A whole new **class** of programming

IST 338 *today*!

CS building blocks: functions and composition

behind the CS curtain: *circuits*, *assembly*, *loops*

**Designing Data!**

CS: *theory + practice*
Foziah: website using Django (or Flask)

Final projects

Not an end, but a beginning

IST 338 Final Projects Page

Welcome!

For the final project in IST 338, you have a choice of

- one of our pre-scaffolded projects, all available here, or
- designing your own project

Either way, here is a calendar of dates relating to the project:

- **due 4/17**: choice of project - or description of a custom project (one page)
- **on 4/22 and 4/30**: there will be a shorter class (45min or less) to allow room to work on projects or ask questions about them
  - there will also be shorter assignments (small computational challenges) those weeks
- **on 5/6**: project presentations (5-10 minutes on your project, which need not be 100% finished)
- (no class on 5/13)
- **on 5/17**: final project is due

Mandelbrot Set!

also: www.cs.hmc.edu/~jgrasel/
Classes and Objects

An object-oriented programming language allows you to build your own customized types of variables.

(1) A class is a type

(2) An object is one such variable.

(instance)

There will typically be MANY objects of a single class.
Everything in Python is an object!

Its capabilities depend on its class.

what's more, you can build your own...
Everything is an object?

Take strings, for example:

```python
>>> s = str(42)  # This calls the str constructor.
>>> type(s)
<type 'str'>  # Shows the type of s is str
>>> dir(s)
# Shows all of the methods (functions) of s
```

Let's try some!
Objects

Like a list, an object is a container, but much more customizable:

1. Its data elements have *names chosen by the programmer.*

2. An object contains its own functions, called *methods.*

3. In its methods, objects refer to themselves as *self.*

4. Python signals special methods with two underscores:

   - `__init__` is called the *constructor*; it creates new objects
   - `__repr__` tells Python how to print its objects

*I guess we should doubly underscore these two methods!*
A `Date` object, \( d \)

\[
\begin{array}{ccc}
8 & 4 & 2015 \\
day & month & year \\
\end{array}
\]

memory location \( \sim 42042778 \)
A **Date** object, \( d \)

- **day**: 8
- **month**: 4
- **year**: 2015

**memory location** ~ 42042778

**It's an alien date!**
class Date:
    """ a blueprint (class) for objects that represent calendar days """

    def __init__(self, mo, dy, yr):
        """ the Date constructor """
        self.month = mo
        self.day = dy
        self.year = yr
class Date:
    """
    a blueprint (class) for objects that represent calendar days
    """

def __init__(self, mo, dy, yr):
    """
    the Date constructor
    """
    self.month = mo
    self.day = dy
    self.year = yr

The Date class

This is the start of a new type called Date
It begins with the keyword class

This is the constructor for Date objects
As is typical, it assigns input data to the data members.

These are data members – they are the information inside every Date object.
This is a class. It is a user-defined datatype that you'll build in Lab 10 this week...

```python
>>> d = Date(4, 8, 2015)
>>> d.month
4
>>> d.day
6

>>> d
04/08/2015

>>> d.isLeapYear()
False
```

The `isLeapYear` method returns True or False. How does it know what year to check?

The representation of an object of type Date is:

```
04/08/2015
```

Constructor!

The `repr`esentation of an object of type Date:

```
```

d contains data members named day, month, and year.
class Date:
    """ a blueprint (class) for objects that represent calendar days """
    def __init__(self, mo, dy, yr):
        """ the Date constructor """
        self.month = mo
        self.day = dy
        self.year = yr

    def __repr__(self):
        """ used for printing Dates """
        s = "%02d/%02d/%04d" % (self.month, self.day, self.year)
        return s

This is the repr for Date objects. It tells Python how to print these objects.

Why self instead of d?
**self** is the variable calling a method

```python
>>> d = Date(4, 8, 2015)
>>> d
04/08/2015
```

```python
>>> d.isLeapYear()
False
```

```python
>>> d2 = Date(1, 1, 2016)
```

```python
>>> d2
01/01/2016
```

```python
>>> d2.isLeapYear()
True
```

These methods need access to the object that calls them: it's **self**
class Date:
    """ a blueprint (class) for objects that represent calendar days """

def __init__( self, mo, dy, yr ):
    """ the Date constructor """
    self.month = mo
    self.day = dy
    self.year = yr

def __repr__( self ):
    """ used for printing Dates """
    s = "%02d/%02d/%04d" % (self.month, self.day, self.year)
    return s

def isLeapYear( self ):
    """ anyone know the rule? """
2.2.1 What years are leap years?

