Data Abstraction

Rob Thompson

UW CSE 160

Winter 2021

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”

Review: Procedural Abstraction

```
def abs(x):
  if x < 0:
    return -1 * x
  else:
    return 1 * x
```

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

```
def abs(x):
  if x < 0:
    return -x
  else:
    return x
```

```
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.

Review:
Using the Graph class in networkx

```
import networkx as nx
```

module name

alias

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

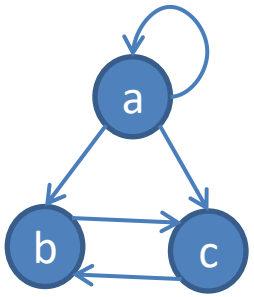
```
from networkx import Graph, DiGraph
```

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(list(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

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

Aside: setdefault

```
def read_words(filename):
 """Given a filename, return a dictionary mapping each word
in filename to its frequency in the file"""
    wordfile = open(filename)
    worddata = wordfile.read()
    word_list = worddata.split()
    wordfile.close()
    wordcounts_dict = {}
    for word in word_list:
        if word in wordcounts_dict:
            wordcounts_dict[word] = wordcounts_dict[word] + 1
        else:
            wordcounts_dict[word] = 1
    return wordcounts_dict
```

This "default" pattern is so common, there is a special method for it.

setdefault

```
def read_words(filename):
 """Given a filename, return a dictionary mapping each
word in filename to its frequency in the file"""
    wordfile = open(filename)
    worddata = wordfile.read()
    word_list = worddata.split()
    wordfile.close()
    wordcounts_dict = {}
    for word in word_list:
        count = wordcounts_dict.setdefault(word, 0)
        wordcounts_dict[word] = count + 1
    return wordcounts_dict
```

This “default” pattern is so common, there is a special method for it.

setdefault

```
for word in word_list:
    if word in wordcounts_dict:
        wordcounts_dict[word] = wordcounts_dict[word] + 1
    else:
        wordcounts_dict[word] = 1
```

VS:

```
for word in word_list:
    count = wordcounts_dict.setdefault(word, 0)
    wordcounts_dict[word] = count + 1
```

`setdefault(`*key*[, *default*]**`)`**

- If *key* is in the dictionary, return its value.
- If *key* is NOT present, insert *key* with a value of *default*, and return *default*.
- If *default* is not specified, the value **`None`** is used.

get

```
def get_count(wordcounts_dict, word):
    """Return count of the word in the dictionary. """
    if word in wordcounts_dict:
        return wordcounts_dict[word]
    else:
        return 0
```

VS:

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

`get(`*key*[, *default*]**`)`**

- Return the value for *key* if *key* is in the dictionary, else *default*.
- If *default* is not given, it defaults to None, so that this method never raises a KeyError

See in CSE 160 Syntax examples:
https://courses.cs.washington.edu/courses/cse160/21wi/computing/syntax_examples.html

Problems with the implementation

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

- The **`wc_dict`** dictionary is exposed to the client: the client 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:
    - **`y = 22`**
      - Type of **`y`** is **`int`**, value of **`y`** 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

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

```
# client 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_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())
```

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

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

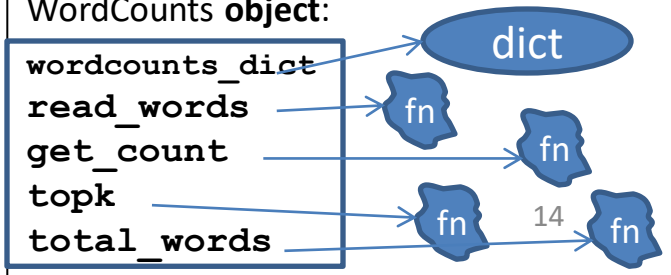
Defines a class (a datatype) named WordCounts

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**:



```
# client 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

```
# client 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_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])
```

__init__ is a special function, a "constructor"

Alternate implementation

```
# client 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_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)
```