Q1. Is it permissible to use several import statements to import the same module? What would the goal be? Can you think of a situation where it would be beneficial?

The module is only loaded the first time the import statement is executed and there is no performance loss by importing it again. Any duplicate import statement for the same module in the same program is ignored by Python.

This is actually the norm in Python. In fact, Python only loads the module when it is imported in the first file and all subsequent files simply set the name to refer to the already loaded module. While it is possible to override this behavior so that each file has its own copy of the module, it is generally not recommended.

To see this in action, import the same module in two different files. Set a variable in the module from one of the files and have the other file retrieve that variable from the module.

It would be beneficial when we are dealing with circular imports. A circular import occurs when two or more modules depend on each other. In this example, m2.py depends on m1.py and m1.py depends on m2.py . Its better to have multiple imports of the same module. In all the .py files.

Q2. What are some of a module's characteristics? (Name at least one.)

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use.

Q3. Circular importing, such as when two modules import each other, can lead to dependencies and bugs that aren't visible. How can you go about creating a program that avoids mutual importing?

A circular import occurs when two or more modules depend on each other. In this example, m2.py depends on m1.py and m1.py depends on m2.py .

Import the module: Avoid importing objects or functions from a module that can cause Circular Imports.

Absolute Import

Just use the first import syntax above. The downside to this method is that the import names can get super long for large packages.

In a.py

import package.b

In b.py

import package.a

Defer import until later

I've seen this method used in lots of packages, but it still feels hacky to me, and I dislike that I can't look at the top of a module and see all its dependencies, I have to go searching through all the functions as well.

In a.py

def func():

from package import b

In b.py

def func():

from package import a

Put all imports in a central module

This also works, but has the same problem as the first method, where all the package and submodule calls get super long. It also has two major flaws -- it forces all the submodules to be imported, even if you're only using one or two, and you still can't look at any of the submodules and quickly see their dependencies at the top, you have to go sifting through functions.

In \_\_init\_\_.py

from . import a

from . import b

In a.py

import package

def func():

package.b.some\_object()

In b.py

import package

def func():

package.a.some\_object()

Q4. Why is \_ \_all\_ \_ in Python?

A list of strings that define what variables have to be imported to another file is known as \_\_all\_\_ in Python. The variables which are declared in that list can only be used in another file after importing this file, the rest variables if called will throw an error.

\_\_all\_\_ is a global variable (remember that globals are per module and not per interpreter; it lists the globals which are imported from that module by a star import.

Q5. In what situation is it useful to refer to the \_ \_name\_ \_ attribute or the string '\_ \_main\_ \_'?

Python files can function as independent applications or as reusable modules. If the file was executed directly, without being imported, then the statement "if \_\_name\_\_ == "main" is used to run some code.

It Allows You to Execute Code When the File Runs as a Script, but Not When It's Imported as a Module. For most practical purposes, you can think of the conditional block that you open with if \_\_name\_\_ == "\_\_main\_\_" as a way to store code that should only run when your file is executed as a script.

Q6. What are some of the benefits of attaching a program counter to the RPN interpreter application, which interprets an RPN script line by line?

Reverse Polish Notation (RPN) Calculator which can be used to evaluate postfix expressions. In a postfix operation, an operator appears after its operands. For example, an infix expression like 25 + 12 would be written as 25 12 + in the postfix notation. A postfix expression is evaluated in the order in which the operations appear (left to right).

For this, we need to define a stack class. This stack class will be the container for the operands as well as intermediate results. The final result also gets stored in the stack and is extracted at the end of the process and displayed.

Background

RPN Calculator, also known as Stack Calculator, is a special type of calculator in which there must be two operands before an operator in an expression. The RPN calculator works by pushing operands into a stack until an operator is encountered. When an operator is encountered, the last two pushed operands are popped, the required operation is performed, and the result of the operation is again pushed into the stack. At the end of the expression, the last value is popped from the stack and displayed as the final result.

Using the Code

Following is the code for the Node class which is the building block for our Stack class. It defines a constructor and the required getter and setter methods.

# Representing the node for the stack.

# The node class defines a constructor and the required getter and setter methods.

class Node:

def \_\_init\_\_(self,d):

self.data = d

def setnext(self,n):

self.next = n

def getdata(self):

return self.data

def getnext(self):

return self.next

Following is the code for the Stack class. It defines a constructor and the push and pop operations in addition to a method to check if the stack is empty.