The Gregorian calendar has 97 leap years every 400 years:

Every year divisible by 4 is a leap year.
However, every year divisible by 100 is not a leap year.
However, every year divisible by 400 is a leap year after all.

So, 1700, 1800, 1900, 2100, and 2200 are not leap years. But 1600, 2000, and 2400 are leap years.

```python
class Date:
    def __init__(self, mo, dy, yr):
        (constructor)

    def __repr__(self):
        (for printing)

    def isLeapYear(self):
        """ here it is ""
        if self.year%400 == 0:
            return True
        if self.year%100 == 0:
            return False
        if self.year%4 == 0:
            return True
        return False
```
There are 12 sequential dates in this century. The next big sequential date is 12/13/14.

"That's on a Saturday so we're hoping to see even bigger numbers," Mills said.

If you put any stock in the idea that sequential dates bode well for a long and happy marriage, though, you better start looking for that special someone — your next opportunity for a wedding on such a date won't happen until 2103.
10/10/10: They Love Just Thinking About It

By JOHN SCHWARTZ  OCT. 8, 2010

Sunday is the big day for saying “I do.”

More than 39,000 couples chose 10/10/10 as their wedding day — a nearly tenfold increase over the number of nuptials on Oct. 11, 2009, the comparable Sunday last year, according to figures gathered by David’s Bridal, the wedding superstore chain.

The reason for the surge is a blend of superstition and symbolism, said Maria McBride, the wedding style director at Brides Magazine. “You cross your fingers and hope it lasts a lifetime,” she said, and so “a perfect 10, times 3” suggests good luck.

Besides, Ms. McBride said, “You’ll never forget your anniversary.”

At the Viva Las Vegas Wedding Chapel, the owner, Ron DeCar, has 150 ceremonies planned in five chapels, beginning at midnight. He has had to hire extra Elvis impersonators, he said, to bring the contingent to six.
For those of a geeky bent, the date has another layer of importanc — it is made up entirely of ones and zeros, the binary language of computing. Kevin Cheng and Coley Wopperer of San Francisco have been waiting nearly two years for their wedding date to roll around, having realized over dinner with friends in 2008 that, as one suggested, “you could have a binary-themed wedding!” he recalled.

“Both of our eyes just lit up,” he said.

“We’re very much technology people,” Mr. Cheng explained, as if it were necessary to point this out.

The dinner group quickly calculated the more familiar base-10 value of the binary number 101010, and found that it was 42. “That totally sealed the deal!” he recalled.

Footnote: For fans of Douglas Adams, author of the series of science fiction comedic novels beginning with “The Hitchhiker’s Guide to the Galaxy,” the number 42 is instantly recognized as the punch line to one of literature’s most revered shaggy dog stories. In it, super-intelligent beings have created the most powerful computer ever to provide the “Answer to the Ultimate Question of Life, The Universe, and Everything.” The computer labors for 7.5 million years. Finally, it answers: “Forty-two.”

Which means, gentle reader, that Kevin Cheng and Coley Wopperer are truly meant for each other.
I think CS is really fun,
so many programs have been run.
Picobot made me a little loopy,
but extra credit has made me Dodd's groupie.
Some problems made me stop and say 'hmmmm,
but none more than Logism.
Now I guess this test is done,
Time to go back to line one!

OMG mamy me!
(or Be a gnubr!)

Emily M
== vs. equals

>>> d = Date(11,12,2013)
>>> d
11/12/2013

>>> d2 = Date(11,12,2013)
>>> d2
11/12/2013

>>> d == d2
False

Python objects are handled by reference...
== compares references!

