More Data Abstraction

UW CSE 190p

Summer 2012

Recap of the Design Exercise

- You were asked to design a module – a set of related functions.
- Some of these functions operated on the same data structure
  - a list of tuples of measurements
  - a dictionary associating words with a frequency count
- Both modules had a common general form
  - One function to create the data structure from some external source
  - Multiple functions to query the data structure in various ways
  - This kind of situation is very common

What we've learned so far

- data structure
  - a collection of related data
  - the relevant functions are provided for you
  - Ex: list allows append, sort, etc.
  - What if we want to make our own kind of "list," with its own special operations?

- module
  - a named collection of related functions
  - but shared data must be passed around explicitly
  - What if we want to be sure that only our own special kind of list is passed to each function?

Terms of Art

- Abstraction: Emphasis on exposing a useful interface.
- Encapsulation: Emphasis on hiding the implementation details.
- Information Hiding: The process by which you achieve encapsulation.
- Procedural abstraction: Hiding implementation details of functions
- Data Abstraction: Hiding implementation details of data types

- Overall:
  – Your job is to choose which details to hide and which details to expose.
  – The language's job is to provide you with machinery to get this done

Tools for abstraction

- Functions
- Modules
  - A named set of related functions
  - A namespace mechanism to refer to avoid conflicts
- What else?

Tools for abstraction: Default Values

- As you generalize a function, you tend to add parameters.
- Downsides:
  - A function with many parameters can be awkward to call
  - Existing uses need to be updated

```
def twittersearch(query):
  """Return the responses from the query"""
  url = "http://search.twitter.com/search.json?q=" + query
  remote_file = urllib.urlopen(url)

  raw_response = remote_file.read()

  response = json.loads(raw_response)
  return [tweet["text"] for tweet in response["results"]]



def twittersearch(query, page=1):
  """Return the responses from the query for the given page"""
  resource = "http://search.twitter.com/search.json"
  qs = "?q=" + query + "&page=" + page
  url = resource + qs
  remote_file = urllib.urlopen(url)

  raw_response = remote_file.read()

  response = json.loads(raw_response)
  return [tweet["text"] for tweet in response["results"]]
```

- Data Abstraction, first attempt:
    - Group related functions in a module

Text Analysis

```
def read_words(filename):
    “””Return a dictionary mapping each word in filename to its 
frequency”””
    words = open(filename).read().split()
    wordcounts = {}
    for w in words:
      cnt = wordcounts.setdefault(w, 0)
      wordcounts[w] = cnt + 1
    return wordcounts

def wordcount(wordcounts, word):
    “””Return the count of the given word”””
    return wordcounts[word]

def topk(wordcounts, k=10):
    “””Return top 10 most frequent words”””
    scores_with_words = [(s,w) for (w,s) in wordcounts.items()]
    scores_with_words.sort()
    return scores_with_words[0:k]

def totalwords(wordcounts):
    “””Return the total number of words in the file”””
    return sum([s for (w,s) in wordcounts])
```

```
# program to compute top 10:
wordcounts = read_words(filename)
result = topk(wordcounts, 10)
```

The wordcount dictionary is exposed to the user.

If we want to change our implementation to use a list, we can’t be sure we won’t break their program.

We want to collect all the data we need and all the functions we need and bring them together into one unit.

- Data Abstraction, first attempt:
  - Group related functions in a module
  - Cons: Doesn’t achieve encapsulation

- Other ideas?

```
my_new_datatype = {}
my_new_datatype[“read_words”] = read_words
my_new_datatype[“topk”] = topk
my_new_datatype[“wordcounts”] = {}

wordcounts = my_new_datatype[“read_words”](“somefile.txt”)
```

We’re no better off. We have everything lumped into one place, but the different functions can’t communicate with each other.

For example, the read_words function can’t pass the wordcounts dictionary directly to the topk function. So the user still has access, and we don’t trust users.

- Data Abstraction, first attempt:
  - Group related functions in a module
  - Cons: Doesn’t achieve encapsulation
- Data Abstraction, second attempt:
  - Group related functions and data into a data structure
  - Didn’t really help, and made the syntax more difficult.
- Other ideas?

```
wordcounts = {}
```

Global variable?

```
def read_words(filename):
    """Return a dictionary mapping each word to its frequency"""
    words = open(filename).read().split()
    for w in words:
      cnt = wordcounts.setdefault(w, 0)
      wordcounts[w] = cnt + 1


def wordcount(word):
    """Return the count of the given word"""
    return wordcounts[word]

def topk(k=10):
    """Return top 10 most frequent words"""
    scores_with_words = [(s,w) for (w,s) in wordcounts.items()]
    scores_with_words.sort()
    return scores_with_words[0:k]

def totalwords():
    """Return the total number of words in the file"""
    return sum([s for (w,s) in wordcounts])
```

```
# program to compute top 10:

read_words(filename)
result = topk(10)
```

We’re no longer passing wordcounts around explicitly!

Problem solved?

