Lab 9 – Interpreted Languages Primer

CPSC 1021 – Spring 2017

CLEMSON School of COMPUTING

Due: Thursday, March 16, 2017, 11:59:59 Midnight

Introduction

Today's lab is designed to introduce interpreted programming languages and inspire you to branch out and experiment with new languages and technologies.

Lab Objectives

- Learn the concepts behind interpreted programming languages
- Examine "Hello World" in Java, Python, and Ruby
- Use the Java compiler and the Python/Ruby command line interpreter
- Write some basic Python/Ruby code
- Use the internet to fill in the gaps!

Prior to Lab

- Download and untar all the code files on Canvas (Lab9Code.tar.gz)
- If interested: https://www.ibm.com/developerworks/java/tutorials/j-introtojava1/
- If interested: https://www.python.org/about/gettingstarted/
- If interested: https://www.ruby-lang.org/en/documentation/quickstart/

Instructions

As you are probably aware, there are many different programming languages and paradigms in use today. While **C/C++** are very popular and important, most projects require a broader depth of programming knowledge. The goal of interpreted languages is to give us programmers even more **flexibility** and **power** while also abstracting away the nitty-gritty details that we must face in C and C++. Most do not require memory management (no malloc or free), compiling, or even variable types! Below we will briefly examine three different interpreted languages and then write a little **OOP** code.

Before we begin, create a Lab9 directory to store your work and download all the code files from Canvas to use in the examples below!

Compiled/Interpreted Language - Java

Java is a good starting point for us because it is somewhat of a hybrid between traditional procedural languages such as C and fully interpreted languages like Ruby. Java programs have a very similar structure to C programs, and even require compilation. Consider the following code:

```
class Hello {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

This is "Hello World!" in Java, and you should notice some familiar elements. A **Main** method where execution begins, a **String[]** of command line arguments, and statements that end in a **semi-colon**. Like C, you have to compile Java code before you can execute it. Use the Java compiler on Hello.java as follows:

```
~$ javac Hello.java
```

This will produce a file Hello.class, which is a **Java bytecde** file. This is very similar to the executable created by the gcc/g++ compiler, but with key differences. Remember that the executable created from a C/C++ program is **directly runnable by your processor.** If you have ever looked into an a.out file, you've probably noticed it's just a bunch of binary gibberish.

If you look into Hello.class however, you will see you can still read it! Which means that your processor probably cannot directly execute this file. So then, how to we run this code? Enter the **JVM** or **Java Virtual Machine**, which is our first look at a code **interpreter**. In an interpreted language, instead of writing code that compiles down to machine-readable binary, we use another program that **interprets** our code and executes the results. The JVM, as its name implies, behaves like a mini-computer and will translate your Java code into something the processor can handle. To use the JVM, run:

```
~$ java Hello
```

You may wonder why we go through this middle man since it all ends up as machine code in the end, but interpreters give us a number of advantages when writing and executing code. Memory management is a big one; consider the following Java code StringThing.java:

```
class StringThing {
  public static void main(String[] args) {
    String cat = new String("cat");
    String dog = new String("dog");

    String combo = new String(cat + dog);
    System.out.println(combo);
  }
```

Notice the use of the String class here; it is very similar to a C++ std::string. Particularly, look at how we always use an explicit **constructor** when trying to get a **new** instance of String. The memory for these new Strings actually goes on the **heap**, meaning it is dynamically allocated. In C/C++ we would normally need to call some kind of free/delete function to ensure we don't have any memory leaks… but not so in Java! The JVM will keep track of allocating and freeing memory for us, no malloc/free required.

Summary – Java

In summary, Java is often seen as a nice compromise between procedural languages and pure interpreted languages. It enforces familiar program structure while also hiding some of the messy details present in lower level languages. Running our code in the JVM means we don't need to worry too much about memory management and we get lots of other features like exception handling. Oracle has some great official Java tutorials if you're interested in more info.

Interpreted Language – Python

Python is probably the most popular interpreted language used today, aside from maybe Javascript. Often Python programs are referred to as **scripts**, because they lack the formal structure present in procedural languages like C/C++. When running a Python script, statements are simply executed from top to bottom. This lack of structure can seem confusing, but leads to quick and elegant code. For example, here is "Hello World!" in Python:

