CS 1
Introduction to Computer Programming

Lecture 9: October 24, 2012

Dictionaries

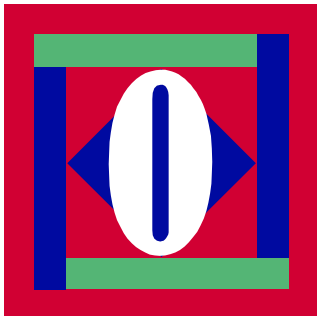
Last time

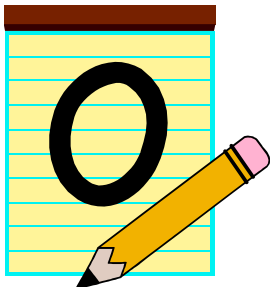
- A bunch of mostly-unrelated topics:
  - Booleans and boolean operators
  - Using `for` loops with files
  - The `in` operator
  - The `range` function
  - Tuples
  - The `enumerate` function
  - Sequence slices

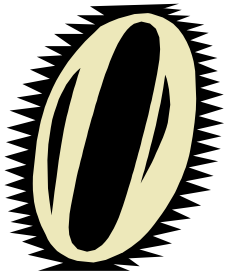
Today

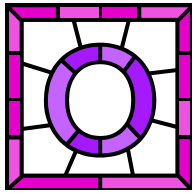
- Binary numbers
  - and hexadecimal numbers
- Dictionaries (a new data type)
  - Dictionaries and `for` loops
  - Dictionary methods

Binary numbers









Binary numbers

- We normally compute using *decimal* numbers
  - numbers composed of ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9
- This is only one of *many* ways to represent numbers
- Other ways include:
  - Roman numerals: MMXI (2011)
  - Sequence of 1s: 1111111111 (10)

Binary numbers

- Computers normally use *binary* numbers
  - only two digits: 0 and 1
- Reason: it's easy to build digital electronic circuits using binary numbers
  - 0 is "low voltage"
  - 1 is "high voltage"
- Any number that can be represented as a decimal number can also be represented as a binary number
  - so computers use binary numbers for convenience

Binary numbers

- Decimal numbers are also called *base 10* numbers
- Binary numbers are *base 2* numbers
- Also often use *base 16* numbers (hexadecimal)
- Here, we'll show how to convert from binary numbers to decimal numbers
  - We'll restrict ourselves to integers >= 0 for simplicity

Binary numbers

- A binary number consists only of 0s and 1s, *e.g.*:

11010

- What is this number in decimal notation?

Binary numbers

- Recall: what does a decimal number mean?
- The number 1234 means:
- $1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$
- $= 1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1$
- $= 1234$
- 1000, 100, 10, and 1 are all powers of 10
- So we say that 1234 is written in *base 10*
- Or 1234_{10} if we want to be very explicit

Binary numbers

- Binary numbers use powers of 2, not 10
- The number 11010 means:
- $1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
- $= 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$
- $= 26$
- 16, 8, 4, 2, and 1 are all powers of 2
- So we say that 11010 is written in *base 2*
- Or 11010_2 if we want to be very explicit

Binary numbers

- Note: the same number (like 11010) can mean something very different in binary and decimal number systems!
- 11010 (binary) = twenty six
- 11010 (decimal) = eleven thousand and ten
- Usually obvious from context which number system we mean
- If not, use a subscript:
  - 11010_{10} for decimal
  - 11010_{2} for binary

Conversions

- We've seen how to convert binary numbers to their decimal equivalents
- Converting decimal to binary is significantly more complicated
  - makes a great lab problem ☺
  - won't cover here

Conversions

- Computer does all of its arithmetic in binary
- We usually input data as decimal
- So computer must
  - convert decimal → binary
  - do computations in binary
  - convert back to decimal for human viewing
- Even when you see decimal numbers, they are really binary numbers inside the computer

Literal binary numbers

- Python allows us to enter literal binary numbers

```
>>> 0b11010
26
```

- The `0b` prefix says "the following number is a binary number"

Hexadecimal numbers

- Large binary numbers are very cumbersome to write out
- For instance:
  - 1000 in binary is 1111101000
  - 1000000 in binary is 11110100001001000000
- Instead, we can write out binary numbers as *hexadecimal* (base 16) numbers

