Data Abstraction

Ruth Anderson

UW CSE 160

Autumn 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

1) Procedural abstraction - Already covered
2) Data abstraction - Topic for today!
Review: Procedural Abstraction

• Define a function specification that describes how to use the function
  – Aside: a function is sometimes called a “procedure”
• Hide implementation details from the user/client
• Examples:
  – You know how to USE the functions sorted and abs
  – You do not know how these functions are IMPLEMENTED
Review: Procedural Abstraction

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

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

```python
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

• Define what the datatype represents
• Define how to create, query, and modify
• Hide implementation details of representation and of operations from the user/client

• Examples:
  – You know how to USE the datatypes int, float, list, dict, set
  – You do not know how these are actually stored in memory or how operations on them are IMPLEMENTED
    • How is .sort() implemented on lists?
    • How is .items() implemented for dictionaries?
    • How is .remove() implemented for sets?
Types and Classes

• Built in types like `int`, `float`, `list`, `dict`, `set` are examples of Data Abstraction

• Python provides a way for users to essentially create their own types by defining a `class`
  
  – You can then create `instances` of that `class` or `objects`

• You have already used a `class` in the `networkx module`!
Review:

Using the Graph class in networkx

```python
import networkx as nx

# Module name alias

g = nx.Graph()

from networkx import Graph, DiGraph

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)))
```

Aside: With this way of importing you need to use: `nx.` before referring to something in `networkx`. With the approach below, you do not.

Graph and DiGraph are the names of classes

Good style for Python class names use `CapWords` (sometimes called CamelCase)

This is a client program that uses the Graph class. The client does not need to know how the class is implemented.
Constructors, Instances & Objects

```python
from networkx import Graph, DiGraph

# Creating a graph object

# Create an empty graph (undirected, as default)
g = Graph()
g.add_node(1)
g.add_edge(1, 2)
...  
print(g.nodes())
print(g.edges())

# Creating instances

rj = Graph()
rj.add_node("Romeo")
...

practice_graph = Graph()
practice_graph.add_node("A")
...  
```

Graph and DiGraph are the names of classes

Graph() is the constructor for the Graph class

g is an instance of the Graph class
We also say that g is a Graph object

rj and practice_graph are also instances of the Graph class or Graph objects
Methods and State

```python
from networkx import Graph, DiGraph

# Create a graph g
g = Graph()
g.add_node(1)
g.add_edge(1, 2)
...
print(g.nodes())
print(g.edges())

# Create another graph rj
rj = Graph()
rj.add_node("Romeo")
...

# Create a practice graph
practice_graph = Graph()
practice_graph.add_node("A")
...
```

dd_node(), add_edge(), nodes() and edges() are methods of the Graph class.

The nodes and edges of Graph object `g` are also known as its state.

Each object has different state.

`rj` and `practice_graph` each have their own nodes and edges with can be different from the nodes and edges in other objects.
from networkx import Graph, DiGraph

g = Graph()
g.add_node(1)
g.add_edge(1, 2)
...
print(g.nodes())
print(g.edges())

rj = Graph()
rj.add_node("Romeo")
...

practice_graph = Graph()
practice_graph.add_node("A")
...
Representing a graph

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

• If you were implementing the **Graph** class, how would you store the nodes and the edges?
  – Would you use **lists**, **sets**, **dicts**?
  – How would you implement methods like:
    • `edges()`
    • `neighbors(a)`
Representing a graph

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

• Possible Representations:
  – set of nodes and set of edges
    • nodes are \{a, b, c\}
    • edges are \{(a, b), (b, c)\}
  – dict with node for key, a list of neighbors as value
    • \{a: [b], b: [a, c], c: [b]\}
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:
wc_dict = read_words(filename)
result = topk(wc_dict, 5)
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
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.
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.
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:
Problems with the implementation

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

```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 or “fields” 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
    • \( g = \text{nx.Graph}() \)
      – 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
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())
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 scores_and_words[0:k]

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

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

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(self.wordcounts_dict.values())

The type of \texttt{self} is \texttt{WordCounts}.

\texttt{get\_count} does not return a value; it mutates \texttt{self}.

\texttt{topk} takes 2 arguments.

The type of \texttt{wc} is \texttt{WordCounts}.

Each function in a class is called a \textit{method}. Its first argument is of the type of the class.
# client program to compute top 5:

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

```python
result = wc.topk(5)
```

```python
result = WordCounts.topk(wc, 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
- 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 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!