The Stack

The PLAS M7383 Stack / Nest Container stacks or nests when turned around by 180º

Stacks
Abstract Data Types
The Stack ADT
Implementing a stack
Complexity
Examples

In the real world, a *stack* is a collection of objects, where, to be safe,

- if we want to take an item, we take it from the top
- if we want to add an item, we add it onto the top

A stack is *last-in, first-out (LIFO)*

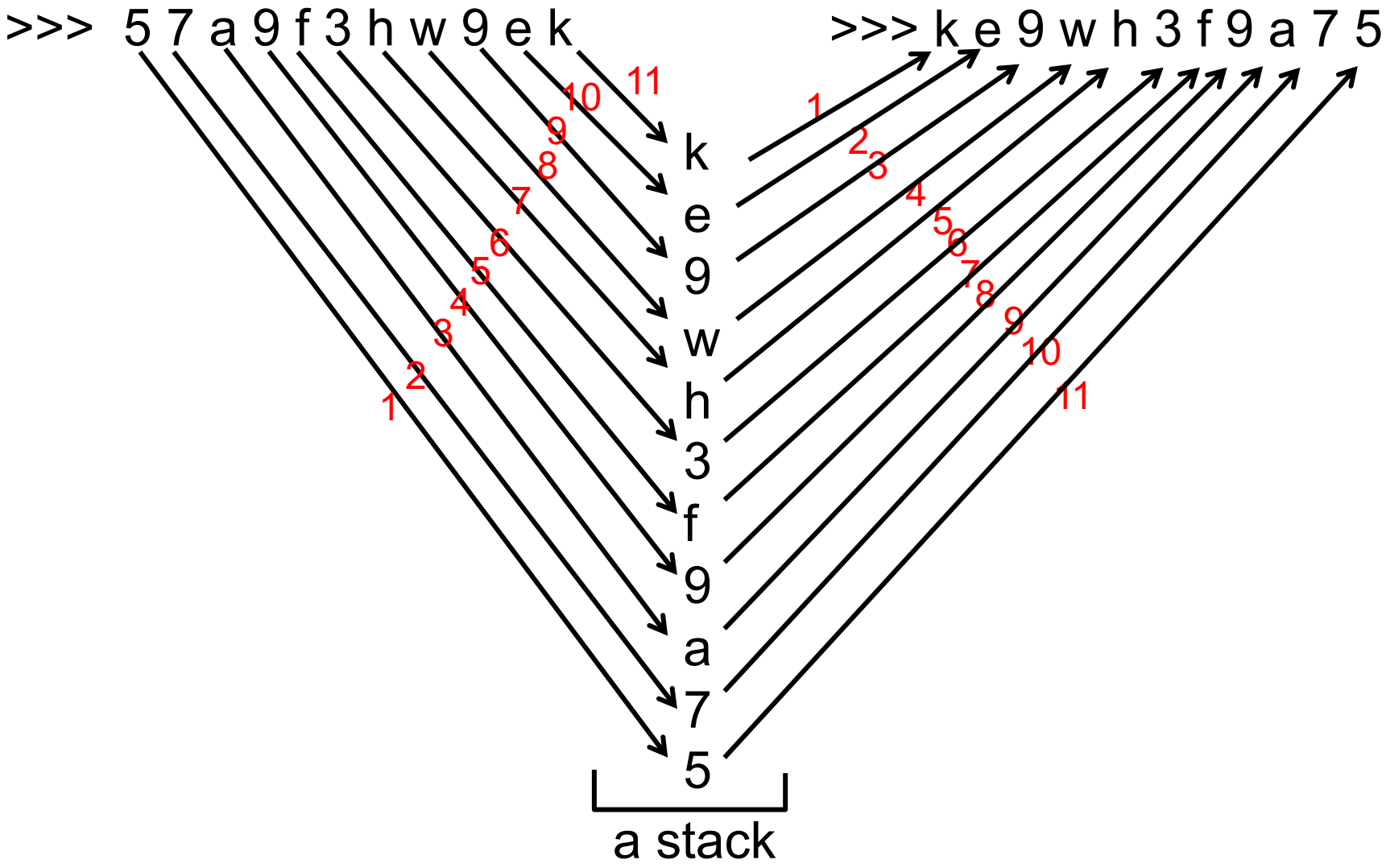
Stacks in Computer Science

The stack idea is surprisingly useful in computer science:

- maintaining a stack of actions taken in a text editor allows an easy UNDO option
- the 'back' button in a web browser returns to the page at the top of a stack of previous pages
- many languages (including Python) are implemented using a stack of active function calls
- evaluating arithmetic expressions without the need for brackets (via postfix notation)
- a quick method for reversing a sequence of input characters

(and many more)

Reversing a sequence of inputs



Postfix

The standard *infix* notation for arithmetic requires brackets to allow operators to be applied against the precedence order

- e.g. 3+5*2 = 13, but (3+5)*2 = 16

It is called "infix" because the operators are located *in*between the values or expressions they operate on.

