CSGE601020 Dasar-Dasar Pemrograman 1

(Foundations of Programming 1)

Programming Assignment 4

LAST DAY for uploading the result of your work to SCeLE: Friday 11 Dec 2017 (11:55 PM).
Don't forget to write enough comments in your Python source code.
Please contact the TA (teaching assistant) for giving a demo of your work as soon as possible, not later than Saturday 23 Dec 2017. The TA will give you a mark after the demo.

Please start working on this assignment immediately.
If you have any questions, please ask the TA or the professor.

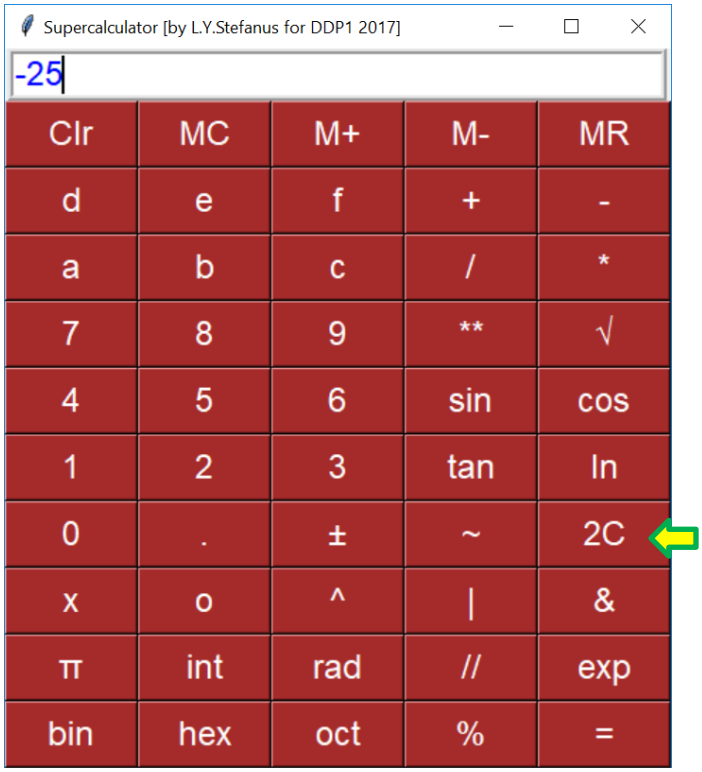
Marking scheme:
60 % correctness
30 % explanation in demo session
10 % program documentation (comments, neatness)

Task Description

- In this assignment, you are asked to write a GUI-based event-driven Python program that works as a supercalculator. Your program should be object-oriented.
- A supercalculator can perform many tasks in addition to the ordinary tasks of the common calculator.
- Your calculator should be able to:
  - ✓ Enter input including binary number (0b...), octal number (0o...), hexadecimal number (0x...), and of course decimal number (without prefix).
  - ✓ Perform conversion between the number representations.
  - ✓ Compute various functions including: add, subtract, multiply, divide, power, square root, mod, trigonometric functions, logarithm, bitwise xor, bitwise or, bitwise and, bitwise complement, 32-bit 2's complement representation of an integer, memory functions, and sign toggle.
- Your program should handle exceptions gracefully.
- By default, the results are displayed in decimal representation.
- When the user hovers the mouse cursor over a button, a tooltip will be displayed to give information about the button. A tooltip is a GUI element in the form of a pop-up text box which appears when the mouse pointer is hovering over a widget.
- Your program can only import the modules: tkinter, math, and idlelib.tooltip.
- The built-in function format( ) will be useful.
- At the end of this document, some help is provided.


Supercalculator [by L.Y.Stefanus for DDP1 2017]
0xface2017
Clr
MC
M+
M-
MR
d
e
f
+
-
a
b
c
/
*
7
8
9
**
√
4
5
6
sin
cos
1
2
3
tan
ln
0
.
±
~
2C
x
o
^
|
&
π
int
rad
//
exp
bin
hex
oct
%
=

Example of a tooltip:



Happy programming! 'Met ngoding!

L. Y. Stefanus

Help for you to get started:

[Adapted from: Ljubomir Perkovic. Introduction to Computing Using Python: An Application Development Focus. 2nd Edition. Wiley, 2015.]

The Calculator Buttons and Passing Arguments to Handlers

The following program fragment creates the 24 buttons of the calculator.

```
1  # calculator button labels in a 2D list
2  buttons = [['MC',     'M+',      'M-', 'MR'],
3             ['C' , '\u221a', 'x\u00b2', '+' ],
4             ['7' ,     '8' ,      '9' , '-' ],
5             ['4' ,     '5' ,      '6' , '*' ],
6             ['1' ,     '2' ,      '3' , '/' ],
7             ['0' ,     '.' ,      '+-', '=' ]]
8
9  # create and place buttons in appropriate row and column
10 for r in range(6):
11     for c in range(4):
12         b = Button(self,       # button for symbol buttons[r][c]
13                    text=buttons[r][c],
14                    width=3,
15                    relief=RAISED,
16                    command=???)          # method ??? to be done
17         b.grid(row = r+1, column = c)    # entry is in row 0
```

What's missing in this code is the name of each event-handling function (note the question marks ??? in line 16). With 24 different buttons, we need to have 24 different event handlers. Writing 24 different handlers would not only be very painful, but it would also be quite repetitive since many of them are essentially the same. For example, the 10 handlers for the 10 "digit" buttons should all do essentially the same thing: append the appropriate digit to the string in the entry field.

