A whole new class of programming

CS's building blocks: functions and composition

CS 5 overview

behind CS's curtain: circuits, assembly, loops

Designing Data!

CS: theory + practice

whose convenience?
Lists ~ 2D data

\[ A = \begin{bmatrix} [1,2,3,4], &[5,6], &[7,8,9,10,11] \end{bmatrix} \]

- \( A[0] \)
- \( A[1] \)

Where's 3?

- \( \text{len}(A) = 3 \)
- \( \text{len}(A[0]) = 4 \)
- \( \text{len}(A[1]) = 2 \)

Replace 10 with 42.
Rectangular 2D data

\[ A = [ [0,0,0,0], [0,0,0,0], [0,0,0,0] ] \]

\[ A[1][2] = 42 \]

\[ A[r][c] = \text{value} \]
Rectangular 2D data

\[ A = [ [0,0,0,0], [0,0,0,0], [0,0,0,0] ] \]

\[ A[1][2] = 42 \]

\[ A[r][c] = \text{value} \]
Rectangular 2D data

\[ A = \begin{bmatrix} 0,0,0,0,0 \end{bmatrix}, \begin{bmatrix} 0,0,0,0,0 \end{bmatrix}, \begin{bmatrix} 0,0,0,0,0 \end{bmatrix} \]

\[ \text{NROWS} = \text{len}(A) \quad \# \text{HEIGHT} \]
\[ \text{NCOLS} = \text{len}(A[0]) \quad \# \text{WIDTH} \]

```
for r in range(0, NROWS):
    for c in range(0, NCOLS):
        if r == c:
            A[r][c] = 4
        else:
            A[r][c] = 2
```
2 in-a-row?

def two_in_a_row(A):
    """ what's happening? """
    NROWS = len(A)
    NCOLS = len(A[0])
    B = deepcopy(A)

    for r in range(0,NROWS):
        for c in range(0,NCOLS):
            if c == NCOLS-1:
                B[r][c] = False
            elif A[r][c] == A[r][c+1]:
                B[r][c] = True
            else:
                B[r][c] = False

A = [ [4, 2, 2, 2], [2, 2, 4, 4], [2, 4, 4, 2] ]

Challenge:
How could we change the code above to check for two-in-a-row SOUTHWARD -- or DIAGONALLY!?
A = [[4, 2, 2, 2],
     [2, 2, 4, 4],
     [2, 4, 4, 2]]

\[ A = \begin{bmatrix}
4 & 2 & 2 & 2 \\
2 & 2 & 4 & 4 \\
2 & 4 & 4 & 2
\end{bmatrix} \]

B = deepcopy(A)

for r in range(0, NROWS):
    for c in range(0, NCOLS):
        if c == NCOLS - 1:
            B[r][c] = True
        elif A[r][c] == A[r][c + 1]:
            B[r][c] = True
        else:
            B[r][c] = False

How could we change the code above to check for two-in-a-row SOUTHWARD or DIAGONALLY?!
def two_in_a_row(A):
    """ what happens here? """
    NROWS = len(A)
    NCOLS = len(A[0])
    B = deepcopy(A)
    for r in range(0,NROWS):
        for c in range(0,NCOLS):
            if c == NCOLS-1:
                B[r][c] = False
            elif A[r][c] == A[r][c+1]:
                B[r][c] = True
            else:
                B[r][c] = False
    return B

Challenge: How could we change the code above to check for two-in-a-row SOUTHWARD or DIAGONALLY?!

What `two_in_a_row(A)` places into B...
First, try it by eye... ... then, on hw9pr2, w/Python!

\[
A = \begin{bmatrix}
\text{row 0} & [ ' ', 'X', 'O', ' ', 'O' ] , \\
\text{row 1} & [ 'X', 'X', 'X', 'O', 'O' ] , \\
\text{row 2} & [ ' ', 'X', 'O', 'X', 'O' ] , \\
\text{row 3} & [ 'X', 'O', 'O', ' ', 'X' ] \\
\end{bmatrix}
\]

\textit{inarow\_3east('X', 1, 0, A)}
First, try it by eye... … then, on hw9pr2, w/Python!

\[
A = \begin{bmatrix}
\text{row 0} & [ & ' ' & , & 'X' & , & 'O' & , & ' ' & , & 'O' & ] & , \\
\text{row 1} & [ & 'X' & , & 'X' & , & 'X' & , & 'O' & , & 'O' & ] & , \\
\text{row 2} & [ & ' ' & , & 'X' & , & 'O' & , & 'X' & , & 'O' & ] & , \\
\text{row 3} & [ & 'X' & , & 'O' & , & 'O' & , & ' ' & , & 'X' & ] & \\
\end{bmatrix}
\]

\[
\text{_checker start row start col LoL } \\
\text{inarow_3east('X', 1, 0, A)} \quad \rightarrow \quad \text{True} \\
\text{inarow_3south('O', 0, 4, A)} \quad \rightarrow \quad \\
\text{inarow_3southeast('X', 2, 3, A)} \quad \rightarrow \quad \\
\text{inarow_3northeast('X', 3, 1, A)} \quad \rightarrow \quad \\
\]

the data doesn't wrap around
Lab Problem: *Creating life*

Many life configurations expand forever...

