Taylor Swift Is Angry, Darn It

“I feel like in my music I can be a rebel,” Ms. Swift said at a quiet restaurant not far from her new Nashville apartment.
Last time

Last time we started by having a look at functions -- We then covered some coding strategies in Python to deal with uncertainties as well as a simple Python debugger to assess the operation of your running code

We ended with a brief introduction to MongoDB, an option we will revisit later in the quarter but one that some of you were interested in for your recipe project

We closed with your next homework assignment!
Re-cap: Naming

So far we’ve seen several examples of naming schemes -- Whether it be URLs (uniform resource locators) on the web, IP addresses or variables in Python

We have seen names functioning as “a communication and an organizing tool”, letting us share information between programs or different parts of a single program -- Names are also useful for system designers as they can defer deciding to which object a name actually refers (The first use of names is known as ***indirection***, while the second is an example of ***binding***)

A naming scheme consists of three elements: A **collection of names** (an alphabet of symbols together with syntax rules describing which names are acceptable), a **name-mapping algorithm** (associating some names with values), and a **universe of values** (in Python, objects, for example)
Re-cap: Naming

When faced with a new naming system, we have (or should have) asked some basic questions

What is the syntax for a name and its values?

What is the name-mapping algorithm?

Where does a name take its context from?
Re-cap: Scoping rules in Python

As we have seen, when we use a name in a program (in an assignment or a simple expression, say), Python needs to associate that name with some object -- This is where the name-mapping algorithm comes in

The rules used by the algorithm in Python depend on where the assignment statement is located in our code -- A key concept being the name’s scope
Re-cap: Scoping rules

When we first started writing Python code, all our variable assignments took place at the top level of a module*; that is, their names were part of the module's namespace, or the “global scope” of the module and we could refer to them simply

Notice that this sense of “global” is really file-based; that is, when we write a module, we don’t have to worry about whether someone using our module has defined variables of the same name in their code

With functions, we introduce a nested namespace (a nested scope) that localizes the names they use so that you can avoid similar kinds of clashes

* If it is typed in at the “>>>” prompt, you are in a module called __main__; otherwise the enclosing module is the file that contains your program
Re-cap: Scoping rules

The execution of a function introduces a new namespace for the “local” variables of the function; all variable assignments in a function store their names in this local namespace.

When we look up variable name (by referring to it in an expression in the function, say), Python first looks in this local namespace; if it can’t find an object of that name locally, it starts a search that moves out to (eventually) the global namespace and then the collection of built-in names.

During this lookup process, Python will return the object associated with the first instance of the name it finds; so local names take precedence over globals.
>>> # built-in scope

>>> import __builtin__

>>> dir(__builtin__)
Re-cap: Scoping rules

Because `def` is a statement like any other, we can certainly define functions within other functions; in that case, **our search for variables works way out through the enclosing functions**

Lutz defines the LEGB rule for resolving a name: When a name is referenced, Python will look it up in the following order:

1. The Local (function) scope
2. The Enclosing function scope
3. The Global (module) scope, and
4. The Built-in scope
The names in your argument list become new local names (local variables) and arguments are passed to a function by assigning objects to local names; that means the variable names in the argument list are assigned references to the objects you specify (this is VERY different from what you will see in R).

For immutable objects like numbers or strings, this is safe enough; for mutable objects like lists and dictionaries, this can produce some unexpected consequences...
Today

We are going to start with a treatment of user-defined objects in Python -- So far we have only used aspects of the object-oriented structure in Python and our own work hasn’t involved creating new objects.

We’ll take the example of a browser and examine how Python implements this object, the attributes and methods it supports -- We’ll then transition to something closer to your new homework assignment.
Coding paradigms*

Programming the first computers (before the 1940s) was a physical process of setting switches; in so doing, the machine was “hard wired” to perform a particular task.

In the 1940s, the mathematician John von Neumann came up with the idea of a central processing unit that would not have to be hard wired, but instead a central processing unit would act on a series of codes or instructions stored as data; this led to “assembly language” a very low-level, machine dependent kind of programming abstraction.

Programmers quickly realized that higher-levels of abstraction could help them write understandable, concise instructions that could operate on any machine; some basic ideas like iteration, assignment, conditional execution were useful no matter what machine was being programmed.

* Summarized from “Programming Languages: Principles and Practice” by Kenneth Louden
Coding paradigms

Early higher-level programming languages were still abstractions of how a “computer” operated; of course the design of these languages depended to a large extent on the kind of computer they had in mind.

Through the work of von Neumann, Church and Turing, an image of a computer emerged that involved a central processing unit that executed instructions sequentially, and a separate memory area that could be used to store data and code (instructions).

This model gave rise to a kind of programming language that computer scientists refer to as an imperative language; that is, “its primary feature is a sequence of statements that represent commands or imperatives.”
Coding paradigms

But imperative programming languages are just one of several paradigms or patterns for a language to follow; for example, with Python we see object-oriented programming and with R we will see something closer to functional programming.

While broad characterizations are important when you start to learn a new language, it’s often the case that you can write programs in a language that make it look like one of the other paradigms -- This is particularly true of many of the popular so-called scripting languages.
Object-oriented programming*

As its name suggests, this style of programming starts with objects -- In the “real world” objects abound

These notes, your chair, the video projector

These objects share two characteristics: **State and behavior**

These notes refer to a particular topic on a particular day (state), the video projector displays images from different computer sources (behavior)

* Borrowed liberally from the Sun Java Tutorial
Object-oriented programming

Software objects are modeled after real world objects in that they also have state and behavior -- A software object **maintains its state in one or more variables**; named pieces of data (a particular set of memory locations)

Software objects **implement behaviors with methods**; a function or operation associated with an object
An object is a software bundle of variables and related methods
Object-oriented programming

Let's take a little time and think about one particular kind of object and think about the data it might contain and the operations it might perform

Because we've been furiously scraping the web, let's look for objects associated with web browsing

From a previous lecture ...
Recall...

