

CS313E: Elements of Software Design

Abstract Data Types

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

Last updated: October 4, 2011 at 13:31

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?

What is data? What is a data type? When is a data type “abstract”?

“Raw” *data* is simply information. Typically in a computer system, it’s stored in a huge array of 0’s and 1’s (bits).

A *data type* is simply an interpretation placed on data. Various languages supply particular data types: **bit string**; **machine instruction**; **integer** (signed or unsigned); **rational**; **float**; **character**; **string**; **pointer** (addresses); **list**; many others.

Associated with a data type is a set of operations, though sometimes an operation may be apparently distinct from a data type.

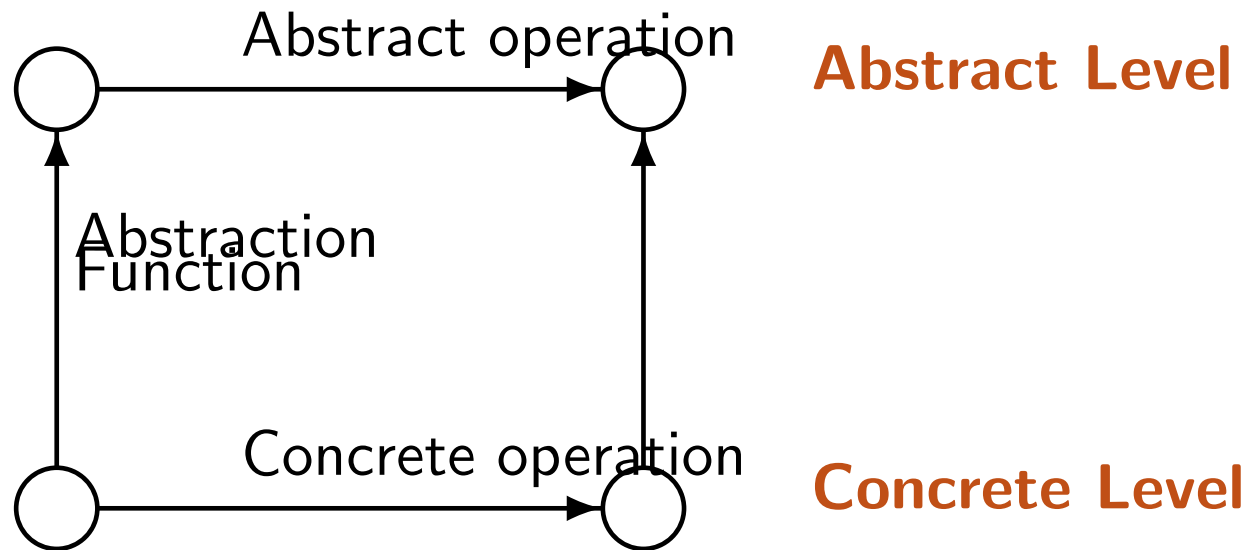
What operations do you associate with the integer data type?

What about operations that take several arguments, whose types may be different? Is there an “object oriented” way of thinking about such things?

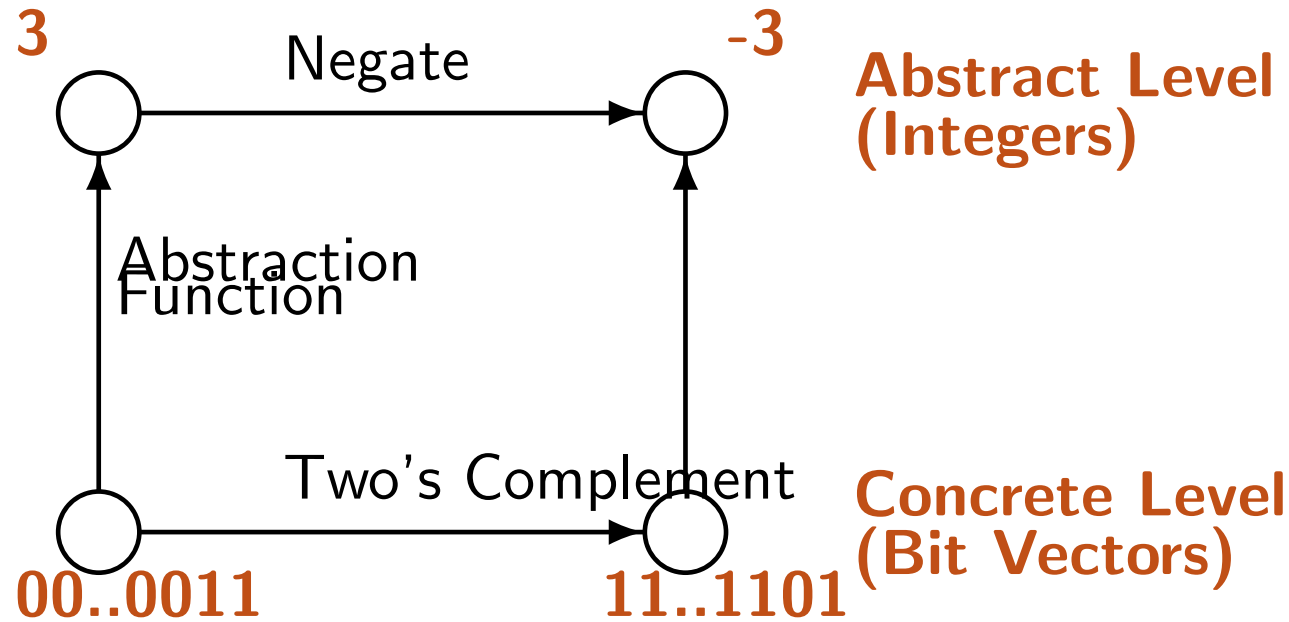
What about operations whose argument(s) can be any of several types? (Overloading, Polymorphism)

Abstract Compared to What?

