A whole new **class** of programming

CS's building blocks: functions and composition

Designing Data!

The Date class

Solutions... (*keep within this term's CS5*)

Exams...

Not taken? Wait about 10 minutes...

whose convenience?
Concerns? Let me know!
no drawing
no sticker
2nd week of CS5...

THE WORLD IS RECURSION IS RECURSION IS RECURSION...

3rd week of CS5...

SO, LIKE, I GUESS THE ANSWER'S ALWAYS 42, RIGHT?

CS5 LAB...

YOU MEAN... USE IT OR LOSE IT?

I HATE PYTHON! I'M GONNA LOSE IT!

Circuits in CS5...

I MISS PACBOT!

Assembly Language...

HMMMM...

Loops in CS5...

JUST KEEP LOOPING JUST KEEP LOOPING
2nd week of CS5...

THE WORLD IS RECURSION.

3rd week of CS5...

SO, LIKE, I GUESS THE ANSWER'S ALWAYS 42, RIGHT?

CS5 LAB...

YOU MEAN... USE IT OR LOSE IT?

I HATE PYTHON!
I'M GONNA LOSE IT!

Circuits in CS5...

I MISS PICOBOT!

!&**!

Pop Tart Day in CS5

WE SHOULD BE CS MAJORS...

MMHMM...
A whole new **class** of programming

CS's building blocks: functions and composition

behind CS's curtain: **circuits**, **assembly**, **loops**

**Designing Data!**

CS: **theory** + **practice**
Lec 18 ~ 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

(2) An object is one such variable.

There will typically be MANY objects of a single class.
Classes and Objects

An object is a type of variable.

1. A **class** is a type.
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.

functions "methods"

example s.split()

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

strings, for example:

```python
In : s = str("the claremont colleges")
In : type(s)
In : 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']
```

Let's try it!
Designing a **student** class!

**Data contained**

- name
- year

**Functions contained**

- two needed by Python
  - `__init__`
  - `__repr__`
- a function...
Designing a **student** class!

**Data contained**

- name
- year

**Functions contained**

- two needed by Python:
  - `__init__`
  - `__repr__`

- a function we design: `defer(numyrs)`
# defining our own Student class

class Student:
    """ a class representing students """
    # [sets initial data]
    def __init__(self, name, yr):
        """ this is the constructor """
        self.name = name
        self.year = yr
    
    # the "REAPER" method (for printing)
    # [let's change from 2021 to '21]
    def __repr__(self):
        """ the not-so-grim reaper: for printing """
        s = self.name + str(self.year)
        return s

    # here's a method of our own
    # (not one of Python's __special__ ones)
    def defer(self, numyrs):
        """ defer for numyrs years """
        self.year += numyrs

    
# This is the end of the Student class

# Now, let's construct two students:
fr = Student("Frosh A.", 2023)
sr = Student("Senior B.", 2020)
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) Inside 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** class and object, $d$

A Date class and object, $d$

memory location ~ 42042778
11/12/13: A good day for a wedding?

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.
A `Date` class and object, `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(11,12,2019)
y = Date(1,1,2020)
c = Date(1,1,2100)
class Date:
    """ a blueprint (class) for objects that represent calendar days """

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.

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

These are data members – they are the information inside every Date object.
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?
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

In : wd = Date(11,12,2013)
In : wd.isLeapYear()
Out: False
In : d = Date(1,1,2020)
In : d.isLeapYear()
Out: True
```
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).
        """
        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

    ny = Date(1,1,2020)
    today = Date(11,12,2019)
    wd = Date(11,12,2013)
    nc = Date(1,1,2100)

    Extra: what should today > wd return?  today > ny ?
    Extra: what should ny – today be?  What about nc – wd?
Quiz ~ names!

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

class keyword (keyword)
class definition (end)
object definitions (4)
members (3) and __repr__
constructor
data member (3)
what prints Dates?

Extra: what should today > wd return? today > ny?

Extra: what should ny – today be? what about nc – wd?
The variable `self` is the object calling a method.

