4，栈和队列-I

- 栈和队列的概念
- 数据的生成，缓存，使用和顺序
- 栈的实现和问题
- Python 的栈实现
- 队列的实现和问题
- Python 的队列实现
- 栈应用实例
- 队列应用实例
- 搜索问题和其他

概述

- 栈（stack）和队列（queue）是两种使用最广泛的数据结构，它们都是保存数据元素的容器，可以将元素存入其中，或者从中取出元素使用（查看，或弹出，即在取得元素的同时将其从容器里删除）
- 容器是一大类具有保存数据作用的数据结构，它们都保证存入的元素可以在将来取到，而取出并删除的元素不再存在于容器之中
- 栈和队列主要用于在计算过程中临时性地保存元素
  - 这些元素是前面计算中发现或产生的中间数据，需要在后面使用
  - 如果工作中产生的中间数据暂时无法使用或用不完，就有必要把已生成出来但当时还不需要或不能用掉的元素暂时保存起来
  - 栈和队列常用于生成和使用之间的缓冲，称为缓冲存储或缓存
- 栈和队列的操作都比较简单
  - 最重要的就是放入元素和取出元素两个操作
  - 还可能有另外一些辅助操作，如创建，检查，判断空/满等

概述

- 中间数据元素的生成有早有晚，有时间上的先后顺序。在实际应用中，使用这些元素也可能需要按时间的顺序。最典型的两种顺序如下：
- 根据数据生成的顺序，后生成并保存的数据先行处理
  - 如做数学题，遇到做不下去，通常是退回一步考虑别的可能性
- 根据数据生成的先后顺序处理，先生成的数据先处理
  - 如银行办事，先到的应先得到服务，具体等待方式不重要
    - 直接排在一个等待队列上
    - 拿（顺序）号后等叫号，每次总叫尚未服务的最早来的顾客
- 在这两种情况下，访问（并可能删除）都按默认的方式确定元素
  - 栈和队列就是支持按这两种顺序使用元素的缓存数据结构
  - 栈和队列存入操作只需要保证元素存入和将来取出的顺序，不需要记录或保证新存入的元素与容器中已有元素之间的任何关系

概述

- 栈和队列在保证元素存取之间的时间顺序关系方面的特点：
  - 栈是保证缓存元素后进先出（**Last In First Out，LIFO**）的结构
  - 队列是保证缓存元素的先进先出（先存的先用，**First In First Out，FIFO**）关系的结构
- 对栈和队列，在任何时候，下次访问或删除的元素都默认地唯一确定。只有新的存入或删除（弹出）操作可能改变下次的默认元素
- 栈和队列的特点完全是抽象的逻辑的，对于如何实现（如何实现相应时间顺序）并没有任何约束，任何能满足要求的技术均可使用
  - 一种很方便的技术是利用元素的排列顺序表示它们的先来后到，也就是说，可以用线性表作为栈和队列的“实现结构”
  - 例如：把元素进入实现为表的后端插入，这样
    - 队列里最老的（下次访问或删除的）元素总是最前面一个元素
    - 栈里最年轻的（下次访问或删除的）元素总是最后一个元素

概述

- 栈和队列是计算中使用最广的缓存结构，其使用环境可以总结如下：
  - 计算过程分为一些顺序进行的步骤
  - 所执行的步骤中会不断产生一些后面可能需要的中间数据
  - 产生的一些数据不能立即使用，但必须保存起来准备以后使用
  - 需要保存的数据的项数不能事先确定

  这种情况下，通常就需要用一个栈或一个队列作为缓存
- 栈和队列是许多重要算法的基础，后面有许多实例。栈和队列的性质和操作效率，也是许多算法的设计中非常关键的因素
- 由于栈和队列在应用中的重要性，**Python** 的基本功能中包括了对栈的支持（可直接用 **list**），标准库提供了支持队列用途的结构
- 下面分别讨论这两种结构的情况，包括