How can this be False?
Two `Date` objects:

```
memory location ~ 42042778
```

```
memory location ~ 42042742
```

`==` compares *memory locations*, not contents.
== vs. **equals**

```python
>>> d = Date(11,12,2013)
>>> d
11/12/2013

>>> d2 = Date(11,12,2013)
>>> d2
11/12/2013

>>> d.equals(d2)
True
```

This constructs a different Date

Python objects are handled by reference... `.equals` compares contents
class Date:

    def __init__(self, mo, dy, yr):
    def __repr__(self):
    def isLeapYear(self):

        def equals(self, d2):
            """ returns True if they represent
            the same date; False otherwise
            """
            if

                return True
            else:
                return False
class Date:

def isBefore(self, d2):
    """ if self is before d2, this should return True; else False """
    if self.year < d2.year: return True
    if self.year > d2.year: return False

    # here, the years are EQUAL!
    if self.month < d2.month: return True
    if self.month > d2.month: return False

    # here, the years and months are EQUAL!
    if self.day < d2.day: return True
    return False

>>> d = Date(1,1,2016)
>>> d2 = Date(4,6,2015)
>>> d.isBefore( d2 )
False
class Date:

def isBefore(self, d2):
    """ if self is before d2, this should return True; else False """
    if [self.year, self.month, self.day] < [d2.year, d2.month, d2.day]:
        return True
    else:
        return False
Date's purpose...?!?

```python
>>> d = Date(5,14,2015)
>>> d
05/14/2015

>>> d.tomorrow()
```

the `tomorrow` method returns nothing at all. Is it doing anything?

```python
>>> d
05/15/2015
```

d has changed!

```python
>>> d.subNDays(37)
```

lots of printing, but no return value!

Why is this important?

Some methods return a value; others change the object that call it!
Lab today – or tomorrow

Add these to your `Date` class!

- `yesterday(self)`
- `tomorrow(self)`
- `addNDays(self, N)`
- `subNDays(self, N)`
- `isBefore(self, d2)`
- `isAfter(self, d2)`
- `diff(self, d2)`
- `dow(self)`

and use your `Date` class to analyze our calendar a bit...

Prof. Benjamin!
no computer required...
class Date:

def tomorrow(self):
    """ moves the date ahead 1 day """

DIM = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

self.day _________

if self.day > ____________:

    first, add 1 to self.day

then, adjust the month and year, but only if needed

Quiz

Don't return anything. This CHANGES the date object that calls it.

NAME(s) _____________________________

Implement tomorrow!

Extra: how could you make this work for leap years, too?
class Date:

    def tomorrow(self):
        """ moves the date ahead 1 day """

        self.day += 1

        if self.day > DIM[self.month]:
            self.day = 1
            self.month += 1

        DIM = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

    Name(s) _____________________________

Quiz

Don't return anything. This CHANGES the date object that calls it.

DIM looks pretty bright to me!

Implement tomorrow!

Extra: how could you make this work for leap years, too?
class Date:

def tomorrow(self):
    """ moves the date ahead 1 day """
    DIM = [0, 31, fdays, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    self.day += 1  # add 1 to the day!
    if self.day > DIM[self.month]:  # check day
        self.month += 1
        self.day = 1
    if self.month > 12:  # check month
        self.year += 1
        self.month = 1
class Date:

def tomorrow(self):
    """ moves the date ahead 1 day """
    if self.isLeapYear() == True:  
        fdays = 29
    else:  
        fdays = 28

    DIM = [0, 31, fdays, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

    self.day += 1  # add 1 to the day!

    if self.day > DIM[self.month]:  # check day
        self.month += 1
        self.day = 1

    if self.month > 12:  # check month
        self.year += 1
        self.month = 1
class Date:

def tomorrow(self):
    """ moves the date ahead 1 day """

    fdays = 28 + self.isLeapYear()  # What ?!  
    DIM = [0, 31, fdays, 31, 30, 31, 30, 31, 30, 31, 30, 31, 31]

    self.day += 1  # add 1 to the day!

    if self.day > DIM[self.month]:  # check day
        self.month += 1
        self.day = 1

    if self.month > 12:  # check month
        self.year += 1
        self.month = 1
Not all years are the same!
Using **classes + objects**

Building classes...

**hw10pr2**

Connect Four **Board** class

... vs. using the library

**hw10pr3**

file and dictionary classes

files
dictionaries

If I had a dictionary, I guess I could look up what it was!
Classes: DIY data

**Class:** a user-defined datatype

**Object:** data or a variable whose type is a class

```
d = Date( 4, 8, 2015 )
d.tomorrow()
print d
```

**Method:** a function defined *in a class* called *by an object*

**self:** in a class, the name of the object calling a method

**Constructor:** the `__init__` function for creating a new object

**repr:** the `__repr__` function returning a string to print

**data member:** the data in `self`: `self.day`, `self.month`, `self.year`
Why classes?

Python has no Connect-four datatype...

