

3.2 Invoking Functions

```
def square(x) {
        return x * x
}

square(15)
```

3.3 Recursion

A classical example is the computation of factorial.

```
def fact(n) {
        if n == 0 {
                return 1
        } else {
                return n * fact(n - 1)
        }
}
fact(10)
```

3.4 Higher-order Functions

The function can be passed as a parameter.

```
def mul(x){
        return x * y
}

def mul2(f, x){
        return f(x, 2)
}
```

Chapter 4

Structures

The structure is a container, used to organize simple data. The syntax is

```
struct('field1', value1, 'field2', value2, ...)
```

The struct can be initialized by "struct" function. The arguments must be in pairs, and each pair consists of a field and a value. Note that the field must be string.

Here is an example:

```
Susan = struct('name', 'Susan', 'age', 31)
=>  struct: {
                name: "Susan"
                age: 31
}
Susan.name
Susan.age
```

The function can return a struct, namely, you can use it as a stucture constructor.

```
def complex(real, imaginary) {
        return struct('real', real,
         'imaginary', imaginary)
}
a = complex(10, 20)
```

Here are some suggestions. Generally, structures perform better if the number of fields is relatively small, and they have difficult in containing complicated logical. They are suitable for simple data organization. If the logical of the data is more complicated, there exist some better choices, such as classes.

Chapter 5

Classes

The data and functions within a class are known as the class's members. The following example displays the usage of the cygni class.

```
class position{
        def __init(nx, ny){
                this.x = nx
                this.y = ny
        }

        def move(nx, ny){
                this.x = nx
                this.y = ny
        }

        def __toStr(){
                printf("({0}, {1})", this.x, this.y)
        }

}
p1 = position(10,20)
print(p1)
p1.move(35, 47)
print(p1)
```

5.1 Data Members

Data members are those members that contain data for the class.

5.2 Function Members

Functions members are functions defined within the class. The "function members" includes not only functions, but also constructors, operators, indexers, etc. An example is as followings:

5.2.1 Constructors

The default constructor is a non-arg function, which will only initialize the already-exist fields. You can overload the constructor by writing a function which is named as "__INIT".

```
class MyClass {
        def __init([arguments]) {
                # initialzze the class
        }
}
```

In the example "position", the constructor of class "position" initialize two fields "x" and "y".

5.2.2 Implementation Inheritance

If you want to declare that a class derives from another class, use the following syntax:

```
class myDerivedClass: myBaseClass {
        # functions and data members here
}
```

The Cygni class does not support multiple inheritance. Namely, a class can only inherit from one class.

```
class person {
        def __init(name, age) {
                this.name = name
                this.age = age
        }

        def say() {
                print('I am a person!')
        }
}
```

```
class employee : person {
        def __init(name, age, salary) {
                this.name = name
                this.age = age
                this.salary = salary
        }

        def say() {
                print('I am an employee!')
        }
}
```

5.2.3 Hiding Function Members

If a function with the same name is declared in both base and derived classs, the function in the derived class will hide the one in the base class.

There are some built-in functions that can be overwrited.

- __init: Constructor for the class. The default constructor is a non-arg constructor.
- __toStr: Output the class instance as a string.
- __add: override '+' operator.
- __sub: override '-' operator.
- __mul: override '*' operator.
- __div: override '/' operator.
- __mod: override '%' operator.
- __pow: override '^' operator.
- __unaryPlus: override '+' operator.
- __unaryMinus: override '-' operator.
- __cmp: return a integer to indicate the comparision result. This function will override the $'>','<','>=','<='$ operators.
- __equals: return a bool value to indicate whether two objects are the same. The function will over load the '==','!=' operators.

5.2.4 Example: vector

```
class vector {
        def __init(values) {
                this.values = values
        }

        def __add(other) {
                if this.values.count != other.values.count {
                        throw ("vector length error")
                }
                result = []
                for i = 0, this.values.count {
                        result.append(this.values[i] + other.values
                }
                return vector(result)
        }

        def __unaryMinus() {
                result = []
                for i = 0, this.values.count {
                        result.append(-this.values[i])
                }
                return vector(result)
        }

        def __sub(other) {
                return this.__add(other.__unaryMinus())
        }

        def __mul(other) {
                if this.values.count != other.values.count {
                        throw ("vector length error")
                }
                sum = 0
                for i = 0, this.value.count {
                        sum = sum + this.values[i] * other.values[i
                }
                return sum
        }
```

```
        def __equals(other) {
                if this.values.count != other.values.count {
                        return false
                }
                for i = 0, this.values.count {
                        if this.values[i] != other.values[i] {
                                return false
                        }
                }
                return true
        }

        def __toStr() {
                return strcat("vector(", toString(this.values), ")")
        }
}
```

Chapter 6

Built-in Commands

Built-in commands have special uses. The syntax is

CommmandName `arg1 , arg2 , ...`

The arguments are separated by commas. Commands should have at least one argument.

