CS313E: Elements of Software Design

Abstract Data Types and Stacks

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: October 4, 2012 at 15:35

Getting More Abstract

We've been looking at defining classes in Python.

Classes are the Python representation for "Abstract Data Types," a very useful notion in any programming language.

What are Abstract Data Types?

A *data type* in any programming language is simply an *interpretation placed on data*.

The same bit string in memory may be interpreted as a:

- machine instruction
- integer (signed or unsigned)
- floating point number
- collection of characters
- memory address
- many others.

Associated with a data type is a set of operations you can perform on the data.

Abstract Data Type

An *Abstract Data Type* is just a data type (typically defined by the user) in terms of its external interface. That is, certain operations are possible on the data type.

For example, the Card class we defined previously has the following interface (signature):

$$
\begin{aligned}
&\text{Card(integer, string)} &&\longrightarrow \text{Card object}\\
&\text{getSuit(Card)} &&\longrightarrow \text{string}\\
&\text{getRank(Card)} &&\longrightarrow \text{integer}\\
&\text{str(Card)} &&\longrightarrow \text{string}
\end{aligned}
$$

The implementation of the type should be irrelevant to the user. *Any attempt to access a Card object except via these functions is a "violation of the abstraction."*

Just Another Type

Once Card is defined, anyone can use it just like any predefined type of the language by importing the Card module.

This has the following two consequences:

1. Anyone who wants to implement Poker, Canasta, Gin Rummy, BlackJack, etc. can use the Card type as if it were any predefined type, without knowing or caring how it is implemented.
2. The implementor is free to change the underlying implementation in Card.py, *as long as the interface is preserved*.

The interface is a *contract* between the implementor and the user.

Why Abstraction Matters

Abstraction is one of the most powerful ideas in computer science. It separates the *what* from the *how*. Abstraction is how we manage complexity.

For example, when defining Deck, I don't need to think about how Card is implemented, just understand the Card interface.

Similarly, the Python `math` module provides many useful functions: acos (arc cosine), acosh (hyperbolic arc cosine), asin (arc sine), atan (arc tangent), many more. I really don't want to think about how they work, but they're there if I need them.

An Example: Stack

A *stack* is one of the most useful ADTs around. It is a LIFO (last in, first out) list with (at least) the following operations:

Stack()	$\longrightarrow$ Stack
Push (Stack, item)	$\longrightarrow$ Stack
Pop (Stack)	$\longrightarrow$ item
Peek (Stack)	$\longrightarrow$ item
IsEmpty (Stack)	$\longrightarrow$ Boolean
Len (Stack)	$\longrightarrow$ Integer

How do you think this could be implemented?

Stacks

Stacks are used for any tasks where you have to "unwind" a state in reverse order.

For example, in any programming language when you call a procedure or function it may call itself or another function, which may call another, and so on. You have to return from the innermost call to the calling function, etc.

The is implemented via the system stack where each call creates a *stack frame* holding the local variables of the function and the return address. An infinite recursion will give you a *stack overflow*.

A Stack in Python

```
class Stack:
    """Define a Stack ADT with operations: Stack, isEmpty,
    push, pop, peek, and len."""

    def __init__(self):
        self._items = []
    def isEmpty(self):
        return not len(self._items)
    def push(self, item):
        self._items.append(item)
    def pop(self):
        return self._items.pop()
    def peek(self):
        return self._items[-1]
    def __len__(self):
        return len(self._items)
    def __str__(self):
        output = ""
        for x in self._items:
            output = str(x) + " " + output
        return "[ " + output + "]"
```

Using the Stack Class

```
>>> from Stack import *
>>> s = Stack()
>>> s.push(3); s.push(1); s.push(5)
>>> print(s)
[ 5 1 3 ]
>>> x = s.pop()
>>> x
5
>>> print(s)
[ 1 3 ]
>>> y = s.peek()
>>> y
1
>>> len(s)
2
>>> s.isEmpty()
False
>>> s.pop(); s.pop()
1
3
```

Using the Stack Class (2)

```
>>> s.isEmpty()
True
>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "Stack.py", line 22, in pop
    return self._items.pop()
IndexError: pop from empty list
>>> s.push("a")
>>> s.push("b")
>>> print(s)
[ b a ]
```

Implementation Issues

What could I change to make the behavior more robust? I.e., how could I avoid runtime errors?