Care for a game?

... but now we can fix that!
Data design...

(Data Members) What data do we need?

(Methods) What are capabilities we want?
Our Board object, \( b \)

How could we set \( ? \) and \( ? \) to 'X'?
__init__

```python
class Board:
    """ a datatype representing a C4 board
    with an arbitrary number of rows and cols
    """

    def __init__( self, width, height ):
        """ the constructor for objects of type Board """
        self.width = width
        self.height = height
        W = self.width
        H = self.height
        self.data = [ [ ' ']*W for row in range(H) ]
```

This list comprehension lets us create

\[ H \] independent rows with

\[ W \] independent columns each.
```python
def __repr__(self):
    """ this method returns a string representation for an object of type Board """
    H = self.height
    W = self.width
    s = ''
    for r in range(H):
        s += '|' + 
        for c in range(W):
            s += self.data[r][c] + '|' + 
        s += '

    s += (2*W+1)*'-' + 

    # what will you need to add right here?
    return s
```
Quiz

```python
class Board:

    def addMove(self, col, ox):
        """ buggy version! """
        H = self.height
        for row in range(0, H):
            if self.data[row][col] != ' ':
                self.data[row-1][col] = ox
        self.data[H][col] = ox
```

(1) Run `b.addMove(3, 'O')`

(2) **Bugs!** Can you fix them?!

Name(s) ____________________________
### Quiz

**class Board:**

```python
def addMove(self, col, ox):
    """ buggy version! """
    H = self.height
    for row in range(0, H):
        if self.data[row][col] != ' ':
            self.data[row-1][col] = ox
    self.data[H][col] = ox
```

1. Run `b.addMove(3, 'O')`

2. **Bugs!** Can you fix them?!

Try this on the back page first...
class Board:

def allowsMove(self, col):

    # True if col is in-bounds + open
    # False otherwise ""
    H = self.height
    W = self.width
    D = self.data

    if

    If it's in-bounds and not full, return True.

    If col is out-of-bounds or full, return False.
hw10pr2: **Board** class

- **The “constructor”**  
  `__init__( self, width, height )`
- Checks if allowed  
  `allowsMove( self, col )`
- Places a checker  
  `addMove( self, col, ox )`
- Removes a checker  
  `delMove( self, col )`
- Outputs a string  
  `__repr__( self )`
- Checks if any space is left  
  `isFull( self )`
- Checks if a player has won  
  `winsFor( self, ox )`
- The game...  
  `hostGame( self )`

Which are similar to others? Which requires the most thought?
def winsFor(self, ox):
    """ does ox win? ""
    H = self.height
    W = self.width
    D = self.data

    for row in range(
        for col in range(

b.winsFor('X')
or 'O')

Watch out for
corner cases!
Why objects and classes?

**Elegance:** Objects hide complexity!

```python
if b.winsFor( 'X' ) == True:
    if d.isBefore( d2 ) == True:
```

Simple – and INVITING -- building blocks!
CS 5 this week

Building classes...

hw10pr2

Connect Four Board class

... vs. using the library

hw10pr3

file and dictionary classes

files
dictionaries

If I had a dictionary, I guess I could look up what it was!

Hw #10 due 4/12
In Python reading files is no problem...

```python
>>> f = open('a.txt')
opens the file and calls it f

>>> text = f.read()
reads the whole file into the string text

>>> f.close()
closes the file (optional)

>>> text
'I like poptarts and 42 and spam.\nWill I

>>> LoW = text.split()
returns a list of each "word"

[ 'I', 'like', 'poptarts', ... ]
```
def word_count(filename):
    """ word-counting program """
    f = open(filename)
    text = f.read()
    f.close()
    
    LoW = text.split()
    print "There are", len(LoW), "words."

What if we wanted the number of different words in the file?

This would be the author's **vocabulary size**, instead of the total word count.
Dictionaries

A dictionary is a set of key-value pairs.

>>> d = {}
creates an empty dictionary, d

>>> d[1996] = 'rat'

>>> d[1995] = 'pig'

>>> d


>>> d[1995]

'pig'

This seems like the key to dictionaries’ value...
More on dictionaries

Strings can be keys, too!

