CSC372 Project - Group 14

The Language:

The language we chose to design ended up being a simple stack-based language oriented around singular instructions per line. We chose a stack-based language because it felt suitably unique compared to our translation language, Python. The syntax reasoning is mostly for the easier parsing, but it also causes rather unique formatting.

The most similar existing language to ours is likely something like Forth or PostScript. Our language shares the stack-based philosophy, postfix operation, and general design choices (e.g. storing variables in a dictionary), but is much more simplistic.

Our language is extremely imperative - it does not support declarative design patterns like pattern matching. This decision was made simply because it would be easier. Our understanding of imperative languages is much higher than that of declarative languages.

The language is weakly typed with no conversions, and follows Python rules for cross-type operations. For example, a user could multiply a string and an integer, but they cannot add a string and an integer.

The language is interpreted line-by-line. This made stack manipulation relatively easy and was faster to implement compared to writing a compiler.

Beyond syntax errors, the language allows for various errors to be translated (type errors, variable errors, etc.) These are left as an exercise for the programmer to solve.

Process:

We initially began with the idea of writing a Java offshoot with weird syntax, but quickly pivoted in favor of a stack-based language instead. The main difficulty with writing a stack-based language is how fundamentally different they are structured compared to a more typical language (e.g. our translation language Python). A great deal of effort was spent just figuring out the logic of the language.

Due to the simple grammar of our language, implementation of basic regex and command parsing went relatively smoothly. Logic errors and other small issues did crop up, but they were not major problems.

One difficulty in writing the translator was interpreting loops. We ended up making them simpler by having only for loops, and only for integer values (or variables representing integers). We dealt with loops in the main parsing loop of the translator, rather than with a loop handler function like most other commands.

The biggest challenge in writing this language was implementing if-else statements. Conceptualizing them in a stack-based language was, logically speaking, very difficult. Even at the end, although there was some path to functioning else statements, we were unable to reach it. However, we were able to implement if statements and nesting them.

Tutorial:

The translator can be used in two different ways.

You can call it normally though Python to use the live shell. On Windows, this would typically be:

```
python translator.py
```

Alternatively, you can call it followed by the name of a file containing some text that can be parsed by the translator. As above:

```
python translator.py <filename> <optional arguments>
```

The output script will be in stdout. To create an output file, you can do something like:

```
python translator.py > output.py, or
python translator.py <filename> <optional arguments> output.py
```

The contents of this file must follow the syntax below.

A single line will be a single command - the language does not support one-liners or other methods of formatting (with the exception of if-statements).

Each command is a single word followed by a value if applicable. All commands are listed below. Commands are case sensitive.

- Comments are declared by preceding a line with “#”, e.g:
    - # this is a comment
- insert <value> : pushes a value onto the stack. This can be an integer, boolean, string, or previously defined variable. To declare an integer, type an integer (insert 1) To declare a string, enclose the value in double quotes (insert “string”). To declare a boolean, declare it as “true” or “false” (insert true).
- remove : removes the top value on the stack if it exists.
- assign <variable name> : assigns the top item on the stack to some variable name. This variable name must be a single contiguous string. This string cannot be an integer. It does not need to follow any case conventions. You cannot assign the variables “a” or “b” because the Python translator backend relies on these two variables. Oops!
- print : prints the top item on the stack to the console.

- print <variable> : prints the value assigned to the specified variable. Does not print a string, integer, or boolean directly.
- add : adds the top two items on the stack if applicable. Does not support type conversion, but can concatenate strings.
- subtract : subtracts the top two items on the stack applicable. Does not support type conversion.
- multiply : multiplies the top two items on the stack if applicable. Can multiply and concatenate strings and integers. Does not support type conversion otherwise.
- divide : divides the top two items on the stack if applicable. Does not support type conversion.
- modulus : performs a modulus operation on the top two items of the stack if applicable. Does not support type conversion.
- equalto : pops the top two items on the stack and compares them. Pushes the boolean true if they are equal, false otherwise. Can compare any types.
- greaterthan : pops the top two items on the stack and compares them. Pushes the boolean true if the second popped is greater than the second, false otherwise. Does not support type conversion.
- lessthan : pops the top two items on the stack and compares them. Pushes the boolean true if the second popped is less than than the second, false otherwise. Does not support type conversion.
- and : pops the top two items on the stack and pushes the result of the logical AND. Follows Python's rules for AND-ing, e.g. 1 and 2 will push 2, 2 and 1 will push 1.
- or : pops the top two items on the stack and pushes the result of the logical OR. Follows Python's rules for OR-ing, e.g. 1 or 2 will push 1, 2 or 1 will push 2.
- not : pops the top item on the stack and pushes its negation. Works on any value, but, because Python evaluates most values as true, the negation of anything that isn't a boolean will typically be false.
- loop

