

# CS I

# Introduction to Computer Programming

Lecture 5: October 8, 2014

Lists, etc.



# Last time

- `raw_input()`
- Objects
- Method syntax (dot syntax)
- Modules
- Module syntax (dot syntax)



# Today

- Documentation (docstrings)
- The **\_\_main\_\_** module
- Lists



# Docstrings

- We've talked about comments already:  
`# This is a comment.`
- Very common use of comments is to describe what a function does:

```
# Print out a hearty greeting.  
def greet(name):  
    print 'Hi there, %s' % name
```



# Docstrings

- Comments describing the purpose of functions are very useful
- However, Python's `help()` function can't use them
- It would be cool if there were a special way to write comments so that they could be used by the `help()` function
- And there is! It's called a **docstring**
  - short for "documentation string"



# Docstrings

- A docstring is just a regular Python string that is the first thing in any of:
  - a function body
  - a module
  - a class (later in course)
- When executing a function body, the docstring doesn't do anything
  - but Python stores it as part of the function
  - same is true for modules and classes



# Docstrings

- `greet` function from last time, with a docstring:

```
def greet(name) :                                     docstring
    '''Print out a hearty greeting.'''
    print 'Hi there, %s' % name
```

- Docstrings usually written using triple-quoted strings (multiline strings)
  - because docstrings often span more than 1 line



# Docstrings

- Assume this is still part of module `greetings`
- Now can do this:

```
>>> help(greetings.greet)
```

```
Help on function greet in module  
greetings:
```

```
greet(name)
```

```
    Print out a hearty greeting.
```



# Module docstrings

- Similarly, can have docstrings for entire module
- New version of file `greetings.py`:

```
'''Module: greetings
Functions to print out greetings.'''

```

module  
docstring

```
def greet(name):
    '''Print out a hearty greeting.'''
    print 'Hi there, %s!' % name
```

```
def insult(name):
    '''Print out a nasty insult.'''
    print 'Get lost, %s!' % name
```



# Module docstrings

```
>>> help(greetings)
```

**FILE**

`greetings.py`

**DESCRIPTION**

`Module: greetings`

`Functions to print out greetings.`

**FUNCTIONS**

`greet(name)`

`Print out a hearty greeting.`

`insult(name)`

`Print out a nasty insult.`



# The point of docstrings

- We will expect you to write docstrings for all your functions and modules
- Docstrings are good documentation
  - for you
  - for you in the future
  - for anyone else that wants to use your modules/ functions



# What to put in docstrings

- For **functions**, a docstring should describe
  1. what the function does
  2. what the function **arguments** represent
  3. what the function **return value** represents
- For **modules**, a docstring should describe
  - the purpose of the module
  - general description of the kinds of functions in the module
    - but *not* a detailed description of the functions!
  - any other relevant information



# Experiment

- Let's type in a function with a docstring into the Python shell:

```
>>> def double(x):  
...     '''This function doubles its argument x.'''  
...     return 2 * x
```



# Experiment

- Now let's get some `help()`:

```
>>> help(double)
```

```
Help on function double in module __main__:
```

```
double(x)
```

This function doubles its argument `x`.

- What's all this `module __main__` stuff?



# The main module

- main is the name that Python gives to either
  - the interactive interpreter
  - the module which was directly invoked by Python
- All other modules are referred to by their own names
  - e.g. **greetings** module we defined before



# name

- Python also defines a variable called name which contains the name of the currently-executing module
  - We'll see a good use for this later on
- Many "special names" in Python have the form xxx for some **xxx**
  - We'll see many more of these



# builtins

- Python contains quite a few built-in functions
  - e.g. `abs`, `max`, `min`
- These functions actually live in a special module called  
`_builtins`
- To get documentation on all of them:

```
>>> help(_builtins_)
```

- Can also call builtin functions with a qualified name:

```
>>> _builtins_.abs(-5)
```

