CptS355 - Assignment 4 (PostScript Interpreter - Part 1)
Fall 2018

An Interpreter for a Simple Postscript-like Language

Assigned: Thursday October 18th, 2018

Due: Monday October 29th, 2018

Weight: The entire interpreter project (Part 1 and Part 2 together) will count for 12% of your course grade. This first part is worth 4% and second part is 8% - the intention is to make sure that you are on the right track and have a chance for mid-course correction before completing Part 2. However, note that the work and amount of code involved in Part 1 is a large fraction of the total project, so you need to get going on this part right away.

This assignment is to be your own work. Refer to the course academic integrity statement in the syllabus.

Turning in your assignment

All the problem solutions should be placed in a single file named HW4_part1.py. When you are done and certain that everything is working correctly, turn in your file by uploading on the Assignment4 (Interpreter-Part1) DROPBOX on Blackboard (under AssignmentSubmisions menu).

The file that you upload must be named HW4_part1.py . Be sure to include your name as a comment at the top of the file. Also in a comment, indicating whether your code is intended for Unix/Linux or Windows. You may turn in your assignment up to 4 times. Only the last one submitted will be graded.

Implement your code for Python 3. The TA will run all assignments using Python3 interpreter. You will lose points if your code is incompatible with Python 3.

The work you turn in is to be **your own personal work**. You may **not** copy another student's code or work together on writing code. You may not copy code from the web, or anything else that lets you avoid solving the problems for yourself.

Grading

The assignment will be marked for good programming style (appropriate algorithms, good indentation and appropriate comments -- refer to the [Python style guide]) -- as well as thoroughness of testing and clean and correct execution. You will lose points if you don't (1) provide test functions / additional test cases, (2) explain your code with appropriate comments, and (3) follow a good programming style.

The Problem

In this assignment you will write an interpreter in Python for a **simplified** PostScript-like language, concentrating on key computational features of the abstract machine, omitting all PS features related to graphics, and using a somewhat-simplified syntax. The simplified language, SPS, has the following features of PS:

- integer constants, e.g. `123`: in Python3 there is no practical limit on the size of integers
- array constants, e.g. `[1 2 3 4] or [[1 2] 3 4]`
- name constants, e.g. `/fact`: start with a / and letter followed by an arbitrary sequence of letters and numbers
- names to be looked up in the dictionary stack, e.g. `fact`: as for name constants, without the /
- code constants: code between matched curly braces { ... }
- built-in operators on numbers: `add, sub, mul, div, eq, lt, gt`
- built-in operators on boolean values: `and, or, not;` these take boolean operands only. Anything else is an error.
- built-in conditional operators: `if`, `ifelse`; make sure that you understand the order of the operands on the stack. Play with Ghostscript if necessary to help understand what is happening.
- built-in loop operator: `for` (you will implement `for` operator in Part 2).
- built-in operators on array values: `length, get, forall.` (You will implement `length` and `get` in Part 1, and `forall` in Part 2). Check lecture notes for more information on array operators.
- stack operators: `dup, exch, pop, copy, clear`
- dictionary creation operator: `dict`; takes one operand from the operand stack, ignores it, and creates a new, empty dictionary on the operand stack
- dictionary stack manipulation operators: `begin`, `end`. `begin` requires one dictionary operand on the operand stack; end has no operands.
- name definition operator: `def`.
- defining (using `def`) and calling functions
- stack printing operator (prints contents of stack without changing it): `stack`

Part 1 - Requirements

In Part 1 you will build some essential pieces of the interpreter but not yet the full interpreter. The pieces you build will be driven by Python test code rather than actual Postscript programs. The pieces you are going to build first are:

1. The operand stack
2. The dictionary stack
3. Defining variables with **`def`**
4. Looking up names
5. The operators that don't involve code arrays: **all of the operators except `if, ifelse, for, forall` operators, and calling functions** (You will complete these in Part 2)

1. The Operand Stack

The operand stack should be implemented as a Python list. The list will contain Python integers, arrays, and later in Part 2 code arrays. Python integers and lists on the stack represent Postscript integer constants and array constants. Python strings which start with a slash / on the stack represent names of Postscript variables. When using a list as a stack, one of the decisions you have to make is where the hot end of the stack is located. (The hot end is where pushing and popping happens. You should use the end of the list as the hot end, i.e., push and pop from the end).

2. The Dictionary Stack

The dictionary stack is also implemented as a Python list. It will contain Python dictionaries which will be the implementation for Postscript dictionaries. The dictionary stack needs to support adding and removing dictionaries at the hot end, as well as defining and looking up names.

3. Operators

Operators will be implemented as **zero-argument Python functions** that manipulate the operand and dictionary stacks. For example, the `div` operator could be implemented as the below Python function (with comments instead of actual implementations)

```python
def div():
    op1 = # pop the top value off the operand stack
    op2 = # pop the top value off the operand stack
    # push (op1 / op2) onto the operand stack
```

The `begin` and `end` operators are a little different in that they manipulate the dictionary stack in addition to or instead of the operand stack.

