Recall the design exercise

• We created a module or library: a set of related functions
• The functions operated on the same data structure
  – a dictionary associating words with a frequency count
• The module contained:
  – A function to create the data structure
  – Functions to query the data structure
  – We could have added functions to modify the data structure
Two types of abstraction

Abstraction: Ignoring/hiding some aspects of a thing
- In programming, ignore everything except the specification or interface
- The program designer decides which details to hide and to expose

Procedural abstraction:
- Define a procedure/function specification
- Hide implementation details

Data abstraction:
- Define what the datatype represents
- Define how to create, query, and modify
- Hide implementation details of representation and of operations
  - Also called “encapsulation” or “information hiding”
def abs(x):
    if val < 0:
        return -1 * val
    else:
        return 1 * val

def abs(x):
    if val < 0:
        result = -val
    else:
        result = val
    return result

def abs(x):
    return math.sqrt(x * x)

We only need to know how to USE abs. We do not need to know how abs is IMPLEMENTED.
Data abstraction

• Describing word counts:
  – “dictionary mapping each word in filename to its frequency (raw count) in the file, represented as an integer”
  – “WordCounts”

• Which do you prefer? Why?

• Hint: This must appear in the doc string of every function related to the word count! Ugh!

In HW5 we used terms like *PollsterPredictions* in doc strings as shorthand for “A dictionary mapping Pollster to a dictionary mapping States to floats”
Review: Using the Graph class in networkx

```python
import networkx as nx

# module name alias

# Create a Graph

g = nx.Graph()

g = Graph()
g.add_node(1)
g.add_node(2)
g.add_node(3)
g.add_edge(1, 2)
g.add_edge(2, 3)

print(g.nodes())
print(g.edges())
print(g.neighbors(2))
```

Graph and DiGraph are now available in the global namespace.
Representing a graph

• A graph consists of:
  – nodes/vertices
  – edges among the nodes

• Representations:
  – Set of edge pairs
    • (a, a), (a, b), (a, c), (b, c), (c, b)
  – For each node, a list of neighbors
    • { a: [a, b, c], b: [c], c: [b] }
  – Matrix with boolean for each entry

\[
\begin{array}{ccc}
  \text{a} & \text{b} & \text{c} \\
  \text{a} & \checkmark & \checkmark & \checkmark \\
  \text{b} & & & \checkmark \\
  \text{c} & & \checkmark & \\
\end{array}
\]
def read_words(filename):
    """Return dictionary mapping each word in filename to its frequency."""
    wordfile = open(filename)
    word_list = wordfile.read().split()
    wordfile.close()
    wordcounts_dict = {}
    for word in word_list:
        count = wordcounts_dict.setdefault(word, 0)
        wordcounts_dict[word] = count + 1
    return wordcounts_dict

def get_count(wordcounts_dict, word):
    """Return count of the word in the dictionary. """
    return wordcounts_dict.get(word, 0)

def topk(wordcounts_dict, k=10):
    """Return list of (count, word) tuples of the top k most frequent words."""
    counts_with_words = [((c, w) for (w, c) in wordcounts_dict.items())]
    counts_with_words.sort(reverse=True)
    return counts_with_words[0:k]

def total_words(wordcounts_dict):
    """Return the total number of words."""
    return sum(wordcounts_dict.values())

# client program to compute top 5:
w_c_dict = read_words(filename)
result = topk(w_c_dict, 5)
Problems with the implementation

• The `wc_dict` dictionary is exposed to the client: the user might corrupt or misuse it.
• If we change our implementation (say, to use a list of tuples), it may break the client program.

We prefer to
  – Hide the implementation details from the client
  – Collect the data and functions together into one unit

```python
# client program to compute top 5:
wc_dict = read_words(filename)
result = topk(wc_dict, 5)
```
Datatypes and Classes

• A **class** creates a namespace for:
  – Variables to hold the data
  – Functions to create, query, and modify
    • Each function defined in the **class** is called a **method**
      – Takes “**self**” (a value of the **class** type) as the first argument