  性质和模型；实现和问题；一些应用；一些一般性问题

栈：概念

- 栈（**stack**），有的书籍里称为堆栈，是一种容器，可存入数据元素、访问元素、删除元素。在这里没有位置的概念

  栈保证任何时刻可以访问、删除的元素都是在此之前最后存入的那个元素。因此确定了一种默认的访问顺序
- 栈的基本操作是一个封闭集合（与表不同），通常包括：
  - 创建空栈
  - 判断栈是否为空（还可能需要判断满），**is_empty()**
  - 向栈中插入（通常称推入/压入，**push**）一个元素，**push(...)**
  - 从栈中删除（弹出，**pop**）一个元素，空栈弹出报错，**pop()**
  - 取当前（最新）元素的值（并不删除），**top()**
- 我们可以为栈建立一个理论模型，见本节最后的幻灯片（可以课后自己看看，不讲了）

栈：特性和基本实现考虑

- 栈可以实现为（可以看作）是只在一端插入和删除的表
  - 因此有人（有书）把栈称为后进先出（**LIFO**）表
  - 进行插入或删除操作的一端称为栈顶，另一端称为栈底
- 用线性表的技术实现栈时，由于只需要在一端操作，自然应该利用实现最方便而且能保证两个主要操作的效率最高的那一端
  - 采用连续表方式实现，在后端插入删除都是 **O(1)** 操作（!!!☹）
  - 采用链接表方式实现，前端插入删除都是 **O(1)** 操作

  栈通常都采用这两种技术实现
- 实现栈之前，我们定义一个自己的异常类（**Python** 的内部异常是一组类，都是 **Exception** 的子类，可以继承已有异常类定义自己的异常类）

```
class StackUnderflow(ValueError): # 栈下溢（空栈访问）
    pass
```


栈：实现

- 采用连续表技术实现，也会遇到连续表的各种相关问题
  - 用简单连续表，还是采用动态连续表（分离式实现的连续表）？
  - 如果用简单连续表，就可能出现栈满的情况
  - 采用动态连续表，栈存储区满时可以置换一个更大的存储区。这时又会出现置换策略问题，以及分期付款式的 **O(1)** 复杂性
- **Python list** 及其操作实际上提供了一种栈功能，可以作为栈使用
  - 建立空栈，对应于创建一个空表 **[ ]**，判空栈对应于判空表
  - 由于 **list** 采用动态连续表技术（分离式实现），作为栈的表不会满
  - 压入元素操作应在表尾端进行，对应于 **lst.append(x)**
  - 弹出操作也应在尾端进行，无参的 **lst.pop()** 默认弹出表尾元素
  - 由于采用动态连续表技术，压入操作具有分期付款式的 **O(1)** 复杂性，其他操作都是 **O(1)** 操作

栈：实现

- 如果为了概念清晰和操作名更易理解，也可以定义一个连续表实现的栈类，用 list 作为实现基础

```
class SStack():
    def __init__(self):
        self.elems = [ ]
    def is_empty(self):
        return self.elems == [ ]
    def top(self):
        if self.elems == [ ]:
            raise StackUnderflow
        return self.elems[len(self.elems)-1]
    def push(self, elem):
        self.elems.append(elem)
    def pop(self):
        if self.elems == [ ]:
            raise StackUnderflow
        return self.elems.pop()
```


栈：链接表实现

- 采用链接技术，可以借用 LNode 类实现一个链接栈：

```
class LStack (): # stack implemented as a linked node list
    def __init__(self):
        self.top = None
    def is_empty(self):
        return self.top is None
    def top(self):
        if self.elems == []:
            raise StackUnderflow
        return self.top.elem
    def push(self, elem):
        self.top = LNode(elem, self.top)
    def pop(self):
        if self.top is None:
            raise StackUnderflow
        p = self.top
        self.top = p.next
        return p.elem
```

栈的应用

- 栈是算法和程序里最常用的辅助结构，基本用途基于两个方面：
  - 如前所述，用栈可以很方便地保存和取用信息，因此常作为算法或程序里的辅助存储结构，临时保存信息，供后面操作中使用
  - 利用栈后进先出的特点，可以得到特定的存储和取用顺序

