Python Front-End for Daikon – Instruction Manual

DOCKER IMAGE: Daikon, Python3.9, and Python front-end source code already included:
https://hub.docker.com/r/spencetk/python_front_end
From your terminal:

1. `docker pull spencetk/python_front_end:latest`
2. `docker exec -it <image id> /bin/bash`

From inside container:

    #traverses to folder with python_front_end.py
1. `cd home/python_front_end/src`

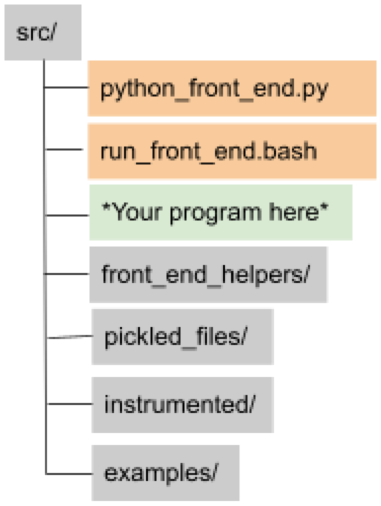
    #copy example program from examples folder to current directory
2. `cp examples/<source.py> .`

    #instruct front-end to perform .decl, .dtrace generation and Daikon to generate results
    #all steps described in detail under Usage section of this README
3. `source run_front_end.bash <source.py>`

Requirements:

- Python v3.9
- Dependencies:
    - ast, pickle, sys, os, collections
- **source.py MUST be on the same file level as `python_front_end.py` to work!**

Project Structure:



1. python_front_end.py
    a. The front end source code that runs and executes a target python program
2. front_end_helpers/
    a. contains helper methods for manipulating the strings outputted by Python's `type(x)` command. The file is opened and writes the methods into an instrumented target file for use before adding anything types or values to a dictionary.
        i. Methods in `prepper.py`
            1. type_prep(x)
                a. Gets corresponding type for a python variable x and modifies output from '<'type' int> to 'int' only
                b. For Lists we check first element's type to determine type of array to use (int[], float[] for Java)
                c. Changes bool to boolean (Java standard)
            2. value_prep(x)
                a. Gets value of variable
                b. List elements are separated by spaces

        c. Boolean values are uppercase in Python, lowercase in Java
            i. True —> true

3. pickled_files/
    a. Contains compact versions of dictionaries d and v for types and runtime values
4. examples/
    a. Example programs that were used to test tool
        i. Includes Paper Examples
            1. int_for_loop.py
            2. sum_of_all_pairs.py
            3. stack.py
        ii. **examples/references** sub-folder holds outputs/data that we could get with front-end ourselves

5. run_front_end.bash
    a. Runs steps in Usage Section

       ```
       $> source run_front_end.bash source.py
       ```

Usage:
The following 3 steps can all be done by run_front_end.bash in Docker Image

1. Inputting program **source.py**, first instrument and run code for type information. This places file "`pickled_types`" into "`pickled_files`" directory . The type instrumented code is placed under the "`instrumented`" directory as "`type_instr.py`".

   ```
   $> python3.9 python_front_end.py source.py
   ```

2. Inputting program **source.py** again, re-instrument and run code for runtime values (use –T flag). This places the file "`pickled_values`" into the "`pickled_files`" directory. The value instrumented code is placed under the "`instrumented`" directory as "`value_instr.py`".

   ```
   $> python3.9 python_front_end.py --T source.py
   ```

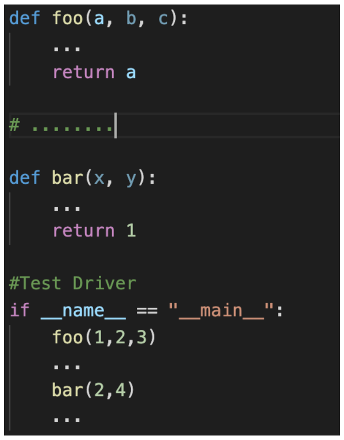
   Outputs: `source.py.decls, source.py.dtrace`

3. Input declaration and dtrace files into a configured Daikon program using the following command:

   ```
   java -cp $DAIKONDIR/daikon.jar daikon.Daikon source.py.decls source.py.dtrace
   ```

Limitations for Input Source Program:

1. Standalone programs only
2. Strictly set of def's with parameters and returns
    a. supports conditional statements but else statements must be "`elif 1`" instead due to ast package limits
3. Test driver must be appended to the bottom of code
4. Supported Python Types:
    a. Primitive Types : int, float, bool
    b. Other: Python Lists
5. Python's type() method seems to work differently for different Python versions. If some of the types in declarations are not parsed properly check the `front_end_helpers/prepper.py` file.



```
def foo(a, b, c):
    ...
    return a

# ........

def bar(x, y):
    ...
    return 1

#Test Driver
if __name__ == "__main__":
    foo(1,2,3)
    ...
    bar(2,4)
    ...
```