《用栈实现计算器》

一、实验题目

用栈实现计算器

二、实验目的：

1、掌握栈的基本操作：插入、删除、查找等运算。

2、掌握栈的存储特点及其实现。

三、实验设备与环境

微型计算机、Windows 系列操作系统 、Visual Studio、Pycharm 系列软件

四、实验内容

利用栈实现一个科学计算器。

五、概要设计（思路、算法、步骤等）

5.1 计算器项目设计思路

5.1.1 项目的初步实现

可视化的实现

针对可视化部分，首先根据实验指导，安装 pyqt5、sip 和 pyqt5-tools 包，然后在 QtDesigner 程序中构建出计算器的初始样式，将其利用工具转化为 python 文件，然后编写 main 文件联合刚刚转化完的文件实现计算器可视化操作。

逻辑功能的实现

针对逻辑实现部分，我考虑先建立一个栈类，栈类的初始化利用常量类的常量进行初始化，栈要能实现从顶部堆入元素、从顶部移出元素、计算堆入元素的数量、返回顶部元素、判断栈是否为空是否为满的操作。

对于一个算数表达式，我们考虑先将其顺序放到一个栈中，称其为中缀栈，然后将中缀栈转化为后缀栈并除去栈中的括号，这实现了中缀表达式转化为后缀表达式的操作，最后利用后缀栈计算算数结果，得到运算答案。

在中缀栈转化成后缀栈和利用后缀栈计算结果的操作中，都需利用中间栈来实现。中缀栈转化为后缀栈操作中，利用运算符中间栈将运算符先寄存到运算符中间栈中，然后根据运算符优先级依次输出到后缀栈中；计算后缀栈结果则是用到了数字栈，将数字堆入栈中，然后再分别根据各个运算符计算对应结果，最后栈中会只剩下一个数字，那便是最终结果。

由此我们便实现了计算器的基本逻辑结构，但是还存在一些非法运算表达式的问题，比如说输入是一个左括号没有对应的右括号时会输出一个 none，这是括号方面的匹配问题，我们考虑设计一个函数判断括号合法性，再设计一个括号异常类，在括号不合法的时候抛出括号异常，完成括号不匹配的非法表达式问题。由此，便可完成计算器的初步实现。

5.1.2 开方运算与幂运算的添入

由于真实计算器还有很多操作，这边我考虑对简易计算器进行创新，添入开方运算操作和幂运算操作。

针对幂运算操作，我们只需将幂运算符“^”的优先级定义高于加减乘除，

其余操作类似加减乘除即可。

针对开方运算，我们考虑将开方运算符设置为“sqrt()”,每个开方运算自带一个括号,否则将会抛出括号异常,对于堆入栈的操作,我们利用中间字符串，将其整体堆入栈中，然后进行对应的栈操作来完成。

5.1.3 计算器 CSS 样式的设计

由于完成的计算器还不够美观,我们考虑加入 CSS 样式对计算器进行样式修改，让其成为一个美观、令人舒适的计算器，直接在 QtDesigner 中编写控件样式便可完成操作。

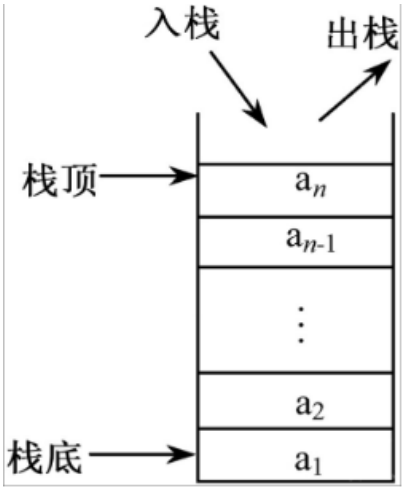
5.1.4 exe 可执行文件的生成与图标修改

为了让我的计算器看起来更像是一个软件,我考虑将其打包成 exe 可执行文件，并利用 ico 文件修改 exe 文件的图标样式，使计算器可以真正的成为软件供用户使用。