- Data Abstraction, first attempt:
    - Group related functions in a module
    - We have to rely on the user to pass the right values around
    - We can't be sure we won't break their program if we change our implementations
- Data Abstraction, second attempt:
    - Group related functions and data into a data structure
    - Didn't really help, and made the syntax more difficult.
- Data Abstraction, third attempt:
    - Use a global variable to manage communication between functions
    - Avoids handing off values to that untrustworthy user
    - But pollutes the global namespace
    - And the user still handles our dictionary
- Other ideas?

```
ourdata = object()
```

Global variable, but we can change how its used.

```
def read_words(filename):
    """Return a dictionary mapping each word to its frequency"""
    ourdata.wordcounts = {}
    words = open(filename).read().split()
    for w in words:
      cnt = ourdata.wordcounts.setdefault(w, 0)
      ourdata.wordcounts[w] = cnt + 1


def wordcount(word):
    """Return the count of the given word"""
    return ourdata.wordcounts[word]

def topk(k=10):
    """Return top 10 most frequent words"""
    scores_with_words = [(s,w) for (w,s) in ourdata.wordcounts.items()]
    scores_with_words.sort()
    return scores_with_words[0:k]

def totalwords():
    """Return the total number of words in the file"""
    return sum([s for (w,s) in ourdata.wordcounts])
```

- Data Abstraction, first attempt:
  - Group related functions in a module
  - We have to rely on the user to pass the right values around
  - We can’t be sure we won’t break their program if we change our implementations
- Data Abstraction, second attempt:
  - Group related functions and data into a data structure
  - Didn’t really help, and made the syntax more difficult.
- Data Abstraction, third attempt:
  - Use a global variable to manage communication between functions
  - Avoids handing off values to that untrustworthy user
  - But pollutes the global namespace
  - And the user still handles our dictionary
- Data Abstraction, fourth attempt:
  - Use a global variable, but make it generic so the user doesn’t see that we’re using a dictionary, a list, or whatever
  - Still pollutes the global namespace
  - The user still might get their grubby paws on our implementation

Classes

- A class is like a module: it provides a namespace for a set of functions
- A class is like a function: it generates a local scope for variables that won’t be seen outside of the class
- A class can like a data structure: it generates a storage area for data that you can retrieve later.

Special syntax

A reference to shared state

```
class WordCounts:
   def read_words(self, filename):
      """Return a dictionary mapping each word to its frequency"""
      words = open(filename).read().split()
      self.wordcounts = {}
      for w in words:
        cnt = self.wordcounts.setdefault(w, 0)
        self.wordcounts[w] = cnt + 1

   def wordcount(self, word):
      """Return the count of the given word"""
      return self.wordcounts[word]

   def topk(self, k=10):
      """Return top 10 most frequent words"""
      scores_with_words = [(s,w) for (w,s) in self.wordcounts.items()]
      scores_with_words.sort()
      return scores_with_words[0:k]

   def totalwords(wordcounts):
      """Return the total number of words in the file"""
      return sum([s for (w,s) in self.wordcounts])
```

```
# program to compute top 10:

wc = WordCounts()
wc.read_words(filename)
result = wc.topk(10)
```

```
# program to compute top 10:

wc = WordCounts()
wc.read_words(filename)


result = wc.topk(10)
```

$$\texttt{result = }\underbrace{\texttt{WordCounts}}_{\text{A namespace, like a module}}\texttt{.}\underbrace{\texttt{topk}}_{\text{A function with two arguments}}\texttt{(}\overbrace{\texttt{wc}}^{\text{The shared state}}\texttt{, 10)}$$

Quantitative Analysis

```
def read_measurements(filename):
    """Return a dictionary mapping column names to data. Assumes
the first line of the file is column names."""
    datafile = open(filename)
    rawcolumns = zip(*[row.split() for row in datafile])
    columns = dict([(col[0], col[1:]) for col in rawcolumn
    return columns

def tofloat(measurements, columnname):
    """Convert each value in the given iterable to a float"""
    return [float(x) for x in measurements[columnname]]

def STplot(measurements):
    """Generate a scatter plot comparing salinity and temperature"""
    xs = tofloat(measurements, "salt")
    ys = tofloat(measurements, "temp")
    plt.plot(xs, ys)
    plt.show()

def minimumO2(measurements):
    """Return the minimum value of the oxygen measurement"""
    return min(tofloat(measurements, "o2"))
```

```
class Measurements:

   def read_measurements(self, filename):
        “””Return a dictionary mapping column names to data. Assumes
   the first line of the file is column names.”””
        datafile = open(filename)
        rawcolumns = zip(*[row.split() for row in datafile])
        self.columns = dict([(col[0], col[1:]) for col in rawcolumn


   def tofloat(self, columnname):
        “””Convert each value in the given iterable to a float”””
        return [float(x) for x in self.columns[columnname]]

  def STplot(self):
       “””Generate a scatter plot comparing salinity and temperature”””
       xs = tofloat(self.columns, “salt”)
       ys = tofloat(self.columns, “temp”)
       plt.plot(xs, ys)
       plt.show()

  def minimumO2(self):
       “””Return the minimum value of the oxygen measurement”””
       return min(tofloat(self.columns, “o2”))
```

```
mm = Measurements()
mm.read_measurements(filename)
result = mm.Stplot()
```

We now come to the decisive step of mathematical abstraction: we forget about what the symbols stand for. ...[The mathematician] need not be idle; there are many operations which he may carry out with these symbols, without ever having to look at the things they stand for.

Hermann Weyl, The Mathematical Way of Thinking

Abstraction and encapsulation are complementary concepts: abstraction focuses on the observable behavior of an object... encapsulation focuses upon the implementation that gives rise to this behavior... encapsulation is most often achieved through information hiding, which is the process of hiding all of the secrets of object that do not contribute to its essential characteristics.

Grady Booch