    - loop <int> : marks the beginning of a loop which will iterate <int> times. The value must be an integer, but it can be a variable assigned to an integer.
    - endloop : marks the end of the loop.
- if (<expression 1>, <expression 2>,...) : evaluates the expressions enclosed in parenthesis if the top item on the stack evaluates to true. This can lead to unintended effects, as Python evaluates most values as True. There must be a space between "if" and the parenthesis. You can nest if statements, e.g.
    - if (insert true, if (<some expression>))

Example programs and demonstrations:

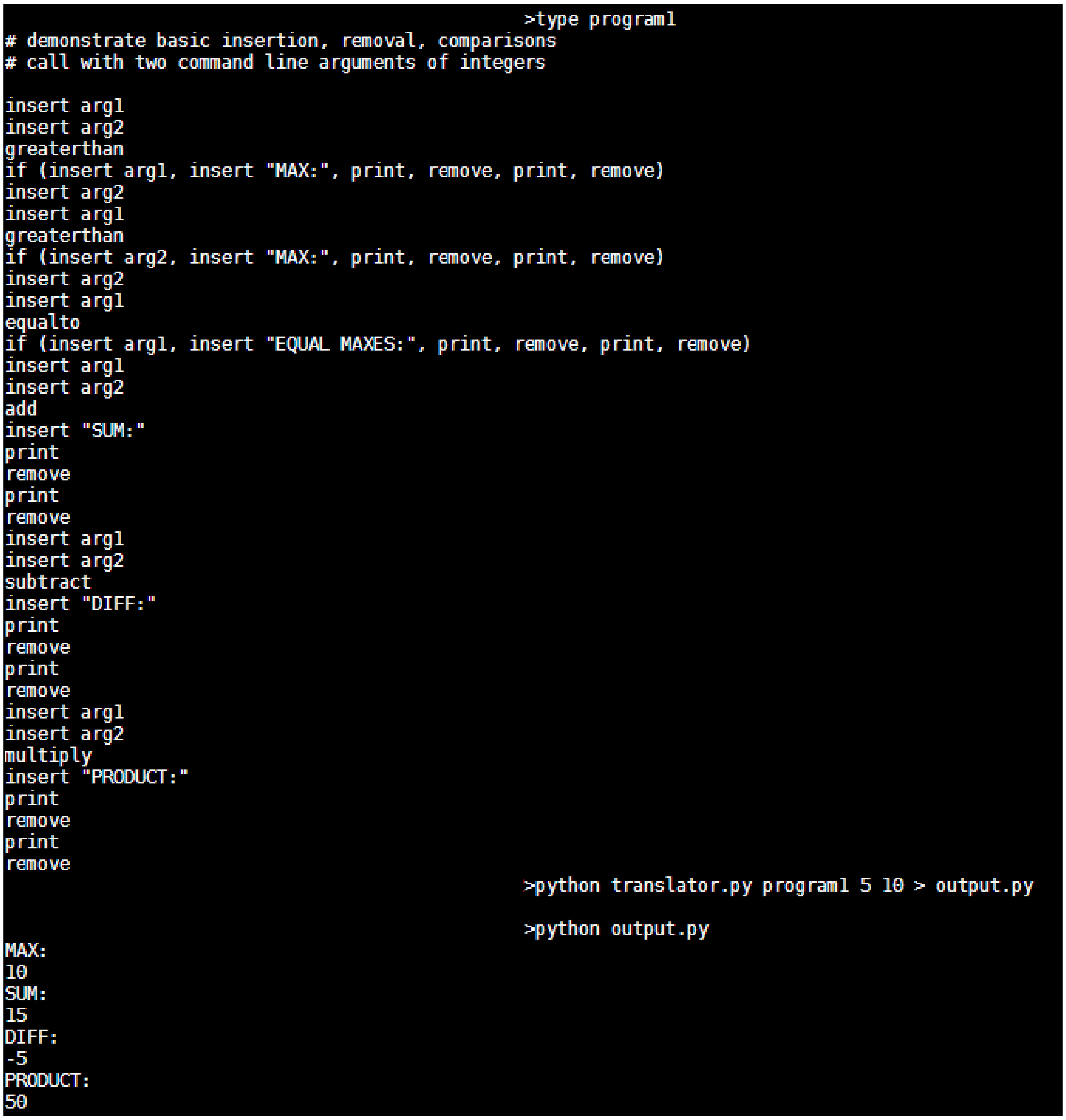
Live Mode:

Live mode is initiated by calling the translator with no arguments. In this mode, you can treat it as an interactive shell.

