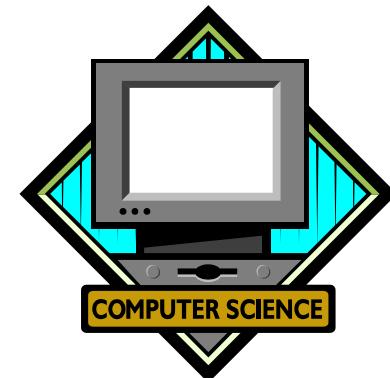
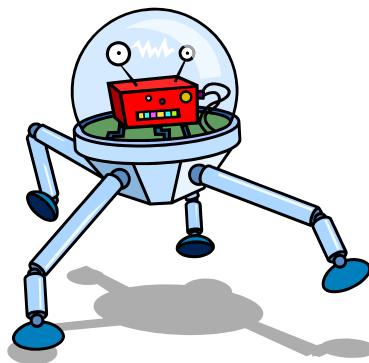


# Caltech/LEAD Summer 2012

## Computer Science



Lecture 5: July 12, 2012  
Modules (Libraries)



Caltech/LEAD CS: Summer 2012

# This lecture

- Modules
  - Importing modules
  - The `help()` function
  - Defining modules
  - Some useful modules



# Modules

- A "**module**" is a chunk of Python code that
  - exists in its own file
  - is intended to be used by Python code outside itself
- Modules can be "**imported**"
  - Functions in imported modules become available to code that imports them



# "Libraries"

- Modules are often informally referred to as "**libraries**"
- They are like little "books" of code that can be "checked out" (imported) and used wherever needed
- "Module" is the accepted Python term, so we'll use that



# What's in a module?

- Modules can contain any Python code
- Most often, modules contain
  - functions
  - values (constants)
  - classes
    - (we don't know what classes are yet)
- For now, we'll mainly be interested in modules that contain functions



# Why are modules important?

- Modules are the basic unit of *code reuse* in the Python language
- Modules allow you to bundle together your code in a form in which it can easily be used later (by yourself, or by others)



# Why are modules important?

- Countless pre-written modules are available in Python
  - math, graphics, strings, networking, etc.
- Generally, don't want to write code that has already been written by someone else
  - that's called "**reinventing the wheel**" and is bad practice



# Why are modules important?

- The best code is code you never had to write!
  - especially if it has been thoroughly debugged, has great documentation etc.
- Modules enable you to "**stand on the shoulders of giants**" and make programming vastly easier



# import

- To use a module, **import** it:

```
>>> import math
```

- This imports a module containing mathematical functions
- Then we can use these functions:

```
>>> math.sqrt(2.0)
```

```
1.4142135623730951
```

- This computes the square root (**sqrt**) of 2.0



# Dot syntax, revisited

- Consider: `math.sqrt(2.0)`
- Recall syntax of method calls on objects:

`object.method(args)`



# Dot syntax, revisited

- Consider: `math.sqrt(2.0)`
- Here we have:

`module.function(args)`

- Is this the same thing?
- Two ways to look at this



# Dot syntax, revisited

- Way 1: Python is overloading the dot syntax to mean two completely different things:
  1. **object.method(args)**
  2. **module.function(args)**



# Dot syntax, revisited

- Way 2: Python is generalizing the dot syntax to mean two kind-of related things:
  1. `object.method(args)`
  2. `module.function(args)`
- **function** and **method** are both basically functions that exist in some kind of container (module or object)



# Dot syntax, revisited

- It doesn't matter which way you think about it
  - just make sure you understand both meanings!
- Usually easy to tell which is which
  - standard module names aren't usually names of objects



# Modules are objects!

- In fact, Python modules are a kind of object!
- That means that you can define functions that act on modules
  - We'll get back to this



# Back to import

- Consider again:

```
>>> import math
```

```
>>> math.sqrt(2.0)
```

```
1.4142135623730951
```

- Problem: writing `math.sqrt` instead of just `sqrt` is pretty verbose
- Is there a shorter way?



# from X import Y

- Instead, can do:

```
>>> from math import sqrt  
>>> sqrt(2.0)
```

```
1.4142135623730951
```

- More concise
- But not necessarily a good thing!
  - We'll get back to this too



# Pitfalls (I)

- Without importing `math`, try:

```
>>> math.sqrt(2.0)
```

```
NameError: name 'math' is not defined
```

- You have to import the `math` module before using any of its functions!



# Pitfalls (2)

- Try:

```
>>> import math
```

```
>>> sqrt(2.0)
```

```
NameError: name 'sqrt' is not defined
```

- Using **import** this way only allows you to use the **module.function** form of the imported functions (**math.sqrt**, not **sqrt**)
- Function names with module prefixes are called "*qualified*" names



# Multiple imports (I)

- Can import more than one module at a time:

```
>>> import math, string, time
```

- Now can use any function in the **math**, **string**, or **time** module as long as you qualify it
  - e.g. **math.sqrt**, **string.upper**, **time.localtime**



# Multiple imports (2)

- Can import more than one name from a particular module at a time:

```
>>> from math import sin, cos, tan
```

- Now can use **sin**, **cos**, and **tan** functions without qualifying them



# Multiple imports (3)

- Can import *all* names from a module:

```
>>> from math import *
```

- The \* means "every name in the module"
- Now can use *any* functions in the **math** module without qualifying them!
- Sounds great, huh? Not so fast...



