CSC 108H: Introduction to Computer Programming

Summer 2012

Marek Janicki

Administration

- Midterm is next week.
  - Room assignments will be posted on Piazza/website tomorrow.
- Assignment typos.
  - Should be fixed now.
- I just realised Monday after the midterm is a holiday.
  - So the assignment deadline has been extended to allow for more help centre access.
- Also, no office hours next Friday or Monday (after the midterm).
  - Friday office hours are moved to next Wednesday.

List Review

- != and == use element by element comparison.
- Lists can be nested.
  - We then use multiple pairs of brackets to index into nested lists.
  - The brackets closes to the list name are the first list, and subsequent brackets go into the nesting one at a time.
  - `list_name[i][j][k]`
- Tuples are non-mutable lists.

List Review

- != and == use element by element comparison.
- Lists can be nested.
  - We then use multiple pairs of brackets to index into nested lists.
  - The brackets closes to the list name are the first list, and subsequent brackets go into the nesting one at a time.
  - `list_name[i][j][k]`
- Tuples are non-mutable lists.

List Review

- != and == use element by element comparison.
- Lists can be nested.
  - We then use multiple pairs of brackets to index into nested lists.
  - The brackets closes to the list name are the first list, and subsequent brackets go into the nesting one at a time.
  - `list_name[i][j][k]`
- Tuples are non-mutable lists.

List Review

- != and == use element by element comparison.
- Lists can be nested.
  - We then use multiple pairs of brackets to index into nested lists.
  - The brackets closes to the list name are the first list, and subsequent brackets go into the nesting one at a time.
  - `list_name[i][j][k]`
- Tuples are non-mutable lists.

Evaluate the Expressions

- `a = [9, 2, 5]`
- `a == [9, 2, 5]`

- `a0 = (9, 2, 5)`
- `a != a0`

- `b = [a, a0, a]`
- `b[1][2] == a[2]`

- a[0] = 10
- b[1][0]

- b[0][0]

- b[1][0] = 11

Evaluate the Expressions

- `a = [9, 2, 5]`
- `a == [9, 2, 5]`

  `True`
- `a0 = (9, 2, 5)`
- `a != a0`

  `True`
- `b = [a, a0, a]`
- `b[1][2] == a[2]`

  `True`

- a[0] = 10
- b[1][0]

  9
- b[2][0]

  10
- b[1][0] = 11

  AssignmentError

While Review

- While loops syntax:

```
while condition:
    block
```

- The block is repeated as long as the condition is true.
- The block may never be executed.
- Every for loop may be rewritten as a while, but the reverse is not true.

How many times do these execute?

```
while True:
   print True
```

```
i = 15
while i > 0:
    i -= 2
```

```
i = 15
while i < 0:
    i-= 2
```

How many times do these execute?

```
while True:
    print True
```

- Infinitely many.

```
i = 15
while i > 0:
    i -= 2
```

- Eight

```
i = 15
while i < 0:
    i-= 2
```

- Never

File Review.

- Files can be opened, closed and written to.
- Can be opened in three modes - `'r', 'w', 'a'`
  - `'r'` allows a file to be read.
  - `'w'` - writes to a file and blanks it if there are things in it.
  - `'a'` - appends to the end of a file.
- Can read the whole file, a line at a time, and some fixed number of characters at a time.
- Close a file after using it.

Consider a file that has 13 characters per line for 5 lines, what character would be read next?

- eg_file.read()

- eg_file.readline()
- eg_file.read(15)
- eg_file.readline()

- eg_file.readline()
- eg_file.readline()
- eg_file.read(15)

Consider a file that has 13 characters per line for 5 lines, what character would be read next?

- eg_file.read()

- An eof character.

- eg_file.readline()
- eg_file.read(15)
- eg_file.readline()

- The first character of the fourth line.

- eg_file.readline()
- eg_file.readline()
- eg_file.read(15)

- The third character of the fourth line.

Lookup Tables

- We saw that python has lookup tables for local and global variables.
- It might be nice to have our own.
  - This would allow use to associate lots of information with a unique piece of information, like a string, or a number.
  - Can store records via student name/date/number/etc.

Lookup tables

- We could implement this with lists and tuples.
- Each element of a list might be a tuple with the format `(id, information)`.
- To get information back about the id we'd need to find out the index and then use `list_name[index][1]`.

