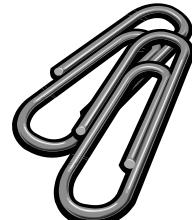


CS I

Introduction to Computer Programming

Lecture 8: October 22, 2012

Odds and Ends



Last time

- **while** loops
- **break**
- Files
 - opening, closing, reading from files



Today

- A smörgasbörd of töpics:
 - More on booleans
 - More on loops
 - looping over files with `for`
 - The `in` operator
 - The `range` function
 - Tuples
 - The `enumerate` function
 - Sequence slices



More on booleans

- Booleans are **True/False** values
- However, Python is not strict about this
- Several things besides the **False** value are considered to be **False** by Python
 - the empty string: ''
 - the empty list: []
 - the number 0
 - ... and most other values can be considered to be **True**



Boolean operators

- Python also supports three boolean operators
 - take booleans arguments, return booleans
 - **not**, **and**, and **or**
- **not** – converts **True** → **False**,
False → **True**

```
>>> not True
```

```
False
```

```
>>> not False
```

```
True
```



Boolean operators

```
>>> not 0
```

```
True
```

```
>>> not 1
```

```
False
```

```
>>> not ''      # empty string
```

```
True
```

```
>>> not []
```

```
True
```



Boolean operators

- **and** and **or** are operators that take two boolean arguments and return a boolean value
- **and** returns **True** if both arguments are **True**, otherwise **False**
- **or** returns **True** if either or both arguments are **True**, otherwise **False**



Boolean operators

```
>>> True and True
```

```
True
```

```
>>> True and False
```

```
False
```

```
>>> False or True
```

```
True
```

```
>>> False or False
```

```
False
```

- Usually don't use **and** and **or** this way, though



Boolean operators

- Generally use **and** and **or** with expressions that *evaluate* to booleans
 - e.g. relational operators

```
a = 10
```

```
b = 30
```

```
# ... later in code ...
```

```
if a > 0 and b < 50: ...
```

```
# ... later in code ...
```

```
if a < 0 or b > 50: ...
```



Example from last time

- Recall the example problem from last time:

```
temp = open('temps.txt', 'r')
sum_nums = 0.0
while True:
    line = temp.readline()
    if line == '':
        break
    sum_nums += float(line)
```



Example from last time

- We had these lines of code in the example:

```
if line == '':
    break
```

- Since the empty string '' can be considered to be a **False** value, can write this in equivalent form:

```
if not line:
    break
```

- **not line** means "if there is no line"



Full example

- With an infinite loop and **not line**:

```
temps = open('temps.txt', 'r')
sum_nums = 0.0
while True:
    line = temps.readline()
    if not line:
        break
    sum_nums += float(line)
```

- This is OK but still somewhat long-winded



Using `for` with files

- Python allows an amazing shortcut using the `for` statement:

```
temp = open('temps.txt', 'r')
sum_nums = 0.0
for line in temp:
    sum_nums += float(line)
```

- This is the preferred way to write this



Using `for` with files

- Previously, we had:

```
for <name> in <something>:  
    # block of code
```

- The `<something>` after the `in` had to be a list or a string
- Python actually allows more than just lists or strings after the `in`
 - files being one example



Using `for` with files

- Conceptually, we have

```
for <name> in <sequence>:  
    # block of code
```

- Lists are sequences
- Strings are sequences
- And files can be considered to be "sequences of lines"



Using `for` with files

- However, just because files work here:

```
for <name> in <sequence>:  
    # block of code
```

- Doesn't mean you can do *all* sequence operations on files!



Using `for` with files

- For instance, this won't work:

```
# assume that the file 'foo.txt' exists
f = open('foo.txt', 'r')
print f[0] # print first line in file?
```

- Result:

TypeError: 'file' object is unsubscriptable

- Moral: files are *not* full-fledged Python sequences
 - but `for` does work with files



The `in` operator

- We have seen the word `in` in the context of a `for` expression:

```
>>> for i in [1, 2, 3]:  
...     print i
```

1

2

3

- However, `in` has a separate meaning when used as an operator (coming between operands)



The `in` operator

- An expression of this form:

`1 in [1, 2, 3]`

- means: does `1` occur in the list `[1, 2, 3]`?
- More generally, `<x> in <s>` asks if a value `<x>` is found in a sequence `<s>` (usually a list or a string)

```
>>> 1 in [1, 2, 3]
```

```
True
```

```
>>> 'b' in 'foobar'
```

```
True
```



The `in` operator

- You can use this with variables too:

```
>>> x = 10
```

```
>>> x in [10, 20, 30]
```

