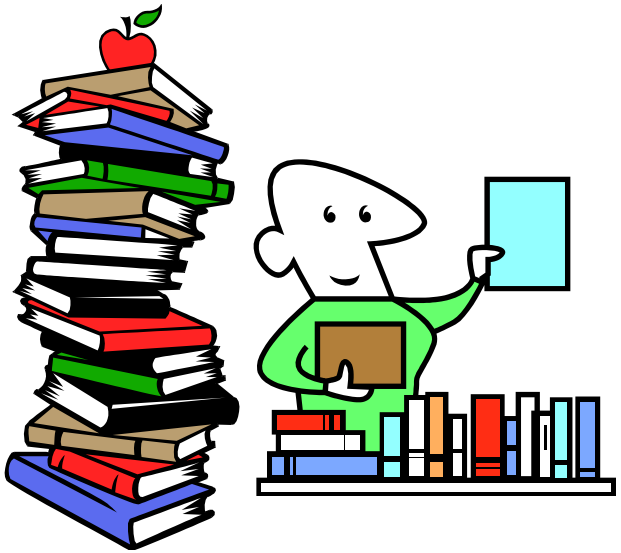
CS 1
Introduction to Computer Programming

Lecture 4: October 6, 2014

Objects and Modules (Libraries)



Last time

- Strings
- String operators
- String formatting
- Linux tutorial

Today

- User input using `raw_input`
- Objects
  - Method syntax (dot syntax)
  - String methods
- Modules
  - Importing modules
  - The `help()` function
  - Defining modules
  - Some useful modules

User input

- Problem: how to write interactive programs?
  - At some point in the execution of the program, the program has to stop and ask the user for information, get it, then continue
  - Various ways exist to do this
  - Today we'll look at one way: the `raw_input()` function

User input

- `raw_input()` returns a single line read from the keyboard:

```
>>> line = raw_input()
I'm a lumberjack and I'm OK.
>>> line
"I'm a lumberjack and I'm OK."
```

[you type this]

- Question: Why does Python print this with double quotes?
- `raw_input()` also removes the end-of-line (`\n`) character from the line read in
  - which you had to type to finish the line

User input

- **`raw_input`** can take a string argument, which is used as a prompt:

```
>>> SAT = raw_input("Enter your SAT score: ")
Enter your SAT score: 2500
>>> SAT
'2500'
```

- Note that **`SAT`** is still a string!
- **`raw_`** in **`raw_input`** means that the input value is just represented as a string

User input

- To convert to an **`int`**, use the **`int`** conversion function:

```
>>> SATs = raw_input("Enter your SAT score: ")
Enter your SAT score: 2500
>>> SAT = int(SATs)
>>> SAT
2500
```

- Usually, this would be written more simply as:

```
>>> SAT = int(raw_input("Enter your SAT score: "))
```

- Other conversion functions exist: **`float`**, **`str`**, etc.
  - convert from some data item to a particular type
  - an error occurs if conversion can't be done

Strings are objects

- Python is what's known as an "object-oriented" programming language
  - What that means will take many lectures to completely describe
- One aspect is that most nontrivial data types are represented as "objects"
- An "object" is some data + some associated functions that work on that data
- Python strings are an example of a Python object

Object syntax

- Functions that are associated with an object are referred to as methods
  - they're like functions, but the syntax is different (and there are other differences which we'll learn about later)
- Example method: the `upper` method returns an upper-cased version of a string

```
>>> 'spam'.upper()
'SPAM'
```

Object syntax

```
'spam'.upper()
```

Object syntax

```
'spam'.upper()
```

the object (being acted upon)

Object syntax

```
'spam'.upper()
```

name of the method

Object syntax

The dot indicates that this is a *method call*
(like a function call, but on this object)

Object syntax

```
'spam'.upper()
```

argument list

(The argument list is empty in this case.)

Object syntax

- Object syntax is used everywhere in Python, so get used to it!
- Can use on variables too:

```
>>> s = 'spam'
>>> s.upper()
'SPAM'
```

- The name `s` is bound to a string, so this works

Dot syntax

- Sometimes I will refer to the `object.method()` syntax as "dot syntax"
- As we'll see, it has wider applicability than just to objects

String methods

- Lots of string methods, *e.g.*:
- `lower` – converts string to lower case
- `strip` – removes leading/trailing spaces
- `endswith` – check if string ends in another string (`'foobar'.endswith('bar')` → `True`)
- `islower` – check if all characters in string are lower-case (`'foobar'.islower()` → `True`)
- *etc.*
- Consult Python documentation for full list

len

- One function you'd expect to be a method is called `len()`
- It calculates the length of a sequence (which includes strings)

```
>>> len('foobar')
6
```

- If you try this:

```
>>> 'foobar'.len()
```

you get:

```
AttributeError: 'str' object has no
  attribute 'len'
```

`len`

- *Moral:* Python doesn't always do everything in a completely consistent way!
- Sometimes there is a choice: a certain operation can be written as either a function or a method
- Which one Python chooses to use is somewhat arbitrary
  - Sometimes you get both a function *and* a method!
- *Most* of the time, operations on objects are done using methods

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)
  - often in multiple unrelated programs

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
```

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 (1)

- 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 (1)

- 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.capitalize`**, **`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* function 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

- We call this a *name clash*
- 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 Y`** *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 as

- Here is a neat variation of the `import` statement

```
>>> import math as m
>>> m.sqrt(2.0)
1.4142135623730951
```

- The `as m` means that you can qualify the name with just the prefix `m.` (or whatever name you choose) instead of `math.`
- This is safe and convenient, so we recommend it!

Documentation

- Modules can contain a *lot* of functions, values etc.
  - *e.g.* the `os` module contains 220 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`**
- Documentation: **`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
- We will revisit this idea much later in course
  - leads to "functional programming"

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, but it works

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
3.141592653589793
>>> math.e
2.718281828459045
>>> 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 interface
- **`re`**: regular expressions (string processing)
- **`email`**: email parsing
- **`HTMLParser`**: web page processing

Next lecture

- Lists
  - provide way to bundle together arbitrary objects into a single object
  - another kind of sequence
- Docstrings
  - a convenient way to document your code