What is the largest amount of the life universe that can be filled with cells?

How sophisticated can Life-structures get?

www.ibiblio.org/lifepatterns/
Life @ HMC?
Life & HMC!

Life, universally!
Mid-term feedback ...

On average, how much time per week do you spend on CS5 *outside class + lab*?

![Graph showing time spent outside class and lab with an annotation indicating 4.7 hours.](image-url)
Mid-term feedback ...

How does CS5's workload compare to other classes you're taking this term?

1 3 5

much lighter  about the same  much heavier

[1, 1.9] (1.9, 2.8] (2.8, 3.7] (3.7, 4.6] (4.6, 5.5]
How would you judge the pace of CS5?
Mid-term feedback ...

On average, how much time per week do you spend on CS5 *outside class + lab*?

How does CS5’s workload compare to other classes you're taking this term?

How would you judge the *pace* of CS5?

Circle your year:  First-year  Sophomore  Junior  Senior  Other

Something you’d *keep* about CS5 …?

Something you’d *change about / get rid of / add to* CS5 …?

Other thoughts optional, but 142% welcome:
Mid-term feedback ...

Something you'd **keep** about CS5 ...?
- Recursion before loops
- Grutoring (and grutors)
- Homework (inc. EC)
- Incentivized labs
- Website, schedule
- Practice problems in lecture

Something you'd **change about** CS5 ...?
- Recursion before loops
- Less 42
- More 42
- Less hardware (circuits, assembly)
- More hardware (circuits, assembly)
- More office hours
- More practice problems
- More topics – OOP, efficiency
- More languages
Mid-term feedback ...

Something you'd **keep** about CS5 ...

- Recursion before loops
- Handouts
- Grutoring (and grutors)
- Homework (inc. EC)
- Incentivized labs
- Website, schedule
- Practice problems in lecture

Something you'd **change about** CS5 ...

- Recursion before loops
- Handouts (wasted paper)
- Less 42
- More 42
- Less hardware (circuits, assembly)
- More hardware (circuits, assembly)
- More office hours
- More practice problems
- More topics – OOP, efficiency
- More languages
Mid-term feedback ...

Something you'd *change about* CS5 ...?

Handouts (wasted paper)

Tell me on today’s worksheet and I will reduce the copies I print!

More office hours

Tues – 1:10-5:15
Thurs – 2-4
Fri – 9-10
Other: https://calendly.com/jmedero

More topics - OOP, efficiency

Coming soon to a CS5 near you!

More languages

Take more CS! CS60 (lots!), CS70 (C++),
...

Object-Oriented Programming
Lec 19 ~ Classes and Objects...

CS-specific **names**
- `class`, `type`, `user-defined type`, `template`
- `object`, `instance`, `self`, `variable`, `container`
- `method`, `function`
- `constructor`, `initializer`, `__init__`
- `__repr__`, `printer`

CS-specific **topics**
- Syntax needed to define a `class`
- Syntax needed to create an `object`
- The use of `self` to refer to a specific `object` + within the definition of a `class`!

Also!
- Midterm exams...
- All Python variables are objects...
- Examples
  - `Student` class (that we define)
  - `str` class (Python-defined)
  - `Date` class (that we define)
Classes and Objects

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

(1) A class is a type (instance)

(2) An object is one such variable.

There will typically be MANY objects of a single class.
(1) A class is a type of object.

(2) An object is one such variable.

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...
Designing a **student** class!

**Data** contained:
- name
- year

**Functions** contained:
- `defer(numyrs)`
- and others needed by Python: `__init__`, `__repr__`
Student is a class

1. constructor, init

2. its string representation

3. change things via methods

fr and sr are objects
Everything is an object! strings, for example:

\[ \text{In : } s = \text{str}(42) \] This calls the \text{str} constructor.

\[ \text{In : } \text{type}(s) \] shows the type of \( s \) is \text{str}

\[ \text{In : } \text{dir}(s) \] shows all of the methods (functions) of \( s \)

```
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

\[ s = \text{"hi there"} \]

\[ s.\text{split}(\text{" ")} \rightarrow [\text{"hi"}, \text{"there"}] \] 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 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 \textbf{Date} class and object, \texttt{d}

\begin{center}
\begin{tabular}{ccc}
11 & 12 & 2013 \\
\text{month} & \text{day} & \text{year} \\
\end{tabular}
\end{center}

memory location $\sim 42042778$
A **Date** class and object, `d`

- **month**: 11
- **day**: 12
- **year**: 2013

Memory location ~ 42042778

It's an alien date!
Date object, d

