CSE 20
Beginning Programming in Python
Lab Assignment 4

In this project you will write another Python script to be run on the timeshare, as you did in lab3. This script will itself be a (very simple) command line interpreter. Basically, it will evaluate python expressions typed at its "command prompt", and print the values of those expressions back to the user. The program is therefore a desktop calculator of sorts. Accordingly, its name will be `Calc`. Several sample runs of `Calc`, with explanations, are given below.

```
$ Calc
A[0]:= 34 + 17
51
A[1]:= 17**5
1419857
A[2]:= A[0]**2 + A[1]**2
2015993903050
A[3]:=

$
```

As you can see, after invoking the program on the Unix command line, the `Calc` prompt `A[0]:=` appears, waiting for the user to enter an expression. Once an expression is entered, and then evaluated, its value is printed on the next line. This is followed by another `Calc` prompt, this time `A[1]:=`, after which another expression is entered and evaluated. Observe that each prompt has its own index, beginning at 0. The values (or *answers*) to all of these arithmetic expressions are saved internally (as a list) and can be referred to as `A[0]`, `A[1]`, etc. in subsequent expressions. To end the interactive session, the user enters a blank expression. When this blank expression is entered, a blank line is printed, the program ends, and the Unix command prompt returns.

All functions belonging to the Python math library, as well as all built-in functions and operations can be used in these expressions. All built-in data types and their corresponding literals can also be used.

```
$ Calc
A[0]:= 23.45
23.45
A[1]:= sin(36.45)
-0.9487041893631407
A[2]:= 'happy'+'day'
happyday
A[3]:= sum([1,2,3,4,5])
15
A[4]:= sum([x**2 for x in range(1,101)])
338350
A[5]:=

$
```

It is not possible however to do variable assignments, and any attempt will result in a syntax error, ending the program. Of course, variables are not necessary since the list of previous answers `A[0],A[1],A[2],...` can be queried any time during the session. Even expressions involving slices of `A` or the list `A` itself are possible.

```
$ Calc
A[0]:= 5
5
A[1]:= 6
6
A[2]:= 7
7
A[3]:= A[0]+A[1]-A[2]
4
A[4]:= sum(A[:3])
18
A[5]:= sum(A)
40
A[6]:= z=12
Traceback (most recent call last):
  ... ... details of syntax error removed
SyntaxError: invalid syntax
$
```

The reason variable assignments fail is that an assignment, such as `z=12`, is not an expression having a value (not even `None`), and therefore its value cannot be printed and appended to the list `A`. Although assignment statements are not allowed, comparison operators and Boolean operators will work.

```
$ Calc
A[0]:= factorial(20)
2432902008176640000
A[1]:= 13**17
8650415919381337933
A[2]:= A[0]<A[1]
True
A[3]:= A[1]==A[2]
False
A[4]:= A[2] and A[3]
False
A[5]:=

$
```

In general, any expression you can evaluate in a Python interactive session (where the math library has been imported) other than an assignment statement, can be evaluated by `Calc`. In addition to its *interactive mode* illustrated above, `Calc` can also evaluate a single expression placed on the command line. We call this `Calc`'s *command line mode*.

```
$ Calc 93847//457
205
$ Calc 93847%457
162
$ Calc 93847/457
205.35448577680526
$
```

A problem arises when we try to evaluate a Python expression (in command line mode) containing parentheses. Such expressions always produce syntax errors. The reason for this has nothing to do with the `Calc` program, or even with Python, but with the Unix command line interpreter.

```
$ Calc factorial(20)
-bash: syntax error near unexpected token `('
$
```

As you can see, the syntax error message does not come from Python but from the Unix `bash` shell. Unix considers parentheses `(` and `)` to be special characters, and will not interpret them literally unless they are enclosed in quotes: `"("` and `")"`. In fact it is easier, and does no harm, to enclose the whole expression in quotes.

```
$ Calc "factorial(20)"
2432902008176640000
$
```

So with this one exception, `Calc`'s command line mode evaluates any Python expression it can evaluate in interactive mode.

```
$ Calc 2**3 + 4**3 + 6**3
288
$ Calc "sum([x**3 for x in [2,4,6]])"
288
$
```