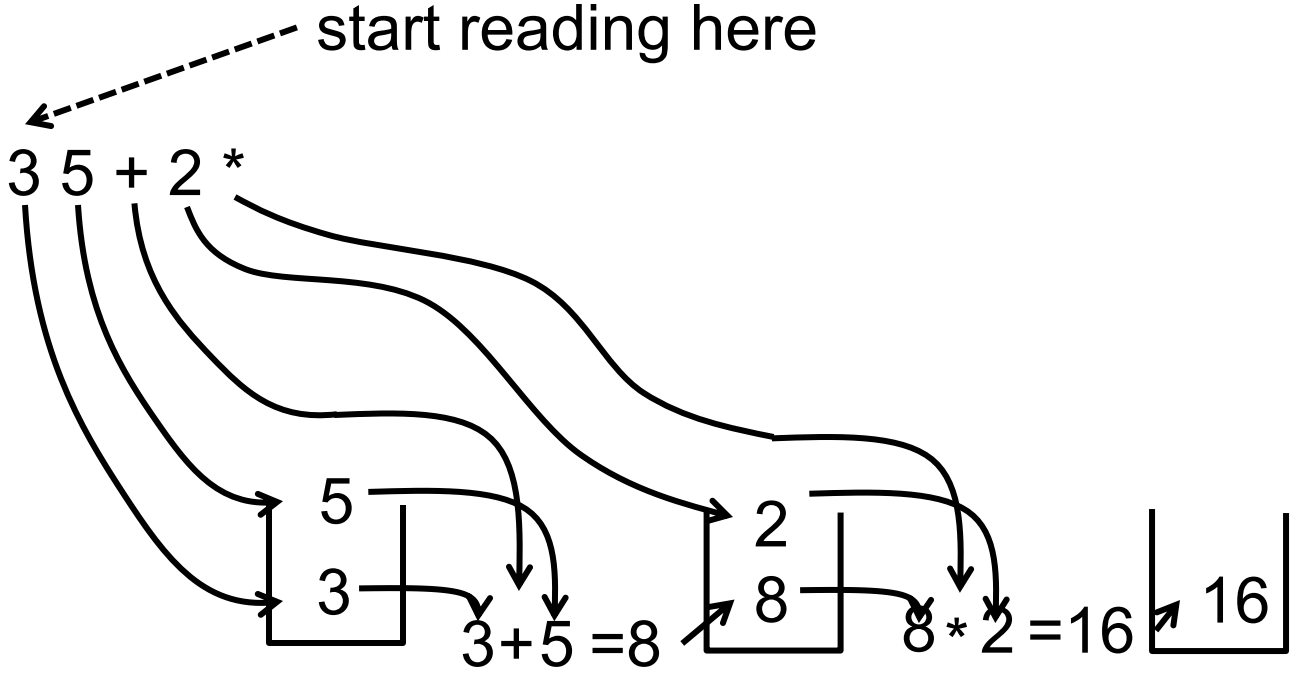
In *postfix*, the operator comes after the values.

- 3 5 + is interpreted as (3 + 5), and so = 8
- 3 5 – is interpreted as (3 - 5), and so = -2
- 5 2 * is interpreted as (5 * 2), and so = 10

Precedence is entirely based on the sequence of the operators. To apply a standard operator, every operator to the left of it must already been applied, and then we can apply it to the values immediately to its left

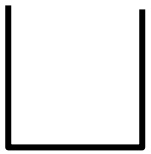
- 3 5 + 2 * = (3 5 +) 2* = 8 2 * = 16
- 3 5 2 * + = 3 (5 2 *) + = 3 10 + = 13

Evaluating Postfix with a Stack

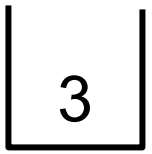


- if you read a number, push it onto the top of the stack
- if you read an operator, pop the top off the stack as the 2nd term, pop the next top off the stack as the 1st term, compute the result, and push it onto the top

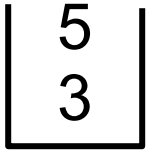
3 5 + 2 *



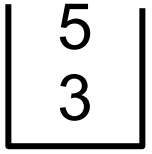
. 5 + 2 *



. . + 2 *

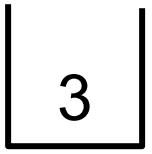


. . . 2 *



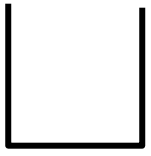
+

. . . 2 *



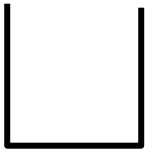
+ 5

. . . 2 *



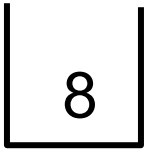
3 + 5

. . . 2 *

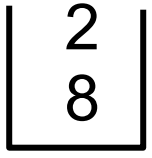


8

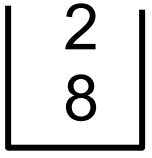
. . . 2 *



. . . . *

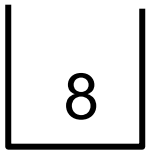


. . . . .