```python
memory location ~ 42042778
```
Quiz ~ naming

point each name to its piece of the code...

A. class keyword (keyword)
B. class definition (end)
C. object creation (4)
D. methods (3)
E. constructor
F. data member (3)
G. what prints Dates?

Extra: when's the next leap year? Is 2100 a L.Y.?

Extra: what should ny - today be? What about nc - d?
class Date:
    ""
    Date is a user-defined data structure -- a class that stores and manipulates dates
    ""

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

    def __repr__(self):
        ""
        This method returns a string representation for the
        object of type Date that calls it (named self).
        ""
        It's called by the print statement!
        ""
        s = '{:02d}/{:02d}/{:04d}'.format(self.month, self.day, self.year)
        return s

    def isLeapYear(self):
        ""
        Returns True if self, the calling object, is
        in a leap year; False otherwise.
        ""
        if self.year % 400 == 0: return True
        if self.year % 100 == 0: return False
        if self.year % 4 == 0: return True
        return False

d = Date(11,12,2013)
today = Date(11,13,2013)
y2000 = Date(1,1,2000)
y2100 = Date(1,1,2100)

Four objects constructed here...
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.

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

In : wd = Date(11,12,2013)
In : wd.isLeapYear()
Out: False

In : d = Date(1,1,2020)
In : d.isLeapYear()
Out: True
Lab Tuesday

You'll create a `Date` class with

```python
yesterday(self) → -= 1
tomorrow(self) → += 1
addNDays(self, N) → += N
subNDays(self, N) → -= N
isBefore(self, d2) → <
isAfter(self, d2) → >
diff(self, d2) → -
dow(self) →
```

methods  operators!

Prof. Benjamin!
`no computer required...`
What's the `diff`?

In : `today = Date(11,13,2018)`  
In : `wd = Date(11,12,2013)`  
In : `today.diff(wd)`  
Out: 1827

In : `today - wd`  
Out: 1827  

In : `wd - today`  
Out: -1827

In : `eraday = Date(1,1,1)`  
In : `today.diff(eraday)`  
Out: 737010

In : `today - eraday`  
Out: 737010

This gives me pause
Where's the dow?

In : `sm1 = Date(10,28,1929)`
Out: 'Monday'

In : `sm2 = Date(10,19,1987)`
Out: 'Monday'

In : `sm1.dow()` uses a *named* object...

In : `sm2.dow()` uses a *named* object...

In : `Date(1,1,1).dow()` unnamed!
Out: 'Monday'

In : `Date(1,1,2100).dow()` unnamed!
Out: 'Friday'

In : `Date(10,10,2010).dow()` popular!
Out: 'Sunday'
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
Special Dates?

10/10/10: They Love Just Thinking About It

By JOHN SCHWARTZ | OCT. 8, 2010

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.
```python
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
```

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 finish building in Lab 10 this week...

Constructor!

In : d = Date(11,12,2013)
In : d.isLeapYear()
False

d contains data members named day, month, and year

>>> d
11/12/2013

The repr! the representation of an object of type Date

>>> d.isLeapYear()
False

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

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}".format( 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**?
is the variable calling a method

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

>>> print d
11/12/2013

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

```python
>>> nd = Date(1,1,2020)

>>> print nd
01/01/2020

>>> nd.isLeapYear()
True
```

These methods need access to the object that calls them: it's **self**
Problems with `==`

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

>>> wd2 = Date(11,12,2013)
>>> wd2
11/12/2013

>>> wd == wd2
False
```

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

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

- `wd`:
  - Month: 11
  - Day: 12
  - Year: 2013
  - Memory location: ~42042778

- `wd2`:
  - Month: 11
  - Day: 12
  - Year: 2013
  - Memory location: ~42042742

`==` compares memory locations, not contents.
class Date:

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

def __repr__(self):

def isLeapYear(self):

def equals(self):
    
    def equals(self, d2):
        
        def isLeapYear(self):
            return False

        def __init__(self, mo, dy, yr):
            return False

    def __init__(self, mo, dy, yr):
        return True

    def __init__(self, mo, dy, yr):
        return False

To use this, write `wd.equals(wd2)`
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 self.year == d2.year and \
       self.month == d2.month and \
       self.day == d2.day:
        return True
    else:
        return False