  许多实际运用结合了这两方面的特性
- 作为最简单的应用实例，栈可以用于颠倒一组元素的顺序
  - 将一组元素依次全部存入后取出，得到反序的序列
  - 通过不同的存入取出操作序列，可以得到不同的元素序列。但请注意，这种做法不能得到任意的排列，结果序列有一定的规律性
- 下面介绍栈的若干典型应用
  - 有些比较具体，是特殊的应用
  - 有些是一大类应用的实例，更具典型性

简单应用：括号配对问题

- 在许多正文中都有括号，特别是在程序、表示数学表达式的正文片段里。括号有正确配对问题。作为例子，下面考虑 Python 程序里的括号
  - 存在不同的括号：如圆括号、方括号和花括号
  - 每种括号都有开括号和闭括号，括号括起的片段可能相互嵌套，各种括号需要分别正确嵌套配对
- 配对的原则
  - 遇到的闭括号应该匹配此前遇到的最近的尚未匹配的对应开括号
  - 由于多种/多次/可能嵌套，为检查配对，遇到的开括号必须保存
  - 由于括号可能嵌套，需要逐对匹配，闭括号应与前面最近的尚未有匹配的开括号匹配，后面括号应与更前面次近的括号匹配
  - 可以删除匹配的括号，为后面的匹配做好准备
- 后遇到并保存的开括号应该先删除，这就是后进先出，而且要按照出现顺序，显然应该/可以用一个栈保存开括号

简单应用：括号配对问题

- 问题处理的线索已经清楚了：
  - 顺序检查所考虑的正文（一个字符串）里的一个个字符
  - 无关字符统统跳过
  - 遇到开括号时将其压入一个栈
  - 遇到闭括号时弹出栈顶元素与之匹配

    匹配则继续；遇到不匹配时检查以失败结束
- 定义一个函数完成这个检查，其中先定义一些检查用的数据和变量

简单应用：括号配对问题

- 函数定义（很简单）：

```
def check_pares(text):
    pares = "()[]{}"
    open_pares = "([{"
    opposite = {")":"(", "]":"[", "}":"{"} # 表示配对关系的字典

    def paretheses(text): ... # 括号生成器，定义见后

    st = SStack()

    for pr, i in paretheses(text): # 对 text 里各括号和位置迭代
        if pr in open_pares: #  开括号，压进栈并继续
            st.push(pr)
        elif st.pop() != opposite[pr]: # 不匹配就是失败，退出
            print("Unmatching is found at", i, "for", pr)
            return False
        # else 是一次括号配对成功，什么也不做，继续
    print("All paretheses are correctly matched.")
    return True
```

简单应用：括号配对问题

- 括号生成器定义为（局部）函数

```
def paretheses(text):
    i, text_len = 0, len(text)
    while True:
        while i < text_len and text[i] not in pares:
            i += 1
        if i >= text_len:
            return
        yield text[i], i
        i += 1
```

- 生成器（回忆一下）：
  - 用 yield 语句产生结果
  - 可以用在需要迭代器的地方
  - 函数结束导致迭代结束

简单应用：括号配对问题

- 总结：
  - 利用了栈的性质，这是关键
  - 准备好所需数据。虽然只用一次，但能使程序清晰易读，易于修改
  - 利用了 Python 丰富的数据结构和操作，如 in，not in 和字典等
  - 定义括号生成器函数，使代码功能清晰便于修改

    例如，要检查 Python 或 C 程序里的括号，就需要跳过注释和字符串，只需修改这个函数。可作为自己的练习

表达式的表示、计算和变换

- 小学生就开始写表达式，先是算术表达式，后来是代数表达式
  - 对二元运算符，通常把它们写在两个运算对象的中间，这种写法称为中缀表示，按中缀表示写出的表达式称为中缀表达式
- 中缀表示很难统一贯彻，一元和多元运算符很难用中缀表示
  - 代数表达式里的函数符号通常写在运算对象前面，这种写法称为前缀表示。为界定函数参数的范围，通常用括号将它们括起
  - 可见，常见的表达式习惯表示是前缀表示和中缀表示的组合
- 实际上，表达式并不一定采用这种习惯写法
  - 可以用纯粹的前缀表示，这样写出的表达式称为前缀表达式
  - 还有一种表达式写法称为后缀表示，其中运算符（函数）总写在它们的运算对象之后，这样写出的表达式称为后缀表达式。实际上，后缀表达式特别适合计算机处理
- 前缀表达式也称为波兰表达式（由波兰数学家 **J. Lukasiewicz** 于**1929** 提出），后缀表达式也称为逆波兰表达式