Now that you know how your program is supposed to operate, you may be thinking that such a program is impossible to write. `Calc` seems to be only a little less powerful than the Python interpreter itself, which you have only just learned. How can you begin such a monumental task? This would indeed be a very difficult project if you had to start from scratch, but Python has a built-in function called `eval()` that allows you to write `Calc` in just a few lines. Function `eval()` takes as input a string containing a valid Python expression, then returns the value of that expression. Let us illustrate `eval()` in Python interactive mode.

```
$ python3
Python 3.8.5 (default, Oct  2 2020, 16:15:42)
[GCC 8.3.1 20190311 (Red Hat 8.3.1-3)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from math import *
>>> a = 'factorial(20)'
>>> b = '23*(12+3**13)'
>>> c = '6'
>>> d = '"happy"'
>>> eval(a)
2432902008176640000
>>> eval(b)
36669705
>>> eval(c)
6
>>> eval(d)
'happy'
```

First, notice how we imported the math library. Normally we do `import math`, then call a function like factorial as `math.factorial(20)`. The different form of the import statement, `from math import *`, allows us to invoke a function without the prefix, as in `factorial(20)`. Read the `*` as "everything", so the import statement says "from math import everything". You'll have to use this form when you import the math library into `Calc`.

Next, observe that the variables `a`, `b`, `c` and `d` are strings. The `eval()` function merely feeds these strings to the Python interpreter, and returns the result. Any valid expression enclosed in quotes will do, even a literal expression like `6`. We can think of `eval()` as stripping off one set of quotes, then evaluating the result. Recall however that a string literal must itself be enclosed in quotes, and hence two sets of quotes are needed for the `eval()` function in this case. In order that the interpreter can tell which open quote matches which close quote, different types of quotes are necessary (double and single). It doesn't matter which type are the outer and which are the inner quotes.

```
>>> eval('"happy"')
'happy'
>>> eval("'happy'")
'happy'
>>> eval('happy')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'happy' is not defined
```

As you can see `eval('happy')` will not work, since after the quotes are stripped off, we're left with just `happy`, which is an invalid expression, unless there is a variable in scope called `happy`.

```
>>> happy=6
>>> eval('happy')
6
```

Hopefully the `Calc` program is now beginning to take shape in your mind. If not, here is a brief outline.

1. Import both the `sys` and `math` libraries, using the proper form for `math`.
2. Examine the list of command line arguments. If there is any expression after `Calc`, feed that expression (as a string) to `eval()`, then print the result and exit.
3. If there is no expression after `Calc`, then the program is to run in interactive mode. Enter a loop that repeatedly prompts for and reads a string from user input, feeds that string to `eval()`, then prints the result. You must append each computed value to a list called `A`, so that subsequent user input can use the expressions `A[0]`, `A[1]`, etc.
4. If at some point user input is blank (i.e. an empty string or just whitespace), then exit the program.

Although this should turn out to be a very short program, there are a few subtleties that were not discussed above. For instance, when the user calculates an expression in command line mode, there may be spaces in the expression. The tokens delimited by these spaces will show up as different strings in the `sys.argv` list. You must join these strings together into a single string before you feed it to `eval()`. There may be one or two other such points that you'll discover only after you have some kind of working program.

Once your program is done, there is one more way to run it in interactive mode. You can prepare a file consisting of exactly those expressions typed by the user at the prompts, including the blank line to quit. When you run `Calc`, use the Unix *input re-direct operator* < to take input from this file instead of from the keyboard.

```
$ Calc < input_file
```

The expected output will appear in the terminal, with output values on the same lines as their respective prompts. You can also use the Unix *output re-direct operator* > to send output to a file, rather than the terminal window.

```
$ Calc < input_file > output_file
```

A matched pair of input/output files are included in `/Examples/lab4` on the class webpage called `testIn` and `testOut`, respectively. Copy these files to the timeshare (in your `cse20/lab4` directory) and do

```
$ Calc < testIn > myOut
```

then

```
$ diff myOut testOut
```

If `diff` gives no output, then the two files are identical. Do not consider this to be a thorough test of your `Calc` program. It just illustrates an efficient way for you to do more extensive tests.

This lab assignment will use everything you've learned about scripts from lab3, and most of the Python you've acquired throughout the quarter. As usual, start early and ask questions if anything is unclear. Submit the file `Calc` to lab4 on Gradescope in the usual way.