```
print "Hello World!"
```

The python interpreter is in a way very similar to the JVM in that it executes a Python script we write and translates it to machine instructions. One big difference you'll notice right off the bat is that Python does **not** require a main method to execute! You can run Hello.py using the following command:

```
~$ python Hello.py
```

Easy right? Python, and many other interpreted languages, strive for simplicity in **syntax** and **readability**. In the case of Python, this is both a blessing and a curse… one of the biggest headaches in Python is making sure your code is properly indented. Unlike C/C++, **indentation matters** and your code will not run if it is not indented properly.

Python **is strongly but not explicitly typed**. This means that when you declare a variable, the interpreter infers its type… but you can change it on the fly with casting. A single variable could be many different types during its lifetime, consider the following code in StringThing.py:

```
month = "10"
day = "5"
year = "1993"

date = month + " " + day + ", " + year
print date

date = int(month+day+year)
print date
```

None of our variables is explicitly typed, and in fact, the "date" variable changes types from String to Int during execution of this code.

Summary - Python

Python is a powerful language and once you master it you'll be amazed at how quickly and elegantly you can solve problems that takes hundreds of lines in C/C++. This comes at a price however… despite how readable Python is, its lack of enforced **structure** means it can be very difficult to understand Python code you have not written. It also means that there are **many ways** to solve a single problem, and having a big tool chest sometimes encourages one-off solutions or quick fixes instead of reliable code. If you want to play around more, you can simply type **python** in the terminal to use the interactive interpreter. Press **Ctrl+D** to exit.

Interpreted Language – Ruby

Ruby is very similar to Python in that it is an interpreted language with fluid typing and simple syntax. Ruby programs are also called scripts and they are executed by the Ruby interpreter which you can invoke using the **irb** command at the terminal. As before, here is "Hello World!" in Ruby:

```
puts "Hello World!"
```

Which can be run using the command:

```
~$ ruby Hello.rb
```

Ruby and Python are quite similar not only in their uses but also in how they are written. Ruby is very human readable; good Ruby code almost looks like pseudocode! Unlike Python, Ruby has explicit scope identifiers instead of relying on indentation (i.e. {} in C/C++). Ruby is also interesting in that it allows for a mystical practice known as "meta-coding", which is essentially writing code that can modify itself on the fly.

For completeness, here is a StringThing example in Ruby:

```
dog_names = ["Watson", "Chloe", "Oliver"]
my_dog = "Watson"

if dog_names.include?(my_dog) then
  puts "Hey there #{my_dog}!"
  puts "Your name is more than 5 letters long!" if my_dog.length() > 5
else
  puts "Who's pupper are you #{my_dog}?"
end
```

One cool feature of Ruby is the ability to have trailing if statements. It is also customary for a method that returns a Boolean to end in a "?". Ruby also support direct embedding of variables in Strings, as shown above.

Summary – Ruby

It is hard to say which pure interpreted language is "better" between Python and Ruby. Both have their own features and quirks, with Python being very useful in data analysis and scientific settings thanks to list comprehensions and data mapping. Ruby is useful in web server applications like Rails and for general OOP projects. I would recommend giving each a shot and seeing which you enjoy more, but learning both would be even better!

Assignment

Included in the file download are **Date.py** and a **Date.rb**. This is an implementation of last week's lab in both Python and Ruby. Open them up and look them over, just to see what a simple object oriented program looks like in these languages. Don't worry if it all doesn't make sense or seems confusing! This is about being exposed to different styles and technologies, to expand your horizons.

Now, if you run either of these programs they will not produce the correct output. This is because I left the Date.older/older? method unimplemented. Pick one of the Date programs and try to implement the missing function, such that it will print "Aug 3, 1895" when run.

I'll be brief on guidance or instructions here… use the other code samples to get a feel for each language and use Google to your advantage! See if you can implement the functions in **one line**.

(HINT: One of the code samples might help you condense to a single line…)

Submission Instructions

Please raise your hand when you are done and your lab TA will check out your work! It's okay if you can't get the code working, but you will only be allowed to leave early if you complete the assignment. This is about participation.