>>> d = {'pig': 1995, 'rat': 1996}

>>> 'pig' in d  
True

>>> 'cat' in d  
False

>>> len(d)  
2

>>> d.keys()  
['pig', 'rat']

>>> d.values()  
[1996, 1995]

>>> d.items()  
[('rat', 1996), ('pig', 1995)]
WOULD YOU LIKE THEM IN A HOUSE?
WOULD YOU LIKE THEN WITH A MOUSE?
I DO NOT LIKE THEM IN A HOUSE.
I DO NOT LIKE THEM WITH A MOUSE.
I DO NOT LIKE THEM HERE OR THERE.
I DO NOT LIKE THEM ANYWHERE.
I DO NOT LIKE GREEN EGGS AND HAM.
I DO NOT LIKE THEM, SAM-I-AM.
def vocab_count( filename ):
    """ vocabulary-counting program """
    f = open( filename )
    text = f.read()  # file handling
    f.close()

    LoW = text.split()
    print "There are", len(LoW), "words."

    d = {}  # Tracking the number of occurrences of each word with a dictionary, d.

    for wd in LoW:
        if wd not in d:
            d[wd] = 1
        else:
            d[wd] += 1

    print "There are", len(d), "distinct words."

    return d  # return d for later use by other code...
Vocabulary, anyone?

Shakespeare used *31,534 different words* -- and a grand total of 884,647 words -- counting repetitions (across all of his works)  

http://www.math.cudenver.edu/~wbriggs/qr/shakespeare.html

**Shakespearean coinages**

- gust
- besmirch
- unreal
- superscript
- watchdog
- swagger
- successful
- unsuccessful

*There's one contemporary author in the Oxford English Dictionary...*

- affined
- rooky
- attasked
- out-villianed

http://www.pathguy.com/shakeswo.htm
http://www.shakespeare-online.com/biography/wordsinvented.html
Vocabulary, anyone?

Shakespeare used **31,534 different words** -- and a grand total of 884,647 words -- counting repetitions (across all of his works)

Shakespearean coinages

- gust
- besmirch
- unreal
- superscript
- watchdog
- swagger
- affined
- rooky
- attasked
- out-villianed
- successful
- unsuccessful

"Muggle" goes into Oxford English Dictionary

JK Rowling's word for non-wizards - "muggle" - has made it into the new edition of the Oxford English Dictionary (OED).

The draft definition according to the dictionary's website says:

- **Muggle**: invented by JK (Joanne Kathleen) Rowling (b. 1965), British author of children's fantasy fiction (see quot. 1997).

  In the fiction of JK Rowling: a person who possesses no magical powers. Hence in allusive and extended uses: a person who lacks a particular skill or skills, or who is regarded as inferior in some way.

J. K. Rowling
Algorithmic authoring?

'Cause somethin' like he left knee and a harp," said he had to the whole school? The shouting and then some strange and Mrs. "Well, I know Hagrid; they spotted handkerchief and get him get rid of course, had a gigantic beet with her," he knew what to all he's

All the sky with the sun in the sun in the church where you're gone Lucy in my eyes. There beneath the girl with an hourglass And then the banker never wears a lot to hold your hand. Can't buy me tight, tight Owww! Love is love I can't hide,

Who is the author? What is the work? What is going on?

This is but ourselves. No, faith, My uncle! O royal bed of confession Of your rue for leave to nature; to this time I should weep for thy life is rotten before he is. have sworn 't. Or my blood. I have closely sent for nine; and unprofitable,

The Senators and the date of a written declaration that Purpose, they shall consist of nine States, shall not, when he shall have such Vacancies. The President pro tempore, in the Desire of a Qualification to the Speaker of the Senate. Article 6. When vacancies by the office upon probable
Markov Models

Techniques for modeling *any* sequence of natural data

*1st-order* Markov Model
*(defining property)*

Each item depends *only* on the *one* immediately before it.
Our Markov Model

Original file:

Markov Model:

is a dictionary!

What are the keys?

What are the values?

What are the missing values?

What is the '$'?

Why do some keys seem missing?

```json
{
  '$': ['I', 'Will', 'I'],
  'I': ['like', 'get', 'like'],
  'like': ['poptarts', 'spam'],
  'poptarts': ['and', 'for'],
  'and': ['42'],
  '42': ['and'],
  'Will': ['I'],
  'the': ['spam'],
  'spam': ['and', 'poptarts!'],
  'get': ['spam'],
  'for': ['the']
}
```
Our Markov Model

Original file:
I like poptarts and 42 and spam.
Will I get spam and poptarts for the holidays? I like spam poptarts!