True

- However, this has no relation to this syntax:

```
>>> for x in [10, 20, 30]:
```

```
...     print x
```

10

20

30



range

- The `range` function creates lists of consecutive integers:

```
>>> range(0, 5)
```

```
[0, 1, 2, 3, 4]
```

```
>>> range(10, 20)
```

```
[10, 11, 12, 13, 14, 15, ..., 19]
```

```
>>> range(-10, -5)
```

```
[-10, -9, -8, -7, -6]
```



range

- The **range** function takes two arguments:
 - the first number in the list
 - the number one after the last number in the list
- So **range(1, 10)** gives you
[1, 2, 3, 4, 5, 6, 7, 8, 9]
and not
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- This seems weird, but will turn out to be useful



range with 3 arguments

- The `range` function can take an optional third argument:

```
>>> range(0, 10, 2)
```

```
[0, 2, 4, 6, 8]
```

- The third argument is the *step size*
 - the difference between consecutive list items
- Can even have negative step sizes:

```
>>> range(10, 0, -2)
```

```
[10, 8, 6, 4, 2]
```



range with 3 arguments

- Another way to think about it: `range` "always" has three arguments, but the last one is `1` by default when you use the two-argument form
- So:

`range(0, 10)`

really means:

`range(0, 10, 1)`

- Python allows you to define functions which can take varying numbers of arguments (will see later in course)



range with 1 argument

- **range** can also be used with *one* argument
- In this case:
 - starting value is **0**
 - step size is **1**

```
>>> range(10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Equivalent to:

```
>>> range(0, 10, 1)
```
- This is the most commonly-used form of **range**



Use of range

- **range** is often used with **for** loops:

```
>>> for i in range(1000):  
...     print i  
  
0  
1  
2  
3  
...
```



Use of `range`

- `range` is often used with `for` loops and lists:

```
>>> mylist = ['Caltech', 'is', 'great']
>>> for i in range(len(mylist)):
...     mylist[i] = mylist[i] + '-YEAH!'
>>> mylist
['Caltech-YEAH!', 'is-YEAH!', 'great-YEAH!']
```

- This `range(len(...))` idiom is quite common in Python code



range(len(<list>))

- `range(len(<list>))` gives you a list of the valid indices of a particular list `<list>`
- For example:

```
>>> lst = ['a', 'list', 'of', 'strings']
```

```
>>> len(lst)
```

```
4
```

```
>>> range(len(lst))
```

```
[0, 1, 2, 3]
```

- 0, 1, 2, and 3 are the valid indices of the list `lst`



for i in range(len(<list>))

- **for i in range(len(<list>))** used when need to have the indices of the list **<list>** inside a **for** loop (e.g. so you can change the list elements)
- If don't need to change list elements, usually can get by with just

for <element> in <list>: ...



Example

- Need to double every element of a list of numbers

```
nums = [23, 12, 45, 68, -101]
for i in range(len(nums)):
    nums[i] = nums[i] * 2
```

- Or could write second line as just

```
    nums[i] *= 2
```

- Result: [46, 24, 90, 136, -202]



Example

- This works, but `range(len(nums))` is kind of chunky to write
- We will shortly see a more elegant way to write this using the `enumerate` built-in function
- But first, we need to talk about tuples
- But before that, we need an...



Interlude

- Computer games!



Tuples

- Tuples are another Python data type
 - basically a read-only list
- Tuple syntax is simple:

(1, 2, 3, 4, 5)

- Like a list, but surrounded by parentheses, not square brackets



Tuples

- Tuples of length 1 written like this:

(1 ,)

- because (1) is just the number **1**
 - (normal use of parentheses)
- Empty tuple is written ()



Tuples

- Tuples support many, but not all, of the list operations
- Tuples are sequences like strings or lists
 - so it's OK to do
`for <item> in <tuple>: ...`
 - with a tuple
 - `len` gives the length of a tuple

```
>>> len((1, 2, 3, 4, 5))
```

5



Tuples

- Can concatenate tuples with the `+` operation

```
>>> (1, 2, 3) + ('foo', 'bar', 'baz')  
(1, 2, 3, 'foo', 'bar', 'baz')
```

- Can index tuples as with lists

```
>>> tup = ('foo', 'bar', 'baz')  
>>> tup[0]  
'foo'  
>>> tup[-1]  
'baz'
```



Tuples

- Cannot change contents of tuple!
 - like strings, tuples are immutable

```
>>> tup = ('foo', 'bar', 'baz')  
>>> tup[0] = 'hello'
```

TypeError: 'tuple' object does not support item assignment



Tuples

- Python sometimes lets you leave off the parentheses in a tuple