6.1 dofile

"dofile" command is used to execute a file, which takes one or two argument. The first argument is the path of the file. The second argument is the encoding of the file. If the second argument is omitted, then the encoding is UTF-8.

6.2 import

"import" command is used to load the module of Cygni. The modules should be put into the 'lib' folder, which is inside the folder of the interpreter.

6.3 loaddll

TO DO:

6.4 scope

The "scope" command takes one argument. The value of the argument can be either 'display' or 'clear'. If the argument is 'display', this command will

print every variable in the global scope. Otherwise, this command will clear everything in the global scope.

6.5 delete

The "delete" command is used to remove variable in the global scope. The arguments are the name of the variables.

6.6 setglobal

The "setglobal" command is used to modify variable in the global scope. The first argument is the name of the variables, and the second one is the new value assigned to the variable.

Chapter 7

Lists and Dictionaries

7.1 Lists

7.1.1 List Initialization

A List is declared by using the following syntax:

```
myList = [1, 2, 3, 4, 5]
```

Once you append elements to the last of the list, the length of the list will increase. To know how many elements in the specific list, you can use function "len":

```
len(myList)
```

Or use the field "count":

```
myList.count
```

7.1.2 Accessing List Elements

After a list is initialized, you can access the list elements using an indexer. Lists only support integer index. The index starts from zero.

The lists are able to contain values from various types.

```
myList = [123, 'apple', false]
v1 = myList[0] # read first element
v2 = myList[1] # read second element
```

Generally, there are two ways to access every element in a list.

```
for i = 0, myList.count {
        print(i)
```

```
}

foreach item in myList {
        print(item)
}
```

7.2 Dictionaries

7.2.1 Dictionary Initialization

A Dictionary is declared by using the following syntax:

```
myDictionary = { }
```

Once you add key-value pairs to the dictionary, the length of the dictionary will increase. To know how many key-value pairs in the specific dictionary, you can use function "len":

```
len(myDictionary)
```

Or use the field "count":

```
myDictionary.count
```

7.2.2 Accessing Dictionary Elements

After a Dictionary is initialized, you can access the Dictionary elements using an indexer. The type of key of Dictionaries only support integer, boolean, string. There is no limit for the types of the values of dictionaries.

```
myDictionaty = {'key1', 123, 'key2', 456}
myDictionary['key1']
myDictionary['key2']
```

You can go through every key-value pair in a dictionary by using "foreach" loop. Every key-value pair is returned as a structure with two fields: key and value.

```
foreach item in myDictionary {
        print(item.key)
        print(item.value)
}
```

Chapter 8

Operators

Cygni supports the operators listed in the following table:
See table 8.

8.1 Arithmetic Operators

Cygni supports the following arithmetic operators:

- Add: +
- Subtract: -
- Multiply: *
- Divide: /
- Modulo: %
- Power: ^
- Unary Plus: +

Category	Operator
Arithmetic	+, -, *, /, %, ^
Logical	and, or, not
Comparison	>, <, =, >=, <=, ==, !=
Assignment	=
Member access	.
Indexing	[]

Table 8.1: Operators

- Unary Minus: -

```
a = 10
b = 99.2
a + b
=> 109.2
```

8.2 Logical Operators

Cygni supports the following logical operators:

- and
- or
- not

```
a and b
=> False
```

8.3 Relation Operators

The following relation operators will return a boolean value:

- >
- <
- >=
- <=
- ==
- !=

8.4 Operator Overloading

There are serveral functions that used to overload the operators of the classes.

- __add
- __sub
- __mul
- __div
- __mod
- __pow
- __unaryPlus
- __unaryMinus
- __cmp
- __equals

Chapter 9

Strings

Strings is a very important type in programming. Cygni supported ample usages of strings.

9.1 String Functions