Project 4

COMP 443 – Programming Languages
Spring 2020

Due Tuesday, May 5 by 11:55 pm

Overview

For this project, you will be writing an interpreter for the Grove language.

You can work by yourself or in a group of two people for this project. All of the work on this project should be your team's own work. Do not show code to anyone outside your group, nor look at anyone else's code for this project. You can discuss general concepts with people outside your group (e.g., How do I create a object when I have a string for its type?), but not how they pertain to this project.

Learning Objectives

Completing this project will give you experience with Python, including...

- Objects
- Duck typing
- Introspection

Program Specifications

The ideas for this project build on parts of the Calc interpreter that we built in class. There are different ways to build your parse trees for the Grove language (detailed below), but you will likely want to add additional classes for some of the new kinds of language expressions and statements.

As with our Calc example, there is a parse step and an evaluate step.

- The parse step should build a parse tree, but should not change any state of the Grove program. Put another way, if your interpreter were to parse the statement one time or a hundred times, it should make no difference to the behavior of the Grove program.
- The eval step should take the parse tree from the parse step and execute the statement or expression.

When errors occur with the parsing or evaluating of expressions, your program should throw a GroveError, defined as follows:

```
class GroveError(Exception):
    def __init__(self,*args,**kwargs):
        Exception.__init__(self,*args,**kwargs)
```

The command

```
raise GroveError("my message")
```

will create and raise the exception.

Raising the GroveError exception is important, because it is how I verify that you are catching invalid commands. Note that some errors should be raised during parsing (e.g., expressions that do not conform to the grammar), while other errors should be raised during evaluation (e.g., expressions that are syntactically correct but have mismatched types or have undefined methods or variable names).

Language

The Grove language is given on the last page of this document. This section explains the aspects of the language that differ from the Calc language we discussed in class.

- Name is more general than in Calc. A valid Name must start with an alpha**betic** character (including _), followed by zero or more alpha**numeric** characters (also including _). This is taken from Python 2's definition, which states the same idea in BNF: https://docs.python.org/2/reference/lexical_analysis.html#identifiers
- Expr no longer supports subtraction, but there are two new kinds of expressions:
    - **String literals**. The rule <Expr> ::= "<Token>" means that the following are valid expressions (**including the quotes**):
        - "hello"
        - "a.b.c"

      whereas these are not:
        - "a b c"
        - "a\"b"

      You should add a StringLiteral class to your available parse tree nodes. It should evaluate to the string itself. For example, StringLiteral("hi").eval() == "hi"
    - **Method calls.** A method call expression starts with the keyword `call`, followed by open parenthesis, a Name, another Name, zero or more Arguments, and closing parenthesis.
        - The first Name is the name of an object in our global variables table. When you evaluate the expression, if that name does not exist in the variables table, raise a GroveError.
        - The second Name is the method name. When you evaluate the expression, you should check that the specified object has the method by using introspection. If the method is not defined, raise a GroveError with a message indicating which method is not defined.
        - The "<Expr>*" syntax in the language specification indicates zero or more expressions come between the method name and the closing ")".
          To parse the arguments, after handling the object Name and method Name, run a while loop:
          while the next token is not ")" and there are still more tokens to parse:
              parse the next token(s)
              verify that the result is an Expr
              append the Expr to a list of arguments (to be evaluated later)
        - The arguments for these method calls are standard, positional arguments (i.e., not like name=value arguments or default arguments). If you have a list of values and you want to split them up so they are separate arguments to a

Python function call, you can use the *operator to do so. See https://docs.python.org/2/tutorial/controlflow.html#unpacking-argument-lists for an example.

- The Addition operation should be modified from the Calc language, since Grove supports more than just integers. When evaluating the expression, throw a GroveError if the two values being added do not have the same type.

- Stmt has changed substantially
  - The "set" keyword denotes a variable assignment, as in the Calc language. However, there are now two kinds of right-hand-sides to accept. The first is an Expr, as in the Calc language.
    The second is "new" <Name> or "new" <Name>.<Name>
    It allows for statements like
    `new list`
    or
    `new calendar.Calendar`
    A "new" statement should create a new Python object of the specified type, which is then assigned to the variable name on the left-hand-side of the set statement. For simplicity, our language does not support constructor arguments.