• A **class** **defines a datatype**
  – An **object** is a value of that type
  – Comparison to other types:
    • \( y = 22 \)
      – Type of \( y \) is \texttt{int} , value of \( y \) is 22
    • \( g = \texttt{nx.Graph}() \)
      – Type of \( g \) is \texttt{Graph} , value of \( g \) is the object that \( g \) is bound to
      – Type is the **class**, value is an **object** also known as an instantiation or **instance** of that type
def read_words(filename):
    """Return dictionary mapping each word in filename to its frequency."""
    wordfile = open(filename)
    word_list = wordfile.read().split()
    wordfile.close()
    wordcounts_dict = {}
    for word in word_list:
        count = wordcounts_dict.setdefault(word, 0)
        wordcounts_dict[word] = count + 1
    return wordcounts_dict

def get_count(wordcounts_dict, word):
    """Return count of the word in the dictionary. """
    return wordcounts_dict.get(word, 0)

def topk(wordcounts_dict, k=10):
    """Return list of (count, word) tuples of the top k most frequent words."""
    counts_with_words = [(c, w) for (w, c) in wordcounts_dict.items()]
    counts_with_words.sort(reverse=True)
    return counts_with_words[0:k]

def total_words(wordcounts_dict):
    """Return the total number of words."""
    return sum(wordcounts_dict.values())
Text analysis, as a class

```python
class WordCounts:
    """Represents the words in a file."""
    # Internal representation:
    # variable wordcounts_dict is a dictionary mapping a word its frequency

def read_words(self, filename):
    """Populate a WordCounts object from the given file"
    word_list = open(filename).read().split()
    self.wordcounts_dict = {}
    for w in word_list:
        self.wordcounts_dict.setdefault(w, 0)
        self.wordcounts_dict[w] += 1

def get_count(self, word):
    """Return the count of the given word"
    return self.wordcounts_dict.get(word, 0)

def topk(self, k=10):
    """Return a list of the top k most frequent words in order"
    scores_and_words = [(c,w) for (w,c) in self.wordcounts_dict.items()]
    scores_and_words.sort(reverse=True)
    return score_and_words[0:k]

def total_words(self):
    """Return the total number of words in the file"
    return sum(self.wordcounts_dict.values())
```

Each function in a class is called a method. Its first argument is of the type of the class.

- The type of `self` is `WordCounts`
- The type of `wc` is `WordCounts`
- `topk` takes 2 arguments
- `read_words` does not return a value; it mutates `self`
- `get_count` queries a `WordCounts` object
- `total_words` modifies a `WordCounts` object
- `topk` queries a `WordCounts` object
- `read_words` modifies a `WordCounts` object

The namespace of a `WordCounts` object:

```
wordcounts_dict
read_words
get_count
topk
total_words
dict
```
# client program to compute top 5:

```python
wc = WordCounts()
wc.read_words(filename)

result = wc.topk(5)
```

A namespace, like a module (the name of the class)
A function that takes two arguments

A value of type `WordCounts`

Two equivalent calls

But no one does it this way! Use the first approach!

Weird constructor: it does not do any work
You have to call a mutator immediately afterward
class WordCounts:
    """Represents the words in a file.""
    # Internal representation:
    # variable wordcounts_dict is a dictionary mapping a word its frequency
    def __init__(self, filename):
        """Create a WordCounts object from the given file""
        words = open(filename).read().split()
        self.wordcounts_dict = {}
        for w in words:
            self.wordcounts_dict.setdefault(w, 0)
            self.wordcounts_dict[w] += 1
    def get_count(self, word):
        """Return the count of the given word""
        return self.wordcounts_dict.get(word, 0)
    def topk(self, k=10):
        """Return a list of the top k most frequent words in order""
        scores_and_words = [(c, w) for (w, c) in self.wordcounts_dict.items()]
        scores_and_words.sort(reverse=True)
        return scores_and_words[0:k]
    def total_words(self):
        """Return the total number of words in the file""
        return sum([c for (w, c) in self.wordcounts_dict])
Alternate implementation

class WordCounts:
    """Represents the words in a file.""
    # Internal representation:
    # variable words_list is a list of the words in the file

    def __init__(self, filename):
        """Create a WordCounts object from the given file""
        self.words_list = open(filename).read().split()

    def get_count(self, word):
        """Return the count of the given word""
        return self.words_list.count(word)

    def topk(self, k=10):
        """Return a list of the top k most frequent words in order""
        scores_with_words = [(self.get_count(w), w) for w in set(self.words_list)]
        scores_with_words.sort(reverse=True)
        return scores_with_words[0:k]

    def total_words(self):
        """Return the total number of words in the file""
        return len(self.words_list)

# client program to compute top 5:
wc = WordCounts(filename)
result = wc.topk(5)

Exact same program!