5.2 计算器项目涉及算法

5.2.1 栈的定义

栈(stack)是限定仅在表尾进行插入或者删除的线性表。对于栈来说，表尾端称为栈顶（top），表头端称为栈低（bottom）。不含元素的空表称为空栈。因为栈限定在表尾进行插入或者删除，所以栈又被称为后进先出的线性表（简称 LIFO:Last in, First out.结构）。



5.2.2 中缀到后缀的转换算法

当读到一个操作数的时候，立即把它放到输出中。操作符不立即输出，所以必须先存在某个地方。正确的做法是将已经见到过的操作符放进栈中而不是输出中。当遇到左圆括号时我们也要将其推入栈中。我们从一个空栈开始计算。

如果见到一个右括号，那么就将栈元素弹出，将弹出的符号写出直到我们遇见一个（对应的）左括号，但是这个左括号只被弹出，并不输出。

如果见到任何其他的符号（“+”、“-”、“×”、“÷”、“sqrt”、“^”、“(”），那么我们从栈中弹出栈元素直到发现优先级更低的元素为止。有一个例外：除非是在处理一个“)”的时候，否则我们绝不从栈中移走“(”。对于这种操作，“+”的优先级最低，而“(”的优先级最高。当从栈中弹出元素的工作完成后，我们再将操作符压入栈中。

最后，如果读到输出的末尾，我们将栈元素弹出直到该栈变成空栈，将符号

写到输出中。

5.2.3 计算后缀栈的算法

我们将中缀栈转化为后缀栈中之后，即可利用一个数字栈来计算求值。当遇到一个数字的时候就将其推入栈中；在遇到一个运算符时，该运算符就作用于从该栈弹出的两个数上（不同的：在我的计算器的 sqrt 运算符中仅弹出 1 个数进行操作），将所得结果推入栈中，最后当栈只剩下 1 个数字时，那便是最终计算的结果。

5.3 计算器项目具体实现步骤

5.3.1 项目的初步实现

可视化的实现

首先，根据实验指导，安装 pyqt5、sip 和 pyqt5-tools 包，但是我的 python 版本是 3.10 的，而我在下载了 pyqt5 和 sip 后下载安装 pyqt5-tools 会报错，查了很多资料之后发现是我的 python 版本太高的缘故导致安装的 pyqt5 和 pyqt5-tools 不匹配，于是我再次查资料发现在本实验中，我们仅需用到 pyqt5-tools 中的 QtDesigner 程序,于是考虑下载替代包 pyqt5Designer，最终成功配置完成实验环境。

然后在 QtDesigner 程序中构建出计算器的初始样式，并修改每个控件的 text 值和对象名，便于后续代码编写使用，然后将计算器的显示框设定为不能编辑和靠右显示，最后将这个 UI 文件保存到计算器主程序目录下，对这个 UI 文件使用 PyUIC 工具，将其转化为 python 文件，成功完成初步可视化。

逻辑功能实现

先根据实验指导完成 main.py 文件的编写，不同的是，我构建了一个 self.flag 属性用于重输入数字时对答案的清空，其余照样画葫芦进行编写。

针对运算逻辑实现部分，我先建立了一个 stack 栈类，栈类的初始化利用常量类的常量进行初始化，栈实现了从顶部堆入元素、从顶部移出元素、计算堆入元素的数量、返回顶部元素、判断栈是否为空是否为满的操作。

然后我设计了三个针对栈操作的函数，一个是将一段字符串算数表达式转化为中缀栈的函数，另一个是将一个中缀栈转化为后缀栈的函数，最后一个是计算后缀栈运算结果的函数。第一个函数的实现，我们利用栈直接推入即可；后面两个函数的实现，我们利用 5.2 中给出的算法进行操作即可实现。