Lookup tables

- We could implement this with lists and tuples.
- Each element of a list might be a tuple with the format `(id, information)`.
- To get information back about the id we'd need to find out the index and then use `list_name[index][1]`.
- Two problems with this:
  - Bulky, requires more than one line of code.
  - Slow, lookup tables are constant, but we need to find the element.

Example

- A lot of searching is based on word counts.
  - This is especially true in fixed data bases like Academic journals.
- One reads through a document, and counts words; and then normalises the word counts.
- Related documents should have similar normalised word counts.
- So we want a (word, frequency) pair, but the number of words could be massive.

Dictionaries

- Dictionaries are (key, value) pairs. Sometimes they are called maps. Can be thought of as lookup tables.
- Python syntax:

```
{key0 : value0, key1 : value1, ...,
keyn : valuen}
```

- Dictionaries are of type `dict`
  - Since they have a type, they can be assigned to a variable.
- To refer to a value associated with a key in a dictionary we use `dictionary_name[key]`

Dictionaries

- Dictionaries are unsorted.
- Dictionary keys must be immutable, but the values can be anything.
  - Keys cannot be `None`.
- Once you've created a dictionary you can add key-value pairs by assigning the value to the key.

```
dictionary_name[key] = value
```

- Keys must be unique.

Parentheses Aside.

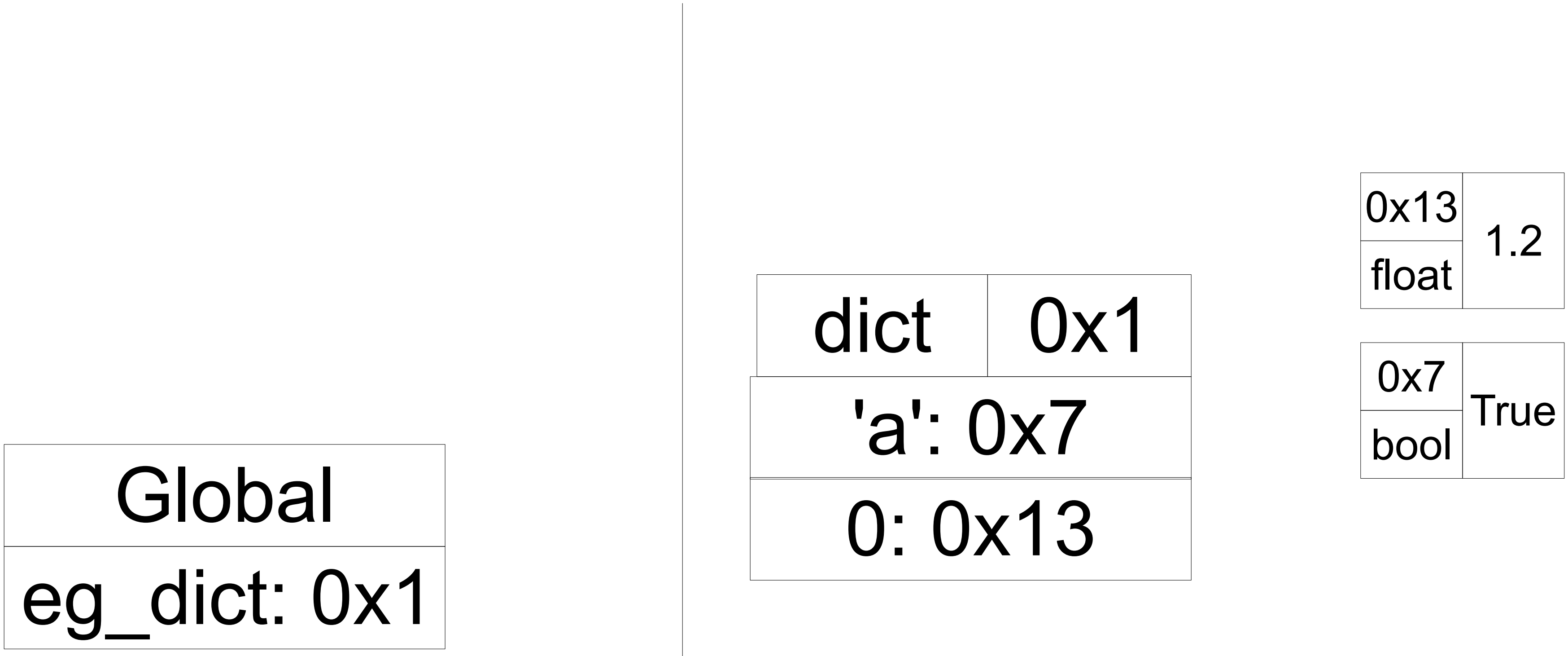
- Python uses three kinds of parenthese (), [], and {}.
- () are used for specifying parameters. This means that parentheses are closely tied to calling functions/methods.
  - Also used to force order of operations.
  - And tuples.
- [] Brackets are used to index into things.
- {} are used to create dictionaries.