We saw how we could issue requests for web pages using the Python module urllib; below, we simply invoke a method from the module

```python
>>> import urllib2
>>> response = urllib2.urlopen("http://www.nytimes.com")
>>> html = response.read()
>>> type(html)
<type 'str'>
```

With this one method call, we can grab the content of a page and, perhaps, parse it with a tool like Beautiful Soup to identify or extract various kinds of data from the page

How does this compare to the way you browse the web?
Afghan Leader Admits His Office Gets Cash From Iran

President Hamid Karzai said on Monday that the United States has known about the Iranian assistance for years.

Canadian Pleads Guilty in Guantanamo Trial

Omar Khadr pleaded guilty on Monday to all five terrorism charges against him as part of a deal.

Cuomo Vows to Stand Up to Labor Unions

Andrew M. Cuomo, the Democratic nominee for governor of New York, is looking ahead to battles after the elections.

OPINION »

Is Outsider Politics Here to Stay?
The new mainstream: Oddball candidates and the voters who support them.

WHAT'S POPULAR NOW

- Krugman: Falling Into the Economic Chasm
- Douthat: The Great Bailout Backlash
- Editorial: An Indefensible Defense
- The Stone: Stories vs. Stats
Object-oriented programming

Let’s consider a particular object, a web browser

1. **The state** or variables describing your browser might include information about the page you’re on like its **links and forms**; data that the site hosting the page wants you to remember in the form of **cookies**; and a record of where you’ve been, **you’re history**

2. **Behaviors or methods** might include navigational tasks like **following links**, **reloading a page** or **hitting the “back” button**; or more service-oriented tasks like **submitting forms**; or more book keeping tasks like **remembering and setting cookies**
Object-oriented programming

One commonly used cartoon describing this programming paradigm places **variables at the center** or nucleus of the object -- **Methods**, then, surround the nucleus and **regulate access to the object’s variables**
Object-oriented programming

When we consider a specific object, it is known as an **instance**; in the case of our example, an instance would be a particular browser.
Object-oriented programming

Thinking about organizing data and computations in this way is said to have two advantages:

**Modularity:** The source code for an object can be written and maintained independently of the source code for other objects -- Also, objects can be shared easily.

**Encapsulation:** An object has a public interface that other objects can use to communicate with it -- The object can maintain information and methods that can be changed at any time without affecting the other objects that depend on it.
Object-oriented programming

You can have many real-world objects of the same kind; as we go through our day, we might interact with many different browsers (on our smartphones or our desktops or a random lab machine), but from the standpoint of this exercise, they're **all the same type**

We can think of our description of a browser, its behaviors and possible states, as a **blueprint for the class**
A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.
Object-oriented programming

We then create instances of the class by assigning values to the instance variables. Remember, an object in Python has a type, an identity and a value.
Object-oriented programming

So, knowing the class of an object tells you a lot about it; we know a browser will have a URL, some history, cookies, etc.

Object oriented programming goes a step farther and lets you define classes in terms of other classes -- A browser is a subclass of a web-based agent; a web-based agent might also be a superclass for a news reader
Object-oriented programming

Each subclass is said to **inherit its state variables and methods from its superclass** -- Subclasses can also **add states and variables** to those provided by its superclass

It is possible that **subclasses override inherited methods**, providing specialized implementations for those methods

Chaining together sub- and superclasses, you obtain an **inheritance tree or class hierarchy that can be quite deep** -- One of the beauties of this kind of programming is that the tree becomes rich rather quickly
Ok, so what?

Many languages support (or rigidly enforce) an object-oriented style of programming -- in terms of the languages we will (or might) see in this class...

- Java is religious about it (and as a result, so is Processing)
- Python is all about objects (although class creation is often left to later chapters in text books), and
- R's object-orientation is really more a dispatch mechanism that might be called object-based programming (a distinction we'll make in a couple weeks)
Object-oriented programming in Python

While we have done quite a lot of analysis so far in Python without explicitly discussing how objects are defined or organized in the language, we have been fed a steady diet of “everything is an object in Python”

And we have certainly seen syntax that comes directly from Python’s object model -- expressions like `x.sort()` for an object of type `str` (a string) is an example

Beyond explaining syntax, it helps to understand how Python implements objects -- In fact, it’s one of the most powerful tools that the language provides
An example

Rather than having you build a browser yourself, I thought we would look at how someone else has gone through the process

We will focus on a module called Mechanize* -- It is designed for “stateful programmatic web browsing in Python”

We will focus specifically on the class Browser; in Python, we capture states and behaviors through a series of attributes that name functions and variables...

*http://wwwsearch.sourceforge.net/mechanize
A browser

Here we have a piece of the Mechanize code for the class Browser; for a first go, I’ve stripped away all the contents of the functions

Notice that the class definition creates another code block with its corresponding indentation

A `def` statement within a class declaration defines a method, a function that operates on instances of the class

Within that block, we see a number of methods that we can access; what do you see?

class Browser(UserAgentBase):
    def __init__(self, history=None):
    def __str__(self):
    def open(self, url, data=None):
    def close(self):
    def reload(self):
    def back(self, n=1):
    def links(self, **kwds):
    def click_link(self, link=None, **kwds):
    def follow_link(self, link=None, **kwds):
    def find_link(self, **kwds):
    def forms(self):
    def select_form(self, name=None, predicate=None, nr=None):
    def submit(self, *args, **kwds):
    def title(self):
    def geturl(self):
    def viewing_html(self):
    def encoding(self):
    def set_cookie(self, cookie_string):
    def response(self):
    def visit_response(self, response, request=None):
    def clear_history(self):
    def open_local_file(self, filename):
Creating a browser

The first function has a special name `__init__` and is a constructor for the class; it takes an (optional) argument `history`, meaning you can create a browser with a previous history of fetching pages.