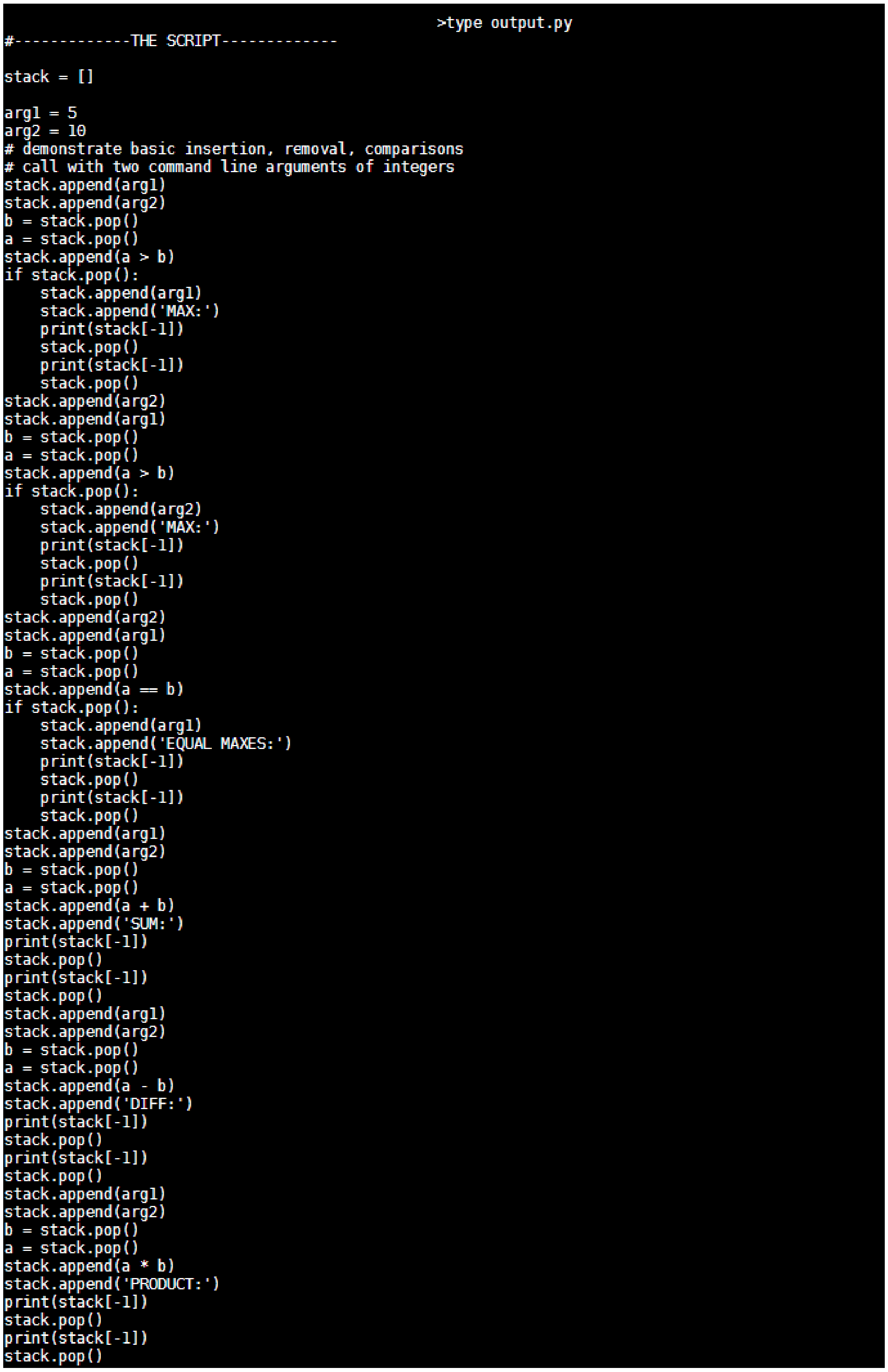
Tutorial2 - YouTube - a more thorough explanation.

A quick, simple example.

program1: a program that demonstrates basic insertion, removal, and comparison.