Representing Dictionaries in the Memory Model.

- Dictionaries are implemented in such a way that it is difficult to accurately represent them in the memory model while also making it easy to see what's going on.
- So instead we'll represent them as lookup tables (on the right of the line) with the evaluation of the key, but the memory address of the value.
  - Using memory addresses for both is more accurate but less useful

Dictionaries and the Memory Model

```
eg_dict = {'a':True, 0:1.2}
```



Global

eg_dict: 0x1

dict 0x1

'a': 0x7

0: 0x13

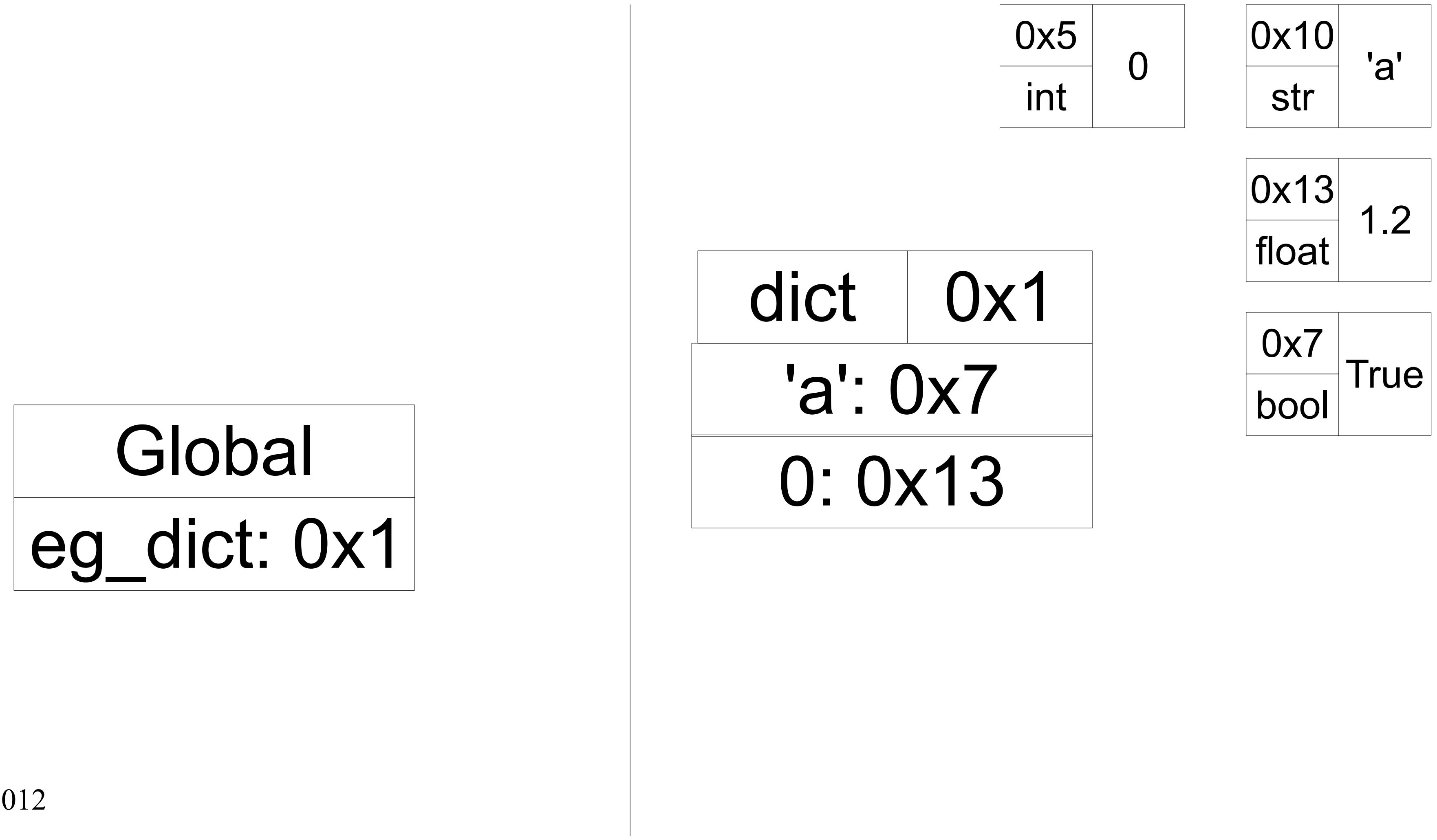
0x13 float 1.2

0x7 bool True

June 14 2012

Dictionaries and the Memory Model

```
eg_dict = {'a':True, 0:1.2}
```



Global
eg_dict: 0x1

dict 0x1
'a': 0x7
0: 0x13

0x5 int 0

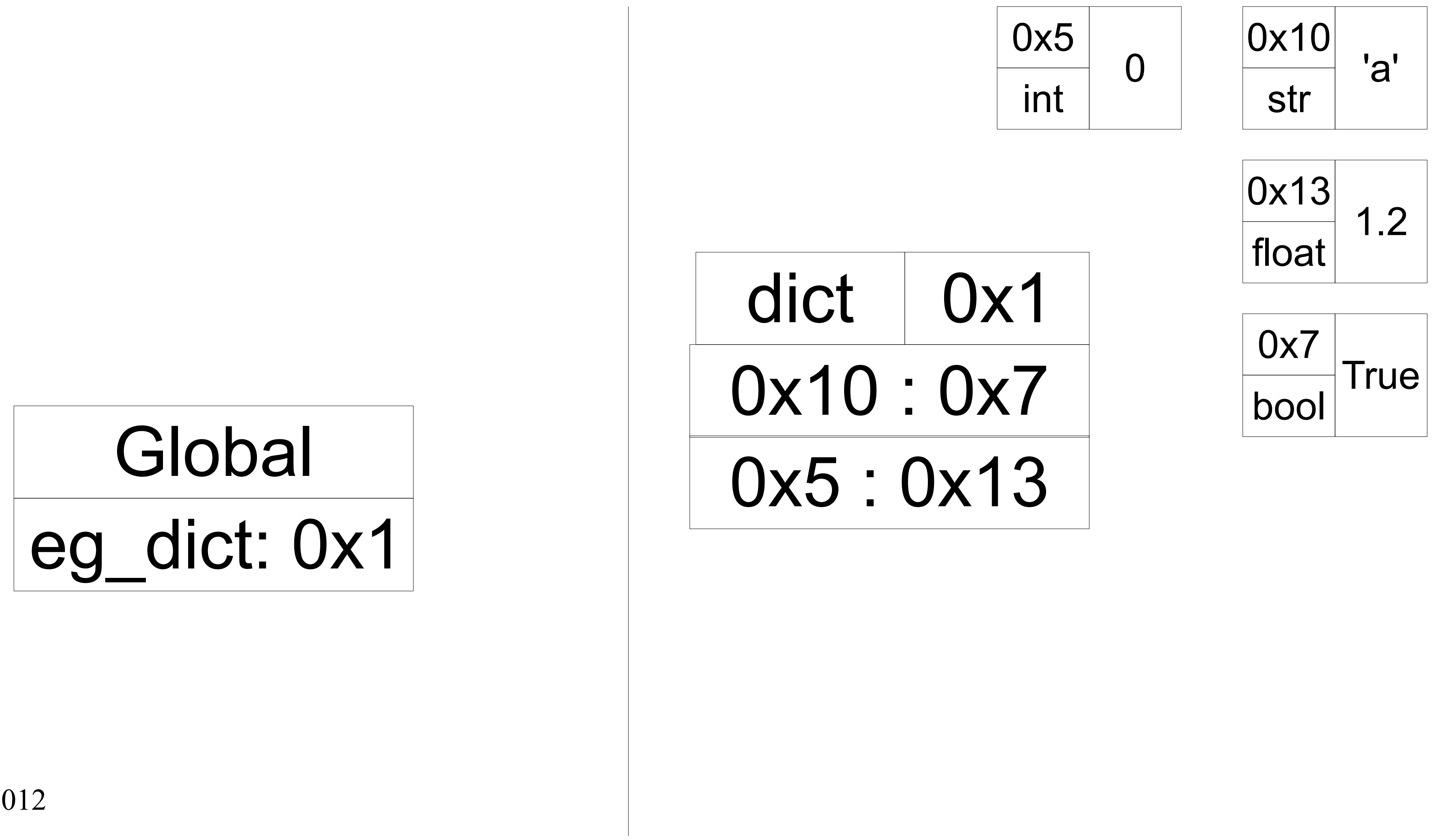
0x10 str 'a'

