Genome 559
Intro to Statistical and Computational Genomics

Lecture 15b:
Classes and Objects, Part II
Larry Ruzzo

Today

More fun with classes

- Summary
- Motivation
- Changing objects vs New objects
- Printing

More Practice

Objects and Classes

A *class* defines the "type" of a variable

    ex: "int", "string", "list", "tuple", "dictionary"

AND defines associated functions relevant to it

    ex: string offers functions such as upper(), lower(), split()

    ex: ints offer arithmetic operations like division

    ex: *both* string and int offer "+", but it's different (Overloaded)

An *object* is an instance of a class - e.g., many string objects, one string class.

Why Classes & Objects

Bundles together data and operations on data

Allows special operations appropriate to data

- “count” or “split” on a string;
- “square root” on numbers

Allows context-specific meaning for common operations

```
x = “a”; x*2  vs  x = 42; x*2
date(Jan,31) + 1
```

Useful to you?

- Biopython (and other tools) use it extensively

More on Classes

Much in modern programming languages is motivated by the need to write large programs

- BioPython is 25 megabytes, ~0.5 million lines. (And that isn't "large.")
- Large programs aren't just small programs on steroids
- (Not always easy to appreciate until it's too late)

Python modules are one such feature

Classes/"object oriented programming" are another

- A key feature in most modern programming languages

Goal is not to make you instant experts at this, but to acquaint you with the issues so you can use "object-oriented" tools, e.g., BioPython, and won't be intimidated by these features.

Issues in Large Programs?

Management of (many!) names is one issue

```
myseq = file.readline()
frags = digest(mysequence)
```

Hmm, did you mean:

EcoR1 + DNA? `frag = dna_digest(myseq)`

trypsin + protein? `frag = tryp_digest(myseq)`

Oh, and your pal sent you `rev_comp_DNA()`

Will you ever forget/use the wrong name/case?

Modules Might Help

Have a module named `DNA` for your DNA-based tools

```
import DNA
antisense = DNA.rev_comp(myseq1)
frags = DNA.digest(myseq1)
```

Have another module named `prot` for protein tools

```
import prot
frags = prot.digest(myseq2)
```

At least you now have consistent spelling

But you might still twitch and call the wrong `.digest()`

“Classes” might help?

Have separate classes for protein vs DNA sequences, each with appropriate methods

```
class SeqDNA:
  def digest(theseq): ...
  def rev_comp(aseq): ...
class SeqProt:
  def digest(someseq): ...
myseq = SeqDNA(file.readline())
frags = SeqDNA.digest(myseq)
```

yes, this really works

A lot like the “module” version: consistent spelling, but still error-prone, *and* extra “constructor” step

Classes help more: methods & the "self" shorthand

Instead of:

```
classname.methodname(class_instance)
```

Do this:

```
class_instance.methodname()
```

Automatically converted

E.g.:

`myseq.digest()` $\xrightarrow{\text{Auto conv}}$ `SeqDNA.digest(myseq)`

How? The class instance knows what class it's in, and effectively "inherits" that class's methods.

Classes help more

Have separate classes for protein vs DNA sequences, each with appropriate methods

```
class SeqDNA:
  def digest(self): ...
  def rev_comp(self): ...
class SeqProt:
  def digest(self): ...
myseq = SeqDNA(file.readline())
frags = myseq.digest()
```

Better than the "module" version: yes, still the extra "constructor" step, but since objects know which class they're in, you *always* get the class-specific method

Change or Make a New One?

```
>>> mybirthday = Date(6,"Jul")
>>> mybirthday.printUS()

Jul 6

>>> party = mybirthday.add(4)
>>> party.printUS()

Jul 10

>>> mybirthday.printUS()

Jul 10                ←——————— Really?
```

`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) :
  newmon = self.mon
  newday = self.day + numdays
  while newday > daysinmonth[newmon] :
    newday = newday - daysinmonth[newmon]
    newmon = nextmonth(newmon)
  return Date(newday,newmon)
```

Make a new “Date” object

Using date.addnew()

```
>>> mybirthday = Date(6,"Jul")
>>> mybirthday.printUS()

Jul 6

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

Jul 10

>>> mybirthday.printUS()

Jul 6
```

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 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 (cont)

```
daysinmonth =(31,28,31,30,31,30,31,31,30,31,30,31)
monthlist = ["Jan", "Feb”, ..., "Dec"]
def nextmonth(thismonth):
  return (thismonth + 1) % 12
def month2str(monthnum):
  return monthlist[monthnum]
def str2month(monthstr):
  return monthlist[monthlist.index(monthstr)+1]
class Date:
  def __init__(self, day, monthstr) :
    self.day = day
    self.mon = str2month(monthstr)
  def print(self) :
    print month2str(self.mon), self.day
  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)
```