

**Genome 559**  
**Intro to Statistical and**  
**Computational Genomics**  
**2009**

**Lecture 16b:**  
**Classes and Objects, III**  
**Larry Ruzzo**

# Continuing “Date” example

```
class Date:
    def __init__(self, day, month) :
        self.myday = day
        self.mymonth = month
    def printdate(self)
        print self.myday, self.mymonth
```

```
mydate = Date(15, "January")
mydate.printdate()
15 January
```

# Practice (cont.)

Write a function for our date class that adds a number to a date

*Algorithm:*

add the number to the day; if this goes past the end of a month, advance to the next month; repeat

*Step 1:* Set up a dictionary mapping month name (key) to number of days in month (value)

*Step 2:* Write a function `nextmonth(month_name)` returning name of the next month.

*Step 3:* Write `add(self, numdays)`. Assume `numdays > 0`. (Use the algorithm above, dictionary to find the number of days in a month, and the `nextmonth` function to find the next month.)

# Practice: Step 1 solution

```
daysinmonth = {  
    "Jan": 31,  
    "Feb": 28,  
    "Mar": 31,  
    "Apr": 30,  
    "May": 31,  
    "Jun": 30,  
    "Jul": 31,  
    "Aug": 31,  
    "Sep": 30,  
    "Oct": 31,  
    "Nov": 30,  
    "Dec": 31  
}
```

## Practice: step 2 solution

```
# It could also be done with 12 if statements  
# but in general, simpler is better
```

```
def nextmonth(thismonth):  
    monthlist = ["Jan", "Feb", "Mar",  
                 "Apr", "May", "Jun",  
                 "Jul", "Aug", "Sep",  
                 "Oct", "Nov", "Dec",  
                 "Jan"]  
    for index in range(0, len(monthlist)) :  
        if (monthlist[index] == thismonth) :  
            return monthlist[index + 1]  
    print "Illegal month", thismonth
```

**Q: What's returned if illegal?**

## Practice step 2, alternate solution a

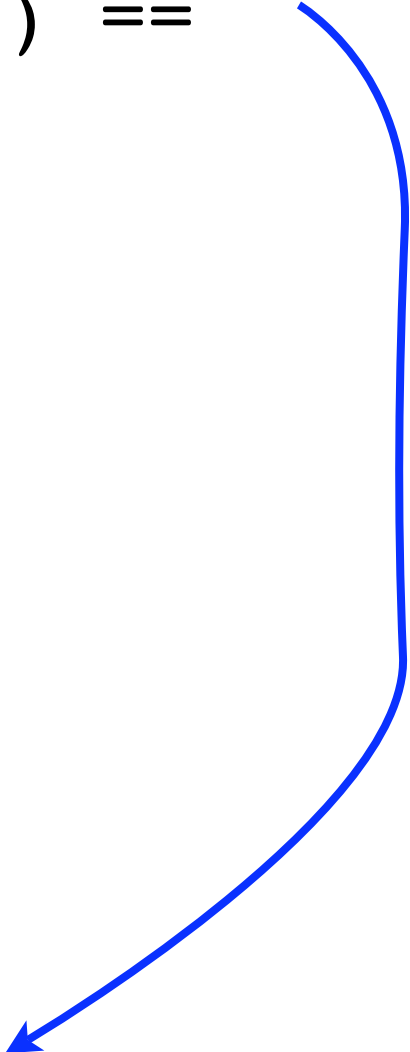
```
# use a dictionary to hold the
# "next month" mapping

def nextmonth(thismonth):
    nextmonthdict = {
        "Jan": "Feb", "Feb": "Mar", "May": "Apr",
        "Apr": "May", "May": "Jun", "Jun": "Jul",
        "Jul": "Aug", "Aug": "Sep", "Sep": "Oct",
        "Oct": "Nov", "Nov": "Dec", "Dec": "Jan"}
    if thismonth in nextmonthdict :
        return nextmonthdict[thismonth]
    else :
        print "Illegal month", thismonth
```

## Practice step 2, alternate solution b

```
# A handy nerdy trick: "a % b", usually read  
# "a mod b", means the remainder when a is  
# divided by b. E.g., (1%12, ..., 11%12) ==  
# (1, ..., 11), but 12%12 == 0, so Dec + 1  
# wraps around to Jan again; sweet!
```

```
def nextmonth(thismonth):  
    monthlist = ["Jan", "Feb", "Mar",  
                 "Apr", "May", "Jun",  
                 "Jul", "Aug", "Sep",  
                 "Oct", "Nov", "Dec"]  
    for index in range(0, len(monthlist)) :  
        if (monthlist[index] == thismonth) :  
            return monthlist[(index + 1) % 12]  
    print "Illegal month", thismonth
```



## Practice step 3 solution

```
class Date:
    def __init__(self, day, month) :
        self.myday = day self.mymonth = month
    def printUS(self) :
        print self.mymonth, self.myday
    def printUK(self) :
        print self.myday, self.mymonth
    def
add(self, numdays) :
    self.myday = self.myday + numdays
    while self.myday > daysinmonth[self.mymonth] :
        self.myday = self.myday-daysinmonth[self.mymonth]
        self.mymonth = nextmonth(self.mymonth)
```

Q: where could/should daysinmonth & nextmonth( ) go?