0x13 float 1.2

0x7 bool True

Dictionaries and the Memory Model

```
eg_dict = {'a':True, 0:1.2}
```



Global

eg_dict: 0x1

dict 0x1

0x10 : 0x7

0x5 : 0x13

0x5 int 0

0x10 str 'a'

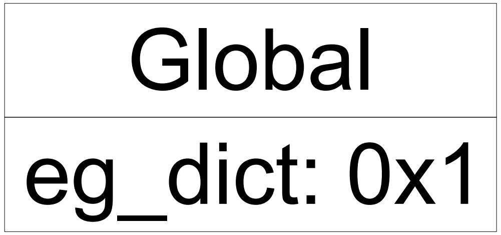
0x13 float 1.2

0x7 bool True

Dictionaries and the Memory Model

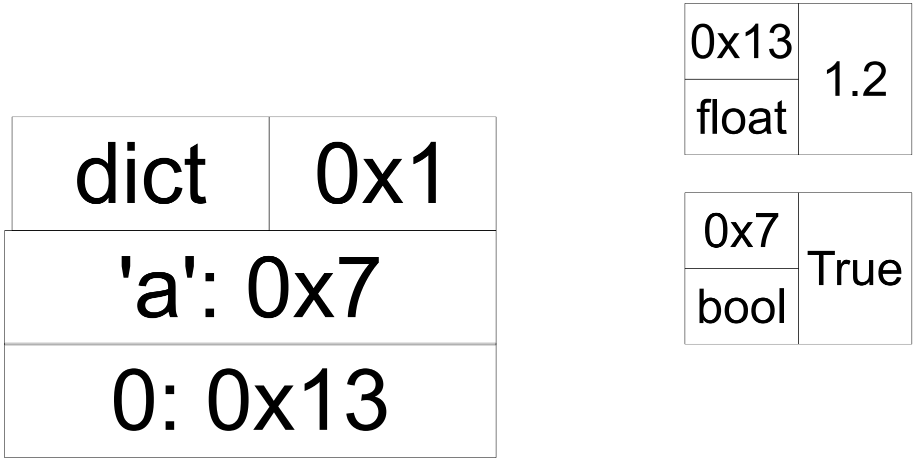
```
eg_dict = {'a':True, 0:1.2}
```

This is the style we want!



Global

eg_dict: 0x1



dict 0x1

'a': 0x7

0: 0x13

0x13 float 1.2

0x7 bool True

Break, the first.

Rewrite this code so that eg_list is a dictionary, not a list.

```
eg_list = [4, 3, 6]
for i in range(2):
    eg_list.append(i*i)
print eg_list[0]
print eg_list[3]
print eg_list[4]
```

Rewrite this code so that eg_list is a dictionary, not a list.

```
eg_list = [4, 3, 6]       eg_dict = {0 : 4, 1 : 3, 2 : 6}
for i in range(2):        for i in range(2):
    eg_list[i].append(i*i)    eg_dict[i + 3] = i*i
print eg_list[0]          print eg_dict[0]
print eg_list[3]          print eg_dict[3]
print eg_list[4]          print eg_dict[4]
```

Dictionary methods.

