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 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**
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"

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

Everything is an object! strings, for example:

In : `s = str(42)`  This calls the `str` constructor.
In : `type(s)`  Shows the type of `s` is `str`
In : `dir(s)`  Shows all of the methods (functions) of `s`

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

Let's try some!
A **Date** class and object, \( d \)

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):
        self.month = mo
        self.day = dy
        self.year = yr

    def __repr__(self):
        return f"{self.month}/{self.day}/{self.year}"

    def isLeapYear(self):
        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)
print(d.isLeapYear())
```

<table>
<thead>
<tr>
<th>Date</th>
<th>memory location ( \sim 42042778 )</th>
</tr>
</thead>
</table>

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):
        self.month = mo
        self.day = dy
        self.year = yr

    def __repr__(self):
        return f"{self.month}/{self.day}/{self.year}"

    def isLeapYear(self):
        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)
print(d.isLeapYear())
```

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

Quiz ~ naming  
point each name to its piece of the code...

| class keyword (keyword) |
| class definition (end) |
| object creation (4) |
| methods (3) |
| constructor |
| data member (3) |
| what prints Dates? |

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)` → \( [ ] \)

Prof. Benjamin!  
no computer required...

Lab Tuesday

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?

In:

- `today = Date(11,13,2018)`
- `wd = Date(11,12,2013)`
- `today.diff(wd)`
  Out: 1827
- `today - wd`
  Out: 1827
- `wd - today`
  Out: -1827
- `eraday = Date(1,1,1)`
- `today.diff(eraday)`
  Out: 737010
- `today - eraday`
  Out: 737010

Where's the dow?

In:

- `sm1 = Date(10,28,1929)`
- `sm2 = Date(10,19,1987)`
- `sm1.dow()`
  Out: 'Monday'
- `sm2.dow()`
  Out: 'Monday'
- `Date(1,1,1).dow()`
  Out: 'Monday'
- `Date(1,1,2100).dow()`
  Out: 'Friday'
- `Date(10,10,2010).dow()`
  Out: 'Sunday'

Special Dates?

The New York Times

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.

The Date class

```
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

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

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.
Date

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

This is a class. It is a user-defined datatype that you’ll finish building in Lab 10 this week...

Constructor!

`d` contains data members named `day`, `month`, and `year`.

**class Date:**

"""
  a blueprint (class) for objects that represent calendar days
""

```python
    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`?

The `repr` for Date objects:

It tells Python how to print these objects.

Why `self` instead of `d`?

Problems with `==`

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

This constructs a different Date.

```python
>>> wd2 = Date(11,12,2013)
>>> print wd2
11/12/2013
```

```python
>>> wd == wd2
False

How can this be False?
```

Python objects are handled by reference…  
`==` compares references!
Two `Date` objects:

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

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

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

```
Solution: `equals`
```

```
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
```

```

equals
```

```
To use this, write  d == d2
```

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

```
>>> wd
11/12/2013
```

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

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

```
Solution: `equals`
```

```
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
```

```
__eq__
```

```
To use this, write  d == d2
```

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

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

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

```
Solution: `equals`
```

```
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
```

```
__eq__
```

```
To use this, write  d == d2
```

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

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

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

```
Solution: `equals`
```

```
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
```

```
__eq__
```

```
To use this, write  d == d2
```

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

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

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

```
Solution: `equals`
```

```
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
```

```
__eq__
```

```
To use this, write  d == d2
```

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

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

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

arithmeti

in-place arithmetic

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

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):
        """ 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 self.isBefore(d2)
The 2 most essential methods

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

>>> wd.tomorrow()          # the tomorrow method returns nothing at all. Is it doing anything?
>>> print(wd)              # wd has changed!
11/13/2013

>>> wd.yesterday()         # yesterday is pretty much just like tomorrow (is this a good thing?)
>>> print(wd)              # Some methods return a value; others change the object that call it!
11/12/2013
```

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

```python
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 = 1
            self.month += 1
            if self.month > 12:
                self.year += 1
                self.month = 1
```

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

def tomorrow(self):
    """ moves the self date ahead 1 day """
    fdays = 28 + self.isLeapYear()  # What?!  "the Luke trick"
    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?!

See you @ next week's lab ...

... it's a Date!

L.A. street sign with typo from 2006
Quiz ~ naming

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

class keyword (keyword)

class definition (end)

object creation (4)

methods (3)

constructor

data member (3)

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?

Your name(s) ____________________________