CS 2316 Data Manipulation for Engineers

Data Representation

Christopher Simpkins
chris.simpkins@gatech.edu

Data Representation[1]

Today we'll do an exercise in representing data, examining:

- records as lists,
- lists of lists as databases,
- records as dictionaries,
- dictonaries as databases,

and culminating in an introduction to objects.

[1] This lecture is based on Chapter 1 of Mark Lutz's Programming Python, which is available as a free sample on the book's web site.

Records

A record is a structured data type, a metaphor that maps a human concept to a computer language representation. For example, we could say that a student is a

- student id
- name
- list of grades

Each student record would contain these three *fields*. The simplest way to represent a record like this in Python is with a list:

```
>>> chris = ['gtg034t', 'Chris', [100, 80, 90]]
>>> chris
['gtg034t', 'Chris', [100, 80, 90]]
```

Notice that we can nest lists within lists.

Records as Lists

With our list-based records, getting a field means indexing into the list:

```
>>> chris_id = chris[0]
>>> chris_id
'gtg034t'
```

We index into sub-lists as well:

```
>>> chris_exam1 = chris[2][0]
>>> chris_exam1
100
```

Notice that we give meaningful names to the variables we store the values in, e.g., `chris_exam1`, but the indexed records themselves don't have meanful names, e.g., it's not obvious that `chris[2][0]` is Chris's Exam 1 score.

Field Labels for List-based Records

We can create contstants to improve readability:

```
>>> ID, NAME, GRADES = range(3)
>>> ID, NAME, GRADES
(0, 1, 2)
```

Note that we name variables that are meant to be constant in ALL CAPS. This is a hint to the programmer that we shouldn't reassign these variables, but Python won't prevent it.
Now we can get fields from list-based student records more readably:

```
>>> chris[ID]
'gtg034t'
```

But it's still brittle. We have to maintain the mapping from index constants to fields. If we re-arrange our lists we have to update the field constants. We'll revisit this idea in a minute...

Databases as Lists of Lists

If we had multiple students we could store them in a list:

```
>>> eva = ['eluator3', 'Eva', [100, 100, 100]]
>>> alyssa = ['ahacker3', 'Alyssa', [100, 100, 100]]
>>> students = [chris, eva, alyssa]
```

We can then use familiar list operations to operate on the student database:

```
>>> for stud in students:
...     stud[GRADES][1] += 10 # give each student 10 points on exam 2
>>> exam2avg = sum([stud[GRADES][1] for stud in students]) /
    len(students)
>>> exam2avg
103.33333333333333
```

Eventually, maintaining the mapping between indexes and their meanings will become hard to maintain. Dictionaries are a better approach.

Records as Dictionaries

Dictonaries maintain an efficient mapping between keys and their values. A dictonary can express a student record more naturally:

```
>>> chris = {'id': 'gtg034t', 'name': 'Chris',
...          'grades': {'exam1': 100, 'exam2': 80, 'exam3': 90}}
>>> import pprint as pp
>>> pp.pprint(chris)
{'grades': {'exam1': 100, 'exam2': 80, 'exam3': 90},
 'id': 'gtg034t',
 'name': 'Chris'}
```

Notice that we're nesting dictionaries just like we nested lists.

Databases as Dictionaries of Dictionaries

As we'll learn later, database records must have a unique key. Dictionaries employ the same idea:

```
>>> students = {}
>>> students[eva['id']] = eva
>>> students[alyssa['id']] = alyssa
>>> students[chris['id']] = chris
>>> pp.pprint(students)
{'ahacker3': {'grades': {'exam1': 100, 'exam2': 100, 'exam3': 100},
              'id': 'ahacker3',
              'name': 'Alyssa'},
 [...]
```

We can use comprehensions to do queries. For example, to get the names of all students who scored higher than 90 on exam2:

```
>>> [students[stud]['name'] for stud in students.keys()
...                         if students[stud]['grades']['exam2'] > 90]
['Alyssa', 'Eva']
```

Classes and Objects

Dictionaries are an improvement, but classes provide the cleanest representation. See student.py for a Student class:

```
class Student():
    def __init__(self, id, name, exams):
        self.id = id
        self.name = name
        self.exams = exams

    def exam(self, number):
        return self.exams[number - 1]
```

Using classes is much clearer:

```
>>> from classes.student import Student
>>> chris = Student('gtg034t', 'Chris Rock', [100, 80, 90])
>>> eva = Student('eluator3', 'Eva Luator', [100, 100, 100])
>>> alyssa = Student('ahacker3', 'Alyssa Hacker', [100, 100, 100])
>>> students = [chris, eva, alyssa]
>>> [stud.name for stud in students if stud.exam(2) > 90]
['Eva Luator', 'Alyssa Hacker']
```

We'll learn about classes and objects in detail next lecture.