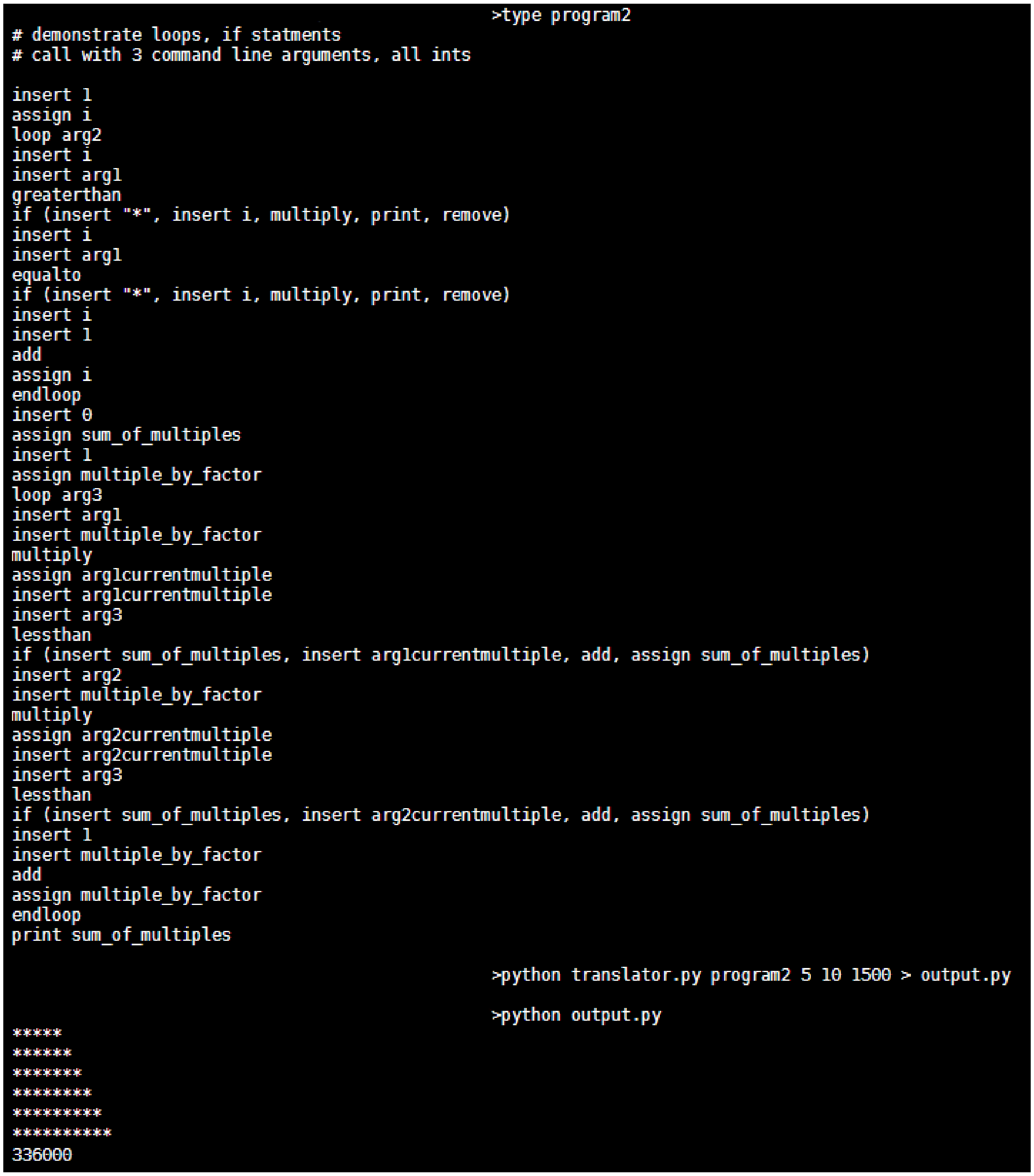


```
                                                >type program1
# demonstrate basic insertion, removal, comparisons
# call with two command line arguments of integers

insert arg1
insert arg2
greaterthan
if (insert arg1, insert "MAX:", print, remove, print, remove)
insert arg2
insert arg1
greaterthan
if (insert arg2, insert "MAX:", print, remove, print, remove)
insert arg2
insert arg1
equalto
if (insert arg1, insert "EQUAL MAXES:", print, remove, print, remove)
insert arg1
insert arg2
add
insert "SUM:"
print
remove
print
remove
insert arg1
insert arg2
subtract
insert "DIFF:"
print
remove
print
remove
insert arg1
insert arg2
multiply
insert "PRODUCT:"
print
remove
print
remove
                                                >python translator.py program1 5 10 > output.py

                                                >python output.py
MAX:
10
SUM:
15
DIFF:
-5
PRODUCT:
50
```