表达式的表示、计算和变换

- 假设我们知道每个运算符的元数（运算对象个数），且元数唯一。在写表达式时，一个重要问题是需要准确描述计算的顺序
- 中缀表达式的一个重要缺点是不能直接表示运算的顺序，需要辅助性的约定和/或辅助描述机制
  - 首先必须引进括号，表示括号里的运算先做（显式描述计算顺序）
  - 为减少总要写括号的麻烦，通常还引进优先级概念，规定各运算符的优先级（或优先级关系），优先级高的运算符结合性强，运算符相邻出现时，优先级高的运算应先做
  - 还需规定具有相同优先级的运算符相邻出现时计算顺序（结合性）
- 与之对应的情况：在前述条件下，前缀表达式和后缀表达式都不需要括号，也不需要规定优先级，足以描述任意复杂的计算顺序
  - 可见，中缀表达式的表达能力最弱
  - 中缀表达式增加了括号后，几种表达方式具有同等表达能力

表达式的表示、计算和变换

- 对比几种不同表达式形式。下面三个算术表达式等价
  - 中缀：**(3 - 5) * (6 + 17 * 4) / 3**
  - 前缀：**/ * - 3 5 + 6 * 17 4 3**
  - 后缀：**3 5 - 6 17 4 * + * 3 /**

  三个表达式描述的是同一个计算过程
- 下面先考虑后缀表达式的求值，假定
  - 要处理的是算术表达式
  - 其中的运算对象是浮点数形式表示的数
  - 运算符只有 "+"、"-"、"*"、"/"，都是二元运算符
- 而后讨论不同表达式形式之间的转换（考虑中缀形式到后缀形式）

  研究相应的转换算法，以及中缀表达式的求值

后缀表达式的计算

- 考虑计算过程，设有函数 **nextItem()** 得到下一个运算对象或运算符：
  - 遇到运算对象，需要记录以备后面使用
  - 遇到运算符（或函数名），需要根据其元数取得前面最近遇到的几个运算对象或已做运算得到的结果，实施计算并记录结果

  用什么结构记录信息？
- 看计算的性质：
  - 需要记录的是已经掌握但还不能立即使用的中间结果，需要缓存
  - 遇到运算符时，要使用的是此前最后记录的几个结果
- 显然应该用栈作为缓存结构
- 上面的分析已经说明了算法的基本结构

  实际程序就是把这个算法用 **Python** 的语言写出来

后缀表达式的计算

- 先考虑算法的框架
- 假定 st 是一个栈，算法的核心是一个循环（）：

```
while 还有输入 :
    x = nextItem()
    if not is_operator(x):
        st.push(float(x))
    else:
        a = st.pop()        # 第二个运算对象
        b = st.pop()        # 第一个运算对象
        ... ...  # 根据运算符 x 对 a 和 b 计算
        ... ...  # 计算结果压入栈
```

- 这里写了几个辅助过程，其具体实现依赖于一些情况
  - 被求值的表达式从哪里获得
  - 表达式里的元素如何表示（假定用字符串）

后缀表达式的计算

- 假定从函数参数获得输入

  参数是一个字符串，内容是需要求值的后缀表达式

- 为程序清晰，定义了一个包装过程：

```
# 定义一个函数把表示表达式的字符串转化为项的表
def suffix_exp_evaluator(line):
    return suf_exp_evaluator(line.split())
```

- 定义一个扩充的栈类，增加一个检查栈深度的方法：

```
class ESStack(SStack):
    def depth(self):
        return len(self.elems)
```