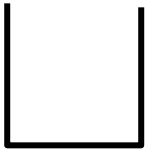
*

. . . . .



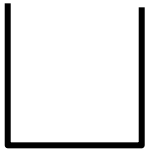
* 2

. . . . .



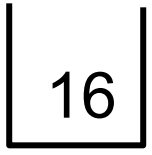
8 * 2

. . . . .



16

. . . . .



16

Evaluate using the stack method: (i) 7 2 – 3 8 + *

(ii) 2 18 8 2 - / +

Abstract Data Type

We will want to use stacks in programming, and we want to know that they will behave like a stack.

We need a way of specify to other programmers (and to ourselves) that we are offering a stack.

Other people don't need to know how it is implemented underneath.

We need a *specification* of a stack, that everyone agrees on, and then we will guarantee that our object behaves according to the specification.

The specification is an Abstract Data Type (ADT)

The Stack ADT

push:	add an item to the stack
pop:	remove the item that was added to the stack most recently
top:	report the item that was added to the stack most recently
length:	report how many elements are in the stack

Note: sometimes an ‘is_empty’ method is also specified, but is simply implemented by '(length == 0)'

The Stack ADT in Python

```
push(element)       #add element onto the stack

pop()               #remove and return the element
                    #that was added to stack most recently
                    #(return None if empty)

top()               #report the element that was added to
                    #stack most recently (None if empty)

length()            #return the number of elements in stack
```

Implementing the Stack

We will define a class, offering those methods.
How should we store the data in the class?

We need to maintain a collection items – use a list?
We need to be able to identify the sequence (to find most recent, etc), so use the list sequence?
We will do a lot of adding to and removing from the sequence. Where should each pushed item go?

To add and delete on a Python list most efficiently, we should do it at the end of the list

- the end of our list will be the 'top' of the stack.

Use `.append()` to push, and `.pop()` to pop.

```
class Stack:
    def __init__(self):
        self.alist = []

    def push(self, element):
        self.alist.append(element)

    def pop(self):
        if len(self.alist) == 0:
            return None
        return self.alist.pop()

    def top(self):
        if len(self.alist) == 0:
            return None
        return self.alist[-1]

    def length(self):
        return len(self.alist)
```

basic implementation
- no private variables
- no __str__() method

```
stack = Stack()  #create a new empty stack object
stack.push(1)
stack.push(2)
stack.push(3)    #should now be |-1-2-3->, length 3
i1 = stack.top() #should be 3, leaving |-1-2-3->
i2 = stack.pop() #should be 3, leaving |-1-2->, length 2
i3 = stack.pop() #should be 2, leaving |-1->, length 1
i4 = stack.pop() #should be 1, leaving |-->, length 0, empty
```

What is happening inside?

Stack: alist: []

Stack: alist: [1]

Stack: alist: [1,2]

Stack: alist: [1,2,3]

Stack: alist: [1,2,3]

Stack: alist: [1,2]

Stack: alist: [1]

Stack: alist: []

Complexity of operations

```
class Stack:
    def __init__(self):
        self.alist = []

    def push(self, element):
        self.alist.append(element)

    def pop(self):
        if len(self.alist) == 0:
            return None
        return self.alist.pop()

    def top(self):
        if len(self.alist) == 0:
            return None
        return self.alist[-1]

    def length(self):
        return len(self.alist)
```

List.append is O(1) *on average*

List.pop() is O(1) *on average* - always pop the last elt, so no copying, unless resizing list

List index lookup is O(1)

List length is O(1)

```
def postfix(string):
    """ Evaluate postfix expressions, using a stack.
        Input must be a string, with space between each item
    """
```

```
def palindrome_check(string):
    """ Determine if string is a palindrome, using a stack.
    """
```

Our stack takes objects of any class

```
def palindrome_check_list(mylist):
    """ Determine if mylist is a palindrome, using a stack.
    """
```

Protecting and providing

```
class Stack:
    def __init__(self):
        self._alist = []

    def __str__(self):
        retstr = '|-'
        for element in self._alist:
            retstr = retstr + str(element) + '-'
        retstr = retstr + '->'
        return retstr

    def pop(self):
        if len(self._alist) == 0:
            return None
        return self._alist.pop()

    #etc.
```

Use the underscore '_' to tell other developers not to access this directly.
Also stops the variable appearing in auto-generated documentation

Provide a method to allow display of the stack as a string

But you have to be consistent. If you use it in __init__, you have to use it everywhere or it won't compile

Next Lecture

The Queue