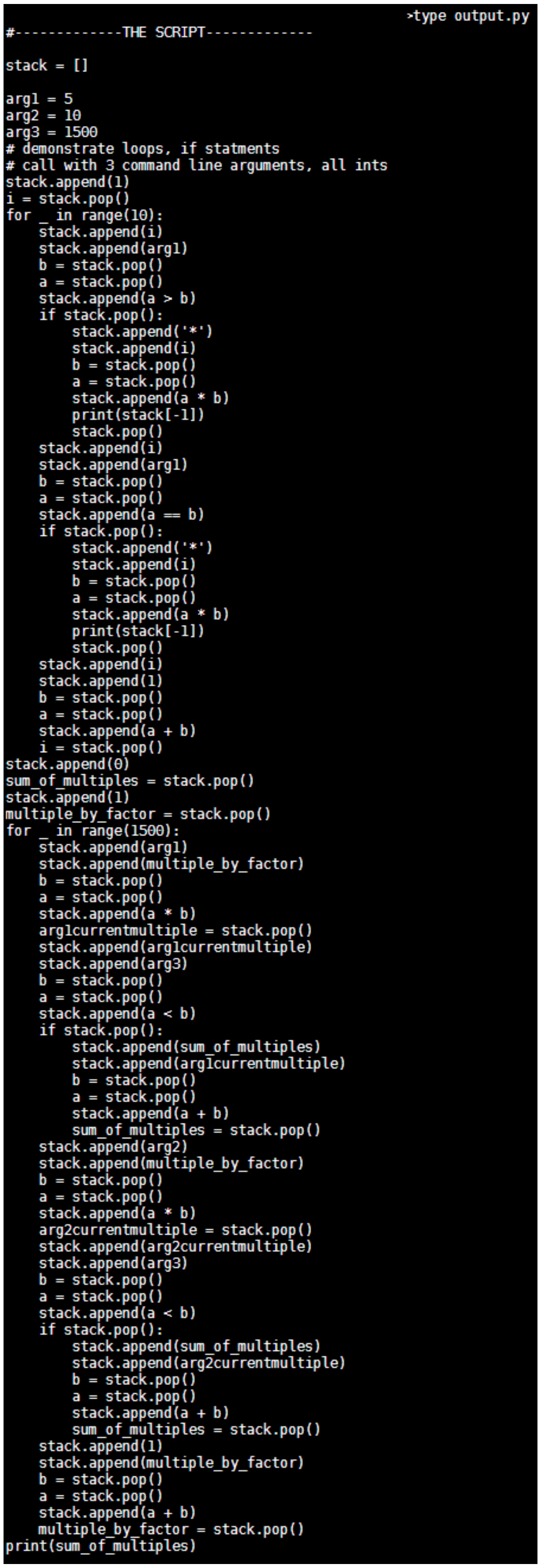


```
                                                >type output.py
#-------------THE SCRIPT-------------

stack = []

arg1 = 5
arg2 = 10
# demonstrate basic insertion, removal, comparisons
# call with two command line arguments of integers
stack.append(arg1)
stack.append(arg2)
b = stack.pop()
a = stack.pop()
stack.append(a > b)
if stack.pop():
    stack.append(arg1)
    stack.append('MAX:')
    print(stack[-1])
    stack.pop()
    print(stack[-1])
    stack.pop()
stack.append(arg2)
stack.append(arg1)
b = stack.pop()
a = stack.pop()
stack.append(a > b)
if stack.pop():
    stack.append(arg2)
    stack.append('MAX:')
    print(stack[-1])
    stack.pop()
    print(stack[-1])
    stack.pop()
stack.append(arg2)
stack.append(arg1)
b = stack.pop()
a = stack.pop()
stack.append(a == b)
if stack.pop():
    stack.append(arg1)
    stack.append('EQUAL MAXES:')
    print(stack[-1])
    stack.pop()
    print(stack[-1])
    stack.pop()
stack.append(arg1)
stack.append(arg2)
b = stack.pop()
a = stack.pop()
stack.append(a + b)
stack.append('SUM:')
print(stack[-1])
stack.pop()
print(stack[-1])
stack.pop()
stack.append(arg1)
stack.append(arg2)
b = stack.pop()
a = stack.pop()
stack.append(a - b)
stack.append('DIFF:')
print(stack[-1])
stack.pop()
print(stack[-1])
stack.pop()
stack.append(arg1)
stack.append(arg2)
b = stack.pop()
a = stack.pop()
stack.append(a * b)
stack.append('PRODUCT:')
print(stack[-1])
stack.pop()
print(stack[-1])
stack.pop()
```

program2: A program that demonstrates loops, if statements, and multiplication.