Notice also that all of the methods we have defined start with a reference to `self`.

class Browser(UserAgentBase):
    def __init__(self, history=None):
    def __str__(self):
    def open(self, url, data=None):
    def close(self):
    def reload(self):
    def back(self, n=1):
    def links(self, **kwds):
    def click_link(self, link=None, **kwds):
    def follow_link(self, link=None, **kwds):
    def find_link(self, **kwds):
    def forms(self):
    def select_form(self, name=None, predicate=None, nr=None):
    def submit(self, *args, **kwds):
    def title(self):
    def geturl(self):
    def viewing_html(self):
    def encoding(self):
    def set_cookie(self, cookie_string):
    def response(self):
    def visit_response(self, response, request=None):
    def clear_history(self):
    def open_local_file(self, filename):
Creating a browser

The use of `self` provides Python a handle back to the instance being processed -- in fact, when we make a method call like

```python
instance.method(args,...)
```

Python will automatically transform it to

```python
class.method(instance,args,...)
```

so we can use any variable name we like -- `self` is customary and will make your code readable by other Python programmers.
Some special methods

Several specially named methods might be included in the class definition

__init__ A customizable constructor -- Called when the object is instantiated

__del__ A customizable destructor -- This is called when the object is deleted

__repr__ A customizable representer -- This is called when "displaying" the object, say after you type its name into the interpreter or use the built-in function repr

__str__ A customizable string-er -- This is called when, for example, we need to print the object (this is an "informal" representation, whereas __repr__ is the "official" representation)
An example

Here is the `__str__` method for the class Browser -- Aside from the string formatting arguments that might look foreign, you can see that the method is building up a list of text components (called `text`) describing the “state” the browser is in and then `join-ing` them together...

```python
def __str__(self):
    text = []
    text.append("<%s " % self.__class__.__name__)
    if self._response:
        text.append("visiting %s" % self._response.geturl())
    else:
        text.append("(not visiting a URL)"
    if self.form:
        text.append("\n selected form:\n %s\n" % str(self.form))
    text.append(">")
    return ".join(text)
```

Let's create a Browser instance and see how it works...
Aside: String formatting

String and unicode objects have one unique built-in operation, the % operator (also known as the string formatting or interpolation operator) -- Given format % values (where format is a string or unicode object), % conversion specifications in format are replaced with zero or more elements of values

```python
>>> "We are counting to %d" % 5
'We are counting to 5'
>>> "We are counting to %.2f" % 5.333333
'We are counting to 5.33'
>>> "We are counting to %s" % "five"
'We are counting to five'

>>> 'Today is %(day)s and the hour is %(hour)03d.' % 
...   {'day': "Monday", "hour": 4}
'Today is Monday and the hour is 004 quote types.'
```

Now, back to browsing...
# import the browser and the html parser...
>>> from mechanize import Browser
>>> from BeautifulSoup import BeautifulSoup

>>> br = Browser() # a “constructor” creating an object of type Browser
>>> print br # calling the __str__ method (next page)
<Browser (not visiting a URL)>

# open a url, the nytimes home page
>>> response = br.open("http://www.nytimes.com")

# when we print the object, we now see reference to the nytimes
>>> print br
<Browser visiting http://www.nytimes.com>

# we can examine the title and url of the page we’re visiting
>>> print br.title()
>>> print br.geturl()
http://www.nytimes.com
Afghan Leader Admits His Office Gets Cash From Iran
By ALISSA J. RUBIN
57 minutes ago
President Hamid Karzai said on Monday that the United States has known about the Iranian assistance for years.

Canadian Pleads Guilty in Guantánamo Trial
By REUTERS 39 minutes ago
Omar Khadr pleaded guilty on Monday to all five terrorism charges against him as part of a deal.

Cuomo Vows to Stand Up to Labor Unions
By NICHOLAS CONFESSORE
Andrew M. Cuomo, the Democratic nominee for governor of New York, is looking ahead to battles after the elections.

Mortgage Lenders Under Review, Bernanke Says
By THE ASSOCIATED PRESS 34 minutes ago

OPINION
Is Outsider Politics Here to Stay?
The new mainstream: Oddball candidates and the voters who support them.

WHAT'S POPULAR NOW
• Falling Into the Economic Chasm
• A Few Transgender Candidates Seem in Line for Victories

Room for Debate
Fallen Giants
• Krugman: Falling Into the Chasm | Comments (95)
• Douthat: The Great Bailout Backlash
• Editorial: An Indefensible Defense
• The Stone: Stories vs. Stats
WASHINGTON — It took President Obama 18 months to invite the Senate Republican leader, Mitch McConnell, to the White House for a one-on-one chat. Their Aug. 4 session in the Oval Office; 30 minutes of private time, interrupted only when the president’s daughter Malia called from summer camp to wish her father a happy 49th birthday; was remarkable, not for what was said, but for what it took to make it happen.
WASHINGTON — It took President Obama 18 months to invite the Senate Republican leader, Mitch McConnell, to the White House for a one-on-one chat. Their Aug. 4 session in the Oval Office — 30 minutes of private time, interrupted only when the president’s daughter Malia called from summer camp to wish her father a happy 49th birthday — was remarkable, not for what was said, but for what it took to make it happen.

