

Classes and Objects

Object Oriented Programming

Genome 559: Introduction to Statistical and
Computational Genomics

Elhanan Borenstein

A quick review

- A class defines variables' types:
 1. What kind of data is stored (members)
 2. What are the available functions (methods)
- An object is an **instance** of a class:
 - `string` is a class;
`my_str = "AGGCGT"` creates an object of the class `string`, called `my_str`.
- **Why classes:**
 - Bundle together data and operations on data
 - Allow special operations appropriate to data
 - Allow context-specific meaning for common operations
 - Help organize your code and facilitates modular design
 - The human factor

A *Date* class example

```
class Date:
```

```
    day = 0
```

```
    month = "None"
```

Define the class *Date*

Create and initialize
class members

```
def printUS(self):
```

```
    print self.month , "/" , self.day
```

```
def printUK(self):
```

```
    print self.day , "." , self.month
```

class functions
(methods)

```
mydate = Date()
```

```
mydate.day = 15
```

```
mydate.month = "Jan"
```

```
mydate.printUS()
```

```
Jan / 15
```

```
mydate.printUK()
```

```
15 . Jan
```

Special name "**self**" refers to the **current** object (no matter what the caller named it).

Access data
members of this
Date object

Access (call)
functions of this
Date object

Where did the
argument go?

An even better *Date* class

```
class Date:
```

```
    def __init__(self, day, month):
```

```
        self.day = day
```

```
        self.month = month
```

```
    def printUS(self):
```

```
        print self.mon , "/" , self.day
```

```
    def printUK(self):
```

```
        print self.day , "." ,
```

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

```
mydate.printUS()
```

```
Jan / 15
```

```
mydate2 = Date(22, "Nov")
```

```
mydate2.printUK()
```

```
22 . Nov
```

Special function "__init__" is called whenever a Date object instance is created. (class constructor)

It makes sure the object is properly initialized

Now, when "constructing" a new Date object, the caller MUST supply required data

Note the magical first arguments:
__init__ defined w/ 3 args; called w/ 2;
printUS defined w/ 1 arg; called w/ 0.

mydate is passed in both cases as 1st arg, so each function knows on which object it is to act

Class declarations and usage - Summary

- The **class** statement defines a new class

```
class <class_name>:  
    <statements>  
    <statements> ...
```

- Remember the colon and indentation
- You can include data members (variables) and functions as part of your class. These will be accessed using the dot (.) notation (e.g., mydate.day)
- The special name **self** means the **current** object
 - *self*.<something> refers to instance variables of the class
 - *self* is automatically passed to each method as a 1st argument

Second thoughts ...

- True, we now have a “print” function, but can we somehow make printing more intuitive?
- Specifically, why is “print” fine for numbers, strings, etc.

```
>>> my_str = "hello"  
>>> my_num = 5  
>>> print my_str, my_num  
"hello" 5
```

but funky for class instances?

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

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

A better way to print objects

- Actually, “print” doesn’t have special knowledge of how to print numbers, lists, etc.
- It just knows how to print strings, and relies on each class to have a `__str__()` method that returns a string representing the object.
- You can write your own, tailored `__str__()` method to give prettier/more useful results

A super *Date* class

```
class Date:
    def __init__(self, day, month):
        self.day = day
        self.month = month
    def __str__(self) :
        day_str = '%s' % self.day
        mon_str = self.month
        return mon_str + "-" + day_str

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

```
It's Sep-3. Happy Birthday!
```


Advanced topic: Allowing the plus sign

- Similarly, how come “+” works (but differently) for numbers and strings but not for dates?
- Yes, we could write a function `addDays(n)` :
`party = birthday.addDays(4)`
- But ... would be much more natural (and way cooler) to be able to write:
`party = birthday + 4`
- Can we do it?

Operator overloading

- Yes! Again, '+' isn't as smart as you thought; it calls class-specific "add" methods `__add__()` to do the real work.

Operator overloading

- Yes! Again, '+' isn't as smart as you thought; it calls class-specific "add" methods `__add__()` to do the real work.
- We can make new classes, like Date, behave like built-in ones
- Common operator overloading methods:
 - `__init__` # object creation
 - `__add__` # addition (+)
 - `__mul__` # multiplication (*)
 - `__sub__` # subtraction (-)
 - `__lt__` # less than (<)
 - `__str__` # printing
 - `__call__` # function calls
 - Many more...

Sample problem #1

- Write a Python class called **HL**, which will be used to include a horizontal line when you print.
- The class constructor should get a string *s* and an integer *l* and when printed it should print *l* repetitions of the string *s* (*and the necessary newline characters*).

```
>>> myHL1 = HL('=',20)
>>> print 'Title', myHL1 , 'The rest of the text'

Title
=====
The rest of the text

>>> myHL2 = HL('*-',5);
>>> print 'Title', myHL2 , 'The rest of the text'

Title
*-*-*-*-*
The rest of the text
```

Solution #1

```
class HL:
    def __init__(self, str, len):
        self.s = str
        self.l = len
    def __str__(self):
        line = self.s * self.l
        return '\n' + line + '\n'
```

Sample problem #2

- As you recall, python can print lists, as well as lists of lists. For example:

```
>>> my_LOL = [ [1,2,3] , [4,5,6] , [7,8,9] ]
>>> print my_LOL
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Since we often use lists of lists to represent matrices, we want a class that stores a list of lists, but that is doing a better job in printing it. Implement a class that gets a list of lists in the constructor, and then, when we print an object of this class, the list of lists will be printed as a matrix, as demonstrated below:

```
>>> my_matrix = NiceMatrix([ [1,2,3] , [4,5,6] , [7,8,9] ])
>>> print my_matrix
[1,2,3]
[4,5,6]
[7,8,9]
```

Sample problem #2 - hints

- Hint 1: Write a `__str__` function to convert the list of lists stored in the `NiceMatrix` object into a string.
- Hint 2: You can include newlines in the string to make it look like a matrix.
- Hint 3: the function `str(list)` converts a simple list into a string – try it.

Challenge Problem

- Overload the operator + for the Date class.
- Now try to overload the operator – for the Date class. Note that there are two fundamentally different ways to subtract dates:
 1. Subtract a given number of days from one date to get another date
 2. Subtract one date from another date to get the number of days between these two dates.

Can you implement both?

