Python Lab Assignment 4

# Program 1

## Code

pl = **lambda** l: l[1] - l[0] *# Profit / loss*

si = **lambda** l: l[0]\*l[1]\*l[2]/100

ci = **lambda** l: l[0]\*((1 + l[1]/100)\*\*l[2] - 1)

nameMap = {"pl":pl, "si":si, "ci":ci}

print("\nSIMPLE FINANCIAL TASKS\n")

print("Format:\nfunction;arg1;arg2...\n")

print("Functions:\npl;cp;sp\nsi;p;r;t\nci;p;r;t\nx to exit\n(Extra arguments will be ignored)\n")

while True:

inp = input(">> ").split(";")

if(inp[0] == "x"): break

try: print("=", nameMap[inp[0]](tuple(map(float, inp[1:]))))

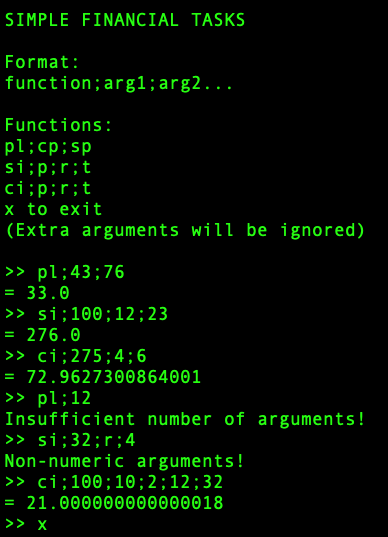
except KeyError: print("Function not found!") *# Wrong function name*

except NameError: print("Extraneous spaces in function name!") *# Extra spaces around the function name*

except ValueError: print("Non-numeric arguments!") *# Non-numeric arguments*

except IndexError:print("Insufficient number of arguments!") *# Some index of inp would be out of bounds*

## Output



## Inferences

The result of the expression in the lambda function is the return value of the function, unlike a regular function, where return values have to be specified, if they are needed at all.

In this program, I used a code that catches one of multiple different exceptions when using the dictionary to use any given function. Here, each exception implies a different error. This makes validation much quicker and more concise, and makes specific error messages much easier to produce.

# Program 2

## Code (shapeClasses.py) (main module)

from abc import ABC, abstractmethod

**class** shape(ABC):

*# Surface area and volume abstract methods...*

@abstractmethod

**def** sa(self): pass

@abstractmethod

**def** vol(self): pass

*# Value input function...*

**def** inp(self, p):

output = p

if input("Input parameters? (n to deny): ") == "n": return output

for i in p.keys():

try: output[i] = float(input("{0}: ".format(i)))

except: output[i] = p[i]

return output

*# Formula calculation methods...*

*# 1. Simple calculations*

**def** calc(self, op, a, b):

try: return {"+":a+b,"-":a-b,"\*":a\*b}[op] *# Unproblematic operations*

except KeyError:

"""

Possible problems in power and division...

# Power may be exceedingly high

# Possible division by zero

"""

try: return {"^":a\*\*b}[op]

except:

try: return {"/":a/b}[op]

except: return "NaN"

*# 2. Precedence of operators and brackets*

**def** precedence(self, op):

try: return {"(":0, "+":1, "-":1, "\*":2, "/":2, "^":3}[op]

except KeyError: return -1 *# For validation purposes*

*# 3. Conversion from infix expression to postfix*

**def** infixToPostfix(self, exp, p):

*# Necessary variables, structures and modifications...*

p["pi"], exp = 3.14159, exp + ");"

postfix, stack, tmp, i = [], ["("], str(), 0

*# Conversion process...*

while exp[i] != ";":

*# To store full numbers...*

if exp[i].isnumeric() or exp[i] == ".":

tmp, i = exp[i], i + 1

while exp[i].isnumeric() or exp[i] == ".":

tmp, i = tmp + exp[i], i + 1

try: p[tmp] = float(tmp) *# To ensure the numeric string is mapped with its value*

except: p[tmp] = tmp *# In case of conversion error*

postfix.append(tmp)

*# To store full variable names...*

if exp[i].isalpha():

tmp, i = exp[i], i + 1

while exp[i].isalpha():

tmp, i = tmp + exp[i], i + 1

postfix.append(tmp)

*# To store operators...*

if exp[i] == "(":

stack.append(exp[i])

elif exp[i] == ")":

while stack[-1] != "(":

postfix.append(stack.pop())

stack.pop() *# Removing the left bracket*

else: *# Here, operator should be encountered...*

while self.precedence(stack[-1]) >= self.precedence(exp[i]):

postfix.append(stack.pop())

stack.append(exp[i])

i = i + 1

return (p, postfix)

*# 4. Evaluation of formula*

**def** formula(self, exp, p):

(p, postfix), i = self.infixToPostfix(exp, p), 0

while i < len(postfix):

if self.precedence(postfix[i]) == -1:

postfix[i], i = p[postfix[i]], i + 1

else:

tmp = self.calc(postfix[i], postfix[i - 2], postfix[i - 1])

if tmp == "NaN": return tmp

del(postfix[i], postfix[i - 1], postfix[i - 2])

postfix.insert(i - 2, tmp)

i = i - 1

return postfix[i - 1]

**class** sphere(shape):

p, \_sa, \_vol = {"r":0}, "4\*pi\*r^2", "(4/3)\*pi\*r^3"

**def** \_\_init\_\_(self): self.p = super().inp(self.p)

**def** sa(self): return super().formula(self.\_sa, dict(self.p))

**def** vol(self): return super().formula(self.\_vol, dict(self.p))

**class** cylinder(shape):

p, \_sa, \_vol = {"r":0, "h":0}, "2\*pi\*r\*(r+h)", "pi\*(r^2)\*h"