What would change if I pushed items onto the other end of the list? Would it matter?

Stack Implementation 2

Study this implementation to see how it differs from the previous one. Would a user see any difference?

```
class Stack:
    """An alternative implementation of Stack.
    Define a Stack ADT with operations: Stack, isEmpty,
    push, pop, peek, and len."""

    def __init__(self):
        self._items = []
    def isEmpty(self):
        return self._items == []
    def push(self, item):
        self._items.insert(0, item)
    def pop(self):
        return self._items.pop(0)
    def peek(self):
        return self._items[0]
    def __len__(self):
        return len(self._items)
    def __str__(self):
        output = ""
        for x in self._items:
            output = output + str(x) + " "
        return "[ " + output + "]"
```

Aside: We Might Care

For most purposes, you *shouldn't* care if a Stack implementation pushes and pops from the front or back of a list. But if your application requires very high efficiency there may be a difference.

For example, Python Lists (the underlying data structure in both of our Stack implementations) are implemented (in C) as resizable arrays. It's more efficient to add elements at the end of an array rather than the front. So, if we were performing billions of Stack pushes and pops, it might make a difference.

Very seldom should you worry about such things. If efficiency is that big a concern, don't use Python at all. Use C or assembly language to code your application.

Methods vs. Functions

When you define a class, the functions/procedures that comprise the interface are the *methods* of the class. They are called using the dot notation.

Instance methods are associated with instances of the class. *Class methods* are associated with the class itself. (Instance methods are those that reference `self`.)

Outside the class definition, you can define your own functions/procedures. These can call methods on visible classes using the dot notation.

Methods vs. Functions

For example, `push` is defined within the `Stack` class as follows:

```
def push(self, item):
    self._items.append(item)
```

That means it's an instance method of the class and must be called on an instance of the `Stack` class.

```
>>> s = Stack()
>>> s.push(3)
>>> s.push("abc")
>>> print (s)           # only if __str__ is defined on Stacks
[ abc 3 ]
>>> x = s.pop()
>>> print(x)
abc
>>> print(s)
[ 3 ]
>>> Stack.push(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: push() takes exactly 2 positional arguments (1 given)
```

Heterogeneity

Did you notice that we can push arbitrary objects onto our Stack?

```
>>> s.push(3)
>>> s.push("abc")
>>> print (s)
[ abc 3 ]
```

In many other languages (like Java), you'd have to declare a `Stack of ints` or a `Stack of strings` and not mix the two. Why don't you have to do that in Python?

Heterogeneity

Did you notice that we can push arbitrary objects onto our Stack?

```
>>> s.push(3)
>>> s.push("abc")
>>> print (s)
[ abc 3 ]
```

In many other languages (like Java), you'd have to declare a `Stack of ints` or a `Stack of strings` and not mix the two. Why don't you have to do that in Python?

Answer: What is pushed onto the stack is the *address* of a value, not the value itself.

Stack Applications

Stacks are used in many different applications, especially for language processing. We'll consider three.

If we've done our work correctly, you should be able to consider the Stack class (type) just as if it were a native Python class supplying a certain functionality defined by the interface.

Stack Application 1: Bracket Matching

Suppose you have a language in which you use various styles of delimiters (parentheses, braces, brackets, etc): {, [, (,),], }. These can be arbitrarily nested, but each left delimiter must match the corresponding right delimiter.

Which of the following are OK and which are not:

```
[a + (b - {c / d} - e) + f] / 3
a + (b - {c / d) - e) + f] / 3
 a + (b - {c / d} - e) + f / 3
(a + b - {c / d} - e) + f / 3)
```

Design an algorithm to check the "well-formedness" of such expressions.

Bracket Matching

Begin with an empty stack, read the input one character at a time.

1. If input is empty, then if the stack is empty, exit with *success*.
   If input is empty, but the stack is not empty, exit with *failure*.
2. Read a character from input.
   1. If it's a left delimiter, push it onto the stack.
   2. If it's a right delimiter, it must match a corresponding left delimiter atop the stack; if so, pop and go to step 1, else exit with *failure*.
   3. Otherwise, discard the character and go to step 1.

Be able to explain why the algorithm works.

