Introduction to Object-Oriented Programming (OOP)

So far, we have learned
- Circuits and basic hardware components in a computer
- Assembly programming language
- Python
  - Recursion
  - Lists
  - Functions and parameters
  - Loops
- The Python elements we learned can be used to accomplish some programming tasks as we have seen.

- However, it is more natural to represent things in the real world as objects in a programming language!
- For example,
  - A car that has an engine, a transmission, … that can move under some instructions …
  - A dog that can walk and bark …
  - A bird that can fly …
  - A book that has chapters and sections, and can be flipped through (read) …

Think, for example, if you are asked to build a program to maintain the information about a collection of books that contains title, author, publisher, date of publishing, and others, how would you do it?

One approach would be to use multiple lists or arrays
- titles
- authors
- publishing_dates

To reference the 4th book in the collection, one would use titles[3], authors[3], publishing_dates[3]

Another approach would be to define a list (or an array) of book **objects**, each of which has an **author** field, a **title** field, a **publishing_date** field. When referring to a book, one would use book[3].author, book[3].title, …

Object Oriented Programming
- An OOP language allows you to create your own types
- A **class** is a *type*
- An **object** is a particular **instance** of that type
- There can be one instance
  - Or many instances
- There can be operations (functions, a.k.a., methods) that apply to the object.

6

Everything in Python is an **object**!

Its capabilities depend on its **class**.

functions

"methods"

class

by the way, you can build your own...

Objects

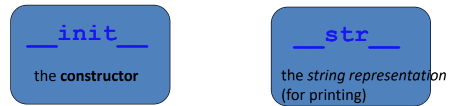
An object is a structure - like a list - except

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

usually called "fields", "attributes"

(2) An object ***contains its own functions*** that it can call (use)!

usually called "**methods**" instead of functions:



Here's what a class may look like (Python syntax)

```
class Bird:
  color = 'Yellow' # class field

  def __init__(self): #Constructor
    self.weight = 10  #instance field

  def fly(self):
    if self.weight >15:
        self.lightenTheLoad() # calling a method
    else:
        print("FLYING!!!!")

  def lightenTheLoad(self):
    print('Splat!')

  def eat(self , food):
    self.weight = self.weight + food
```

bigBird = Bird()

bigBird.fly()

bigBird.eat(20)

bigBird.fly()

Compared to what we know already …

The use of "Bird" class	The use of "String" class
sparrow = Bird() print(sparrow) sparrow.fly() sparrow.eat(20) sparrow.fly()	myString = "Hello World!" capital = myString.caplitalize() words = myString.split() print(capital) letterO = words[0].endswith('o') print(letterO)

Date

This is a class. It is a user-defined datatype (that you'll build in Lab!)

```
>>> d = Date(3,30,2011)
>>> print(d)
03/30/2011

>>> d.is_leap_year()
False

>>> d2 = Date(1,1,2012)
>>> print(d2)
01/01/2012
>>> d2.is_leap_year()
True
```

this calls a **CONSTRUCTOR** …

this is an object of type Date

the `repr`esentation of a particular object of type Date

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

Another object of type Date - again, from the constructor.

How does it know to return True, instead of False in this case ??

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 __str__( self ):
        """ used for printing Dates """
        s = "{:02d}/{:02d}/{:04d}".format(self.month,self.day,self.year)
        return s

    def is_leap_year( self ):
        """ anyone know the rule? """
```

C/C++ printf style string formatting. See Python string documentation

self is the specific OBJECT THAT CALLS A METHOD

```
>>> d = Date(11,8,2011)
>>> print(d)
11/08/2011

>>> d.is_leap_year()
False
```

These methods need access to the object that calls them

```
>>> d2 = Date(1,1,2012)
>>> print(d2)
01/01/2012
>>> d2.is_leap_year()
True
```

These methods need access to the object that calls them

13

a Leap of faith….