```
>>> 1, 2, 3
```

```
(1, 2, 3)
```

```
>>> tup = 4, 5, 6
```

```
>>> tup
```

```
(4, 5, 6)
```

- We recommend that you keep the parentheses (more obvious what the object is)
 - with one exception (coming up)



Tuples

- Tuples are thus basically a more restricted kind of list
- So what good are tuples, anyway?
- Not really an essential language feature
 - Python could do fine without them
- There are some cases where tuples are convenient:
 1. returning multiple values from functions
 2. tuple unpacking
 3. **for** loops with multiple bindings



Multiple return values

- Functions in Python can only return a single value
 - most of the time, this is all we need
- Sometimes, it's useful to be able to return more than one value from a function
- We do this by creating a tuple of the return values, and returning that one value



Multiple return values

- Consider the built-in function **divmod**:

```
>>> divmod(10, 3)
```

```
(3, 1)
```

```
>>> divmod(42, 7)
```

```
(6, 0)
```

```
>>> divmod(101, 5)
```

```
(20, 1)
```

- divmod** returns the quotient and remainder of its two arguments, as a tuple



Multiple return values

- We could define `divmod` ourselves:

```
def divmod(m, n):  
    return (m / n, m % n)
```

- (Actual definition is more complex)
- Now we can write:

```
>>> qr = divmod(101, 5)  
>>> quotient = qr[0]  
>>> remainder = qr[1]
```



Tuple unpacking

- We can use `divmod` even more elegantly:

```
>>> qr = divmod(101, 5)
```

```
>>> (quotient, remainder) = qr
```

- Now `quotient` is 20 and `remainder` is 1
- This is called "*tuple unpacking*"
 - like a multiple assignment statement
- Python also lets us write this as:

```
>>> quotient, remainder = qr
```

- i.e. without the parentheses



Tuple unpacking

- Structure of a tuple unpacking:

$$(a, b, c) = t$$

- (where **t** is a tuple)
- **t[0]** is assigned to **a**, **t[1]** is assigned to **b**, **t[2]** is assigned to **c**
- Tuple **t** must have the same number of elements as tuple on left-hand side
 - or it's an error



Tuple unpacking

```
>>> tup = (1, 2, 3)
>>> a, b, c = tup
>>> a
1
>>> b
2
>>> c
3
```



Tuple unpacking

```
>>> tup = (1, 2, 3)
```

```
>>> a, b, c, d = tup
```

*ValueError: need more than 3 values to
unpack*

```
>>> a, b = tup
```

ValueError: too many values to unpack



Swapping two variables

- Tuple unpacking lets us do a cute trick: swapping the value of two variables

```
>>> a = 10  
>>> b = 20  
>>> (a, b) = (b, a)  
>>> a  
20  
>>> b  
10
```

- Why does this work?



Swapping two variables

- Like all assignments, the right-hand side is evaluated first:

```
>>> (a, b) = (b, a) evaluate
```

- **(b, a)** evaluates to the tuple **(20, 10)**
- Then this is unpacked into **(a, b)**
 - **a** becomes **20**, **b** becomes **10**
 - That's all there is to it!
 - Can also write as:

```
>>> a, b = b, a
```



for loops with multiple bindings

- We can also use tuples with **for** loops to assign ("bind") values to multiple names every time through the loop

```
>>> for (n, s) in [(1, 'a'), (2, 'b')]:  
...     print "num = %d, char = %s" % (n, s)  
num = 1, char = a  
num = 2, char = b
```

- We'll get back to this



The enumerate function

- Earlier we saw this code:

```
nums = [23, 12, 45, 68, -101]
for i in range(len(nums)):
    nums[i] *= 2
```

- The purpose of `range(len(nums))` is to produce a list of the indices of `nums`
 - which are `[0, 1, 2, 3, 4]`
- Seems like a lot of work for something so simple



The `enumerate` function

- We can also write this code like this:

```
nums = [23, 12, 45, 68, -101]
for (i, e) in enumerate(nums):
    nums[i] *= 2
```

- What `enumerate` does is take a sequence and outputs the indices (`i`) and elements (`e`) of the sequence one by one



The `enumerate` function

- We can leave off the parentheses from the tuple:

```
nums = [23, 12, 45, 68, -101]
for i, e in enumerate(nums):
    nums[i] *= 2
```



The `enumerate` function

- Conceptually, it's as if `enumerate` produces a list of (index, element) tuples:

```
>>> enumerate(['a', 'b', 'c'])  
[(0, 'a'), (1, 'b'), (2, 'c')]
```

- This is not actually what happens!
- In reality, you get something like:

```
<enumerate object at 0x6f940>
```

- but this object behaves like a list of tuples inside a `for` loop



The enumerate function

- Can convert an enumerate object to a list using the `list` built-in function:

```
>>> en = enumerate(['a', 'b', 'c'])
>>> en
<enumerate object at 0x6f940>
>>> list(en)
[(0, 'a'), (1, 'b'), (2, 'c')]
```

- Note that `0`, `1`, and `2` are the indices into the list we want



The `enumerate` function

- Enumerate objects are actually examples of something called an iterator
 - basically, something that can be looped over in a `for` loop
- Python has several built-in iterators, and allows you to define your own
 - Will talk about this later in course
 - This is also why files work with `for`



The enumerate function

- Back to our example:

```
nums = [23, 12, 45, 68, -101]
for i, e in enumerate(nums):
    nums[i] *= 2
```

- Note that we don't need the `e` variable anywhere in the example
- Can name `e` anything we want (doesn't matter)
 - but can't leave it out, or `i` will become a tuple!
 - Usually we name it `_` (indicates unused name)



The enumerate function

- What happens when we leave `e` out?

```
>>> nums = [23, 12, 45, 68, -101]
```

```
>>> for i in enumerate(nums) :
```

```
...     print i
```

```
(0, 23)
```

```
(1, 12)
```

```
(2, 45)
```

```
(3, 68)
```

```
(4, -101)
```



The enumerate function

- Compare:

```
for i in range(len(nums)) :
```

to:

```
for i, e in enumerate(nums) :
```

- Which one you use is up to you
 - both are acceptable



Sequence slices

- Python allows you to get a single element from a sequence using the square bracket notation:

```
>>> lst = [1, 2, 3, 4, 5]  
>>> lst[2]  
3
```

- This works for all sequences (tuples, strings), not just lists



Sequence slices

- Python also lets you get more than one element from a sequence using a **slice**:

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[2:4] # a slice of a list
[3, 4]
```

- Again, this works for all sequences



Slice notation

- Anatomy of a slice:

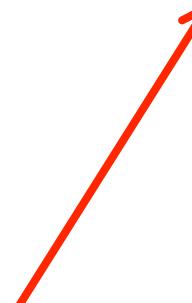
1st[2:4]



Slice notation

- Anatomy of a slice:

1st [2: 4]



starting index of the slice



Slice notation

- Anatomy of a slice:

1st[2:4]

one past final index of the slice



Slice notation

- Anatomy of a slice:

1st[2:
colon (separator)



Sequence slices

- A sequence slice makes a *copy* of a chunk of the sequence
- First element of the copy is the element of the original sequence at the starting index of the slice
- Last element of the copy is the element of the original sequence *one before* the final index of the slice



Sequence slices

- Examples:

```
>>> lst = [1, 2, 3, 4, 5]
>>> tup = (6, 7, 8, 9, 10)
>>> s = 'abcdef'
>>> lst[0:5]
[1, 2, 3, 4, 5]
>>> tup[1:5]
(7, 8, 9, 10)
>>> s[1:4]
'bcd'
```



Sequence slices

- If the slice's final index is larger than the length of the sequence, the slice ends at the last element

```
>>> lst = [1, 2, 3, 4, 5]
```

```
>>> lst[3:1000]
```

```
[4, 5]
```

- If the slice's final index is left out, it's assumed to be equal to the length of the sequence

```
>>> lst[3:]
```

```
[4, 5]
```



Sequence slices

- If the slice's starting index is left out, the slice starts at the first element

```
>>> lst = [1, 2, 3, 4, 5]
```

```
>>> lst[0:3]
```

```
[1, 2, 3]
```

```
>>> lst[:3]
```

```
[1, 2, 3]
```



Sequence slices

- If *both* the slice's starting index and the slice's ending index are left out, the slice is a copy of the entire list!

```
>>> lst = [1, 2, 3, 4, 5]
```

```
>>> lst[0:5]
```

```
[1, 2, 3, 4, 5]
```

```
>>> lst[:]
```

```
[1, 2, 3, 4, 5]
```



Sequence slices

- Slices can use negative indices (counting from the end of the sequence)

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[:-1] # -1: last element of list
[1, 2, 3, 4]
>>> lst[:-2]
[1, 2, 3]
>>> lst[-3:-1]
[3, 4]
```



Sequence slices

- Common application: remove newline character from the end of a string:

```
>>> s = 'Hello, world!\n'  
>>> s[:-1]  
'Hello, world!'
```



Wrap-up

- There are more "odds and ends" still to learn
- Will cover lots more in future lectures
 - as well as major new topics



Next time

- Binary numbers
- Dictionaries