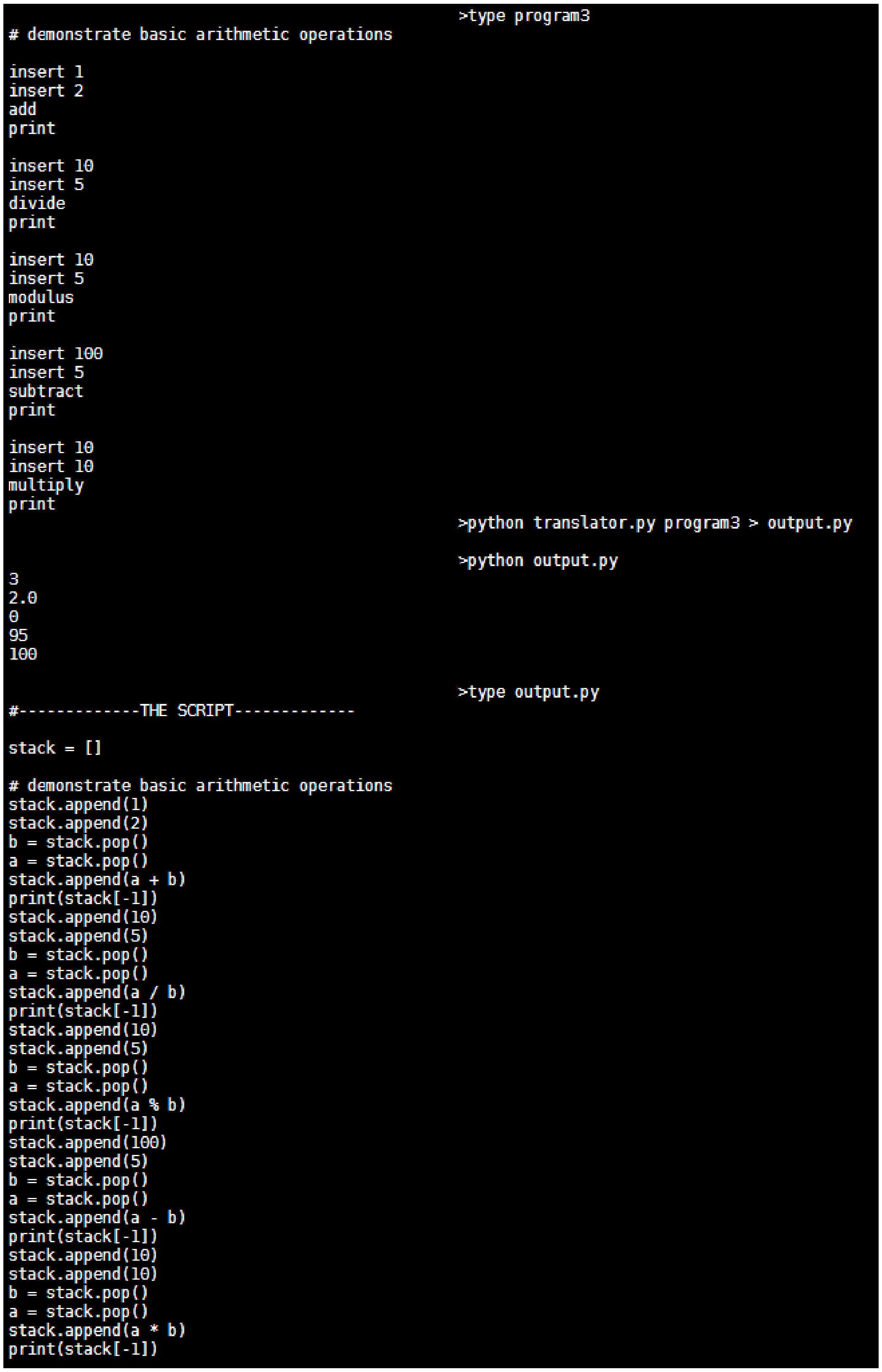


```
                                                >type program2
# demonstrate loops, if statments
# call with 3 command line arguments, all ints

insert 1
assign i
loop arg2
insert i
insert arg1
greaterthan
if (insert "*", insert i, multiply, print, remove)
insert i
insert arg1
equalto
if (insert "*", insert i, multiply, print, remove)
insert i
insert 1
add
assign i
endloop
insert 0
assign sum_of_multiples
insert 1
assign multiple_by_factor
loop arg3
insert arg1
insert multiple_by_factor
multiply
assign arg1currentmultiple
insert arg1currentmultiple
insert arg3
lessthan
if (insert sum_of_multiples, insert arg1currentmultiple, add, assign sum_of_multiples)
insert arg2
insert multiple_by_factor
multiply
assign arg2currentmultiple
insert arg2currentmultiple
insert arg3
lessthan
if (insert sum_of_multiples, insert arg2currentmultiple, add, assign sum_of_multiples)
insert 1
insert multiple_by_factor
add
assign multiple_by_factor
endloop
print sum_of_multiples

                                                >python translator.py program2 5 10 1500 > output.py

                                                >python output.py
*****
******
*******
********
*********
**********
336000
```



```
                                                >type output.py
#-------------THE SCRIPT-------------

stack = []

arg1 = 5
arg2 = 10
arg3 = 1500
# demonstrate loops, if statments
# call with 3 command line arguments, all ints
stack.append(1)
i = stack.pop()
for _ in range(10):
    stack.append(i)
    stack.append(arg1)
    b = stack.pop()
    a = stack.pop()
    stack.append(a > b)
    if stack.pop():
        stack.append('*')
        stack.append(i)
        b = stack.pop()
        a = stack.pop()
        stack.append(a * b)
        print(stack[-1])
        stack.pop()
    stack.append(i)
    stack.append(arg1)
    b = stack.pop()
    a = stack.pop()
    stack.append(a == b)
    if stack.pop():
        stack.append('*')
        stack.append(i)
        b = stack.pop()
        a = stack.pop()
        stack.append(a * b)
        print(stack[-1])
        stack.pop()
    stack.append(i)
    stack.append(1)
    b = stack.pop()
    a = stack.pop()
    stack.append(a + b)
    i = stack.pop()
stack.append(0)
sum_of_multiples = stack.pop()
stack.append(1)
multiple_by_factor = stack.pop()
for _ in range(1500):
    stack.append(arg1)
    stack.append(multiple_by_factor)
    b = stack.pop()
    a = stack.pop()
    stack.append(a * b)
    arg1currentmultiple = stack.pop()
    stack.append(arg1currentmultiple)
    stack.append(arg3)
    b = stack.pop()
    a = stack.pop()
    stack.append(a < b)
    if stack.pop():
        stack.append(sum_of_multiples)
        stack.append(arg1currentmultiple)
        b = stack.pop()
        a = stack.pop()
        stack.append(a + b)
        sum_of_multiples = stack.pop()
    stack.append(arg2)
    stack.append(multiple_by_factor)
    b = stack.pop()
    a = stack.pop()
    stack.append(a * b)
    arg2currentmultiple = stack.pop()
    stack.append(arg2currentmultiple)
    stack.append(arg3)
    b = stack.pop()
    a = stack.pop()
    stack.append(a < b)
    if stack.pop():
        stack.append(sum_of_multiples)
        stack.append(arg2currentmultiple)
        b = stack.pop()
        a = stack.pop()
        stack.append(a + b)
        sum_of_multiples = stack.pop()
    stack.append(1)
    stack.append(multiple_by_factor)
    b = stack.pop()
    a = stack.pop()
    stack.append(a + b)
    multiple_by_factor = stack.pop()
print(sum_of_multiples)
```