Not long before the meeting, Trent Lott, the former Republican Senate leader, lamented to his onetime Democratic counterpart, Tom Daschle, that Mr. Obama would never get an important nuclear arms treaty with Russia ratified until he consulted top Republicans. Mr. Lott, who recounted the exchange in an interview, was counting on Mr. Daschle, a close Obama ally, to convey the message; lo and behold, Mr. McConnell soon had an audience with the president.
White House Memo

Obama’s Playbook After Nov. 2

By Sheryl Gay Stolberg

More Articles by Sheryl Gay Stolberg

Published: October 24, 2010
# now, a little navigation -- having followed a link, we can move back;
# the equivalent of hitting the “back” button on a browser

```python
>>> response = br.back()
```

# the following would also specify the same article
```python
>>> response = br.follow_link(url_regex="war-logs")
```

# and back to the home page
```python
>>> response = br.back()
```

# we can also specify links using the text of the url; here we use
# the fact that urls of stories in the times are formatted with their date -- here
# are links to stories on the home page that were published on Sunday...

```python
>>> for link in br.links(url_regex=r"2010/10/25"):
...    print link
...    response = br.follow_link(link)
...    print br
...    response = br.back()
...```
Forms

With Mechanize, you can submit forms fairly easily, illustrated here by listing out all the forms on a page and documenting the various inputs -- As we do this, it should be clear (for the second time, perhaps) the ease with which one can automatically scrape data from the web.

We should add that Mechanize, while a wonderful tool, has had much of its functionality subsumed into urllib2 -- It is often simpler to use this module rather than Mechanize, but the object structure in Mechanize makes a good lesson.
# clear things up and open the homepage fresh again

```python
>>> response = br.open("http://www.nytimes.com")
```

# now, let’s consider the forms on the page; that is, places where you are asked to type input or make selections

```python
>>> for f in br.forms():
...    print "===== Form %s =====" % f.name
...    print f
...
```

===== Form  searchForm =====
```
    <TextControl(query=)>
    <HiddenControl(type=nyt) (readonly)>
    <ImageControl(<None>=)>
```

===== Form  None =====
```
    <TextControl(symb=Stock, ETFs, Funds)>
    <ImageControl(<None>=)>
```

[output truncated]
# Now, let’s submit a search

```python
>>> response = br.select_form("searchForm")
>>> br["query"] = "iraq war logs"  # (the method here is __setitem__)

# here our query string (the query field in the form); we now submit the form
# and catch the response

>>> response = br.submit()
```  

```python
>>> print br

>>> bs = BeautifulSoup(response.read())

# and here we can use beautifulsoup to extract the bylines associated with the
# different stories

>>> results = bs.findAll(attrs={"class":"byline"})

>>> results[0]
<span class="byline">Atwar Blog</span>
>>> results[1]
<span class="byline"></span>
```
Iraq war logs — NYT Times.com Search

Your Search: iraq war logs

Today | Past 7 Days | Past 30 Days | Past 12 Months | All Results Since 1851

All Result Types | Articles | Multimedia

1-10 of 10,000+ Results

Sponsored Links

Log files Log Analysis & Reporting Software For Windows, Linux, & Cisco Devices
ManageEngine.com/EventLogAnalyzer

Iraq Conflict Our Troops are There Right Now. Show Your Support for Our Troops!
USO.org

Ask The New York Times: Wikileaks Iraq War Logs
The New York Times reporters who wrote about a trove of classified Iraq war military documents disclosed by Wikileaks last week will be ...
Atwar Blog

Iraq War Logs Show Grim Portrait of Civilian Deaths – Iraq War ...
The reports show that most civilians were killed by other Iraqis, but they also reveal many instances in which American soldiers killed ...

The War Logs: Reaction to Newly Disclosed Iraq War Reports ...
The At War blog is following reaction to the release of classified American field reports from the Iraq War, which contain information about ...
Atwar Blog

Growing Use of Contractors Added to Iraq War’s Chaos - Iraq War ...
Contractors were necessary at the start of the Iraq war because there simply were not .... Detainees Fared Worse in Iraqi Hands, Logs Say ...
Creating a browser

One more look at the block statement; notice that first line makes reference to
UserAgentBase

This syntax means that Browser is extending the class
UserAgentBase

class Browser(UserAgentBase):
    def __init__(self, history=None):
    def __str__(self):
    def open(self, url, data=None):
    def close(self):
    def reload(self):
    def back(self, n=1):
    def links(self, **kwds):
    def click_link(self, link=None, **kwds):
    def follow_link(self, link=None, **kwds):
    def find_link(self, **kwds):
    def forms(self):
    def select_form(self, name=None, predicate=None, nr=None):
    def submit(self, *args, **kwds):
    def title(self):
    def geturl(self):
    def viewing_html(self):
    def encoding(self):
    def set_cookie(self, cookie_string):
    def response(self):
    def visit_response(self, response, request=None):
    def clear_history(self):
    def open_local_file(self, filename):
Describing history

By contrast, the class History (used to keep track of where you’ve been) is not an extension, but something entirely new.

It is, however, made up of simpler built-in data types, in this case a list (note the assignment of the variable history in the __init__ constructor).

While the method to move back is a bit hard to read out of context, what is going on with the constructor, the add method and the clear method?

class History:
    def __init__(self):
        self._history = []
    def add(self, request, response):
        self._history.append((request, response))
    def clear(self):
        del self._history[:]
    def back(self, n, _response):
        response = _response
        while n > 0 or response is None:
            try:
                request, response = self._history.pop()
            except IndexError:
                raise BrowserStateError()
            n -= 1
        return request, response
Under the hood

The code (or its complete version), when executed, will create an (as you might expect) object, this time of type `classobj` (just like `def` creates objects of type `function`) -- `classobj` objects let you do two things: **Reference attributes** (selecting data and functions) and **instantiate instances** of the class.

The attributes (the data and functions) are created when the class definition is executed (say, by importing the Mechanize module) and live in the class's namespace.

The instances you create are again objects, this time of class `instance` -- and they support only one operation -- attribute reference (specifying data and methods) using the `obj.name` syntax.
# import the browser and the html parser...