Markov Model:

```json
{
    '$': ['I', 'Will', 'I'],
    'I': ['like', 'get', 'like'],
    'like': ['poptarts', 'spam'],
    'poptarts': ['and', 'for'],
    'and': ['42', 'spam.', 'poptarts'],
    '42': ['and'],
    'Will': ['I'],
    'the': ['holidays?'],
    'spam': ['and', 'poptarts!'],
    'get': ['spam'],
    'for': ['the']
}
```

A dictionary!
Model creation:

1) start with the `prevwd` as `'$'`
2) for each `nextwd` in the list of words, add it in ...
3) then change `nextwd` to `prevwd` or `'$'...` if `nextwd[-1]` is punctuation.

```python
d = {}

# Example text
I like poptarts and 42 and spam. Will I get spam and poptarts for the holidays? I like spam poptarts!

# Initial dictionary

d['$'] = ['I']
d['I'] =
d['like'] =
```

```python
# Update dictionary

d['I'] +=
```
Model creation in Python

\[ \text{pw} \]

\[ \text{nw} \quad \text{I like spam. I eat poptarts!} \]

d = {}

pw =

for nw in LoW:
    if pw not in d:
        d[pw] =
    else:
        d[pw] +=

# what variables need to change here? how do they?

\[ \text{goal for } d \]

\[ $: \ [ l, l ]$

\[ l : \ [ \text{like, eat} ]$

\[ \text{like} : \ [ \text{spam} ]$

\[ \text{eat} : \ [ \text{poptarts} ] \]
```python
def createDictionary(filename):
    """ creates a 1st-order M.Model """
    f = open(filename)
    text = f.read()
    f.close()
    LoW = text.split()

    d = {}
    prevwd = '$'

    for nextwd in LoW:
        if prevwd not in d:
            d[prevwd] = []
        else:
            d[prevwd] += [nextwd]

    return d
```

Model creation

see previous slide for the "key" idea!

We want the **KEY** to be `prevwd`. We want the **VALUE** to be the list of words *following* `prevwd`.

reset variables appropriately here – be sure not to forget to check if the sentence has ended!
Markov Models are *generative*!

A key benefit of Markov Models is that they can *generate* feasible data!

**Original file:**

I like poptarts and 42 and spam. Will I get spam and poptarts for the holidays? I like spam poptarts!

**Generated text:**

I get spam poptarts! I like poptarts and 42 and spam. I like spam and 42 and 42 and spam. Will I like poptarts and 42 and poptarts and 42 and poptarts and 42 and 42 and poptarts and spam. I get spam and 42 and 42 and...
Model creation:

1) start with the `prevwd` as '$'
2) for each `nextwd` in the list of words, add it in ...
3) then change `nextwd` to `prevwd` or '$'

---

Generating text:

1) start with `prevwd` as the '$' string
2) choose a `nextwd` that follows `prevwd`, at random.
3) print `nextwd`
4) `nextwd` or '$' becomes `prevwd`

---

`demo..."
2nd CFP - Systemics, Informatics and Cybernetics

Dear Zachary Dodds:

We invite you to submit a paper/abstract to The 15th World Multi-Conference on Systemics, Cybernetics and Informatics: WMSCI 2011, to be held in Orlando, Florida, USA, on July 19th - July 22nd, 2011 (www.2011iiisciorganisations.org/wmsci)

If you have any colleagues who might be interested in making a submission to the conference, please feel free to forward this e-mail to them.

Below are the next deadlines for WMSCI 2011 (Check the web site for possible extensions or new set of deadlines):

Papers/Abstracts Submission and Invited Session Proposals: November 25th, 2010
Authors Notifications: January 31st, 2011
Camera-ready, full papers: February 28th, 2011
WMSCI 2005

Rooter: A Methodology for the Typical Unification of Access Points and Redundancy

Jeremy Stribling, Daniel Aguayo and Maxwell Krohn

http://pdos.csail.mit.edu/scigen/

Markov-generated submission accepted to WMSCI 2005
theirs was more than a first-order model...
presentation... in costume!
Thesis worries?
Other papers due?

Let Python write the rest of your papers for you...
... and you're still the author!

Have a worry-free weekend!