Wouldn't it be nicer if we could write just one event handler called click() for all 24 buttons? This handler would take one input argument, the label of the clicked button, and then handle the button click depending on what the label is. The problem is that a button event handler cannot take an input argument. In other words, the command option in the Button constructor must refer to a function that can and will be called without arguments. So how?

There is actually a solution to the problem, and it uses the fact that Python functions can be defined so that when called without an input value, the input argument receives a default value. Instead of having function click() be the official handler, we define, inside the nested for loop, the handler to be a function cmd() that takes one input argument x—which defaults to the label buttons[r][c]—and calls self.click(x). The next module includes this approach (and the code that creates the Entry widget):

```
# use Entry widget for display
self.entry = Entry(self, relief=RIDGE, borderwidth=3,
                   width=20, bg='gray',
                   font=('Helvetica', 18))
self.entry.grid(row=0, column=0, columnspan=5)

# create and place buttons in appropriate row and column
for r in range(6):
    for c in range(4):

        # function cmd() is defined so that when it is
        # called without an input argument, it executes
        # self.click(buttons[r][c])
        def cmd(x=buttons[r][c]):
            self.click(x)

        b = Button(self,       # button for symbol buttons[r][c]
                   text=buttons[r][c],
                   width=3,
                   relief=RAISED,
                   command=cmd)          # cmd() is the handler
        b.grid(row=r+1, column=c)        # entry is in row 0
```

In every iteration of the innermost for loop, a new function cmd is defined. It is defined so that when called without an input value, it executes self.click(buttons[r][c]). The label buttons[r][c] is the label of the button being created in the same iteration.The button constructor will set cmd() to be the button's event handler.

In summary, when the calculator button with label key is clicked, the Python interpreter will execute self.click(key). To complete the calculator, we need only to implement the event handler click().

Implementing the Event Handler click()

The function click() actually handles every button click. It takes the text label key of the clicked button as input and, depending on what the button label is, does one of several things. If key is one of the digits 0 through 9 or the dot, then key should simply be appended to the digits already in the Entry widget: self.entry.insert(END, key)

If key is one of the operators +, -, *, or /, it means that we just finished typing an operand, which is displayed in the entry widget, and are about to start typing the next operand. To handle this, we use an instance variable self.expr that will store the expression typed so far, as a string. This means that we need to append the operand currently displayed in the entry box and also the operator key:

```
self.expr += self.entry.get()
self.expr += key
```

In addition, we need to somehow indicate that the next digit typed is the start of the next operand and should not be appended to the current value in the Entry widget. We do this by setting a flag:

```
self.startOfNextOperand = True
```

This means that we need to rethink what needs to be done when key is one of the digits 0 through 9. If startOfNextOperand is True, we need to first delete the operand currently displayed in the entry and reset the flag to False:

```
if self.startOfNextOperand:
        self.entry.delete(0, END)
        self.startOfNextOperand = False
self.entry.insert(END, key)
```

What should be done if key is =? The expression typed so far should be evaluated and displayed in the entry. The expression consists of everything stored in self.expr and the operand currently in the entry. Before displaying the result of the evaluation, the operand currently in the entry should be deleted. Because the user may have typed an illegal expression, we need to do all this inside a try block; the exception handler will display an error message if an exception is raised while evaluating the expression.

We can now implement a part of the click() function:

```
def click(self, key):
    'handler for event of pressing button labeled key'

    if key == '=':
        # evaluate the expression, including the value
        # displayed in entry and display result
        try:
            result = eval(self.expr + self.entry.get())
            self.entry.delete(0, END)
            self.entry.insert(END, result)
            self.expr = ''
        except:
            self.entry.delete(0, END)
            self.entry.insert(END, 'Error')

    elif key in '+*-/':
        # add operand displayed in entry and operator key
        # to expression and prepare for next operand
        self.expr += self.entry.get()
        self.expr += key
        self.startOfNextOperand = True
```

```
22      # the cases when key is '\u221a', 'x\u00b2', 'C',
23      # 'M+', 'M-', 'MR', 'MC' are left as an exercise
24
25      elif key == '+-':
26          # switch entry from positive to negative or vice versa
27          # if there is no value in entry, do nothing
28          try:
29              if self.entry.get()[0] == '-':
30                  self.entry.delete(0)
31              else:
32                  self.entry.insert(0, '-')
33          except IndexError:
34              pass
35
36      else:
37          # insert digit at end of entry, or as the first
38          # figit if start of next operand
39          if self.startOfNextOperand:
40              self.entry.delete(0, END)
41              self.startOfNextOperand = False
42          self.entry.insert(END, key)
```

Note that the case when the user types the +- button is also shown. Each press of this button should either insert a - operator in front of the operand in the entry if it is positive, or remove the - operator if it is negative.

Lastly, we implement the constructor. We have already written the code that creates the entry and the buttons. Instance variables self.expr and self.startOfNextOperand should also be initialized there. In addition, we should initialize an instance variable that will represent the calculator's memory.

```
def __init__(self):
    'calculator constructor'
      .
      .
      .
    self.memory = ''  # memory
    self.expr = ''    # current expression
    self.startOfNextOperand = True    # start of new operand

      . # entry and buttons code
      .
      .
```

Now, you have enough to get going and get things done! Allez! Allez!