```python
>>> from mechanize import Browser
```

# first the type of the class

```python
>>> type(Browser)
<type 'classobj'>
```

```python
>>> dir(Browser)
['BLOCK_SIZE', '__doc__', '__getattr__', '__init__', '__module__', '__str__', '_add_referer_header',
'_call_chain', '_filter_links', '_maybe_reindex_handlers', '_mech_open', '_open', '_replace_handler',
'_request', '_set_handler', '_set_response', '_visit_request', 'add_client_certificate', 'add_client_certificate',
'add_password', 'add_proxy_password', 'back', 'clear_history', 'click', 'click_link', 'close',
default_features', 'default_others', 'default_schemes', 'encoding', 'error', 'find_link', 'force'
'forms', 'geturl', 'global_form', 'handler_classes', 'links', 'open', 'open_local_file', 'open_redis',
'reload', 'response', 'retrieve', 'select_form', 'set_client_cert_manager', 'set_cookie', 'set_cookie',
'set_handle_redirect', 'set_handle_referer', 'set_handle_refresh', 'set_handle_robots',
'set_handled_schemes', 'set_password_manager', 'set_proxies', 'set_proxy_password_manager', 'submit',
'title', 'viewing_html', 'visit_response']
```

# instantiating a Browser

```python
>>> br = Browser()
```

# and noting it is an object of type “instance”

```python
>>> type(br)
<type 'instance'>
```

```python
>>> isinstance(br,Browser)
True
```
Scoping: Class attributes v. instance attributes

An attribute defined in the class (textually in the class definition or later by assignment to the class’s namespace) is a **class attribute** -- An attribute defined in an instance by assignment is known as an an **instance attribute**

When you refer to an attribute in `br.name` notation (looking up a name in an instance object), Python’s scoping rules kick in -- First we look in the instance itself and see if there are any variables that have been set with that name, and if not it looks to the class namespace

There’s more to it than this, and you should consult Lutz for the finer points on classes, scoping and inheritance
>>> from mechanize import Browser

>>> Browser.title            # an unbound method (hasn't been bound to an instance)
<unbound method Browser.title>

# instantiating a Browser

>>> br = Browser()

# when we access data in the form br.name, Python looks in the instance br first to
# see if there is a variable of that name there -- if not, it looks to the class
# namespace.

# now the method has been “bound” and the translation using “self” is used automatically

>>> br.title
<bound method Browser.title of <mechanize._mechanize.Browser instance at 0x796d28>>
# you can overwrite the attribute by creating a local version in the instance’s
# namespace... this is dangerous!

```python
>>> br.title = "yikes!"
>>> br.title
'yikes!'

>>> Browser.title
<unbound method Browser.title>
```

# typically, we should avoid adding attributes to instances unless we know what
# we’re doing -- the whole point of encapsulation is to protect us from the details...

# to further complicate things, we can make changes to the attributes in the
# class Browser which could influence all its instances since they look to Browser
# to help resolve names
Predefined class attributes

There are five predefined class attributes

__dict__ The class’ namespace represented as a dictionary

__name__ The name of the class

__bases__ The classes from which the class inherits

__doc__ A documentation string (top of the class definition)

__module__ The name of the module where the class was defined

Instances have two default attributes

__dict__ The instance’s namespace

__class__ The name of the associated class
>> Browser.__dict__
{
'__module__': 'mechanize._mechanize',
'links': <function links at 0x1692ed8>,
'__str__': <function __str__ at 0x1692938>,
'set_cookie': <function set_cookie at 0x1692e60>,
'clear_history': <function clear_history at 0x1692de8>,
'back': <function back at 0x1692d70>,
'_filter_links': <function _filter_links at 0x169b578>,
'global_form': <function global_form at 0x169b050>,
'__getattr__': <function __getattr__ at 0x169b500>,
'visit_response': <function visit_response at 0x1692b90>,
'close': <function close at 0x1692668>,
'_add_referer_header': <function _add_referer_header at 0x1692758>,
'viewing_html': <function viewing_html at 0x169b0c8>,
'open': <function open at 0x1692848>,
'click': <function click at 0x169b2a8>,
'__init__': <function __init__ at 0x16925f0>,
'_set_response': <function _set_response at 0x1692b18>,
'title': <function title at 0x169b1b8>,
'encoding': <function encoding at 0x169b140>,
'submit': <function submit at 0x169b320>,
'forms': <function forms at 0x1692f50>,
'open_novisit': <function open_novisit at 0x16927d0>,
'__doc__': 'Browser-like class with support for history, forms and links.

... "

[output truncated]
A short introduction

This was an admittedly short introduction to the subject of object-oriented programming in Python -- You have enough now to read code and perhaps we will have the opportunity to write code!

Let’s now have another example, making a class of our own...
Aside: A command line tool

Twill is a command line tool that is a wrapper around Mechanize (and remember that Mechanize is, in turn a wrapper around urllib2) -- I present this because it’s worth seeing how small tools emerge and find use
twill: a simple scripting language for Web browsing

twill is a simple language that allows users to browse the Web from a command-line interface. With twill, you can navigate through Web sites that use forms, cookies, and most standard Web features.

twill supports automated Web testing and has a simple Python interface. Check out the examples!

twill is open source and written in Python.

Downloading twill

The latest release of twill is twill 0.9, released Thursday, December 27th, 2007; it is available for download at http://dares.idyll.org/~t/projects/twill-0.9.tar.gz. You can also use Python's easy_install to install or upgrade twill.

twill works with Python 2.3 or later.

To start using twill, install it and then type twill-sh. At the prompt type:

```
go http://www.slashdot.org/
show
showforms
showhistory
```

Documentation

The documentation for the latest release is always at http://twill.idyll.org/.

The in-development version of the docs can be found at http://dares.idyll.org/~t/projects/twill/doc/.

Documentation is available for the following topics:

- Examples -- some short examples.
- Web browsing with twill -- General introduction to twill.
- Language reference -- the twill scripting language.
- Testing Web sites with twill -- how to use twill to test Web sites.
- Extension modules -- extension modules that come with twill.
- Python API -- for Python programmers interested in using twill from Python.
- Developer information -- for Python developers interested in extending or fixing twill.
- Other tools and packages -- projects relevant to, or based upon, twill.
- Known problems -- those problems that are still lurking in the code.
cocteau@homework:~$ twill-sh

-- Welcome to twill! --

current page: *empty page*
>> go http://www.nytimes.com
==> at http://www.nytimes.com

current page: http://www.nytimes.com
>> showforms

Form #1
  ## ## __Name__________________ __Type___ __ID________ __Value__________________
1  video_permalink          text      video_pe ...  
2  embed_code               text      embed_code

Form name=searchForm (#2)
  ## ## __Name__________________ __Type___ __ID________ __Value__________________
1  query                    text      hpSearch ...  
2  type                     hidden    (None)       nyt  
3  1 None                     image     searchSubmit

[output truncated]

current page: http://www.nytimes.com
>> showhistory

History: (0 pages total)
current page: http://www.nytimes.com
>>
Another attempt

Last time we ended with your next homework assignment (Yes, we’re stretching you like crazy this year!) -- What the treatment last time was lacking was some basic motivation for the task beyond its pedagogical goals

So let’s start with the task of searching for media -- We are all accustomed to performing searches on text, but what about other forms of media, say, songs? What kinds of search engines exist?
Finding music

Recommender systems of various kinds leverage either communities of people and the music they like (either direct social networking or “inferred” from your previous “tagging” history) and some systems make use of various digital representations of a song, building a feature set that seems to capture various aspects of a song’s “style”

We’re being overly broad here, but in some cases this feature space is used because songs “cluster” based on common stylistic qualities, while in other cases, the space might allow one to uniquely identify a song (as in the case of Shazam)
The Music Genome Project®

On January 6, 2000 a group of musicians and music-loving technologists came together with the idea of creating the most comprehensive analysis of music ever.

Together we set out to capture the essence of music at the most fundamental level. We ended up assembling literally hundreds of musical attributes or "genes" into a very large Music Genome. Taken together these genes capture the unique and magical musical identity of a song - everything from melody, harmony and rhythm, to instrumentation, orchestration, arrangement, lyrics, and of course the rich world of singing and vocal harmony. It's not about what a band looks like, or what genre they supposedly belong to, or about who buys their records - it's about what each individual song sounds like.

Since we started back in 2000, we've carefully listened to the songs of tens of thousands of different artists - ranging from popular to obscure - and analyzed the musical qualities of each song one attribute at a time. This work continues each and every day as we endeavor to include all the great new stuff coming out of studios, clubs and garages around the world.

It has been quite an adventure, you could say a little crazy - but now that we've created this extraordinary collection of music analysis, we think we can help be your guide as you explore your favorite parts of the music universe.

We hope you enjoy the journey.

Tim Westergren
Founder
The Music Genome Project
Stylometrics

There is a very rich history of feature extraction of this kind for text analysis -- From early work on “bags of words” to more “sophisticated” attempts that examine parts of speech used, the complexity of sentences

Applications include authorship attribution and a host of “predictive” questions that attempt to infer everything about an author, from their gender to their Meyers-Briggs Type Indicator (oy)

She explained to me that a suitably programmed computer can read a novel in a few minutes and record the list of all the words contained in the text, in order of frequency. "That way I can have an already completed reading at hand," Lotaria says, "with an incalculable saving of time. What is the reading of a text, in fact, except the recording of certain thematic recurrences, certain insistences of forms and meanings?

"Words that appear eighteen times: boys, cap, come, dead, eat, enough, evening, French, go, handsome, new, passes, period, potatoes, those, until…

"Don’t you already have a clear idea what it’s about?" Lotario says. "There’s no question: it’s a war novel, all action, brisk writing, with a certain underlying violence.

From Calvino's novel "If on a winter's night a traveller"
Broadly...

The goal is to extract features that we can compute with, be it for search or recommendation -- Features that somehow capture something of the style of a piece of writing or sample of recorded music.

In general, this is a hard problem and last time we saw one kind of transformation that’s proved useful for “fingerprinting” a song -- Today we’ll take a step back and look at something easier.
Musipedia

The Open Music Encyclopedia

Welcome to Musipedia! Inspired by, but not affiliated with Wikipedia, we are building a searchable, editable, and expandable collection of tunes, melodies, and musical themes.

Every entry can be edited by anybody. An entry can contain a bit of sheet music, a MIDI file, textual information about the work and the composer, and last but not least the Parsons Code, a rough description of the melodic contour.

Name that Tune

Musipedia uses the "Melodyhound" melody search engine. You can find and identify a tune even if the melody is all you know. You can play it on a piano keyboard, whistle it to the computer, simply tap the rhythm on the computer keyboard or use the Parsons code.

Music Search

You can base your search on melody (i.e., pitch and rhythm), melodic contour, or just rhythm.

Melody Search

Identify a melody by entering it
• with a Flash-based piano,
• a JavaScript-based piano,
• by dragging the mouse,
• or whistle it to the computer using a microphone.

Contour-based Search

If you are unsure about the exact intervals between notes, try the Melodic Contour Search. This way, all you need to know is whether the tune goes up, down, or if the pitch stays the same.