由此我们便实现了计算器的基本逻辑结构，但是还存在一些非法运算表达式的问题，比如说输入是一个左括号没有对应的右括号时会输出一个 none，这是括号方面的匹配问题，我们考虑设计一个函数判断括号合法性，该函数也利用栈来操作匹配括号，然后再设计一个括号异常类，在括号不合法的时候抛出括号异常，完成括号不匹配的非法表达式问题。由此，我便实现了计算器的雏形。

5.3.2 开方运算与幂运算的添入

由于真实计算器还有很多操作，于是我对简易计算器进行创新，添入开方运算操作和幂运算操作。

针对幂运算操作，我们只需在优先级函数中将幂运算符“^”的优先级定义高于加减乘除，其余操作类似加减乘除即可。

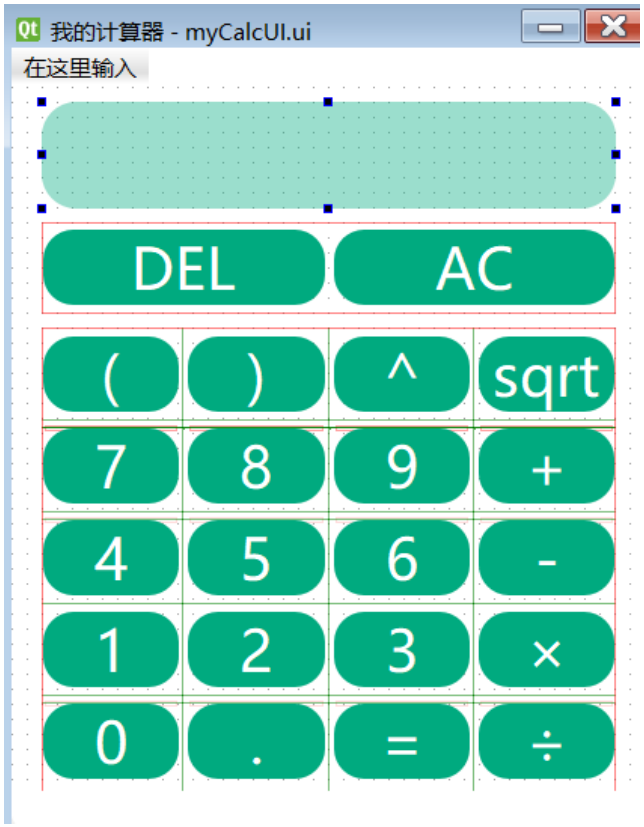
针对开方运算，我们将开方运算符设置为“sqrt()”,每个开方运算自带一个括号，否则将会抛出括号异常。对于堆入栈的操作，我们利用中间字符串，将其整体堆入栈中，将“sqrt”运算符的优先级与“^”定义成一致，都比加减乘

除高，然后进行类似的栈操作来完成，不同的是对于 sqrt 运算，我们在计算后缀栈结果时只需从栈中弹出 1 个数进行 sqrt 运算即可。

5.3.3 计算器 CSS 样式的设计

由于完成的计算器还不够美观，我们考虑加入 CSS 样式对计算器进行样式修改，让其成为一个美观、令人舒适的计算器。

我在 QtDesigner 中编写各控件样式完成了下图所示的计算器样式：



其中按钮样式设计如下：

```
background-color:rgb(0, 170, 127);
font-size:46px;
color:rgb(255, 255, 255);
font-weight:bold;
border-radius: 20px;
font: 23pt "微软雅黑";
```

文本框样式设计如下：

```
font-size:56px;
font-weight:bold;
color:rgb(255, 255, 255);
letter-spacing:10px;
outline-color:rgb(255, 255, 255);
border-radius:23px;
font: 20pt "微软雅黑";
background-color:rgba(0, 170, 127, 100)
```