```python
>>> wd = Date(11, 12, 2013)
>>> print wd
11/12/2013
>>> wd.isLeapYear()
False
```

```python
>>> d = Date(1, 1, 2020)
>>> print d
01/01/2020
>>> d.isLeapYear()
True
```

These methods need access to the object that calls them: it's `self`.

*Note: The code examples are in Python.*
You'll create a `Date` class with

```
- 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!
What's the **diff**?

<table>
<thead>
<tr>
<th>In</th>
<th>Method/Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>today = Date(11,12,2019)</td>
<td>method</td>
</tr>
<tr>
<td>wd = Date(11,12,2013)</td>
<td>operator</td>
</tr>
<tr>
<td>today.diff(wd)</td>
<td>operator</td>
</tr>
<tr>
<td>eraday = Date(1,1,1)</td>
<td>method</td>
</tr>
<tr>
<td>today.diff(eraday)</td>
<td>operator</td>
</tr>
</tbody>
</table>

```text
In : today = Date(11,12,2019)
In : wd = Date(11,12,2013)
In : today.diff(wd)
Out: 2191

In : today - wd
Out: 2191

In : wd - today
Out: -2191

In : eraday = Date(1,1,1)
In : today.diff(eraday)
Out: 737374

In : today - eraday
Out: 737374
```
### Where's the dow?

<table>
<thead>
<tr>
<th>In</th>
<th>sm1 = Date(10,28,1929)</th>
<th>sm2 = Date(10,19,1987)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>sm1.dow() In: sm2.dow()</td>
<td></td>
</tr>
<tr>
<td>Out</td>
<td>'Monday'</td>
<td>'Monday'</td>
</tr>
</tbody>
</table>

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

```
In : Date(1,1,2020).dow()
Out: 'Wednesday'
```

```
In : Date(10,10,2010).dow()
Out: 'Sunday'
```

- The dow looks down to me!
- uses a **named** object...
- unnamed!
- unnamed!
- popular!
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.
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.
Wopperer of San Francisco have been waiting nearly two years for their wedding date to roll around, having realized ever dinner with friends in 2008 that, as one suggested, 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, “so 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.
Problems with  ==

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

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

>>> wd == wd2
False
```

How can this be False?

This constructs a different Date object, but with the same mo/dy/yr.
Problems with `==`

```python
>>> wd = Date(11, 12, 2013)
```
```python
>>> wd
```
```python
11/12/2013
```
```
>>> wd2 = Date(11, 12, 2013)
```
```python
>>> wd2
```
```python
11/12/2013
```
```
>>> wd == wd2
```
```python
False
```
```
How can this be False?
```
```
Python objects are handled by reference...
```
```
== compares references!
```
Two **Date** objects:

```
11 12 2013
month day year
```

```
11 12 2013
month day year
```

memory location ~ 42042778

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, 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

Let's write our own equality-tester

wd.equals(wd2)  
wd2.equals(wd)
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

wd.equals(wd2)  # 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
```

*This constructs a different Date object, but with the same mo/dy/yr*

**.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!

To use this, write d == d2

redefined for our convenience!
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

In-place arithmetic

https://docs.python.org/3/reference/datamodel.html#special-method-names
Lab today – or tomorrow()

You'll create a Date class with

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

methods  \(\uparrow\)  operators!

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

When doesn't this function work?!

Date(12,31,1999).isBefore(Date(11,12,2019))

Date(11,12,2019).isBefore(Date(12,31,1999))
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
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
class Date:

def __lt__(self, d2):
    """ is self < d2? (earlier) ""

    return self.isBefore(d2)
class Date:

def __lt__(self, d2):
    """ is self less than d2? (before) ""
    return self.isBefore(d2)

def __gt__(self, d2):
    """ is self greater than d2? (after) ""
    return _____.isBefore(____)
The 2 *most essential methods*

```python
>>> wd = Date(11,12,2013)  # construct with the
CONSTRUCTOR ...
>>> print wd
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
    self.day
if self.month

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

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

Use for hw10pr1 this week!
class Date:

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

    better as a variable!

    DIM = [0, 31, fdays, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31, 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
```python
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
```

the "Luke trick"
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)

January
February
March
April
May
June
July
August
September
October
November
December

Calendar for year 1712 (Sweden)

January
February
March
April
May
June
July
August
September
October
November
December
See you @ this week's lab ... 

... it's a **Date**!

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