(Note about `dict`: Remember that the `dict` operator takes an integer operand from the operand stack and pushes an empty dictionary to the operand stack (affects only the operand stack). The initial size argument is ignored – Postscript requires it for backward compatibility of `dict` operator with the early Postscript versions).

The `def` operator takes two operands from the operand stack: a string (recall that strings that start with "/" in the operand stack represent names of postscript variables) and a value. It changes the dictionary at the hot end of the dictionary stack so that the string is mapped to the value by that dictionary. Notice that `def` does not change the number of dictionaries on the dictionary stack!

4. Name Lookup

Name lookup is implemented by a Python function:

```python
def lookup(name):
    # search the dictionaries on the dictionary stack starting at the top to find one that contains name
    # return the value associated with name
```

Note that name `lookup` is not a Postscript operator, but you will implement it in your interpreter. In Part 2, when you interpret simple Postscript expressions, you will call this function for variable lookups and function calls.

You may start your implementation using the below skeleton code:

```
#-------------------------- 10% -------------------------------------
# The operand stack: define the operand stack and its operations
opstack = []

# now define functions to push and pop values to the top of to/from the top of
the stack (end of the list). Recall that `pass` in Python is a space
holder: replace it with your code.

def opPop():
    pass

def opPush(value):
    pass

# Remember that there is a Postscript operator called "pop" so we choose
different names for these functions.

#-------------------------- 20% -------------------------------------
# The dictionary stack: define the dictionary stack and its operations

dictstack = []

# now define functions to push and pop dictionaries on the dictstack, to define
 name, and to lookup a name

def dictPop():
    pass
# dictPop pops the top dictionary from the dictionary stack.

def dictPush():     OR      def dictPush(d):
    pass                        pass
#dictPush pushes a new dictionary to the dictstack. Note that, your interpreter
will call dictPush only when Postscript “begin” operator is called. “begin”
should pop the empty dictionary from the opstack and push it onto the dictstack
by calling dictPush. You may either pass this dictionary (which you popped from
opstack) to dictPush as a parameter or just simply push a new empty dictionary
in dictPush.

def define(name, value):
    pass
#add name:value to the top dictionary in the dictionary stack. (Keep the ‘/’ in
name when you add it to the top dictionary) Your psDef function should pop the
name and value from operand stack and call the “define” function.

def lookup(name):
    pass
# return the value associated with name.
# What is your design decision about what to do when there is no definition for
# name? If “name” is not defined, your program should not break, but should
# give an appropriate error message.

#-------------------------- 15% -------------------------------------
# Arithmetic and comparison operators:   define all the arithmetic and
comparison operators here --  add, sub, mul, div, eq, lt, gt
#Make sure to check the operand stack has the correct number of parameters and
types of the parameters are correct.
```

```

#--------------------------- 15% -------------------------------------
# Array operators: define the array operators length, get
#--------------------------- 15% -------------------------------------
# Boolean operators: define the boolean operators psAnd, psOr, psNot
Remember that these take boolean operands only. Anything else is an error

#--------------------------- 25% -------------------------------------
# Define the stack manipulation  and print operators:    dup, exch, pop, copy,
clear, stack

#--------------------------- 20% -------------------------------------
# Define the dictionary manipulation operators: psDict, begin, end, psDef
# name the function for the def operator psDef because def is reserved in
Python.
# Note: The psDef operator will pop the value and name from the opstack and
call your own "define" operator (pass those values as parameters). Note that
psDef()won't have any parameters.
```

Important Note: For all operators you need to implement basic checks, i.e., check whether there are sufficient number of values in the operand stack and check whether those values have correct types.
Examples:
`def` operator: the operands stack should have 2 values where the second value from top of the stack is a string starting with '/'
`get` operator : the operand stack should have 2 values; the top value on the stack should be an integer and the second value should be an array constant.

Test your code:

```
def testAdd():
    opPush(1)
    opPush(2)
    add()
    return (opPop() == 3)

def testpsDef():
    opPush("\n1")
    opPush(3)
    psDef()
    return (lookup("n1") == 3)

........
# go on writing test code for ALL of your code here; think about edge cases,
and other points where you are likely to make a mistake.
```

Main Program

To run all the tests, so your main should look like:

```
#--------------------------- TEST FUNCTIONS -----------------------------------
# Include your test functions here

#now an easy way to run all the test cases and make sure that they all return true is

if __name__ == '__main__':
    # add you test functions to this list along with suitable names
```

```
    testCases = [('add', testAdd), ('psDef', testpsDef)]


 for (testName, testProc) in testCases:
        if not testProc():
            return False
    return True
```

```
#but wouldn't it be nice to run all the tests, instead of stopping on the first failure, and see which ones failed
# How about something like:
if __name__ == '__main__':
    # add you test functions to this list along with suitable names
    testCases = [('add', testAdd), ('psDef', testpsDef)]

    failedTests = [testName for (testName, testProc) in testCases if not testProc()]

    if failedTests:
        return ('Some tests failed', failedTests)
    else:
        return ('All tests OK')
```