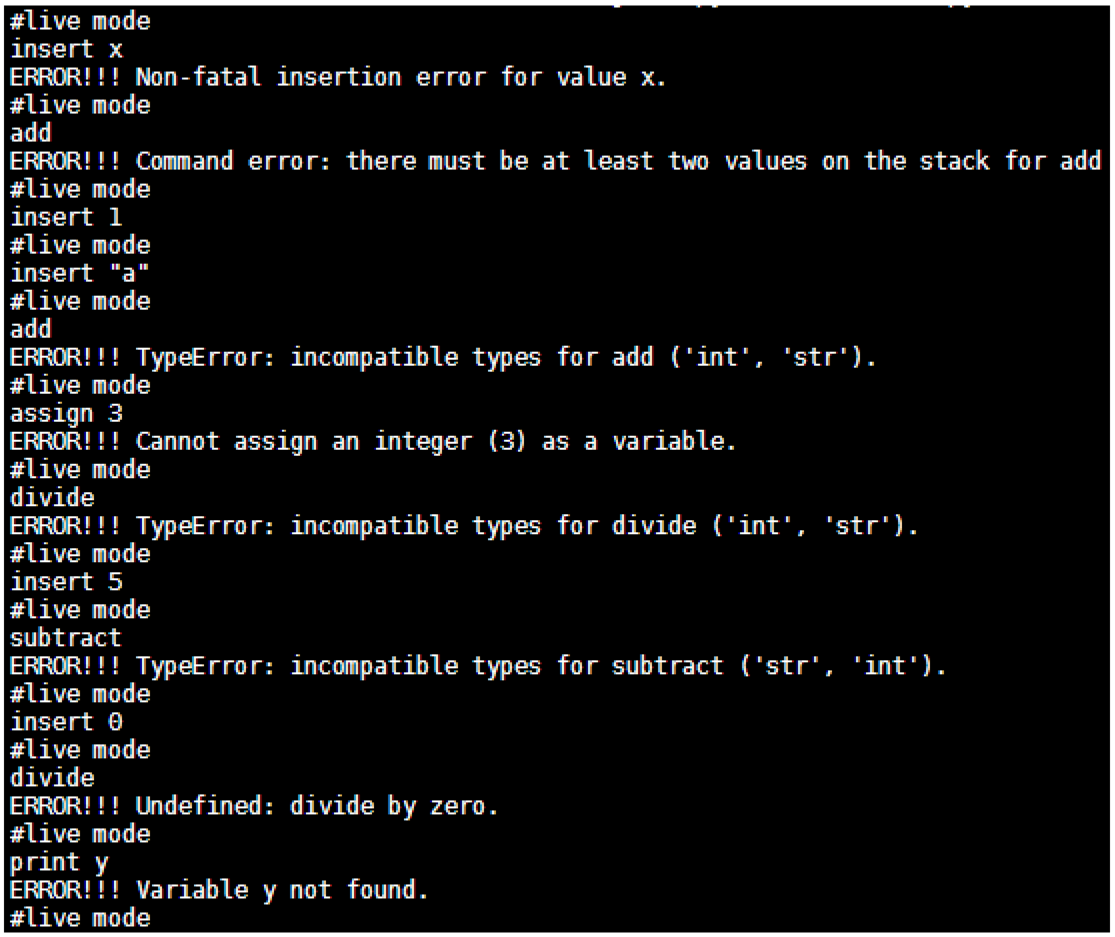
program3: A program that demonstrates basic arithmetic.



```
>type program3
# demonstrate basic arithmetic operations

insert 1
insert 2
add
print

insert 10
insert 5
divide
print

insert 10
insert 5
modulus
print

insert 100
insert 5
subtract
print

insert 10
insert 10
multiply
print
>python translator.py program3 > output.py

>python output.py
3
2.0
0
95
100

>type output.py
#-------------THE SCRIPT-------------

stack = []

# demonstrate basic arithmetic operations
stack.append(1)
stack.append(2)
b = stack.pop()
a = stack.pop()
stack.append(a + b)
print(stack[-1])
stack.append(10)
stack.append(5)
b = stack.pop()
a = stack.pop()
stack.append(a / b)
print(stack[-1])
stack.append(10)
stack.append(5)
b = stack.pop()
a = stack.pop()
stack.append(a % b)
print(stack[-1])
stack.append(100)
stack.append(5)
b = stack.pop()
a = stack.pop()
stack.append(a - b)
print(stack[-1])
stack.append(10)
stack.append(10)
b = stack.pop()
a = stack.pop()
stack.append(a * b)
print(stack[-1])
```