# date.add() changes its argument

If you say `mybirthday.add(8)` you change `mybirthday`  
It might be better to return a *new* date object:

```
def addnew(self, numdays) :  
    newmonth = self.mymonth  
    newday = self.myday + numdays  
    while newday > daysinmonth[newmonth] :  
        newday = newday - daysinmonth[newmonth]  
        newmonth = nextmonth(newmonth)  
    return Date(newday, newmonth)
```

Make a new  
"Date" object



# Using date.addnew()

```
>>> mybirthday = Date(6, "July")
```

```
>>> mybirthday.printUS()
```

```
July 6
```

```
>>> party = mybirthday.addnew(4)
```

```
>>> party.printUS()
```

```
July 10
```

```
>>> mybirthday.printUS()
```

```
July 6
```

# How to print a date

Why is “print” fine for numbers, tuples, etc.

```
>>> print ("Jan", 5)
('Jan', 5)
```

but funky for class instances?

```
print mydate
<__main__.date instance at 0x247468>
```

Yes, `mydate.printUS()` works, but seems clunky

# Here's another way

Actually, “print” doesn't need special knowledge of how to print numbers, strings, tuples, ...

It just knows how to print strings, and relies on each class to have a `__str__()` method that returns a pretty string representing the object.

(“<\_\_main\_\_.date instance at 0x247468>” is the result of calling the *default* `__str__()` method.)

# Printing dates

```
class Date:
    def __init__(self, day, month) :
        self.myday = day
        self.mymonth = month

    def __str__(self) :
        return '%s %s'%(self.mymonth, self.myday)

    add(self, numdays) :
        (etc., as before)

birthday = date(3, "Sep")
print "It's", birthday, ". Happy Birthday!"
```

# Advanced topic: Allowing the plus sign

While we're at it, how come “+” works (but differently) for numbers and strings and tuples and ..., but not for dates?

Yes,

```
“party = mybirthday.addnew(4)”
```

works to add numbers to dates, but

```
“party = mybirthday + 4”
```

seems so much more natural. Can we do it?

# Advanced topic: Overloading “+”

Yes! Again, ‘+’ isn’t as smart as you thought; it calls class-specific “add” methods (“\_\_add\_\_”) to do the real work:

```
def __add__(self, numdays) :
    newmonth = self.mymonth
    newday = self.myday + numdays
    while newday > daysinmonth[newmonth] :
        newday = newday - daysinmonth[newmonth]
        newmonth = nextmonth(newmonth)
    return Date(newday, newmonth)

# usage example
mybirthday = Date(6, "July")
party = mybirthday + 4
print mybirthday, party
July 6 July 10
```

 mybirthday.\_\_add\_\_(4)

# Operator overloading

This shows some of the power of classes in Python; we can make new classes, like Date, behave like built-in ones

Operator overloads involve names with underscores

Common operator overloading methods

```
__init__ # object creation
__add__  # addition (+)
__mul__  # multiplication (*)
__sub__  # subtraction (-)
__lt__   # less than (<)
__str__  # printing
__call__ # function calls
...      # And more for indexing, slicing, iteration...
```

Try “>>>dir(object)” in Python to see what’s there



# Pros and Cons

## Good aspects of operator overloading

- Make the date class easier to use
- Can use your own classes just as you use built-in ones
- It's very cool

## Bad aspects:

- If you overload the + sign to do subtraction, you will make your life miserable
- Must be sure that the resulting functions don't contain boobytraps
- Cool code can distract you from getting the job done

Bottom line: this is an advanced technique which you may or may not need

One exception: almost all classes will need init functions

# Inheritance:

## do the common parts once

```
class Seq:
    def print_FASTA(self): ...
class DNA(Seq):
    def digest(self): ...
    def rev_comp(self): ...
class Prot(Seq):
    def digest(self): ...
```

*Superclass* for seqs in general, with appropriate methods common to all

Separate *subclasses* for protein vs DNA sequences, with methods appropriate to each

```
myseq = DNA(file.readline())
frags = myseq.digest()
myseq.print_FASTA()
```

myseq is a “DNA” object; doesn’t have a “print\_FASTA” method, but *inherits* it from Seq superclass

# “Classes”: Summary

Most useful in (but not restricted to) large programs

Classes package together related data plus the functions (“methods”) appropriate thereto

Method calls automatically find the def of the given name within their own class, not some other one spelled the same

The relevant object is always passed to the method as its 1st parameter, called “self” by convention

Method names starting & ending with “\_\_” are special, allowing “operator overloading” and other emulation of “standard” behavior

# Practice Problem 4

After using “date” for a while, you decide that it was a mistake to keep “mymonth” as a string. Instead, you now want to keep it as an integer 0..11. Change your class definition to do this, but leave the *interface* to users of the class unchanged. In particular the constructor and print methods should still take/print the month as a string.

# Practice 4 solution

```
daysinmonth = (31,28,31,30,31,30,31,31,30,31,30,31)
monthlist = ["January", "February", ..., "December"]
def nextmonth(thismonth):
    if 0 <= thismonth < 12 :
        return (thismonth + 1) % 12
    print "Illegal month", thismonth
def month2str(monthnum):
    return monthlist[monthnum]
def str2month(monthstr):
    for index in range(0,len(monthlist)) :
        if (monthlist[index] == monthstr) : return index
    print "Illegal month", monthstr
```

# Practice 4 solution

```
class date:
    def __init__(self, day, monthstr) :
        self.myday = day
        self.mymonth = str2month(monthstr)
    def print(self) :
        print month2str(self.mymonth), self.myday
    def add(self, numdays) :
        self.myday = self.myday + numdays
        while self.myday > daysinmonth[self.mymonth] :
            self.myday = self.myday - daysinmonth[self.mymonth]
            self.mymonth = nextmonth(self.mymonth)
```