# Name clashes

- Suppose two modules define a `sin` function:

```
>>> from math import sin  
>>> from evil import sin
```

- What will happen?
- Only one definition of `sin` can exist at any given point in the program
- `sin` means "the most recently defined or imported `sin`"



# Name clashes

- When *name clashes* happen, some functions become unusable because their name is being used by another function
- Indiscriminate use of `from X import Y` form of the `import` statement increases the possibility of name clashes of this kind
- Using `from X import *` form makes name clashes even more likely!



# Advice on import

- Avoid using the `from X import *` form of the `import` statement!
- If you use plain `import`, no problems occur:

```
>>> import math
```

```
>>> import evil
```

```
>>> math.sin(0.5)
```

```
0.47942553860420301
```

```
>>> evil.sin("bear false witness")
```

"The check is in the mail!"



# import X as Y

- There's a cool trick you can use to get shorter module names

```
>>> import math as m
```

```
>>> m.sin(0.5)
```

```
0.47942553860420301
```

- The `m.` takes the place of `math.` but you don't have to worry about name conflicts
  - the best of both worlds!



# Documentation

- Modules can contain a *lot* of functions, values etc.
  - e.g. the `os` module contains 217 different functions/values, the `math` module has 46, the `string` module has 60
- How do we learn about what's in a module?



# Documentation

- *Approach 1:* go to the Python web site and look it up
- Python web site: [www.python.org](http://www.python.org)
- Documentation: [docs.python.org/library](http://docs.python.org/library)
- Useful for browsing, especially when you don't know the name of the module you want



# Documentation

- Approach 2: use the `help()` function built in to Python:

```
>>> help(math.sqrt)
```

```
Help on built-in function sqrt in module  
math:
```

```
sqrt(...)
```

```
sqrt(x)
```

Return the square root of x.



# Documentation

- This works for entire modules too:

```
>>> help(math)
```

Help on built-in module math:

...

## FUNCTIONS

acos(...) ...

asin(...) ...

(etc.)



# Notes

- Where does all this documentation come from?
  - we'll see shortly
- `help(math.sqrt)` is a normal function call
  - which means that `math.sqrt` is something you can pass as an argument to a function!
- `help(math)` is also a normal function call
  - which means that `math` is something you can pass as an argument to a function!



# "First class"

- In Python, both functions and modules are "first-class"
- This means that they are valid Python objects, and can be passed as arguments to functions
- This leads to "functional programming"
  - (covered in supplemental lab)



# Pitfalls again

```
>>> import math  
>>> help(math.sqrt)
```

[OK]

```
>>> help(sqrt)
```

```
NameError: name 'sqrt' is not defined
```

```
>>> from math import sqrt
```

```
>>> help(sqrt)
```

[OK]

- `help()` will only work if the name of the argument is known to Python (has been imported or defined locally)



# Defining modules

- Many modules are available ("bundled") with Python, so you can use them immediately
  - We say that Python comes with "**batteries included**"
- Many others can be added later by e.g. downloading them from the internet
- Often you want to define your *own* modules
  - allows you to re-use useful code in multiple programs
- How can you do this?



# Good news, everyone!

- Defining modules in Python is trivially easy!
- Recall that we put our Python code in files which have a filename that ends in "**.py**"
  - the standard Python source code filename extension
- If you do this, your code *automatically* becomes part of a module!
- So if your file is named "**widgets.py**"
  - then that becomes the **widgets** module
  - can be imported/used like any other module



# Simple module

- We'll define a very simple module called **greetings** which prints out greeting messages
- It will go into a file named **greetings.py**



# Simple module

```
# Module: greetings
# Filename: greetings.py

def greet(name):
    print "Hi there, %s!" % name

def insult(name):
    print "Get lost, %s!" % name
```



# Simple module

- Type this code into a file called "**greetings.py**" and save it
- Now from a Python shell, can use it:

```
>>> import greetings
>>> greetings.greet("Mike")
Hi there, Mike!
>>> greetings.insult("Mike")
Get lost, Mike!
```



# Simple module

- Or, can import the other way:

```
>>> from greetings import *
>>> greet("Mike")
Hi there, Mike!
>>> insult("Mike")
Get lost, Mike!
```

- This is *not* recommended!



# The point

- In Python, writing normal code is the *same* as writing a module!
- Downside: Python modules lack some features that exist in some other languages
  - e.g. all names defined in a module are exported by default
  - but this is rarely a problem in practice



# Module contents

- Modules can contain things other than functions:

```
>>> import math
```

```
>>> math.pi
```

```
3.1415926535897931
```

```
>>> math.e
```

```
2.7182818284590451
```

```
>>> math.pi = 3      # !!!
```

- (No constants in Python)



# Module contents

- Modules can also execute code while being imported
  - in fact, that *is* what happens during import
- But will not normally re-execute the code if imported again
  - Python "knows" that module has already been imported, doesn't redo it



# Module contents

```
# Module: test.py
print "This is module test.py"
def triple(x):
    return 3 * x
```

- And in the Python shell:

```
>>> import test
This is module test.py
>>> import test
>>>
```

(module test not reloaded)



# Some useful modules

- **math**: standard math functions and values
- **cmath**: complex number math
- **string**: string functions and values
- **random**: random numbers
- **sys**: system-specific functions and values
- **os**: operating system functions
- **email**: email parsing
- **HTMLParser**: web page processing



# 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

- Same function as before, 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 want 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



# Next lecture

- Function evaluation in depth
- Coding style
- The `range()` function
- `while` loops and `break`
- Random numbers and the `random` module