To use this, write    wd.equals(wd2)
Solution: `equals`

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

>>> wd2 = Date(11, 12, 2013)
>>> wd2
11/12/2013

>>> wd.equals(wd2)
True
```

`.equals` compares mo/dy/yr – because *we asked it to!*

But *who* is this convenient for?!
class Date:

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

def __repr__(self):

def isLeapYear(self):

def __eq__(self, d2):
    """ returns True if they represent the same date; False otherwise """
    if self.year == d2.year and \
    self.month == d2.month and \
    self.day == d2.day:
        return True
    else:
        return False

L==k! This is T== C==L!

redefined for our convenience!

To use this, write  \[ d == d2 \]
DIY operators ...

___eq___(self, other) defines the equality operator, ==
___ne___(self, other) defines the inequality operator, !=
___lt___(self, other) defines the less-than operator, <
___gt___(self, other) defines the greater-than operator, >
___le___(self, other) defines the less-or-equal-to operator, <=
___ge___(self, other) defines the gr.-or-equal-to operator, >=

___add___(self, other) defines the addition operator, +
___sub___(self, other) defines the subtraction operator, -

... and many more! Use dir('')

there are two underscores on each side here

I should underscore this unusual syntax!
More operators!

**Booleans**

- `__lt__(self, other)`
- `__le__(self, other)`
- `__eq__(self, other)`
- `__ne__(self, other)`
- `__gt__(self, other)`
- `__ge__(self, other)`

**arithmetic**

- `__add__(self, other)` +
- `__sub__(self, other)` -
- `__mul__(self, other)` *
- `__matmul__(self, other)` @
- `__truediv__(self, other)`
- `__floordiv__(self, other)`
- `__mod__(self, other)`
- `__divmod__(self, other)`
- `__pow__(self, other[, modulo])`
- `__lshift__(self, other)`
- `__rshift__(self, other)`
- `__and__(self, other)`
- `__xor__(self, other)`
- `__or__(self, other)`

**in-place arithmetic**

- `__iadd__(self, other)` +=
- `__isub__(self, other)` -=
- `__imul__(self, other)` *=
- `__imatmul__(self, other)` @=

[https://docs.python.org/3/reference/datamodel.html#special-method-names](https://docs.python.org/3/reference/datamodel.html#special-method-names)
Lab Tuesday

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 isBefore(self, d2):
        """ True if self is before d2, else False """
        if self.year < d2.year:
            return True
        elif self.month < d2.month:
            return True
        elif self.day < d2.day:
            return True
        else:
            return False

Date(12,31,1999).isBefore(Date(11,13,2018))

Date(11,13,2018).isBefore(Date(12,31,1999))

Why doesn't this function work correctly?!
class Date:

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

        elif self.month < d2.month and self.year == d2.year:
            return True

        elif self.day < d2.day and self.year == d2.year and self.month == d2.month:
            return True

        else:
            return False

I <3 Elf! But what about Elif?
class Date:

def __lt__(self, d2):
    """ if self is before d2, this should return True; else False """

    if self.isBefore(d2) == True:
        return True
    else:
        return False
```python
class Date:
    def __lt__(self, d2):
        """ this is less than most code! ""
        return self.isBefore(d2)
```

class Date:

def __lt__(self, d2):
    """ this is less than most code! """
    return self.isBefore(d2)

def __gt__(self, d2):
    """ this is less than most code! """
    return __.isBefore(____)
The 2 **most essential** *methods*

```python
>>> wd = Date(11,12,2013)  # construct with the CONSTRUCTOR ...
```

```text
11/12/2013
```

```python
>>> print(wd)  # print uses __repr__
```

```text
11/12/2013
```

```python
>>> wd.tomorrow()
```

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

```python
>>> print(wd)  # wd has changed!
```

```text
11/13/2013
```

```python
>>> wd.yesterday()
```

```text
*yesterday* is pretty much just like *tomorrow* (is this a good thing!?)
```

```python
>>> print(wd)  # Some methods return a value; others change the object that call it!
```

```text
11/12/2013
```
class Date:

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

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

        self.day += 1

        if self.day >= DIM[self.month]:
            self.day = 0
            self.month += 1

    if self.month == 13:
        self.month = 0
        self.year += 1

Extra How could we make this work for leap years, too?
class Date:

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

    fdays = 28 + self.isLeapYear()    # What ?!

    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 yesterday(self):
    """ moves the self date backwards 1 day """

    fdays = 28 + self.isLeapYear()  # Yay!

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

For lab: how will "wrap-around" work in this case? What cases do we need to worry about?!
Not all years are the same!

Calendar for year 1752 (United States)

Calendar for year 1712 (Sweden)
See you @ next week's lab ...

... it's a Date!

L.A. street sign *with typo* from 2006

real or otherwise!