```
class Date:
    def __init__( self, mo, dy, yr ): (constructor)
    def __str__( self ): (for printing)

    def is_leap_year( self ):
        if self.year%400 == 0:
            return True
        if self.year%100 == 0:
            return False
        return self.year % 4 == 0
```

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 ye

John Herschel

14

Classes – DIY data

Class: a user-defined datatype

Object: data or a variable whose type is a class

```
d = Date( 11, 11, 2011 )
d.tomorrow()
print(d)
```

object — constructor — method — uses str

d would be named **self** inside the **Date** class…

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

str: the `__str__` function returning a string to print

data members: the data in **self**: `self.day, self.month, self.year`

15

Date ids

```
>>> d = Date(11,8,2011)
>>> print(d)
11/08/2011

>>> d2 = Date(11,9,2011)
>>> print(d2)
11/09/2011

>>> d == d2
False
```

this creates a different Date object!

```
>>> d2.yesterday()
>>> d == d2
False
```

Date ids

```
>>> d = Date(11,8,2011)
>>> print(d)
11/08/2011

>>> d2 = Date(11,9,2011)
>>> print(d2)
11/09/2011

>>> d == d2
False
```

this initializes a different Date!

Need an **equals** method to check if their VALUES are equal, not their MEMORY ADDRESSES

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

18

equals

```
class Date:
    def __init__( self, mo, dy, yr ):
    def __str__(self):
    def isLeapYear(self):

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

19

References

```
>>> d = Date(11,8,2011)
>>> print(d)
11/08/2011

>>> d2 = d
>>> print(d2)
11/08/2011

>>> d.yesterday()
>>> print(d2)
11/07/2011
```

These refer to the same object!

We need a way to make a copy of an object! Simple assignment will not work!

copy

```
class Date:

    def __init__( self, mo, dy, yr ):
    def __str__(self):
    def is_leap_year(self):
    def equals(self):

    def copy(self):
        """ returns a DIFFERENT object w/SAME date value! """
    return Date(self.month, self.day, self.year)
```

Whenever you want to create a brand-new object, use the appropriate CONSTRUCTOR

21

What's wrong?

d = Date(1,1,2012)
d2 = Date(11,8,2011)
d.is_before(d2)
True

```
class Date:

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

     if self.month < d2.month:
       return True

     if self.day   < d2.day:
       return True

     if self.year  < d2.year:
       return True

     return False
```

22

Better

d = Date(1,1,2012)
d2 = Date(11,8,2011)
d.is_before(d2)
False

```
class Date:

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

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

      if self.day < d2.day and self.month == d2.month:
       return True

      return False
```

23

tomorrow()

Write this tomorrow method.

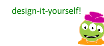
```
class Date:

   def tomorrow(self):
      """ moves the date that calls it ahead 1 day """
      monthLen = [0,31,28,31,30,31,30,31,31,30,31,30,31]
```

first, add 1 to self.day

then, adjust month and year, if needed

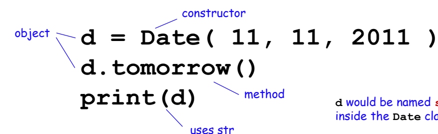
24

Classes – DIY data



Class: a user-defined datatype

Object: data or a variable whose type is a class



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

str: the `__str__` function returning a string to print

data members: the data in **self**: `self.day`, `self.month`, `self.year`

25

Operator Overload

- If we define a class of objects, we may be able to or need to reuse some common operators
- For example, to compare two date objects, can we say something like ‘d1 > d2’ if d1 is AFTER d2?
- Or for two objects in a rational number class, can we say something like ‘r1 > r2’?
- Python and any other modern programming languages allow ‘operator overload,’ that is, re-define the meaning of a common operator.

The example of the Rational class

- From our text book
- r1 = 1/3, r2 = 2/5, how to do operations such as r1 + r1, or comparisons such as if r1 == r2, or if r1 > r2?
- We need overload the operators such as < or ==
- How to do it?
  - Define specially named methods, `__add__()`, `__eq__()`, `__ge__()`, `__gt__()`

Show Rational.py