  - The "quit" and "exit" keywords should each exit your interpreter when "evaluated." You can use Python's sys.exit() to do so.

  - The "import" keyword allows access to existing Python modules. The evaluation of a Stmt with the "import" keyword should import the specified module so it is available for later commands. Use importlib.import_module to do this. Note that importlib.import_module returns the module, which you will need to store in the globals() dictionary. To figure out the key you should use, look at how globals() changes when you run a normal import statement in Python, like `import os`

Interpreter

Once you have the classes defined for your parse tree nodes, you are ready to create them and evaluate them.

Write **a parse function** that **takes a string and returns the root of a parse tree.**

That **root should have an eval() method** that "runs" the command, producing the behavior described in the "Language" section above. If the command is an expression, the eval() method should return the value of the expression.

One of your Python files should be called **grove.py**. When run as a script (i.e., when the __name__ is "__main__"), that file should enter a **read-evaluate-print-loop** until the user enters a quit or exit command. In your loop, you should do the following:

- Prompt the user for input with "`Grove>> `"
- Read the line that the user enters

- Parse the line
- Evaluate the command
- If the command is an expression, print the result to the screen. However, if the result is None, do not print anything.
- Repeat

If there is a GroveError with either the parsing or evaluation, you should **catch it in your loop** and print out an error message for the user before prompting for the next command.

Written Question

In a comment at the top of grove.py, answer the following question [5 points]:

Is your Grove interpreter using a static or dynamic type system? Briefly explain what aspects of the interpreter make it so.

Handin and Grading

Zip your .py files and submit to mygcc. Test grading (out of 95 points maximum) is based on how many test cases you pass. For most of the test cases, you must have your grove.py file correctly launch the REPL, which should display the "Grove>> " prompt. You also need to have the quit and exit statements correctly implemented. Each test case should **match exactly in order to receive credit** for that test case.

Optional: Testing Code

On myGCC, I provide the testing files which will be used for grading in case you want to run them on your machine. Those files include the following:

- no_parse.txt is a list of commands, one on each line, that should raise GroveErrors when you try to parse them. [16 points]
- no_eval.txt is a list of commands, one on each line, that should parse correctly, but raise GroveErrors when you try to evaluate them. [14 points]
- bad_call.txt has an example of trying to call a method that does not exist. The associated GroveError message should indicate that the method does not exist. [4 points]
- bad_var_types.txt has an example of trying to add two variables with different types. This should parse correctly, but raise a GroveError when you try to evaluate the addition. [4 points]
- singles.txt is a list of expressions, one on each line, that should parse correctly and evaluate to the respective values in singles_answers.txt. Each line can be evaluated independently of the others. [10 points]
- single_stmts.txt is a list of statements that should parse and evaluate correctly. Each statement can be evaluated independently of the others. [11 points]
- test1.txt (and similar) are lists of commands that should parse and evaluate correctly when run in order. The corresponding answer1.txt contains the output that your program should produce for test1.txt. [36 points]

The check_errors.py checks the no_parse.txt, no_eval.txt, and bad_var_types.txt cases. It also prints out the errors received from the bad_call.txt test, which should indicate that the notHere method was not found. The run_tests.sh is a bash shell script that I will use to run all of the tests and calculate points. You can use Linux or a Cygwin installation on Windows to run it.

The Grove Language

Supports importing Python modules, creating objects, and calling methods.

```
<Command>        ::=  <Expr>      |  <Stmt>


// Expressions evaluate to a value
<Expr>           ::=  <Num>       |  <Name>
                                  |  "+" "(" <Expr> ")" "(" <Expr> ")"
                                  |  "<Str>"
                                | "call" "(" <Name> <Name> <Expr>* ")"


<Str>            ::=  any string not containing whitespace or quotes

// Numbers are constant, non-negative integers
<Num>            ::=  string of one or more digits

// Names are variable names, module names, method names, etc.
<Name>                      ::=  a string of one or more alphanumeric
                     characters, starting with an alphabetic character
                     For us, alphabetic includes _


// Statements do not evaluate to a value
// Instead, they have some side effect
<Stmt>           ::=  "set" <Name> "=" <Expr>
                                | "set" <Name> "=" "new" <Name>
                                | "set" <Name> "=" "new" <Name>"."<Name>
                                | "quit"
                                | "exit"
                                | "import" <Name>
```