Show the behavior of the algorithm on input:

```
[a + (b - {c / d} - e) + f] / 3
```

Bracket Matching

```
def bracketChecker(symbolString):
    """Check whether parentheses, brackets, and curly
    braces are matched.  Ignore all other characters."""

    stack = Stack()
    for c in symbolString:
        if c == "(" or c == "[" or c == "{":
            stack.push(c)
        elif c == ")" or c == "]" or c == "}":
            if stack.isEmpty():
                return False
            else:
                top = stack.pop()
                # You'll need to write the matches function.
                if not matches(top, c):
                    return False
        else:
            pass
    return True
```

There's a bug in this code. Can you spot it?

Aside: Infix Notation

Our traditional mathematical notation is called *infix*. But it has some shortcomings:

- It requires rather complex *precedence rules* to assign meaning. Consider:

$$a + b * c - d / e$$

- To override the operator precedence requires the addition of parentheses.
- Even though it's infix, there are prefix and postfix operators in the language that have to be handled specially. Name some.
- It works fine for binary operators, but not so well for operators with other arities (numbers of arguments).

Stack Applications 2: Postfix Evaluation

There are several common notations for arithmetic expressions including infix, prefix, postfix. Prefix is sometimes called "Polish notation" and postfix is called "Reverse Polish."

Any mathematical expression can be written into any of the three standard forms:

Infix	$(a + b) * (c + d)$
Prefix	$* + ab + ac$
Postfix	$ab + cd + *$

But prefix and postfix have certain advantages. Such as what?

Stack Applications 2: Postfix Evaluation

There are several common notations for arithmetic expressions including infix, prefix, postfix. Prefix is sometimes called "Polish notation" and postfix is called "Reverse Polish."

Any mathematical expression can be written into any of the three standard forms:

Infix	$(a + b) * (c + d)$
Prefix	$* + ab + ac$
Postfix	$ab + cd + *$

But prefix and postfix have certain advantages. Such as what?

Both are *unambiguous* and postfix is easier to *evaluate* than infix notation. Consider a postfix expression language containing only decimal integers and the binary operators: `+`, `-`, `*`, `/`. Design an algorithm to evaluate expressions in this language.

Infix, Postfix and Prefix

Suppose you have the expression:

$$3 * 4 - 2 * 5 + 7$$

What is the corresponding form in postfix?

Infix, Postfix and Prefix

Suppose you have the expression:

$$3 * 4 - 2 * 5 + 7$$

What is the corresponding form in postfix?

$$3\ 4\ *\ 2\ 5\ *\ -\ 7\ +$$

In prefix?

Infix, Postfix and Prefix

Suppose you have the expression:

$$3 * 4 - 2 * 5 + 7$$

What is the corresponding form in postfix?

$$3\ 4\ *\ 2\ 5\ *\ -\ 7\ +$$

In prefix?

$$+\ -\ *\ 3\ 4\ *\ 2\ 5\ 7$$

Postfix Evaluation

Again, use an initially empty stack and read the input expression from left to right one character (or token) at a time. We assume that the input is a legal non-empty postfix expression.

1. If the input is empty, the answer is on top of the stack.
2. Read a token from the input.
    1. If it's a number, push it onto the stack.
    2. If it's an operator, pop the appropriate number of arguments into variables, perform the indicated operation, and push the result onto the stack.
3. Go to step 1.

Postfix Evaluation

Perform the algorithm for the following input:

```
6 5 2 3 + 8 * + 3 + -
```

How might you add variables to your input language?

Warning: Make sure you note the way that non-commutative operators handle the top two items on the stack.

Think about how you'd design an algorithm for the evaluation of prefix expressions?

A Simple Version

```
def postEval0():
    """Given a postfix expression, evaluate it"""
    symbolString = input("Input a postfix expression: ")
    tokens = symbolString.split()
    stack = Stack()
    for token in tokens:
        # Is the token an integer?
        if token.isdigit():
            stack.push(int(token))
        # otherwise, it must be an operator
        else:
            arg1 = stack.pop()
            arg2 = stack.pop()
            # You'll have to write the applyOp function
            val = applyOp(token, arg1, arg2)
            stack.push(val)
    return stack.pop()
```

Using postEva0

```
>>> from Stack1 import *
>>> postEval0()
Input a postfix expression: 2 3 + 1 5 * -
0
>>> postEval0()
Input a postfix expression: 9 2 * 7 - 4 5 * +
31
```

What are some of the weaknesses of this implementation? How might you correct them? What does it assume about the input?

