Assignment 24 Solutions

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?

Ans: If a module has already been imported, it's not loaded again. we will simply get a reference to the module that has already been imported

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

Ans: The following are some of a module's characteristics:

- `__name__` : It returns the name of the module
- `__doc__` : It denotes the documentation string line written in a module code.
- `__file__` : It holds the name and path of the module file from which it is loaded
- `__dict__` : It returns a dictionary object of module attributes, functions and other definitions and their respective values

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?

Ans: Circular importing means importing the two modules in each other. If suppose we are wokring in `MOD1.py` file and it is importing some function say F2() from some other module say `MOD2.PY` file or we can do vice-versa. What will happen is: This will give an `import error` .

This is because when we import F2() function from module `MOD2.py` , then this will execute `MOD2.py` file and in `MOD2.py` file, there is another statement of importing `MOD1.py` module.

This will result in endless loop. To avoid this error, we can use `if __name__ == '__main__'`

In this function, we can't directly refer to the function in the program. The addition of this sentence avoids the endless loop of the program . We can use an if **name** == "**main**" block to allow or prevent parts of code from being run when the modules are imported. When the Python interpreter reads a file, the **name** variable is set as **main** if the module being run, or as the module's name if it is imported

Q4. Why is `_ _all_ _` in Python?

Ans: The `__all__` tells the semantically “public” names from the module. If there is a name in `__all__` , the users are expected to use it, and they can expect that it will not change. By default, Python will export all names that do not start with an _. we certainly could rely on this mechanism.

Q5. In what situation is it useful to refer to the `__name__` attribute or the string `__main_ _` ?

Ans: During the time of execution of the code, if we want to refer the module in which we are working on, then we use `__name__` attribute. In that case it will return the module in which we are working on. Suppose if the module is being imported from some other module, then name will have the name of that module from where the current module has been imported. The current module in which we are working is refer to the string `__main __` .

This built-in attributes prints the name of the class, type, function, method, descriptor, or generator instance.

For example, if the python interpreter is running that module (the source file) as the main program, it sets the special **name** variable to have a value "**main**". If this file is being imported from another module, **name** will be set to the module's name.

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?

Ans: An advantage of reverse Polish notation is that it removes the need for parentheses that are required by infix notation. While 3 − 4 × 5 can also be written 3 − (4 × 5), that means something quite different from (3 − 4) × 5. A program counter is also known as an instruction counter, instruction pointer, instruction address register or sequence control register. It is a digital counter needed for faster execution of tasks as well as for tracking the current execution point

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?

Ans:

In [6]:

```python
# Reverse Polish Notation (RPN) Evaluator in Python
import logging
import operator as op
import sys
from typing import Any, List, Union

logging.getLogger(__name__).setLevel("INFO")
supported_operators = {"+": op.add, "-": op.sub, "*": op.mul, "/": op.truediv}
Number = Union[int, float]

def tokenize(expr: str) -> List[str]:
    """Breaks expression `expr` into a list of tokens"""
    return expr.split(" ")

def mpop(stack: List[Any], n: int = 1) -> List[Any]:
    """Pops and returns `n` items from a stack. Mutates `stack`"""
    return [stack.pop() for _ in range(n)]

def to_num(x: Any) -> Number:
    """Converts a value to a its appropriate numeric type"""
    n = float(x)
    return int(n) if n.is_integer() else n

def consume_token(token: str, stack: List[Number]) -> List[Number]:
    """Consumes a token given the current stack and returns the updated stack"""
    if token in supported_operators:
        try:
            num1, num2 = mpop(stack, 2)
        except IndexError:
            logging.error("SyntaxError: Malformed expression")
            sys.exit(1)

        result = supported_operators[token](num2, num1)
        return [*stack, result]
    else:
        try:
            return [*stack, to_num(token)]
        except ValueError:
            logging.error("SyntaxError: Unsupported token '%s'", token)
            sys.exit(1)

def get_result_from_stack(stack: List[Number]) -> Number:
    """Gets the result from `stack`"""
    result, *rest = mpop(stack, 1)
    if rest:
        logging.error("SyntaxError: Found extra tokens")
        sys.exit(1)
    return result

def evaluate_v1(tokens: List[str]) -> Number:
    """Evaluates a tokenized expression and returns the result"""
    stack: List = []

    for token in tokens:
        stack = consume_token(token, stack)

    return get_result_from_stack(stack)

def evaluate_v2(tokens: List[str]) -> Number:
    """Evaluates a tokenized expression and returns the result"""

    def _evaluate(tokens: List[str], stack: List) -> Number:
        if not tokens:
            return get_result_from_stack(stack)

        stack = consume_token(tokens[0], stack)

        return _evaluate(tokens[1:], stack)

    return _evaluate(tokens, [])

if __name__ == "__main__":

    print(evaluate_v2(tokenize(input())))
```

```
6 3 /
2.0
```