- 下面是核心求值过程的定义（与前面设计相比有一些小修改）

  由于函数的输入就是一个项的表

```
def suf_exp_evaluator(exp):
    operators = "+-*/"
    st = ESStack() # 扩充功能的栈，可用 depth() 检查栈内元素个数
    for x in exp:
        if not x in operators:
            st.push(float(x)); continue
        if st.depth() < 2:
            raise SyntaxError("Short of operand(s).")
        a = st.pop() # second argument
        b = st.pop() # first argument
        if x == "+":     c = b + a
        elif x == "-":   c = b - a
        elif x == "*":   c = b * a
        elif x == "/":
            if a == 0: raise ZeroDivisionError
            c = b / a
        else: pass # This branch is not possible
        st.push(c)
    if st.depth() == 1:     return st.pop()
    raise SyntaxError("Extra operand(s).")
```


后缀表达式的计算

- 定义一个交互式的驱动函数（主函数）：

```
def suffix_exp_calculator():
    while True:
        try:
            line = input("Suffix Expression: ")
            if line == "end": return
            res = suffix_exp_evaluator(line)
            print(res)
        except Exception as ex:
            print("Error:", type(ex), ex.args)
```

至此后缀表达式计算器的开发工作完成

- 注意：
  - 这里用一个 **try** 块捕捉用户使用时的异常，保证我们的计算器不会因为用户输入错误而结束
  - **except Exception as ex** 两个用途：**Exception** 表示捕捉所有异常保证交互继续；**ex** 将约束到所捕捉异常，使处理器能用相关信息

中缀表达式到后缀表达式的转换

- 中缀表达式的情况比较复杂，求值不容易直接处理
  - 可以考虑将其转换为后缀表达式，然后就可以借用前面定义的后缀表达式求值器
- 考虑这里的情况，例如
  - 中缀：**(3 - 5) * (6 + 17 * 4) / 3**
  - 后缀：**3 5 - 6 17 4 * + * 3 /**
- 分析情况：
  - 运算对象应直接输出（因为运算符应该在它们的后面）
  - 处理中缀运算符的优先级（注意，输出运算符就是要求运算）

    遇到运算符不能简单地输出，只有下一运算符的优先级不高于本运算符时，才能做本运算符要求的计算（应输出本运算符）
  - 也就是说，读到运算符 **o** 时，需要用它与前一运算符 **o'** 比较，如果 **o** 的优先级不高于 **o'**，就做 **o'**（输出 **o'**），而后记住 **o**

中缀表达式到后缀表达式的转换

- 运行过程中可能需要记录多个运算符，新运算符需要与前面最后遇到的运算符比较，因此应该用一个栈
  - 处理中缀表达式里的括号：遇到左括号时应该记录它；遇到右括号时反向逐个输出所记录的运算符（排在后面的肯定优先级更高），直到遇到左括号将其抛弃
  - 最后可能剩下一些记录的运算符，应反向将其输出

  几种情况都是后来的先用，只用一个运算符栈就够了

  操作中需要特别注意检查栈空的情况
- 准备操作中使用的数据：

  priority = {"(":1, "+":3, "-":3, "*":5, "/":5}

  infix_operators = "+-*/()" # 把 '(', ')' 也看作运算符特殊处理
  - **priority** 给每个运算符关联一个优先级。给 **"("** 一个很低的优先级，可以保证它不会被其他运算符强制弹出，只有对应的 **")"** 弹出它

中缀表达式到后缀表达式的转换

```
def trans_infix_suffix(line):
    st = SStack(); llen = len(line); exp = [ ]
    for x in tokens(line): # tokens 是一个待定义的生成器
        if x not in infix_operators: # 运算对象直接送出
            exp.append(x)
        elif st.is_empty() or x == '(': # 左括号进栈
            st.push(x)
        elif x == ')': # 处理右括号的分支
            while not st.is_empty() and st.top() != "(":
                exp.append(st.pop())
            if st.is_empty(): # 没找到左括号，就是不配对
                raise SyntaxError("Missing \'(\'.")
            st.pop() # 弹出左括号，右括号也不进栈
        else: # 处理算术运算符，运算符都看作是左结合
            while (not st.is_empty() and
                   priority[st.top()] >= priority[x]):
                exp.append(st.pop())
            st.push(x) # 算术运算符进栈
```