Stack Application 3: Infix to Postfix

Given the ease of evaluating postfix expressions, it often makes sense to translate infix notation to postfix. This requires dealing with parentheses and operator precedence.

Notice that `a + b * c` yields a different result from `(a + b) * c`.

Consider an infix expression language containing only decimal integers, the binary operators `+`, `-`, `*`, `/`, and parentheses.

Design an algorithm to translate expressions in this language into the appropriate postfix form.

Infix to Postfix Translation

We use the following "precedence chart":

low $\longrightarrow$ high				
$\emptyset$	"(" on stack	$+$ $-$	$*$ $/$	"(" in input

Here $\emptyset$ means the stack is empty. We will need to compare tokens on the input with the top of the stack. We'll assume that the input is well-formed.

As usual, begin with an empty stack. Read the input from left to right one token at a time. In this case, we also have an output stream that is the postfix translation.

Infix to Postfix Translation

1. If the input is empty, pop to output until the stack is empty and exit. The translation will be on the output.
2. Read a token from the input.
    1. If it's a number, output it.
    2. If it's a ")", pop to output until you encounter a "(". Discard both parentheses.
    3. If it's an operator or "(", compare it to the top of the stack.
        1. If precendence of peep is less than the precendence of the input, push input onto the stack.
        2. Otherwise, pop to output until peep holds a lower precedence operator. Then push input onto the stack
3. Go to step 1.

Infix to Postfix Example

Follow the algorithm to translate the following infix expression to postfix:

```
a + b * c + ( d * e + f ) * g
```

Consider how you might translate postfix into infix.

Infix to Postfix

This first part sets up the structures for the loop:

```
import string
def infixToPostfix(infixexpr):
    # First set up a dictionary mapping symbols to
    # precedence.
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    # Split the input into tokens.
    tokenList = infixexpr.split()
    # We'll need a stack and an output stream.
    opStack = Stack()
    outputList = []
```

Infix to Postfix (2)

```
for token in tokenList:
    # Is token a number?
    if token.isdigit():
        outputList.append(token)
    elif token == '(':
        opStack.push(token)
    elif token == ')':
        topToken = opStack.pop()
        while topToken != '(':
            outputList.append(topToken)
            topToken = opStack.pop()
    else:
        while (not opStack.isEmpty()) and \
              (prec[opStack.peek()] >= prec[token]):
            outputList.append(opStack.pop())

        opStack.push(token)

while not opStack.isEmpty():
    outputList.append(opStack.pop())
return ' '.join(outputList)
```

Using infixToPostfix

```
from Stack1 import *
>>> infixToPostfix( "4 + 5 * 6 - 7 / 2" )
'4 5 6 * + 7 2 / -'
>>> infixToPostfix( "( 4 + 5 ) * ( 6 - 7 ) / 2" )
'4 5 + 6 7 - * 2 /'
```

Aside: Adding Variables using a Dictionary

Suppose you wanted to add variables to our language. What might this look like? After entering: `= x 4` and `= y 2`, then evaluating `x y * 2 +` should yield 10.

Store the variable values in a Python dictionary:

```
>>> d = {}        # create an empty dictionary
>>> d["x"] = 4    # associate 4 with the variable name x
>>> d["y"] = 2    # associate 2 with the variable name y
>>> d["x"] * d["y"]  # access associated values and multiply
8
>>> d
{'y': 2, 'x': 4}
>>> d["z"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'z'
>>> d.get('x')
4
>>> d.get('z')
>>>
```

Dictionary Methods

Recall that `d[key]` is used to reference a value in a dictionary, replace the value, or insert a new values.

Here are some other common methods on Python dictionaries:

len(d)	how many items in dictionary d
d.get(k, default)	return value if key in d, else default
d.pop(k, default)	remove key and return value if key in d, else default
list(d.keys())	return a list of the keys
list(d.values())	return a list of the values
list(d.items())	return a list of tuples (key, value)
d.has_key(k)	return boolean if key in d
d.clear()	remove all keys
for k in d:	iterate over keys