5



# Adminterlude

- Get registered if you aren't already!
- Problems with building access after hours:
  - Jeri Chittum: [jeri@cms.caltech.edu](mailto:jeri@cms.caltech.edu)
- You should be getting course-related emails from me
- Read the "Policies/FAQ" page on the web site before emailing me questions



# Adminterlude

- Lab sections will be posted shortly
  - will start on Sunday afternoon
- People who filled out questionnaire very late may not be in initial list
  - I will fill in ASAP
- Let me know if there is a problem



# Adminterlude

- Let's choose ombudspersons for each Caltech house!
- Get to have lunch with me every other Thursday at noon at the Athenaeum
  - starting tomorrow!
- Bring comments/criticism/praise/vilification from your house (anonymous feedback)



# Lists

- A *list* is a sequence of Python values
  - a way to take multiple values and create a single value that contains all the other values
  - Recall: strings are a sequence of characters
  - Lists are a sequence of *any* kind of value



# Why lists?

- Often have many related values that you'd like to store in a single object
  - e.g. average temperature each day for the last week
- Could define separate variables for each value
  - but it would quickly become tedious



# Without lists

```
temp_sunday      = 59.6
temp_monday      = 72.4
temp_tuesday     = 68.5
temp_wednesday   = 79.0
temp_thursday    = 66.4
temp_friday      = 77.1
temp_saturday    = -126.0 # new ice age?
```

- Hard to use this for anything



# Without lists

```
avg_temp = (temp_sunday + ...) / 7.0
```

- Tedious, inflexible



# With lists

```
temps = [59.6, 72.4, 68.5, 79.0,  
         66.4, 77.1, -126.0]
```

- Much simpler, easier to work with:

```
avg_temp = sum(temps) / 7.0
```

- Makes working with multiple values as easy as working with single ones
- Lists are used everywhere in Python code!



# Creating lists

- Create lists by putting Python values or expressions inside square brackets, separated by commas:

[1, 2, 3, 4, 5]

- Items inside list are called the *elements* of the list



# Creating lists

- Create lists by putting Python **values** or **expressions** inside square brackets, separated by commas:

[1, 2, 3, 4, 5]

↑  
↑  
↑  
↑  
↑

Python values



# Creating lists

- Create lists by putting Python values or expressions **inside square brackets**, separated by commas:

[1, 2, 3, 4, 5]

square brackets

A diagram illustrating the creation of a list in Python. It shows a list represented by square brackets containing five integers: 1, 2, 3, 4, and 5. Two red arrows point from the word "square brackets" at the bottom to the opening and closing square brackets of the list. The text "square brackets" is written in red.

# Creating lists

- Create lists by putting Python values or expressions inside square brackets, separated by commas:

[1, 2, 3, 4, 5]

commas



# Creating lists

- Any Python expression can be inside a list:

[1 + 3, 2 \* 2, 4]

- The expressions get evaluated when the list as a whole gets evaluated
  - so this list becomes [4, 4, 4]



# Creating lists

- Lists can contain expressions with variables:

```
>>> a = 10  
>>> [a, 2*a, 3*a]  
[10, 20, 30]
```



# Creating lists

- Lists can contain expressions with function calls:

```
>>> a = 4
```

```
>>> [a, 2*a, 3 * math.sqrt(a)]  
[4, 8, 6.0]
```

- (or any other Python expression)



# Creating lists

- Lists can contain other lists:

```
>>> a = 4
```

```
>>> [[a, 2*a], [3*a, 4*a]]
```

```
[[4, 8], [12, 16]]
```

- This is called a *nested list*



# Creating lists

- Lists can contain values of different types:  
`[1, 3.14, 'foobar', [0, 1]]`
- But most of the time they have values of the same type:

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

`[3.14, 2.718, 1.618]`

`['foo', 'bar', 'baz']`



# Accessing list elements

- Once a list is created, need to be able to get elements of list:

```
>>> temps = [59.6, 72.4, 68.5, 79.0,  
             66.4, 77.1, -126.0]
```

```
>>> temps[0]
```

```
59.6
```

```
>>> temps[2]
```

```
68.5
```



