CptS355 - Assignment 4 (PostScript Interpreter - Part 1)
Fall 2023

An Interpreter for a Simple PostScript-like Language

Weight: The entire interpreter project (Part 1 and Part 2 together) will count for 11.5% of your course grade. This first part is worth 3% and second part is 8.5% - the goal is to make sure you are on the right track and have a chance for mid-course correction before completing Part 2. However, the work and amount of code involved in Part 1 is still 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

The **`HW4_part1.zip`** includes the skeleton code provided for part1.

- **`psItems.py`** : You will not make any changes to this file in **part-1**. It includes the class definitions for representing array and code-array constants. (i.e., `Value, ArrayValue`, and `FunctionValue`).
- **`psOperators.py`** : You will write your PostScript operator implementations in this file. It defines the class `Operators` ; the operator functions you write will be methods of this class.
- **`tests_part1.py`:** Includes the Python `unittests` for the PostScript operator functions
- **`colors.py`:** Includes color definitions. You don't need to make any changes to this file.

All your operator implementations should be added to the **`psOperators.py`** file. When you are done and certain that everything is working correctly, turn in your **`psOperators.py`** and **`psItems.py`** files by uploading them on the HW4-SPS (Part1) DROPBOX on Canvas (on Assignments page). You don't need to make any changes to **`psItems.py`** file; but in case you do we would like to have a copy of your file.

At the top of the file in a comment, please include your name and the **names of the students with whom you discussed any of the problems in this homework**. This is an individual assignment and the final writing in the submitted file should be *solely yours*. 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.

You may turn in your assignment up to 3 times. Only the last one submitted will be graded.

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

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.

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 PostScript features related to graphics, and using a somewhat-simplified syntax. The simplified language, SPS, has the following features of PostScript:

Constants:

- ***integer constants***, e.g. `1, 2, 3, -4, -5.` We will represent integer constants as Python "`int`" values in `opstack` and `dictstack`.
- ***boolean constants***, e.g. `, true , false.` We will represent boolean constants as Python "`bool`" values in `opstack` and `dictstack`.
- ***array constants***, e.g. `[1 2 3 4], [-1 2 3 -4], [1 x 3 4 add 2 sub], [1 [3 5 5] dup length], [1 2 x 4]` where `x` is a variable. We will represent array constants as `ArrayValue` objects (see `psItems.py` file.) Note that arrays can also have subarrays.
- ***name constants***, e.g. `/fact, /x.` A name constant starts with a '/' and letter followed by an arbitrary sequence of letters and numbers. We will represent name constants as Python "`str`" values in `opstack` and `dictstack`.
- ***names*** to be looked up in the dictionary stack, e.g., `fact, x;` as for name constants, without the '/'
- ***code constants (i.e., code-arrays);*** code between matched curly braces { ... }

Operators:

- ***built-in operators on numbers***: `add, sub, mul, mod, eq, lt, gt`
- ***built-in operators on array values***: `array, length, getinterval, putinterval, aload, astore.` These operators should support arguments of type `ArrayValue`.
- ***built-in conditional operators***: `if, ifelse` (you will implement `if/ifelse` operators in Part2)
- ***built-in loop operators***: `repeat, for` (you will implement `repeat` and `for` operators in Part 2).
- ***stack operators***: `dup, copy, count, pop, clear, exch, roll, stack`
- ***dictionary creation operator***: `dict`; takes one operand from the operand stack, ignores it, and creates a new, empty dictionary on the operand stack (we will call this `psDict`)
- ***dictionary stack manipulation operators*** **:** `begin, end.`
  - `begin` requires one dictionary operand on the operand stack; `end` has no operands.
- ***name definition operator***: `def`. We will call this `psDef.`
- ***stack printing operator***: `stack.` Prints contents of `opstack` without changing it.

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 and dictionary stacks
2. Defining variables (with **`def`**`) and l`ooking up names
3. The operators that don't involve code-arrays: all of the operators **except `repeat` loop operator, `if/ifelse` operators, and calling functions** (You will complete these in Part 2)

1. The Operand and Dictionary Stacks – `opstack and dictstack`

In our interpreter we will define the operand and dictionary stacks as attributes of the `Operators` class in `psOperators.py` file. Both stacks will be represented as Python lists. When using a list as a stack, we will assume that the top of the stack is the end of the list (i.e., the pushing and popping happens at the end of the list).

The `opstack` will only include the evaluated values, i.e. the values that the PostScript expressions evaluate to. In the `opstack`:

- The primitive values (i.e., integers, booleans) are represented as Python "`int`" and "`bool`" values, respectively.
- The function and variable names will be represented as Python "`str`" values where the first character is `'/'`.
- Array constants will be represented as `ArrayValue` objects.
- Code-array (i.e., function bodies, and bodies of `repeat`, `if`, `ifelse` expressions) are represented as `FunctionValue` objects.