Hexadecimal numbers

- Hexadecimal numbers have single "digits" for all the numbers from 0 to 15:
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    a (10), b (11), c (12), d (13), e (14), f (15)
- A number in hexadecimal is represented as a sum of powers of 16
- Example: $1a2e_{16}$
  - $= 1 \times 16^3 + a \times 16^2 + 2 \times 16^1 + e \times 16^0$
  - $= 1 \times 4096 + 10 \times 256 + 2 \times 16 + 14 \times 1$
  - $= 6702$

Hexadecimal numbers

- Any 4 binary digits = 1 hexadecimal digit
- 0000 = 0
- 0001 = 1
- ...
- 1001 = 9
- 1010 = a
- 1011 = b
- ...
- 1111 = f

Hexadecimal numbers

- Hexadecimal numbers can be used as more compact versions of binary numbers
- For every 4 binary digits, can write 1 hexadecimal digit
- Example:
- 1111010000100100000_2
- $1111\ \ 0100\ \ 0010\ \ 0100\ \ 0000_2$
- $f\ \ 4\ \ 2\ \ 4\ \ 0_{16}$
- $f4240_{16}$

Hexadecimal numbers

- Python also allows you to write literal hexadecimal numbers:

```
>>> 0xf4240
1000000
```

- The `0x` prefix means "this number is a hexadecimal number"
- Hexadecimal numbers are more compact than decimal or binary numbers

Interlude

- More "Python"

Dictionaries

- A *dictionary* is a new kind of Python data type
- Before we describe what it is, let's describe a problem it could solve

Phone numbers

- You want to keep track of your friends' phone numbers
- But you (naturally) have so many friends, this is a difficult job
- How can the computer help?

Phone numbers

- For each friend, need to store:
  - the *name* of the friend
  - the *phone number* of the friend
- Also, want to be able to retrieve the phone number for a given friend
- Given what you know now, how can you do this?

List?

- You could have a list of names and phone numbers:

```
phone_numbers = ['Joe', '567-8901',
                 'Jane', '123-4567',
                 ...]
```

- But it would not be easy to find the number corresponding to a different name
- It would be better if a name and the corresponding phone number were connected in some way

List of tuples?

- You could have a list of (*name*, *phone number*) tuples:

```
phone_numbers = [('Joe', '567-8901'),
                 ('Jane', '123-4567'),
                 ...]
```

- Let's see what we would need to do in order to find the phone number corresponding to a particular name
  - *e.g.* `'Donnie'`

List of tuples?

- We could write code like this:

```
for (name, phone) in friends:
    if name == 'Donnie':
        print 'Phone number: %s' % phone
```

- This is not too bad, but
  - can't modify the phone number!
    - (tuples are immutable)
  - have to look through entire list in worst case to find one number
  - cumbersome!

Dictionaries

- A dictionary is a data structure that stores associations between *keys* and *values*
- In the previous example:
  - *key*: the name of the friend
  - *value*: the phone number
- Dictionaries make it easy to:
  - find the value given the key
  - change the value given the key
  - add more key/value associations
- And they're fast!

Keys and values

- The *values* stored in a dictionary can be any Python value
- *Keys* can only be *immutable* (unchangeable) Python values, *e.g.*
  - strings
  - tuples
  - numbers (rare)
- There is a technical reason for this (which we won't go into)
  - Usually use strings as keys

Dictionary syntax

- The contents of a dictionary are written between curly braces ({ and })
- The empty dictionary (no key/value pairs) is written like this:

```
{ }
```

Dictionary syntax

- A typical dictionary might look like this:

```
{'Joe'   : '567-8910',
 'Jane'  : '123-4567' }
```