# Accessing list elements

- Syntax:

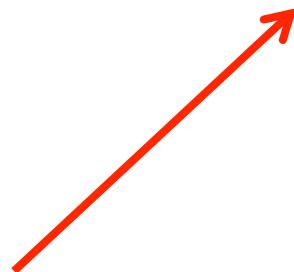
**temps [2]**



# Accessing list elements

- Syntax:

**temps [2]**



**name of the list**



# Accessing list elements

- Syntax:

**temps [2]**



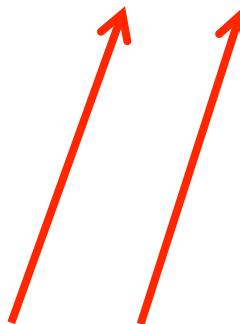
location (index) of desired element



# Accessing list elements

- Syntax:

**temps [2]**



square brackets separate name of list  
from index of element



# Accessing list elements

- NOTE: square brackets are being used for two distinct things:
  1. *creating lists*: `[1, 2, 3, 4, 5]`
  2. *accessing elements* of lists: `temps[2]`
- These are *completely different!*
  - Recall: Python likes to 'overload' syntax to mean different (but sometimes related) things



# Accessing list elements

- Can even have both meanings in one expression:

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

```
3
```

- (almost never see this in practice)



# Accessing list elements

- List indices start at 0, not 1

```
>>> nums = [12, 42, 31, 51, -32]  
>>> nums[0] # first element of nums  
12  
>>> nums[1] # second element of nums  
42
```

- This is common in computer languages
  - but easy to make mistakes if not careful



# Accessing list elements

- Can also access from the end of a list!

```
>>> nums = [12, 42, 31, 51, -32]  
>>> nums[-1] # last element  
-32  
>>> nums[-2] # second-last element  
51
```

- (but can't "wrap around")



# Accessing list elements

- Accessing off the ends of the list is an error:

```
>>> nums = [12, 42, 31, 51, -32]  
>>> nums[5]    # last element: nums[4]  
IndexError: list index out of range  
>>> nums[-6]   # first element: nums[-5]  
IndexError: list index out of range
```



# Empty list

- The empty list is written [ ]

```
>>> empty = []
```

```
>>> empty[0]
```

*IndexError: list index out of range*

- We'll see uses for this later



# Modifying lists

- Recall: Python strings are *immutable*
  - means: can't change them after making them
- Lists are *mutable*
  - means: can change them after making them



# Modifying lists

- Example:

```
>>> nums = [4, 6, 19, 2, -3]
```

```
>>> nums
```

```
[4, 6, 19, 2, -3]
```

```
>>> nums[2] = 501
```

```
>>> nums
```

```
[4, 6, 501, 2, -3]
```



# Modifying lists

- Syntax:

```
nums[2] = 501
```



# Modifying lists

- Syntax:

**nums [2] = 501**



element being modified



# Modifying lists

- Syntax:

**nums [2] = 501**



new value at that  
location in list



# Modifying lists

- Evaluation rule:
  - evaluate right-hand side expression
  - assign to location in list on left-hand side

```
>>> nums[2] = 3 * nums[2]
```

- `nums[2]` is 501, so `3 * nums[2]` is 1503

```
>>> nums
```

```
[4, 6, 1503, 2, -3]
```



# Modifying lists

- Can change an element to an element with a different type:

```
>>> nums  
[4, 6, 1503, 2, -3]  
>>> nums[2] = 'foobar'  
[4, 6, 'foobar', 2, -3]  
>>> nums[0] = [42, 'hello']  
>>> nums  
[[42, 'hello'], 6, 'foobar', 2, -3]
```



# Very quick demo

- Making music with Python!