Rhythm-based Search

For an even simpler (but less specific) way of searching, try the Rhythm-based "Query by Tapping" search method.

SOAP Interface

If you are a computer, or want to integrate the Musipedia search into your own web service, you can use the SOAP interface, which makes it possible to search based on melody, contour, or rhythm.
Parsons’ code

Denys Parsons, a scientist at the British Library, created a simple code for describing a melody -- He examined the pattern formed by the sequence of steps between each note in turn and referred to his invention as a “pitch profile”

The code itself consists of 3 symbols -- Each pair of consecutive notes is coded as “U”, or up, if the second note is higher than the first; “R”, or repeat, if the second note is the same and “D”, or down, if the second note is lower than the first -- An * is used to indicate the first note

In 1975, he published his Directory of Tunes and Musical Themes, a collection of more than 10,000 classical themes and nearly 4,000 popular tunes, all grouped so those melodies with common initial profiles could be scanned through easily (search)
Computers unveil the shape of melody

Computers can make a fair attempt at composing melodies—if their programs are good enough.

Efforts to produce better music from computers may tell us a lot about how people compose melodies.

Robin Maconie       Can computers compose melodies?
Chris Cunningham    Not the random streams of sounds
that any half-tutored programmer can summon from a machine, but coherent music that bears at least some resemblance to the efforts of humans? Even if they can, why bother? The answer is that making music is one of the most advanced of human thought processes. Because a computer needs human intelligence in the form of programs before it can even blink, programs that generate melody could tell us a great deal more than we now know about those thought processes.

Computers and composers are hardly strangers to each other. In recent years, powerful minicomputers that can manipulate sounds at high speed have enabled many composers to work with unfamiliar sounds and to write works that are not restricted to the range of conventional musical instruments. At the main centres of computer music, Stanford University in California, and the Institut pour la Recherche de Coordination Acoustique-Musique in Paris (run by Pierre Boulez), composers have expanded the range of sounds available to them in their works, and have written complex musical scores that rely equally on their cerebral skills and the processing power of their computers. They are not simply doing an academic exercise; their pioneering work would eventually enable anyone to compose and to perform music, even those who are, not unreasonably, unwilling to spend years learning the conventional language and skills of music.

Others are interested in computer music not for its own sake as a means of expression, but because music almost certainly can tell us much about the workings of the human brain. Marvin Minsky of the Massachusetts Institute of Technology, who is one of the so-called “fathers of artificial intelligence” bases much of his research on the logic behind computer music. He reasons that the different steps of composing in the sequence of logic—the algorithm—that forms the bones of a computer program, can reveal frequency (or “pitch”) between notes and is never a loose succession of notes. Armed with a few notes or a longer melodic phrase, a composer can set about constructing the orderly succession of notes that melody comprises. A satisfying melody never loses its main point of reference—the beginning—but neither should it be chained too strongly to that starting point, lacking the leaps and runs that an audience has to listen for rather than predict. That already says quite a lot both about how people listen to music, and how a composer (human or computer) sets about the process of creating melodies.

To instil some sort of order into melody, a composer needs rules that describe as concisely as possible the organisation of notes in music. In all cultures, rules have evolved through common consensus and practice. For instance, the structure of most melodies that belong to the culture of Western Europe rely on rules that became widely accepted only 500 years ago. Before the 17th century, there were as many as 20 different ways of defining the range of notes that appear in a melody.

We can set limits for that range of notes—a base, or “tonic” note and a note of exactly twice the tonic note’s frequency—and call that range a scale. The difference—an interval of one octave—between the two notes can then be divided up into smaller intervals. In theory there is an infinity of ways to divide up a scale, but in practice a culture will settle on a fixed number of divisions. The piano keyboard for instance divides an octave into simple intervals called semitones. Each octave comprises 12 different notes, each of which has an alphabetic name—C, G, D sharp, B flat, and so on—and each of which can be the tonic note of a scale.

The scales from which Johann Sebastian Bach wrote his St Matthew Passion and from which Burt Bacharach wrote Raindrops Keep Falling on my Head are based on seven notes. The scales are still an octave in length, but the division of each scale is not so straightforward as in a simple,
Aside

Parsons found that in a sample of more than 7,000 classical themes, the most common initial profile was “U-U” -- That is, the “preferred pitch profile is a rising interval from the first to the second note and from the second to the third”

The next most popular is “U-D” followed by “D-U” and “D-D” -- The least popular was “D-R”
So let’s build something

Let’s start with a MIDI description of music -- MIDI stands for Musical Instrument Digital Interface and it is binary format for communicating between “electronic musical instruments” and other devices

Among its different messages, it allows one to specify the pitch, volume and timing of notes that are represented as a number between 0-127 -- One the next page we have an example of one such scaling

An excellent reference on MIDI can be found at the following URL (although we really need just the most terse treatment to motivate what comes next)

http://museinfo.sapp.org/doc/formats/midi/
class midi_song:

    def __init__(self):
        self.notes = []

    def __str__(self):
        if self.notes:
            return "our song: " + \".join([str(n) for n in self.notes])\n        else:
            return "empty song -- enter some notes!"

    def add_note(self,n):
        if n in range(128): self.notes.append(n)

    def parsons(self,start=0):

        p = "*"

        for i in range(start+1,len(self.notes)):
            if self.notes[i-1]<self.notes[i]: p += "U"
            elif self.notes[i-1]==self.notes[i]: p += "R"
            else: p += "D"

        return p
```python
>>> from random import sample
>>> from midi_test import *