# Representing the stack class.

# The stack class defines a constructor and the implementations for the

# push and pop operations. It also contains a method to check if the stack is empty.

class Stack:

def \_\_init\_\_(self):

self.top = None

def push(self,d):

self.newnode = Node(d)

self.newnode.setnext(self.top)

self.top = self.newnode

def pop(self):

temp = self.top

self.top = self.top.getnext()

n = temp.getdata()

del temp

return n

def isempty(self):

return self.top == None

Following is the code for the main program. The main program involves accepting input from the user and performing the calculations by using the functions of our stack class. The expression entered to be evaluated must have a space between two operands as well as between an operand and operator, for example, 78 100 + 200 - 5 \*

The code uses the sub() function of the regular expression module (re) to subsitute multiple spaces in the postfix expression with one space and the split() function to parse the expression and extract the elements of the expression (operators and operands) into a list. The strip() function is used to remove the leading and trailing spaces. It pushes the operands into the stack and pops the last two operands and pushes their result when an operator is encountered. At the end, it pops and displays the value from the stack as the final result.

import re

if \_\_name\_\_ == "\_\_main\_\_":

try:

mystack = Stack()

expr = input("Enter expression with space between numbers and operators: ")

expr = re.sub(" +"," ",expr)

expr = expr.strip()

elements = re.split(r"[\s]",expr)

for x in elements:

if x == "+":

n1 = int(mystack.pop())

n2 = int(mystack.pop())

n3 = n2 + n1

mystack.push(n3)

elif x == "-":

n1 = int(mystack.pop())

n2 = int(mystack.pop())

n3 = n2 - n1

mystack.push(n3)

elif x == "\*":

n1 = int(mystack.pop())

n2 = int(mystack.pop())

n3 = n2 \* n1

mystack.push(n3)

elif x == "//":

n1 = int(mystack.pop())

n2 = int(mystack.pop())

n3 = n2 // n1

mystack.push(n3)

elif x == "/":

n1 = int(mystack.pop())

n2 = int(mystack.pop())

n3 = n2 / n1

mystack.push(n3)

else:

mystack.push(x)

print("Result: " + str(mystack.pop()))

except AttributeError as e:

print("Invalid Expression: " + str(e))

Q7. What are the minimum expressions or statements (or both) that you'd need to render a basic programming language like RPN primitive but complete— that is, capable of carrying out any computerised task theoretically possible?

Reverse Polish ‘Notation is postfix notation which in terms of mathematical notion signifies operators following operands. Let’s take a problem statement to implement RPN

Problem Statement: The task is to find the value of the arithmetic expression present in the array using valid operators like +, -, \*, /. Each operand may be an integer or another expression.

Note:

The division between two integers should truncate toward zero.

The given RPN expression is always valid. That means the expression would always evaluate to a result and there won’t be any divide by zero operation.

Layman Working of RPN as shown

Input: ["2", "1", "+", "3", "\*"]

Output: 9

Explanation: ((2 + 1) \* 3) = 9

Input: ["4", "13", "5", "/", "+"]

Output: 6

Explanation: (4 + (13 / 5)) = 6

Input: ["10", "6", "9", "3", "+", "-11", "\*", "/", "\*", "17", "+", "5", "+"]

Output: 22

Explanation:

((10 \* (6 / ((9 + 3) \* -11))) + 17) + 5

= ((10 \* (6 / (12 \* -11))) + 17) + 5

= ((10 \* (6 / -132)) + 17) + 5

= ((10 \* 0) + 17) + 5

= (0 + 17) + 5

= 17 + 5

= 22

Approach:

The basic approach for the problem is using the stack.

Accessing all elements in the array, if the element is not matching with the special character (‘+’, ‘-‘,’\*’, ‘/’) then push the element to the stack.

Then whenever the special character is found then pop the first two-element from the stack and perform the action and then push the element to stack again.

Repeat the above two process to all elements in the array

At last pop the element from the stack and print the Result

# Python 3 code to evaluate reverse polish notation

# function to evaluate reverse polish notation

def evaluate(expression):

# splitting expression at whitespaces

expression = expression.split()

# stack

stack = []

# iterating expression

for ele in expression:

# ele is a number

if ele not in '/\*+-':

stack.append(int(ele))

# ele is an operator

else:

# getting operands

right = stack.pop()

left = stack.pop()

# performing operation according to operator

if ele == '+':

stack.append(left + right)

elif ele == '-':

stack.append(left - right)

elif ele == '\*':

stack.append(left \* right)

elif ele == '/':

stack.append(int(left / right))

# return final answer.

return stack.pop()

# input expression

arr = "10 6 9 3 + -11 \* / \* 17 + 5 +"

# calling evaluate()

answer = evaluate(arr)

# printing final value of the expression

print(f"Value of given expression'{arr}' = {answer}")