

Genome 559

Intro to Statistical and Computational Genomics

Lecture 14b:
Classes and Objects, Part I
Larry Ruzzo

Classes and Objects

What is a class ?

What is an object?

Why use them?

How to define and use them

A class is a defined data type

- Built-in classes in Python include string and dictionary
- A class defines the kinds of data and functions that are available

An object is an instance (example) of a class

- For example:
 - string is a class
 - `mystring = "AGGCGT"` creates an object of class string
- You can only have one class named "string"
- You can have many objects which all belong to class string:
 - `mystring = "AGGCGT"`
 - `yourstring = "Fred"`
- The string class provides many useful functions which all string objects can use
- `mystring.upper()`, `yourstring.split()`, etc.

Why use classes

Keep related data together

Keep functions connected to the data they work on

Example:

A “Date” class could keep the day and month together

It could offer functions such as “add a number to a date”

Could be done without classes, but classes conveniently organize it all, e.g., avoiding errors such as

Setting month to “January”

Copying the month without the associated day

14 days after Feb 18 probably shouldn't be Feb 32

Plus Biopython and many other tools use them extensively

Using Objects

(Surprise: you've been doing so all along)

```
mystring = "ATCCGCG"
```

```
print mystring.find("C")
```

2

← position of first "C"

```
print mystring.count("C")
```

3

← number of "C"s

objects

object *attributes*:
count is a function;
pi is a number

```
print math.pi
```

3.1415926535897931

Defining a new class

As an example, we'll build a simple "Date" class:

A date consists of a month and a day

We will also provide a function to add a number to a date

A useful date class would need

more data (year?)

more functions (subtract two dates?)

more error checking

but this is a start. See the "datetime" library module...

A very very simple “Date” class

Every class is a subclass of another, and “object” is a generic choice...

```
class Date(object):  
    "Dates => day, month."
```

A class definition

```
mydate = Date()  
mydate.day = 15  
mydate.mon = "Jan"  
print mydate
```

Creates a class *instance*

Creates/initializes *attributes*

```
<__main__.Date instance at 0x1005380e0>  
print mydate.day, mydate.mon
```

```
15 Jan
```

```
yourdate = mydate
```

Copies a class instance

Hmmm... Not so useful

That's completely legal

Copying the whole thing at once is handy

But otherwise, that's not so useful

E.g., still possible to forget to include both a day and a month

Try again...

Continuing “Date” example

```
class Date(object):  
    "Dates => day,month."  
    def __init__(self, day, month):  
        self.day = day  
        self.mon = month
```

Special names



```
mydate = Date(15, "Jan")  
print mydate  
<__main__.Date instance at 0x1005380e0>  
print mydate.day, mydate.mon  
15 Jan
```

That's better, but...

Special function “`__init__`” is called whenever a Date object instance is created. (A class *constructor*.)

It makes sure the object is properly initialized. In this case, every Date object will contain day and month attributes. Special name “`self`” lets it access the object in question no matter what the caller named it.

But printing is still clumsy.

Try again ...

Continuing “Date” example

```
class Date(object):  
    "Dates => day,month."  
    def __init__(self, day, month) :  
        self.day = day  
        self.mon = month  
    def printdate(self)  
        print self.day, self.mon
```

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

Magic first arguments:

`__init__` defined w/ 3 args; called w/ 2;
`printdate` defined w/ 1 arg; called w/ 0.
mydate passed in both cases, so function
knows on which object it is to act.

Class Declarations - Summary

The class statement defines a new class

Inside the class (note the colon and indentation), the special name `__init__` is the class constructor – called whenever a new instance of the class is created, to initialize it

The special name “self” means the current object of that class

Variables named `self.something` are *instance variables* of the class

Every instance of the class will have all instance variables defined in the constructor

E.g., this class has instance variables “day” and “mon”. (Spelling it out as “month” is better, I’m just saving space on the slides...)

More features of our class

- All functions in a class start with “self” as an argument
- `printdate(self)` is a straightforward function
- It prints the object’s day and month
- `__init__` is a special function that is run whenever an object of this class is created
- We use it to give the new object its values
- Almost all classes will want an `init` function

A fancier date class

```
class date(object):
    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
```

```
mydate = date(15,"January")
mydate.printUK()
15 January
mydate.printUS()
January 15
```

Adding a number to a date

- We would like a function on our date class that allows us to add a number to a date
- This is fairly tricky; we'll build it in stages
- Rules:
 - Try adding the number to the day
 - If this goes past the end of a month, advance to the next month
 - Ignore the leap year problem

Practice problem 1

- Create and fill up a dictionary:
 - Key is name of month
 - Entry is number of days in month

Practice problem 2

- Write a function `nextmonth()`
- Argument: name of a month
- Return value: name of the next month
 - If it receives “July” it should return “August”
 - If it receives “December” it should return “January”
- You can do this with a big if statement, but there are easier ways
- (Hint: make a list of months with an extra “January” at the end)

Practice Problem 3

- Copy the class definition into your program file
- Add a new class function `add(self, numdays)`
- Its net effect should be to *change* the Date object appropriately (not, e.g., print the new date)
- You may assume that “numdays” is a positive integer.
- Use the dictionary to find the number of days in a month, and the `nextmonth` function (if needed) to advance to the next month
- Note that if the number added is large, you may need to advance more than one month. (Hint: try a while loop...)

Use your new date class

- Create an object of your date class, containing a date:
- `birthday = date(6, "July")`
- Try adding various numbers to it:

```
birthday.printUS()
```

```
July 6
```

```
birthday.add(8)
```

```
birthday.printUS()
```

```
July 14
```

```
birthday.add(30)
```

```
birthday.printUS()
```

```
August 13
```

Practice: Step I 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(len(monthlist)) :  
        if (monthlist[index] == thismonth) :  
            return monthlist[index + 1]  
    print "Illegal month", thismonth  
    # alt: return monthlist[monthlist.index(thismonth)+1]
```

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 nerd trick: "a%b" (read "a mod b")  
# means the remainder when a is divided  
# by b. E.g., (0%12,%12, ..., 11%12) ==  
# (0,1,...,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(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.day = day
        self.mon = month
    def printUS(self) :
        print self.mon, self.day
    def printUK(self) :
        print self.day, self.mon
    def add(self, numdays) :
        self.day = self.day + numdays
        while self.day > daysinmonth[self.mon] :
            self.day = self.day - daysinmonth[self.mon]
            self.mon = nextmonth(self.mon)
```

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

date.add() changes its argument


Calling `mybirthday.add(8)` *changes* `mybirthday`

Maybe `.increment()` would be a better name

Perhaps even better: return a *new* date object:

```
def addnew(self, numdays) :  
    newmonth = self.mon  
    newday = self.day + 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
```