5.3.4 exe 可执行文件的生成与图标修改

为了让我的计算器看起来更像是一个软件，我将其打包成 exe 可执行文件，并利用 ico 文件修改 exe 文件的图标样式，使计算器可以真正的成为软件供用户使用。具体操作步骤如下：

1、安装打包所需要的包（pyinstaller）
在终端窗口中输入 pip install pyinstaller 即可完成下载安装。

2、修改路径
转到计算器项目目录下。

3、寻找一个计算器 jpg 图片将其转化为 ico 文件
在下面链接的网站上即可完成 jpg 图片转化为 ico 文件，并下载到电脑。
下载完成后我们将其放到计算器项目目录下。
制作 ico 图标 | 在线 ico 图标转换工具 方便制作 favicon.ico - 比特虫 - Bitbug.net

4、输入打包命令
在计算器项目目录下输入 pyinstaller -F -w -i calc.ico main.py 即可完成计算器项目的打包。（其中 calc.ico 是刚刚生成的 ico 文件）

5、修改 exe 文件的名称为“我的计算器”。

最后生成的 exe 文件如下图所示：

六、详细设计（核心代码、算法流程图等）

6.1 计算器项目核心代码

6.1.1 常量的设置

文件 const.py 中

```
import sys

class _const:
    class ConstError(TypeError):
        pass

    def __setattr__(self, name, value):
        if name in self.__dict__:
            raise self.ConstError("Can't rebind const (%s)" % name)
        self.__dict__[name] = value
sys.modules[__name__] = _const()
```

文件 stack.py 中

```
import const

const.EmptyTOS = -1
const.MinStackSize = 5
```

6.1.2 stack 栈的定义

文件 stack.py 中

```
class stack(object):
    def __init__(self):
        self.top = const.EmptyTOS
        self.cap = const.MinStackSize
        self.listarray = []

    # 判断栈是否为空(bool)
    def isEmpty(self):
        return self.top == const.EmptyTOS

    # 判断栈是否满了(bool)
    def isFull(self):
        return self.top + 1 == self.cap

    # push 操作,向栈内堆入元素
    def push(self, item):
        if self.isFull():
            self.cap = self.cap * 2
        self.top += 1
        self.listarray.append(item)

    # pop 操作,将栈顶元素移除
    def pop(self):
        self.listarray.pop()
        self.top -= 1

    # peek 操作,取出栈顶元素
    def peek(self):
        if not self.isEmpty():
            return self.listarray[self.top]

    # 返回栈内数据总量(int)
    def size(self):
        return len(self.listarray)
```

6.1.3 字符串算数表达式存入中缀栈

文件 stack.py 中

```
# 将运算表达式放入中缀表达式栈中(stack 型)
def infix(text):
    infix_stack = stack()
    size = len(text)
    num = ''
    sqrt = ''  # 用于存放 sqrt
```

```
    for i in range(size):
        # 判断当前选中字符是否为数字或小数点,若是则存入 num 中
        if identify_num(text[i]):
            num += text[i]
        # 如果当前选中字符是 sqrt 的元素,则将其存入 sqrt 中
        elif text[i] == 's' or text[i] == 'q' or text[i] == 'r' or
text[i] == 't':
            sqrt += text[i]
        else:
            # 若 sqrt 存满,则将 sqrt 堆入栈中,并将 sqrt 初始化
            if sqrt == 'sqrt':
                infix_stack.push(item=sqrt)
                sqrt = ''
            # 若 num 中存有数字则将其转化为 float 型,堆入栈中
            if num != '':
                infix_stack.push(item=float(num))
                num = ''
            # 将当前运算符堆入栈中
            infix_stack.push(item=text[i])
    # 若最后 num 中还存有数字,则堆入栈中
    if num != '':
        infix_stack.push(item=float(num))
    # 返回中缀栈
    return infix_stack
```

