Data Abstraction

UW CSE 160

Winter 2017

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
  - a list of tuples of measurements
- Each 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”

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?

(This must appear in the documentation string of every function related to field measurements!)

Review:
Using the Graph class in networkx

```
import networkx as nx
```

module name

alias

```
g = nx.Graph()
```

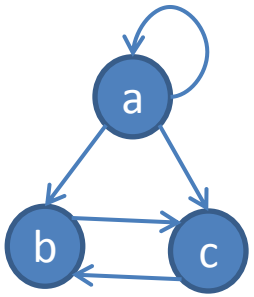
```
from networkx import Graph, DiGraph
```

Graph and DiGraph are now available in the global namespace

```
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)
```

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



	a	b	c
a	✓	✓	✓
b			✓
c		✓	

Text analysis module

(group of related functions)

representation = dictionary

```
# program to compute top 5:
wordcounts = read_words(filename)
result = topk(wordcounts, 5)
```

```
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())
```

Problems with the implementation

```
# program to compute top 5:
wordcounts = read_words(filename)
result = topk(wordcounts, 5)
```

- The **`wordcounts`** 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

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:
    - Type is **`int`**, value is 22
    - **`g = nx.Graph()`**
    - Type of **`g`** is **`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

Text analysis module

(group of related functions)

representation = dictionary

```
# program to compute top 5:
wordcounts = read_words(filename)
result = topk(wordcounts, 5)
```

```
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

The type of **wc** is **WordCounts**

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

topk takes 2 arguments

```
class WordCounts:
  """Represents the words in a file."""
  # Internal representation:
  # variable wordcounts is a dictionary mapping words to their frequency

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

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

  def topk(self, k=10):
    """Return a list of the top k most frequent words in order"""
    scores_with_words = [(c,w) for (w,c) in self.wordcounts.items()]
    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 sum(self.wordcounts.values())
```

The type of **self** is **WordCounts**

read_words does not return a value; it mutates **self**

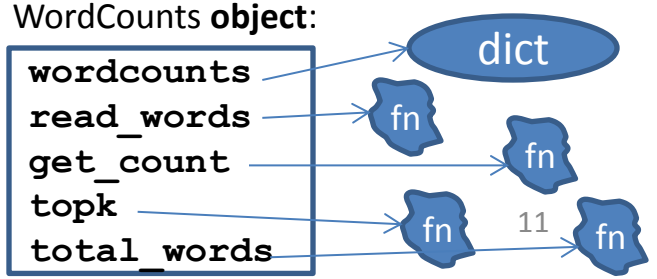
Defines a class (a datatype) named WordCounts

Modifies a WordCounts object

Queries a WordCounts object

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

The namespace of a WordCounts **object**:



```
# program to compute top 5:

wc = WordCounts()
wc.read_words(filename)

result = wc.topk(5)
```

Weird constructor: it does not do any work

You have to call a mutator immediately afterward

A value of type WordCounts

Two equivalent calls

```
result = WordCounts.topk(wc, 5)
```

A namespace, like a module (the name of the class)

A function that takes two arguments

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

Class with constructor

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

The constructor now needs a parameter

```
class WordCounts:
  """Represents the words in a file."""
  # Internal representation:
  # variable wordcounts is a dictionary mapping words to their frequency

  def __init__(self, filename):
    """Create a WordCounts object from the given file"""
    words = open(filename).read().split()
    self.wordcounts = {}
    for w in words:
      self.wordcounts.setdefault(w, 0)
      self.wordcounts[w] += 1

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

  def topk(self, k=10):
    """Return a list of the top k most frequent words in order"""
    scores_with_words = [(c,w) for (w,c) in self.wordcounts.items()]
    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 sum([c for (w,c) in self.wordcounts])
```

__init__ is a special function, a "constructor"

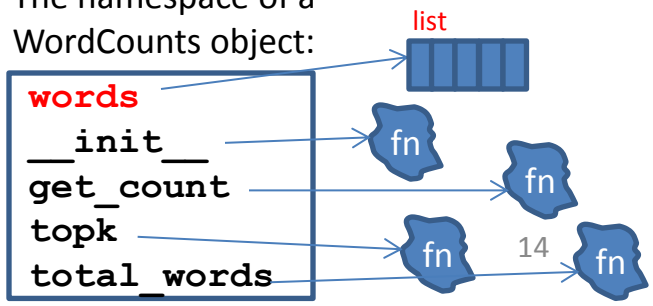
Alternate implementation

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

Exact same program!

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

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

  def get_count(self, word):
    """Return the count of the given word"""
    return self.words.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)]
    scores_with_words.sort(reverse=True)
    return scores_with_words[0:k]

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

The namespace of a WordCounts object:



Quantitative analysis

```
# Program to plot
mydict = read_measurements(filename)
STplot(mydict)
```

```
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"))
```

Quantitative analysis, as a class

```
# Program to plot
mm = Measurements()
mm.read_measurements(filename)
mm.STplot()
```

```
class Measurements:
  """Represents a set of measurements in UWFORMAT."""

  def read_measurements(self, filename):
    """Populate a Measurements object from the given file.
       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])
    return columns

  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"))
```

Quantitative analysis, with a constructor

```
# Program to plot
mm = Measurements(filename)
mm.STplot()
```

```
class Measurements:
  """Represents a set of measurements in UWFORMAT."""

  def __init__(self, filename):
    """Create a Measurements object from the given file.
       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"))
```