README: Checking if input program is in purely functional program style

Attachments: The submission is_pure_functional_program.py, the code check_pure_functional_program.py that I used to run my code, this README file, test cases.

This is the documentation for the Python file `is_pure_functional_program.py`, that, given a Python program as a string input, outputs the Boolean value True, if the input program is a purely functional program, and False if it is not purely functional.

Validation Logic Used

The program `is_pure_functional_program.py` consists of the method `is_purely_functional_program(code: str)->bool`. This method uses the Abstract Syntax Tree (referred as AST) of the program input as code, and runs checks on some nodes of the tree to ensure that it follows the main tenets of purely functional programming:

1. **Input Immutability**
   No function should modify the arguments passed to them. A copy can be created using a local variable for computation. In the attached code, this is done by the method `is_input_immutable(node: ast.FunctionDef)`.

   The method takes in the node `node` of the AST where the subtree parsing a function is rooted, and constructs a list of all the arguments, `arglist`, passed to the function. AST documentation shows that variables are denoted by `ast.Name` nodes with the attribute `id` denoting its name as a string, and the attribute `ctx`, short for context denoting if the variable occurs on the left side (`Store()`) or right side (`Load()`) of an assignment or is being deleted (`Delete()`). Therefore, we walk the subtree rooted at node, and for every `ast.Name` node occurring on the left side of an assignment (i.e., being modified) we check if `Name.id` is in `arglist`.

2. **Single change in variable value**
   Every local variable can be used exactly once. This implies that a local variable can be assigned a value only once, and then used on the right side of an assignment exactly once. In the attached code, this is done by the method `limit_local_var_modification(node: ast.FunctionDef)`.

   The method takes in the node of the AST where the subtree parses the function in question. We then create two Hash Maps (Python dictionaries) for `ast.Name` nodes where `Name.id` is not in the function `arglist`. The dictionaries are named `loaddict`, for variables whose context is an `ast.Load` node i.e., the variable occurs on the left of assignments; and `storedict` for variables with `ast.Store` context node i.e., variables occurring on the right side of assignments.

In both dictionaries, variable name is the key and its frequency of being used in a `Load()` context (or `Store()` context, respectively) is the value corresponding to the key in `loaddict` (or `storedict`, respectively). A local variable should occur in both dictionaries with frequency exactly one.

Additionally, the use of for or while loops is forbidden as they change the value of at least one variable repeatedly. Recursion in preferred in purely functional programs. The methods `visit_For(node: ast.For)` and `visit_While(node: ast.While)` check the input program for For and While loops respectively.

3. No Side Effects of a Function Call

A function call in a purely functional program should not change the value of non-local and global variables, and should not change the global state except for returning the values required. This implies that a function should not use (load, store, or delete) non-local and global variables; and a function should call only pure functions defined inside the input program. In the attached code, several methods are used together to ensure these conditions.

The `visit_FunctionDef(node: ast.FunctionDef)` method ascertains the purity of a function. Due to the recursive visitation of child nodes of each node in the AST, this purity check applies to all functions in the input program.

The `visit_Call(node: ast.Call)` method checks if the function called in defined yet, by checking the list of defined functions and permitted built-in functions, `funclist` for the name of the method called. If it is not there, the function name is pushed into the list of undefined functions, `undeffunc` list. The order in which AST nodes are visited is unknown, so the root node of the AST for a function definition can be visited after the node for the function call. When the function definition node is visited after the function call node, its name is deleted from the `undeffunc` list. At the end of AST traversal, we check if there are function names in the `undeffunc` list, and print an error message if so.

The `visit_Import(node: ast.Import)` and `visit_ImportFrom(node: ast.ImportFrom)` methods prevent "`import X`" or "`from Y import X`" statements in the input program.

Thus, the input program can only call functions that are defined in it, except for some built-in functions hardcoded into the `funclist` list.

In the method `limit_local_var_modification`, if a variable is being used i.e., occurs in `loaddict` dictionary without being defined first i.e., does not occur in `storedict` dictionary, it must be non-local or global. We print an error message accordingly. Additionally `ast.Global` and `ast.NonLocal` nodes are checked for.

4. **No OOP constructs**
   OOP constructs such as classes enable functions to change non-local values. For example, in the attached code, all the methods of `class fpChecker` modify the attributes of the `self` object. So we rule out `ast.ClassDef` and `ast.Attribute` nodes (except for attributes of math, string, list, dict, tuple modules hardcoded into the list `allowlist`).

Exploring the Code

- **Import statement**
  `import ast` is used to import the Python `ast` package.

- **Try Except block**
  According to AST documentation, the Python Interpreter can throw a `ValueError` and possibly other errors, since the input program to a parse tree can have Syntax errors and Runtime errors. I put the code snippet `tree = ast.parse(code)` inside a try except block to minimise the instances of the code crashing.

- **Class fpChecker**
  This class implements the `NodeVisitor` interface of the `ast` package to visit each node of the AST. The constructor for the class initialises a Boolean variable `retval` that will contain the output of the validation code, a list `arglist` that will be used to store input arguments of every function parsed, a list `funclist` containing math and string built-in function names that is also used to store the names of parsed functions and finally, a list `undeffunc` used to contain names of functions called in the input program but not defined in it.

- **Method report**
  Checks for presence of undefined functions and prints an error message accordingly. Returns the final value of `retval`.

- **Method is_input_immutable**
  Builds `arglist`, list of arguments of a function. Checks if arguments passed into this function are being modified.

- **Method limit_local_var_modification**
  Builds the hash maps `loaddict`, `storedict` to check if local variables are

being used more than once. Checks if the function uses a non-local or global variable.

- **Method check_global_nonlocal**
  Check for global or non-local nodes inside the parse tree of the function being inspected.

- **Method visit_FunctionDef**
  Adds function name to defined and allowed built-in functions list, `funclist`; removes function name from `undeffunc` list, if present. Calls methods `is_input_immutable`, `limit_local_var_modification` and `check_global_nonlocal` to validate the function. Resets the function argument list `arglist` to empty at the end.

- **Method visit_Call**
  Adds the name of called function to `undeffunc` list, if not present in `funclist`.

- **Method visit_ClassDef**
  Rules out class definitions.

- **Method visit_Attribute**
  Rules out usage of attributes except for modules in allowlist.

- **Method visit_For**
  Rules out For loops.

- **Method visit_While**
  Rules out While loops.

- **Method visit_Import**
  Rules out importing packages.

- **Method visit_ImportFrom**
  Rules out importing subroutines from packages.

- **Object checkfp**
  Instance of class `fpChecker`, used to call the visit function on the AST.