6.1.4 中缀栈转化为后缀栈

文件 stack.py 中

```
# 将中缀表达式栈转化为后缀表达式栈(stack 型)
def infix_to_suffix(infix_stack):
    suffix_stack = stack()  # 后缀表达式栈
    ope = stack()  # 运算符栈(中间栈)
    size = infix_stack.size()  # 中缀表达式栈的大小
    flag = 1
    for i in range(size):
        temp = infix_stack.listarray[i]
        # 若栈当前元素是数字,则直接压入后缀表达式栈中
        if not identify_ope(temp):
            suffix_stack.push(item=temp)
        # 左括号'('优先级最高,直接入运算符栈
        if temp == '(':
            ope.push(item=temp)
        # 如果是'+'、'-'、'×'、'÷'、'sqrt'根据算数符号优先级入栈
        if temp == '+' or temp == '×' or temp == '-' or temp == '÷'
or temp == 'sqrt' or temp == '^':
            # 若运算符栈空,则算数符号入栈
```

```
            if ope.isEmpty():
                ope.push(item=temp)
            # 否则,根据算数符号优先级入栈与出栈
            else:
                while True:
                    # 取出栈顶算数符号
                    ope_top = ope.peek()
                    # 若运算符栈顶算术符号优先级低于当前算术符号,则直接入运算符
栈
                    if priority(temp, ope_top):
                        ope.push(item=temp)
                        break
                    # 否则,运算符栈顶算术符号压入后缀表达式栈中
                    else:
                        suffix_stack.push(item=ope_top)
                        ope.pop()  # 算数符号出运算符栈直到当前算术符号优先级
大于运算符栈栈顶算术符号
                        # 如果栈空,那么当前算数符号入栈
                        if ope.isEmpty():
                            ope.push(item=temp)
                            break
        # 如果栈当前元素是右括号')',算数符号出运算符栈,直到遇到左括号'('为止
        if temp == ')':
            while ope.peek() != '(':
                suffix_stack.push(item=ope.peek())
                ope.pop()
            ope.pop()  # '('出栈,且不放入算数表达式
    # 运算符栈中剩余算术符号压入后缀表达式栈中
    while not ope.isEmpty():
        suffix_stack.push(item=ope.peek())
        ope.pop()
    # 返回后缀表达式栈
    return suffix_stack
```

6.1.5 计算后缀栈结果

文件 stack.py 中

```
# 利用后缀表达式栈计算后缀表达式结果(float 型)
def compute_suffix(suffix_stack):
    num = stack()  # 数字栈
    size = suffix_stack.size()  # 后缀表达式栈的大小
    for i in range(size):
        temp = suffix_stack.listarray[i]
        # 数字直接入栈
        if not identify_ope(temp):
            num.push(item=temp)
```

```
        # 执行加法'+'
        if temp == '+':
            b = num.peek()
            num.pop()
            a = num.peek()
            if a is None:
                num.push(item=float(b))
                continue
            num.pop()
            num.push(item=float(a + b))
        # 执行减法'-'
        if temp == '-':
            b = num.peek()
            num.pop()
            a = num.peek()
            if a is None:
                num.push(item=float(-b))
                continue
            num.pop()
            num.push(item=float(a - b))
        # 执行乘法'×'
        if temp == 'x':
            b = num.peek()
            num.pop()
            a = num.peek()
            num.pop()
            num.push(item=float(a * b))
        # 执行除法'÷'
        if temp == '÷':
            b = num.peek()
            num.pop()
            a = num.peek()
            num.pop()
            num.push(item=float(a / b))
        # 执行开方运算'sqrt'
        if temp == 'sqrt':
            a = num.peek()
            num.pop()
            num.push(item=math.sqrt(a))
        # 执行幂运算'^'
        if temp == '^':
            b = num.peek()
            num.pop()
            a = num.peek()
```

```
            num.pop()
            num.push(item=pow(a, b))
    if num.size() == 1:
        return num.peek()  # 返回后缀表达式结果
    raise Exception
```