The `dictstack` will include the dictionaries where the PostScript variable and function definitions are stored. The dictionary stack needs to support adding and removing dictionaries at the top of the stack (i.e., end of the list), as well as defining and looking up names. Note that `dictstack` holds all the local and global variables accessible at a particular point in the program, i.e., the referencing environment.

Note about array constants:
As mentioned above, in our interpreter we will represent array constants as `ArrayValue` objects. Remember that, when the PostScript code is interpreted, the array constant values will be evaluated. If the array constant value includes variables, operators, functions, or other arrays, those will be evaluated before the array constant is pushed onto the stack.
In part-2, we will represent the non-evaluated arrays as `Array` objects. When `Array` objects are evaluated, they will be converted to `ArrayValue` values and then pushed onto the `opstack`. For example,

- `Array([1 1 add 5 2 sub])` will be evaluated to `ArrayValue([2,3])` before it is pushed onto the `opstack`.
- `Array([1 2 3 add 5 eq])` will be evaluated to `ArrayValue([1,True])`
- `Array([1 x 3 4 add x sub])` will be evaluated to `ArrayValue([1,2,5])` where `x`'s value is `2`.
- `Array([1 x y 4])` will be evaluated to `ArrayValue([1,2,3,4])` where `x`'s value is `2` and `y`'s value is `3`.

- `Array [1 [2 3 4] dup length]` will be evaluated to
  `ArrayValue([1, ArrayValue([2,3,4]), 3])`
-

Remember that the `array` operator will pop thse array length (an `int` value) from the opstack and push an array of given length to the `opstack`. Assume the `array` operator initializes all elements of the new array to `None`.

2. `define` and `lookup`

You will write two helper functions, `define` and `lookup`, to define a variable and to lookup the values of a variable, respectively.

The `define` function adds the "`name:value`" pair to the top dictionary in the dictionary stack. Your `psDef` function (i.e., your implementation of the PostScript `def` operator) should pop the name and value from `opstack` and call the "`define`" function.

You should keep the `'/'` in the name constant when you store it in the `dictStack`.

```
"""Helper function. Adds name:value pair to the top dictionary in the dictstack.
(Note: If the dictstack is empty, first adds an empty dictionary to the dictstack
then adds the name:value to that."""
def define(self, name, value):
    pass
```

The `lookup` function should look-up the value of a given variable in the dictionary stack. In Part 2, when you interpret simple PostScript expressions, you will call this function for variable lookups and function calls.

```
"""Helper function. Searches the dictstack for a variable or function and returns its
 value. (Starts searching at the top of the opstack; returns None and prints an error
 message if name is not found. Make sure to add '/' to the beginning of the name.)"""
def lookup(self,name):
    pass
```

3. Operators

Operators will be implemented as **zero-argument Python functions** that manipulate the operand and dictionary stacks. For example, the `add` operator could be implemented as follows.

```
"""Pops 2 values from opstack; checks if they are numerical (int); adds them; then
   pushes the result back to opstack.
"""
def add(self):
    if len(self.opstack) > 1:
        op1 = self.opPop()
        op2 = self.opPop()
        if isinstance(op1,int) and isinstance(op2,int):
            self.opPush(op1 + op2)
        else:
            print("Error: add - one of the operands is not a number value")
            self.opPush(op2)
            self.opPush(op1)
    else:
```

```
        print("Error: add expects 2 operands")
```

- 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. Remember that the `dict` operator (i.e., `psDict` function) affects only the operand stack.

  (**Note about `dict`**: `dict` 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 (i.e., `psDef` function) takes two operands from the `opstack`: a string and a value - recall that strings that start with "/" in the operand stack represent names of PostScript variables. It calls `define` function to add the name and value to the top dictionary.

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. For example,

- `def` operator: the operand stack should have 2 values where the second value from top of the stack is a string starting with '/'
- `getinterval` operator : the operand stack should have 3 values; the top two valuea on the stack should be integers (count and index) and the third value should be an `ArrayValue`.

Also, see the `add` implementation on page 3. You will be deducted points if you don't do error checking.

4. Testing Your Code

We will be using the `unittest` Python testing framework in this assignment. See https://docs.python.org/3/library/unittest.html for additional documentation.

The file `tests_part1.py` provides sample test cases for the SPS operators. This file imports the `Operators` module (`psOperators.py` file) which will include your implementations of the SPS operators.

You don't need to provide new tests in this assignment.

In Python `unittest` framework, each test function has a "`test_`" prefix. To run all tests, execute the following command on the command line.

```
python -m unittest tests_part1.py
```

You can run tests with more detail (higher verbosity) by passing in the -v flag:

```
python -m unittest -v tests_part1.py
```