Notice that some types are provided directly or almost directly by the underlying machine, others by the programming language, others by standard libraries, others by the programmer.



An Example



Notice that such abstractions can be “stacked” to arbitrary depth. I.e., we use **integer** to build other abstractions such as lists, stacks, etc., which are then used to build priority queues, hash tables, and so on.

Abstraction is one of the most powerful and ubiquitous ideas in computer science. It separates the *what* from the *how*.

There are various types of abstraction:

- **Data abstraction:** separation of the logical properties of data from the implementation details.
- **Procedural abstraction:** separation of the logical properties of an action from the implementation details.

There are other types of abstraction, but these are what we need most for ADTs. An ADT involves both data and operations on that data.

Why Abstraction Matters

Abstraction provides modularity (also intelligibility, maintainability, communication, and modifiability).

A **module** is a “black box,” a subsystem with a well-defined interface.

An **interface** is a boundary (“logical firewall”) between:

- a subsystem’s exterior and interior;
- a subsystem’s specification and its implementation.

Modularity is an essential tool in the design and implementation of complex systems.

- Designers can cooperate by working on different modules with minimal cooperation.
- A module's implementation can be modified freely so long as the interface remains unchanged.
- A module can be re-used under any conditions that satisfy the interface specification.

Defining an ADT

An **abstract data type** (ADT) is a class of objects whose logical behavior is defined by a set of values and a set of operations (methods) defined on those values.

An ADT definition specifies:

- an interface (a set of data items and methods), and
- an implementation.

Typically the ADT methods are the only ones that can access the ADT's implementation.

An interface specification is a contract between the implementors and the clients.

What Does an ADT Do?

So how do we know what an ADT is supposed to do? This is the *design* problem.

A different question: how do you *say* what an ADT is supposed to do? This is the *specification* problem.

The specification is a *contract* between the ADT designer (me) and the user (you).

- You can rely on the advertised behavior.
- Anything not advertised isn't guaranteed.
- I'm free to change anything I like, as long as the contract is fulfilled.

An Example: Stack

A **stack** is a LIFO (last in, first out) list with (at least) the following operations: Create, Peek, Pop, Push, IsEmpty. (Such a list is called a *Signature* or the *Interface* to the ADT.)

Create	→ Stack
Push (Stack, ItemType)	→ Stack
Pop (Stack)	→ ItemType
Peek (Stack)	→ ItemType
IsEmpty (Stack)	→ Boolean
Len (Stack)	→ Integer

What happens if you Pop an empty Stack?

Stack Axiomatic Specification

An *axiomatic specification* is a bunch of axioms that you expect to be true of any legal implementation of the ADT.

IsEmpty (Create) = True
IsEmpty (Push (S, i)) = False
Peek (Create) = Error
Peek (Push (S, i)) = i
Pop (Create) = Error
Pop (Push (S, i)) = i

Such axioms might be called *invariants*. Does it make sense to prove them? How would you go about it?

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 self._items == []

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

    def pop(self):
        return self._items.pop()

    def peek(self):
        return self._items[len(self._items) - 1]

    def __len__(self):
        return len(self._items)
```

Implementation Issues

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

As with any other Python data types, you probably want to define some functions that allow displaying the stack.

What would that look like?

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 involve `self`.)

Outside the class definition, you can define your own functions/procedures. These can call methods on visible classes, but use the function 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)           # if I've defined __str__ on Stacks
[ 3 abc ]
>>> 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)
```

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

Parenthesis 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 with the corresponding right delimiter.

Design an algorithm to do this.

Parenthesis 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 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.

Show the behavior of the algorithm on input:

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

Parenthesis Matching

```
def parenthesisChecker(symbolString):
    """Check whether parentheses, brackets, and curly
    braces are matched.  Ignore 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()
                if not matches(top, c):
                    return False
        else:
            pass
    return True
```

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

Stack Applications 2: Postfix Evaluation

There are many possible notations for arithmetic expressions including infix, prefix, postfix. Traditional mathematical notation is a mixture of these.

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 many possible notations for arithmetic expressions including infix, prefix, postfix. Traditional mathematical notation is a mixture of these.

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 we 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, 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 would 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.

Can you design an algorithm for the evaluation of prefix notation?

A Simple Version

```
def postEval0():
    """Given a postfix expression, evaluate it"""
    stack = Stack()
    symbolString = input("Input a postfix expression: ")
    for c in symbolString:
        if c.isdigit():
            stack.push(int(c))
        else:
            arg1 = stack.pop()
            arg2 = stack.pop()
            val = applyOp(c, arg1, arg2)
            stack.push(val)
    return stack.pop()
```

What are some of the weaknesses of this implementation? What does it assume about this input?

Aside: Split a String into Tokens

Suppose you want to split an input sentence into words. You can use the Python `split` string method.

```
>>> sentence = "the cat sat on the mat"
>>> print (sentence.split())
['the', 'cat', 'sat', 'on', 'the', 'mat']
```

If you don't give an argument to `split`, it splits on arbitrary whitespace.

To put a list of strings together, use the `join` method:

```
>>> ' '.join(['the', 'cat', 'sat', 'on', 'the', 'mat'])
'the cat sat on the mat'
```

How could you use this to improve our `postEval0` function?

Stack Applications: 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
		-	/	

We will need to compare tokens on the input with the top of the stack. We'll assume that the input is well-formed.

Again we begin with an empty stack. We 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 precedence of peep is less than the precedence 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.

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

    opStack = Stack()
    outputList = []
    tokenList = infixexpr.split()
```

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 string.join(outputList)
```

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