6.1.6 运算符优先级定义

文件 stack.py 中

```
# 算数符号优先度(int 型)
def symbol_priority(symbol):
    # 定义在栈中左括号的优先度为 0
    if symbol == '(':
        return 0
    # 定义加减运算的优先度为 1
    if symbol == '+' or symbol == '-':
        return 1
    # 定义乘除运算的优先度为 2
    if symbol == '×' or symbol == '÷':
        return 2
    # 定义 sqrt 运算的优先度为 3
    if symbol == 'sqrt' or symbol == '^':
        return 3


# 算术优先级判断(bool 型)
def priority(a, b):
    # 算术优先级 a > b,则返回 true
    return symbol_priority(a) > symbol_priority(b)
```

6.1.7 括号合法性的判断

文件 stack.py 中

```
# 判断括号合法性(bool),括号不合法则输出 False
def brackets_legal(infix_stack):
    brackets_stack = stack()
    size = infix_stack.size()
    for i in range(size):
        if infix_stack.listarray[i] == '(':
            brackets_stack.push(item='(')
        if infix_stack.listarray[i] == ')':
            if brackets_stack.isEmpty():
                return False
            else:
                brackets_stack.pop()
    if brackets_stack.isEmpty():
        return True
    return False
```

文件 MyError.py 中

```
# 括号异常类
class brackets_error(Exception):
    def __init__(self, value):
        self.value = value
```

6.1.8 主要按钮的设计

文件 main.py 中

```
# 按钮=
def press_equal(self):
    text = self.lineEdit.text()
    try:
        # 先判断括号合法性
        if not brackets_legal(infix(text)):
            raise brackets_error("括号有误")
        # result 表示结果
        self.result = compute_suffix(infix_to_suffix(infix(text)))
        self.flag = 1
        self.lineEdit.setText(str(self.result))
    except:
        self.flag = 2
        self.lineEdit.setText("Invalid syntax!")

# 按钮 DEL
def press_del(self):
    text = self.lineEdit.text()
    nexttext = ''
    for i in range(len(text)):
        if i != len(text)-1:
            nexttext += text[i]
    self.lineEdit.setText(nexttext)
    if self.flag == 2:
        self.press_ac()
    self.flag = 0

# 按钮 AC
def press_ac(self):
    self.flag = 0
    self.lineEdit.setText("")

# 根据 flag 值判断再次输入数字时是否要清空数值
def re_enter(self):
    if self.flag != 0:
        self.press_ac()
```

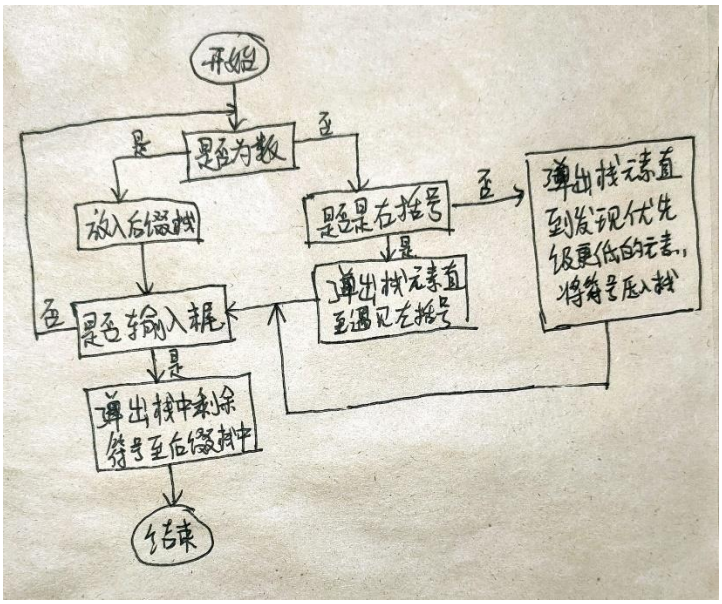
```
# 按钮 0
def press_0(self):
    self.re_enter()
    self.lineEdit.insert("0")

# 按钮+
def press_plus(self):
    if self.flag == 2:
        self.press_ac()
    self.flag = 0
    self.lineEdit.insert("+")
```