# List operators

- Operators on lists behave much like operators on strings
- The **+** operator on lists means list concatenation
  - (like **+** with strings means string concatenation)

```
>>> [1, 2, 3] + [4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

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

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



# List operators

- The `*` operator on lists means list replication
  - (like `*` with strings means string replication)

```
>>> [1, 2, 3] * 3  
[1, 2, 3, 1, 2, 3, 1, 2, 3]  
>>> 0 * [1, 2, 3]  
[]  
>>> [1, 2, 3] * -1  
[]
```



# List functions

- Some built-in functions work on lists
- The **len** function returns the length of a list:

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

```
5
```

- The **list** function converts other sequences to lists, if possible:

```
>>> list('foobar')
```

```
['f', 'o', 'o', 'b', 'a', 'r']
```



# List methods

- Lots of useful methods on lists
- **append**

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

```
>>> lst.append(6)
```

```
>>> lst
```

```
[1,2,3,4,5,6]
```

- **append** adds a new element to the end of a list



# List methods

- Note that `append` changes the list it acts on
- Can use this to build up lists, starting from empty list

```
>>> lst = []
>>> lst.append(1)
>>> lst.append(2)
>>> lst.append(3)
>>> lst
[1, 2, 3]
```



# List methods

- To find an element's index (location) in a list, use the `index` method:

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

```
>>> lst.index(2)
```

```
1
```

```
>>> lst.index(42)
```

```
ValueError: list.index(x): x not  
in list
```



# List methods

- If an element has multiple copies in a list, `index` returns the index of first copy:

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

```
>>> lst.index(2)
```

```
1 # index of 1st 2 in list
```



# List methods

- To remove an element from a list, use the **remove** method:

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

```
>>> lst.remove(2)
```

```
>>> lst
```

```
[1, 3]
```

- Only removes first occurrence of element in list
- Error if element not found in list



# List methods

- To reverse a list, use the `reverse` method:

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

```
>>> lst.reverse()
```

```
>>> lst
```

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

- NOTE: the `reverse` method doesn't return the reversed list!
  - it reverses the list 'in-place' and doesn't return anything



# Pitfall: Aliasing

- *Aliasing* is a subtle issue which can come up when assigning lists to variables

```
>>> nums  
[4, 6, 1503, 2, -3]  
>>> nums2 = nums      # copy of nums?  
>>> nums2[0] = 0  
>>> nums2  
[0, 6, 1503, 2, -3]  
>>> nums  
[0, 6, 1503, 2, -3]      # !!!
```



# Aliasing

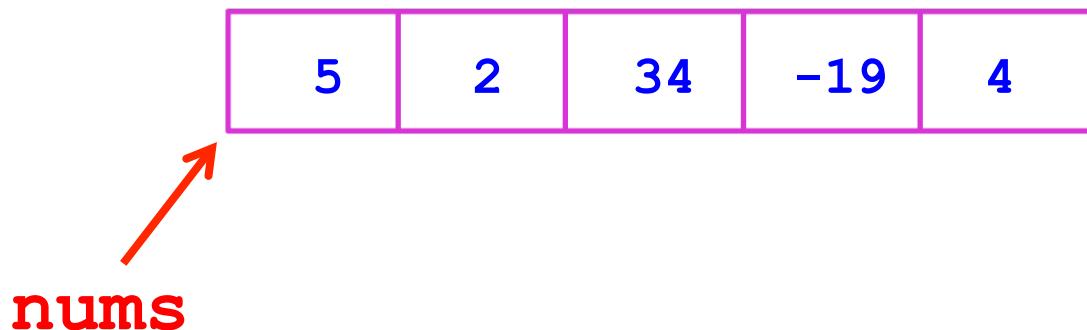
- The point: assigning a list to a variable does not copy the list!
- Python variables are not 'locations' or 'boxes' where stuff can be copied into
  - (unlike many other programming languages)
- Instead, after the assignment the variable name simply refers to the value on the right-hand side
  - we say that it's a *reference* to that value
  - so assigning doesn't copy anything (just get a new reference to the same value)



# Aliasing

- Visually:

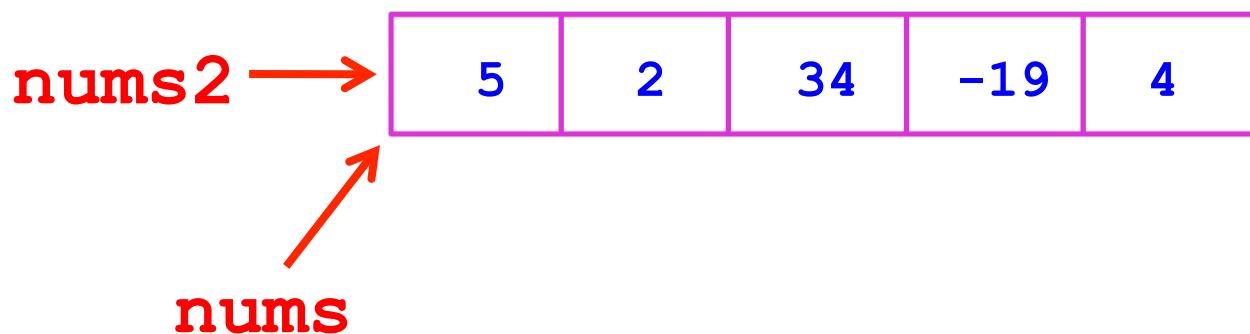
```
>>> nums = [5, 2, 34, -19, 4]
```



# Aliasing

- Visually:

```
>>> nums2 = nums
```



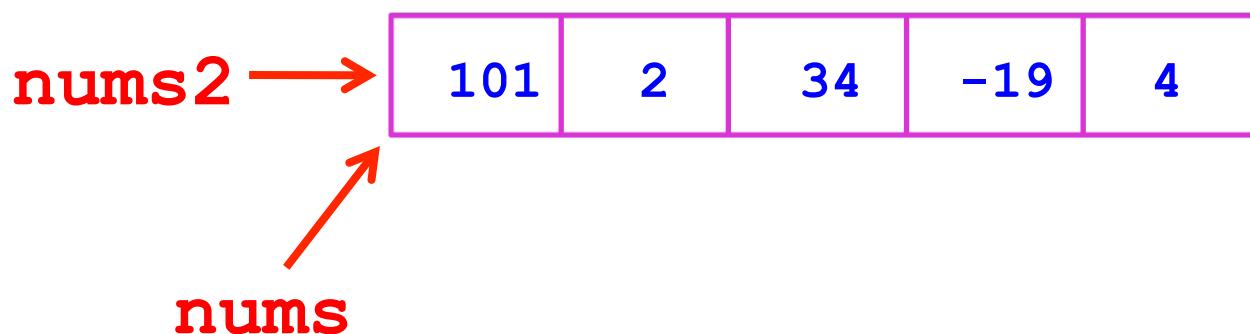
- `nums2` now refers to the same list as `nums`



# Aliasing

- Visually:

```
>>> nums2[0] = 101
```



- `nums` and `nums2` are the same list!
- So when `nums2[0]` changes, so does `nums[0]`



# Aliasing

- Aliasing does *not* happen with numbers or strings
  - only with container-like objects such as lists
- If we want to force Python to copy a list, we can do that too (see how later)



# Nested lists

- Lists of lists (nested lists) are fairly common:

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

- Can access elements like this:

```
lst[0][0] # 1
```

```
lst[0][1] # 2
```

```
lst[1][0] # 3
```

```
lst[1][1] # 4
```



# Nested lists

- This is not special syntax!
- Read it like this:

```
lst[0][0]
= (lst[0])[0]
= ([1, 2])[0]
= 1
```

- Can also assign to nested list location:

```
>>> lst[0][0] = 100
>>> lst  # → [[100, 2], [3, 4]]
```



# Next time

- Loops!
- Loops!
- Loops!
- Loops!
- Loops!