Dictionary syntax

- A typical dictionary might look like this:

key

```
{'Joe'   : '567-8910',
 'Jane'  : '123-4567' }
```

Dictionary syntax

- A typical dictionary might look like this:

value

```
{'Joe'   : '567-8910',
 'Jane'  : '123-4567' }
```

Dictionary syntax

- A typical dictionary might look like this:

colon

```
{'Joe' : '567-8910',
 'Jane' : '123-4567' }
```

- Colon (:) separates a key from its value

Dictionary syntax

- A typical dictionary might look like this:

```
{'Joe'   : '567-8910',
 'Jane'  : '123-4567' }
```

comma

- Comma (,) separates different key/value pairs

Dictionary syntax

- A typical dictionary might look like this:

key/value pair #1

```
{'Joe'   : '567-8910',
 'Jane'  : '123-4567' }
```

Dictionary syntax

- A typical dictionary might look like this:

```
{'Joe'   : '567-8910',
 'Jane'  : '123-4567' }
```

key/value pair #2

Dictionary syntax

- Dictionary values can be expressions:

```
{'Joe'  : 2 + 3,
 'Jane' : '123-' + '4567' }
```

Dictionary syntax

- Dictionary keys can also be expressions:

```
{'Joe' + ' Blow' : '567-8901',
 'Jane' + ' Doe' : '123-4567' }
```

- (This is very rare, though)
- Expressions are always evaluated while creating the dictionary

Getting a value given a key

- We have:

```
phone_numbers = {'Joe'  : '567-8901',
                 'Jane' : '123-4567' }
```

- To get Joe's phone number:

```
Joes_phone_number = phone_numbers['Joe']
```

Getting a value given a key

- Notice that:

```
phone_numbers['Joe']
```

  looks like accessing a list with a value of `'Joe'`
- Python is *overloading* the meaning of the square brackets
- Before, the value inside the brackets could only be an integer
- With a dictionary, it's any key value

Changing a value at a key

- Let's say that Joe's phone number changes
- Can change the dictionary value too:

```
phone_numbers['Joe'] = '314-1592'
```

- Like the syntax for changing a list value
  - except that the "index" is a string, not a number

Adding a new key/value pair

- Add a new key/value pair by "changing" a key that wasn't there before:

```
phone_numbers['Donnie'] = '111-1111'
phone_numbers['Mike'] = '000-0000'
```

Accessing a nonexistent key

- Here's what happens if you try to access a key that isn't in the dictionary:

```
>>> phone_numbers['Quentin']
KeyError: 'Quentin'
```

Deleting a key/value pair

- To remove a key/value pair from a dictionary:

```
>>> del phone_numbers['Joe']
>>> phone_numbers
{ 'Jane' : '123-4567',
  'Mike' : '000-0000',
  'Donnie' : '111-1111' }
```

`del`

- **`del`** is actually a special Python statement, like **`print`**
  - it's not a function, so no parentheses around its argument
- **`del`** can remove elements from things other than dictionaries (*e.g.* lists)
  - but more useful with dictionaries than lists

Back to the example

- Let's improve the example by using a tuple of first and last names as keys:

```
phone_numbers = \
   { ('Joe', 'Smith') : '567-8910',
     ('Jane', 'Doe') : '123-4567',
     ('Mike', 'Vanier') : '000-0000',
     ('Donnie', 'Pinkston') : '111-1111' }
```

- This is OK, because both tuples and strings are immutable
  - so tuple of strings is immutable too, hence OK as key

Back to the example

- Can access phone numbers using tuple as key:

```
>>> phone_numbers[('Joe', 'Smith')]
'567-8910'
>>> phone_numbers['Joe']
KeyError: 'Joe'
>>> phone_numbers['Smith']
KeyError: 'Smith'
```

- You have to use the correct type of key for the dictionary, or it's an error!

Dictionaries and `for` loops

- We've seen many things that can be looped over using `for` loops:
  - lists
  - strings
  - files
- Should it surprise you to learn that dictionaries can also be looped over in a `for` loop?
  - We hope not!