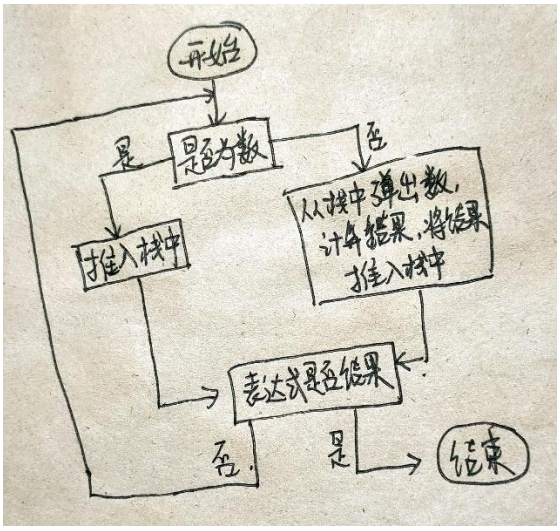
相似代码按钮不列出。

6.2 计算器项目算法流程图

6.2.1 中缀到后缀的转换算法流程图



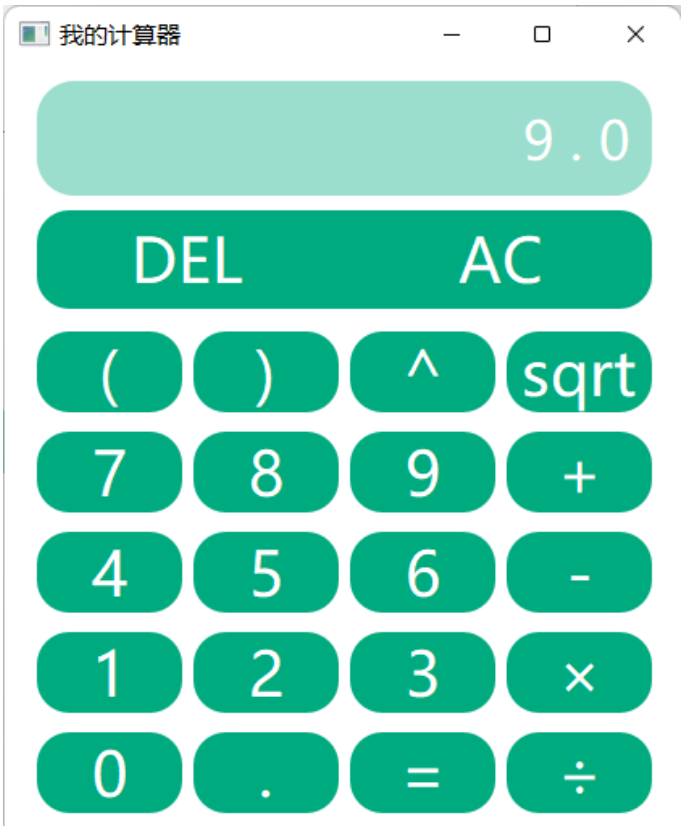
6.2.2 计算后缀栈结果的算法流程图



七、测试结果及分析：

7.1 综合运算正确性测试结果

1、计算表达式 2^3+sqrt(12-3)÷(2×1.5)=9



由于设计初衷是设计一个简易计算器，故文本框设置的比较小，不足以装下整个表达式，故分两段截图截出文本框中的表达式。

对比计算结果可正确答案可知，计算器计算结果正确！

2、计算表达式-3÷1.5=-2

3、计算表达式 sqrt(2)=1.414…..