**def** \_\_init\_\_(self): self.p = super().inp(self.p)

**def** sa(self): return super().formula(self.\_sa, dict(self.p))

**def** vol(self): return super().formula(self.\_vol, dict(self.p))

**class** cone(shape):

p, \_sa, \_vol = {"r":0, "h":0, "s":0}, "pi\*r\*(s+r)", "(1/3)\*pi\*(r^2)\*h"

**def** \_\_init\_\_(self): self.p = super().inp(self.p)

**def** sa(self): return super().formula(self.\_sa, dict(self.p))

**def** vol(self): return super().formula(self.\_vol, dict(self.p))

**class** rectangularPrism(shape):

p, \_sa, \_vol = {"l":0, "w":0, "h":0}, "2\*(l\*w+w\*h+h\*l)", "l\*w\*h"

**def** \_\_init\_\_(self): self.p = super().inp(self.p)

**def** sa(self): return super().formula(self.\_sa, dict(self.p))

**def** vol(self): return super().formula(self.\_vol, dict(self.p))

**class** triangularPrism(shape):

p, \_sa, \_vol = {"b":0, "l":0, "w":0, "h":0, "s":0}, "b\*h+2\*l\*s+l\*b", "(1/2)\*b\*l\*h"

**def** \_\_init\_\_(self): self.p = super().inp(self.p)

**def** sa(self): return super().formula(self.\_sa, dict(self.p))

**def** vol(self): return super().formula(self.\_vol, dict(self.p))

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

print("\nContains class definitions for various shapes.\n")

## Code (shapeHandlingFunctions.py) (secondary module)

*# MODULE WITH REQUIRED CLASSES*

from shapeClasses import \*

*# DICTIONARY OF SHAPES*

shapes = {

"sphere":sphere,

"cylinder":cylinder,

"cone":cone,

"r-prism":rectangularPrism,

"t-prism":triangularPrism}

*# FUNCTIONS*

**def** printShapes():

print("------------\nAVAILABLE SHAPES")

for i in shapes.items():

print(i)

**def** createShape():

print("------------\nCREATE SHAPE")

x = input(">> ")

try: x = shapes[x]()

except KeyError:

if x == "?": printShapes()

else: print("Shape not found!")

else: print("Shape created.")

return x

**def** shapeFormulae(x):

try:

print("------------\nSHAPE FORMULAE")

print("Surface area:", x.\_sa)

print("Volume:", x.\_vol)

except: print("No shape created yet!")

**def** printFunctions():

print("------------\nAVAILABLE SHAPE FUNCTIONS")

print(">> sa for surface area")

print(">> vol for volume")

**def** shapeFunction(x):

print("------------\nSHAPE FUNCTIONS")

try:

functions = {"sa":x.sa, "vol":x.vol}

*# The above immediately checks if the shape is created*

inp = input(">> ")

try: print(functions[inp]())

except KeyError:

if inp == "?": printFunctions()

else: print("Function not found!")

except: print("No shape created yet!")

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

print("\nContains extra functions for handling shape objects created in 'shapeClasses.py'.\n")

## Code (funWithShapes.py) (main program)

*# MODULE CONTAINING REQUIRED FUNCTIONS*

from shapeHandlingFunctions import \*

*# HELP FUNCTION*

**def** help():

print("------------\nAVAILABLE OPTIONS")

print(">> create makes a new shape object.")

print(">> formulae prints all formulae for the shape.")

print(">> fun allows application of function for the shape.")

print(">> ? shows available options in any section.")

print(">> x enables you to exit from main code.")

*# DICTIONARY OF FUNCTIONS*

argOptions = {

"formulae":shapeFormulae,

"fun":shapeFunction}

noArgOptions = {

"?":help,

"x":exit}

*# MAIN FUNCTION*

x = 0

print("========================")

print("FUN WITH SHAPES\n(Enter '?' for help)")

while True:

print("------------\nMAIN CODE")

option = input(">> ")

if option == "create": x = createShape()

else:

try: argOptions[option](x)

except KeyError:

try: noArgOptions[option]()

except KeyError: print("Option unavailable!")

## Inferences

**DUPLICATING DICTIONARY FOR PASSING AS ARGUMENT**

In "shapeClasses.py", in every class constructor for every child of "shape" class, we pass dict(self.p) and not self.p, because the dictionary of the particular shape's parameters is being modified in the function, and since it is a collection, even if it is passed as an argument, the copy of the identifier is merely another name for the same memory addresses i.e the same dictionary internally. Hence, we create a new dictionary object with the same values using dict(self.p), and pass that to the function so that the original dictionary remains untouched.

**WHY WE NEED NOT DO THE SAME FOR STRINGS**

Technically, a string are also a collection (of characters). However, a string is also immutable. Hence, we need not worry about unwillingly changing the original string. This also applies to other immutable collections such as tuples and frozen sets.

**EMULATING SWITCH-CASE USING DICTIONARIES**

Python does not have an in-build switch-case mechanism. However, we can use dictionaries, where the keys serve as cases, and the values serve as case-related code. Now, the values in a dictionary are usually objects. However, as done in this program, we can put function names in the value slot, and call the function through the dictionary.

This, however, has limitations. For example, for a basic calculator function, I could use a dictionary, where every operation symbol is associated with the respective operation on the arguments. However, if one operation throws an exception, such as division by zero, then the entire dictionary becomes unusable.