Dictionaries and `for` loops

- Looping over a dictionary looks like this:

```
for key in phone_numbers:
    print key
```

- Looping over a dictionary loops over the *keys* in the dictionary (not the values)

Dictionaries and `for` loops

```
for key in phone_numbers:
    print key
```

- This will print:

```
('Mike', 'Vanier')
('Joe', 'Smith')
('Donnie', 'Pinkston')
('Jane', 'Doe')
```

Dictionaries and `for` loops

- Note:

```
('Mike', 'Vanier')
('Joe', 'Smith')
('Donnie', 'Pinkston')
('Jane', 'Doe')
```

- is *not* the order in which keys were originally entered in dictionary
- Dictionaries are *unordered* (not a sequence)
  - the "location" of any key/value pair is unimportant

Dictionaries and `for` loops

- We usually want the values, not the keys:

```
for key in phone_numbers:
    print phone_numbers[key]
```

- gives:

```
'000-0000'
'567-8910'
'111-1111'
'123-4567'
```

Searching

- Problem: print out the phone number of every person whose first name is `'Joe'`
  - using `(<first name>, <last name>)` tuples as keys in the dictionary

```
for key in phone_numbers:
    first_name, last_name = key
    if first_name == 'Joe':
        print phone_numbers[key]
```

- Easy peasy!

Dictionary methods

- Dictionaries are objects in Python
  - like lists, and strings, and files
- Therefore, they have methods
- We will discuss these methods:
  - `clear`
  - `keys`
  - `has_key`
  - `values`
  - `update`
- though there are many more

clear

- The `clear` method just empties out the dictionary:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.clear()
>>> d
{}
```

keys

- The **keys** method returns a list of all the keys in the dictionary:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.keys()
['baz', 'foo', 'bar']
```

has_key

- The `has_key` method returns `True` if its argument is a key in the dictionary:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.has_key('foo')
True
>>> d.has_key('fnord')
False
```

values

- The `values` method returns a list of all the values in the dictionary:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.values()
[3, 1, 2]
```

update

- The `update` method adds the key/value pairs from another dictionary into this one
  - overwriting old values if other dictionary has same keys with different values

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.update({'xxx' : 4, 'yyy' : 5})
>>> d
{'baz' : 3, 'xxx' : 4, 'foo' : 1,
 'bar' : 2, 'yyy' : 5}
```

update

- Another example:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.update({'foo' : 4, 'yyy' : 5})
>>> d
{'baz' : 3, 'foo' : 4, 'bar' : 2,
 'yyy' : 5}
```

- New value of key `'foo'` overwrites the old one

What about **append**?

- There is no **`append`** method for dictionaries
  - not needed!
- To add a new key/value pair, just use normal assignment syntax:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d['fnord'] = 4  # add key/value pair
>>> d
{'fnord' : 4, 'baz' : 3, 'foo' : 1,
 'bar' : 2}
```

The `in` operator

- Last time we saw the `in` operator for sequences
- Can also use in with dictionaries
- `<key> in <dictionary>` means: is the key `<key>` one of the keys in the dictionary `<dictionary>`?

```
>>> 'foo' in { 'foo' : 1, 'bar' : 2 }
True
```

- Nicer than using `has_key` method

```
>>> { 'foo' : 1, 'bar' : 2 }.has_key('foo')
```

New example

- We have a list of words
- Want to create a frequency table
  - for each word, how many times does it occur in list?
- Solve by creating a dictionary
  - *key*: a word in the list
  - *value*: the count of that word
- Let's write the code...

New example

```
words = [ ... ]  # whatever
freqs = {}
for word in words:
    if freqs.has_key(word):
        freqs[word] += 1
    else:
        freqs[word] = 1
```

- And we're done!
- But wait! We can make this a bit prettier...

New example

```
words = [ ... ]  # whatever
freqs = {}
for word in words:
    if word in freqs:
        freqs[word] += 1
    else:
        freqs[word] = 1
```

- And we're done again!
- But wait! We want to print out the results...

New example

```
for key in freqs:
    print "Word %s occurs %d times" % \
        (key, freqs[key])
```

- Now we're really done
- Dictionaries are awesome!
  - used in most Python programs
  - makes it much easier to write programs to solve a wide variety of tasks

Summary

- Binary and hexadecimal numbers are a different way to represent integers
  - closer to the way the computer itself represents them
- Dictionaries are a built-in Python data type that allows us to associate keys to values
  - They have many useful methods
  - They can be iterated over in `for` loops
  - They have lots of applications
  - They are used all the time in Python code
  - They are awesome!

Next time

- Friday lecture!
- "How to think about programming"

Next time after that

- Systematic ways of testing your code
- Graphics!