7.2 非法输入测试结果

当输入算数表达式有误时，计算器会提示非法的输入
1、括号的非法输入

2、小数点的非法输入

3、运算符的非法输入

7.3 DEL 按钮测试结果

单击 DEL 按钮时会将算数表达式最后一位删除。

7.4 AC 按钮测试结果

单击 DEL 按钮时会将算数表达式清空。

7.5 运算结果的继承和清除测试结果

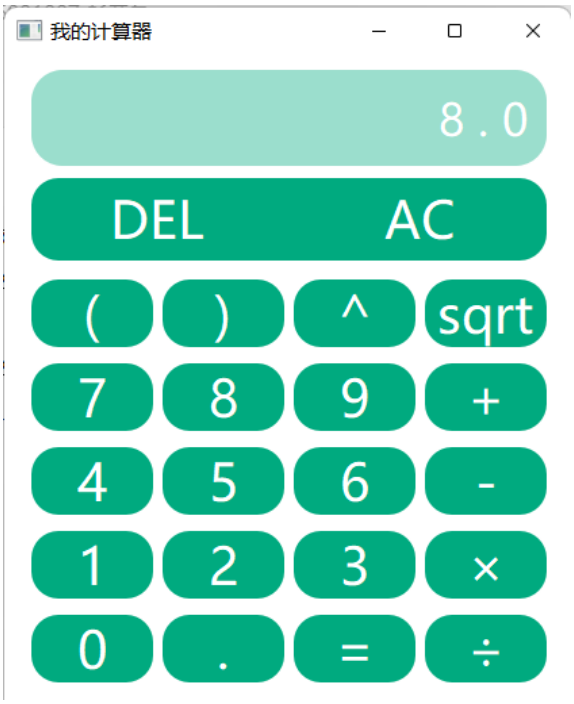
当计算器已经计算完一个结果时，单击数字按钮或小数点按钮或括号按钮时都会清空原计算并转化为键入内容，单击运算符按钮时则会继承运算结果添加运算符继续计算。

1、在 1+2=3 的运算结果下单击数字 8

2、在 1+2=3 的运算结果下单击运算符“+”

7.6 exe 程序运行测试结果

将 exe 程序移到桌面后双击，发现可以正常运行且数值计算结果都正确。



八、总结

8.1 实验感想以及学习心得

本次实验让我系统化的了解到了一个基于 python 的 Windows 窗口软件的制作过程，让我对栈的数据结构的了解更加深入了，熟练掌握了使用栈的操作，深刻理解了中缀表达式转化为后缀表达式和计算后缀表达式结果的算法，以及让我对断点调试程序和如何寻找程序错误更加熟练了。同时本次实验我还进行了实验创新，实现了开方运算与幂运算的操作，同时完成了将一个 python 程序打包成 exe 可执行文件并设定可执行文件的图标的操作，在实验中的探索过程中让我受益匪浅，犹如是在知识的海洋中畅游。综上所述，本次实验收获颇多，对我的数据结果学习提升很大。

8.2 整个实验项目的优缺点自评

优点

本次实验除了老师要求的界面以及功能实现，我还进行了创新，添加了“sqrt”开方运算和“^”幂运算，同时完成了 python 程序的 exe 文件打包和 ico 图标设置，实验结果更趋于是一个 Windows 应用程序。

缺点

计算器未实现保留前一个计算结果的操作，计算器的文本框设计的过于小难以装下很多复杂运算。

改进想法

对于整个项目的改进我的思考如下，分为 3 个方面

1、可以对整个项目的可观性与可用性进行改进，设计一个更好的 CSS 样式，并且将文本框设置的长一些，让它能装下更复杂的算数表达式。

2、可以让计算器的功能更加完善一些，填入更多的函数运算（三角函数、对数等），设置一个 ANS 按钮用于返回上一个计算结果。

3、可以添加存储计算器历史记录的功能，考虑将其设置在菜单栏中，在菜单栏中点击查看计算器历史记录的按钮便可查看计算器的所有历史记录。