中缀表达式到后缀表达式的转换

```
    while not st.is_empty(): # 送出栈里剩下的运算符
        if st.top() == "(":  # 如果还有左括号，就是不配对
            raise SyntaxError("Extra \'(\' in expression.")
        exp.append(st.pop())
    return exp
```

- 为测试方便，定义一个专门用于测试的辅助函数：

```
def test_trans_infix_suffix(s):
    print(s)
    print(trans_infix_suffix(s))
    print("Value:", suf_exp_evaluator(trans_infix_suffix(s)))
```

中缀表达式到后缀表达式的转换

- 把生成表达式里各个项的工作定义为一个生成器：

```
def tokens(line):
    i, llen = 0, len(line)
    while i < llen:
        while line[i].isspace():  i += 1
        if i >= llen: break
        if line[i] in infix_operators: # 运算符的情况
            yield line[i]; i += 1; continue
        j = i + 1 # 下面处理运算对象
        while (j < llen and not line[j].isspace() and
               line[j] not in infix_operators):
            if ((line[j] == 'e' or line[j] == 'E')  # 处理负指数
                and j+1 < llen and line[j+1] == '-'): j += 1
            j += 1
        yield line[i:j] # 生成运算对象子串
        i = j
```

这个计算器不能处理负号（负数）和一元运算符

要完全地处理表达式，需要做语法分析。有关情况不进一步讨论

存在较简单的处理方式，请考虑

中缀表达式的求值

- 中缀表达式的求值比后缀表达式复杂，需要考虑
  - 运算符的优先级
  - 括号的作用
  - 根据读入中遇到的情况，确定完成各运算的时机
  - 运算时需要找到相应的运算对象
- 从中缀形式到后缀形式的转换算法已经解决了前三个问题
  - 需要用一个运算符栈，在读入表达式的过程中比较运算符的优先级，并根据情况压入弹出
  - 生成后缀表达式需要输出运算符的时刻，就是执行运算的时刻（后缀表达式的性质，计算这种表达式时，遇到运算符就计算）
  - 根据后缀表达式的计算规则，每次执行运算时，相应的运算对象应该是最近遇到的数据，或最近的计算得到的结果

中缀表达式的求值

- 计算中缀表达式，可以用一个栈保存运算对象和中间结果
- 总结一下：
  - 求值中缀表达式，一种方法是用两个栈，其中一个保存运算符，另一个保存运算对象和中间结果
  - 遇到运算对象时入栈
  - 遇到运算符时，根据情况考虑计算、入栈的问题
  - 如果需要计算，数据在运算对象栈里
  - 如果做了计算，结果还需要压入运算对象栈
  - 如果表达式读完了，运算符栈里剩下的运算符逐个弹出完成计算
- 其他细节可以参考前一个例子（优先级处理，**tokens** 可以直接使用）

  这一问题作为本次课的一个练习

栈：抽象数据类型（代数描述）

类型 元素和栈: $e, e_i \in D \qquad s, s_i \in Stack$

运算

$$
\begin{aligned}
newstack &: () \to Stack \\
push &: D \times Stack \to Stack \\
top &: Stack \to D \\
pop &: Stack \to Stack \\
empty &: Stack \to Boolean
\end{aligned}
$$

代数法则

$$
\begin{aligned}
top(push(e, s)) &= e \\
pop(push(e, s)) &= s \\
empty(newstack()) &= \text{true} \\
empty(push(e, s)) &= \text{false}
\end{aligned}
$$

限制

$$
\begin{aligned}
top(newstack()) &= \text{error} \\
pop(newstack()) &= \text{error}
\end{aligned}
$$

满足这组规范的系统（抽象的或具体的）都是栈（的模型）

满足这组规范的程序都是栈的实现（无论采用什么技术）