- `len(dict_name)` works in the same way as it does for strings and lists.
- + and * are not defined for dictionaries.
- `dict.keys()` - returns the keys in some order.
- `dict.values()` - returns the values in some order.
- `dict.items()` - returns the (key, value) pairs in some order.
  - All of these methods have iter* variants that return the keys|values|key-value pairs one by one.

Dictionary methods.

- `dict.has_key(key)` - returns `True` iff the dictionary has the key in it.
- `dict.get(key)` – returns the value that is paired with the key, or `None` if no such key exists.
  - `get(key, d)` returns d rather than `None` if no such key exists.
- `dict.clear()` - removes all the key-value pairs from the dictionary.

Dictionary methods.

- `dict.copy() - copy the entire dictionary.`
  - Be wary if the dictionary has mutable objects.
  - Can have the same issue has with nested lists.
- `dict.update(dict_name)` - adds the key-value pairs in dict_name to dict.
- `dict.pop(key)` – removes and returns the key-value pair indexed by the key.
  - `popitem` returns the `(key, value)` pair.

Why dictionaries?

- Dictionaries are useful if you want to have really big sparse data structures.
  - You can implement spreadsheet, or alarms with dictionaries.
- Or if you get a big amount of data but you're not quite sure how complete it is.
  - So you have a bunch of names, but don't know how many of them you'll actually see.

Looping over dictionaries.

```
for key in d:
  print key, d[keys]
```

- Works, but is a bit slow.

```
for key in d.iterkeys():
  print key, d[keys]
```

- This is a bit better.
- However, the order is still arbitrary.
- How can we make the loop ordered?

Inverting a dictionary.

- Sometimes we want to figure out what the key corresponding to a given value is.
  - This is impossible to do naively.
  - That is, `dict[value]` will not return the key.
- That is we want an identical dictionary, except with keys and values switched.
- If we haven't built the dictionary yet, then we can build two at the same time, where they are inverses of each other.
- Otherwise we need to build an inverse dictionary.

A problem.

- While the keys in a dictionary must be unique, the values don't have this restriction.
- So multiple keys can have the same value.
- How do we build our reverse dictionary?
- We still need to make the values into keys, but we won't have enough values to give each key a unique value.
- We can solve this by pairing the original values with lists of original keys.

Break, the second.

Write Code to reverse a dictionary.

Write Code to reverse a dictionary.

```
def rev_dict(dict_in):
    dict_out = {}
    for key in dict_in:
        if dict_in[key] in dict_out:
            dict_out[dict_in[key]].append(key)
        else:
            dict_out[dict_in[key]] = [key]
    return dict_out
```

Function Review

- Now that we've seen mutable objects, we can see that there are essentially three kinds of functions:
  - Functions that return things.
  - Functions that change mutable objects.
  - Functions that do neither.

Functions that return things.

- These are closest to the mathematical definition of a function.
- They take input parameters and produce an output parameter.
- $f(x) = x^2$ takes in numbers and produces numbers.
- To get the value of f(9) and replace the xs on the right with 9s, and evaluate the expression.
  - functions defined in code work in a similar way.

Functions that change mutable objects

- These are functions that take in lists and dictionary and modify them according to the input parameters.
- These functions don't need have return statements.
  - Note, this does not mean they need print statements or pass statements.

Functions that change mutable objects

- These are functions that take in lists and dictionary and modify them according to the input parameters.
- These functions don't need have return statements.
  - Note, this does not mean they need print statements or pass statements.
  - Nothing needs pass statements.

Functions that do neither

- These will generally show something to the user.
- They might print something to the screen, or load an image or play a sound file, etc.
- Don't need return statements.

Midterm Review

- Will cover everything up to (but not including) this lecture.
  - ints, floats, bools, strings, lists.
  - functions, local scope, global scope.
  - print. vs. return.
  - Modules, importing, if __name__ == '__main__'
  - For loops and while loops.
  - Files.
  - Docstrings, function design.

Midterm Review

- There will generally be three types of questions.
  - Questions that ask you to read/understand code.
  - Questions that ask you to convert one set of code to an equivalent set of code.
    - This is a new style of question. I will be posting a bunch of practice questions on Friday from this style.
    - Basically will involve re-writing code to use functions, or writing while loops as for loops, etc.
  - Questions that ask you to generate code.
  - Will be 90 minutes.