>>> song = midi_song()  # make an empty song
>>> print song  # and have a look
empty song -- enter some notes!

>>> song.notes
[]

>>> song.add_note(100)  # add a note, and then a bunch of random notes
>>> for i in range(20): song.add_note(sample(range(128),1)[0])
...

>>> print song
our song: 100 57 124 121 80 109 48 41 120 105 10 72 115 15 119 74 116 2 31 74 18

>>> song.parsons()
'*DUDDUDDUDUUDUDUDUUDUUDUUD'```
Searched Musipedia for:
Melodic contour (Parsons code): DUDDUDDUUDDUDDUDDUUD,
which can be visualized as follows:

Search Category: Popular

[0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90]

[P] *Blue Oyster Cult*: Burnin' For You
From here

Obviously, from here we might create new objects that represent notes (because notes have more than just pitch associated with them) and then methods to write out the binary format for the MIDI file.

The point was to simply show how easily our own objects can be created and quickly start to multiply -- We also wanted to put the whole task of searching for multimedia files into some kind of context.

Shazam’s algorithm may seem a little complex given the number of different terms you are learning all at once, but morally your work would be the same if we were using Parsons’ code.
An Industrial-Strength Audio Search Algorithm

Avery Li-Chun Wang
avery@shazamteam.com
Shazam Entertainment, Ltd.
USA:
2925 Ross Road
Palo Alto, CA 94303

United Kingdom:
375 Kensington High Street
4th Floor Block F
London W14 8Q

We have developed and commercially deployed a flexible audio search engine. The algorithm is noise and distortion resistant, computationally efficient, and massively scalable, capable of quickly identifying a short segment of music captured through a cellphone microphone in the presence of foreground voices and other dominant noise, and through voice codec compression, out of a database of over a million tracks. The algorithm uses a combinatorially hashed time-frequency constellation analysis of the audio, yielding unusual properties such as transparency, in which multiple tracks mixed together may each be identified. Furthermore, for applications such as radio monitoring, search times on the order of a few milliseconds per query are attained, even on a massive music database.

1 Introduction

Shazam Entertainment, Ltd. was started in 2000 with the idea of providing a service that could connect people to music by recognizing music in the environment by using their mobile phones to recognize the music directly. The algorithm had to be able to recognize a short audio sample of music that had been broadcast, mixed with heavy ambient noise, subject to reverber and other processing, captured by a little cellphone microphone, subjected to voice codec compression, and network dropouts, all before arriving at our servers. The algorithm also had to perform the recognition quickly over a large database of music with nearly 2M tracks, and furthermore have a low number of false positives while having a high recognition rate.

This was a hard problem, and at the time there were no algorithms known to us that could satisfy all these constraints. We eventually developed our own technique that met all the operational constraints [1].

We have deployed the algorithm to scale in our commercial music recognition service, with over 1.8M tracks in the database. The service is currently live in Germany, Finland, and the UK, with over a half million users, and will soon be available in additional countries in Europe, Asia, and the USA. The user experience is as follows: A user hears music playing in the environment. She calls up our service using her mobile phone and samples up to 15 seconds of audio. An identification is performed on the sample at our server, then the track title and artist are sent back to the user via SMS text messaging. The information is also made available on a web site, where the user may register and log in with her mobile phone number and password. At the web site, or on a smart phone, the user may view her tagged track list and buy the CD. The user may also download the ringtone corresponding to the tagged track, if it is available. The user may also send a 30-second clip of the song to a friend. Other services, such as purchasing an MP3 download may become available soon.

A variety of similar consumer services has sprung up recently. Musiwave has deployed a similar mobile-phone music identification service on the Spanish mobile carrier Amena using Philips’ robust hashing algorithm [2-4]. Using the algorithm from Relatable, Neuros has included a sampling feature on their MP3 player which allows a user to collect a 30-second sample from the built-in radio, then later plug into an online server to identify the music [5,6]. Audible Magic uses the Muscle Fish algorithm to offer the Clango service for identifying audio streaming from an internet radio station [7-9].

The Shazam algorithm can be used in many applications besides just music recognition over a mobile phone. Due to the ability to dig deep into noise we can identify music hidden behind a loud voiceover, such as in a radio advert. On the other hand, the algorithm is also very fast and can be used for copyright monitoring at a search speed of over 1000 times real-time, thus enabling a modest server to monitor significantly many media streams. The algorithm is also suitable for content-based cueing and indexing for library and archival uses.

2 Basic principle of operation

Each audio file is “fingerprinted,” a process in which reproducible hash tokens are extracted. Both “database” and “sample” audio files are subjected to the same analysis. The fingerprints from the unknown sample are matched against a large set of fingerprints derived from the music database. The candidate matches are subsequently evaluated for correctness of match. Some guiding principles for the attributes to use as fingerprints are that they should be temporally localized, translation-invariant, robust, and sufficiently entropic. The temporal locality
The algorithm behind Shazam is fairly straightforward -- A time-frequency decomposition is performed examining which frequencies are dominant at which times in a song.

The peaks in this map form a kind of constellation -- Relationships between the individual elements are then encoded using something called geometric hashing (don’t worry about this yet).

Given a sample of audio, the same process is repeated and a search is made to see if there are matching patterns of peaks...
Scatterplot of matching hash locations: No diagonal

Fig. 2A

Histogram of differences of time offsets: signals do not match

Fig. 2B
Scatterplot of matching hash locations: Diagonal Present

Sample soundfile time

Database soundfile time

Histogram of differences of time offsets: signals match

Offset $t_{database} - t_{sample}$