Additional features:

In live mode, the translator supports recognizing various errors: type errors, empty stacks, bad variable assignments, bad operations, etc



```
#live mode
insert x
ERROR!!! Non-fatal insertion error for value x.
#live mode
add
ERROR!!! Command error: there must be at least two values on the stack for add
#live mode
insert 1
#live mode
insert "a"
#live mode
add
ERROR!!! TypeError: incompatible types for add ('int', 'str').
#live mode
assign 3
ERROR!!! Cannot assign an integer (3) as a variable.
#live mode
divide
ERROR!!! TypeError: incompatible types for divide ('int', 'str').
#live mode
insert 5
#live mode
subtract
ERROR!!! TypeError: incompatible types for subtract ('str', 'int').
#live mode
insert 0
#live mode
divide
ERROR!!! Undefined: divide by zero.
#live mode
print y
ERROR!!! Variable y not found.
#live mode
```

.

Grammar:

```
<program> ::= <statements>

<statements> ::= <statement> | <statement> "\n" <statements>

<statement> ::= <empty> | <comment> | <command>

<empty> ::= /^$/

<comment> ::= /#.*$/

<command> ::= <insert-command> | <print-command> | <remove-command> |

<assign-command> |            <arithmetic-command> | <logical-command> | <if-command>

| <loop-command> | <endloop-command>

<insert-command> ::= "insert" <value>

<value> ::= <integer> | <boolean> | <string> | <variable>

<integer> ::= /\d+/

<boolean> ::= "true" | "false"

<string> ::= /"[^"]*"/

<variable> ::= /\w+/

<print-command> ::= "print" | "print" <variable>

<remove-command> ::= "remove"

<assign-command> ::= "assign" <variable>

<arithmetic-command> ::= "add" | "subtract" | "multiply" | "divide" | "modulus"

<logical-command> ::= "and" | "or" | "not"

<if-command> ::= "if ("<command-list>")"

<command-list> ::= <command> | <command> "," <command-list>

<loop-command> ::= "loop" <loop-count>

<loop-count> ::= <integer> | <variable>

<endloop-command> ::= "endloop"
```