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 slightly better *Date* class

```
class Date:
    day = 0
    month = "None"

    def printUS(self):
        print self.month , "/" , self.day
    def printUK(self):
        print self.day , "." , self.month

mydate = Date()
mydate.day = 15
mydate.month= "Jan"

mydate.printUS()
Jan / 15
mydate.printUK()
15 . Jan
```

class functions **(methods)**

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

Call method functions of this Date object

Where did the argument go? Answer to come .

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

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

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

Dreams do come true (sometimes)

- What do we have so far:
  - Date data are bundled together (sort of …)
  - Copying the whole thing at once is very handy
  - Printing is easy and provided as a service by the class
  - **User MUST provide data when generating a new Date object**

- Still on our wish-list:
  - ~~We still have to handle printing the various details~~
  - ~~Error checking - e.g., possible to forget to fill in the month~~
  - **No Date operations (add, subtract, etc.)**

Class declarations and usage - Summary

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

```
class <class_name>:
    <statements>
    <statements> …
```

  - Remember the colon and indentation

- 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

- The special name _ _**init**_ _ is the class constructor
  - Called whenever a new instance of the class is created
  - Every instance of the class will have all instance variables defined in the constructor
  - **Use it well!**

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!
```

Operator overloading

- 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
    ```

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

- Add a year data member to the *Date* class:

1. Allow the class constructor to get an additional argument denoting the year
2. If the year is not provided in the constructor, the class should assume it is 2018
   (Hint: remember the default value option in function definition)
3. When printing in US format, print all 4 digits of the year. When printing in UK format, print only the last 2 digits.
   (Hint: str(x) will convert an integer X into a string)

```
>>> mydate = Date(15,"Jan",1976)
>>> mydate.printUK()
15 . Jan . 76
>>> mydate = Date(21,"Feb")
>>> mydate.printUS()
Feb / 21 / 2018
```

Solution #1

```
class Date:
    def __init__(self, day, month, year=2018):
        self.day = day
        self.mon = month
        self.year = year

    def printUS(self):
        print self.mon , "/" , self.day , "/" , self.year

    def printUK(self):
        print self.day , "." , self.mon , "." , str(self.year)[2:]
```

Sample problem #2

- Change the Date class such that the month is represented as a number rather than as a string.
  (What did you have to do to make this change?)
- Add the function addMonths(n) to the class *Date.* This function should add *n* months to the current date. Make sure to correctly handle transitions across years.
  (Hint: the modulo operator, %, returns the remainder in division: 8 % 3→2)

```
>>> mydate = Date(22, 11, 1976)
>>> mydate.printUK()
22 . 11 . 76
>>> mydate.addMonths(1)
>>> mydate.printUK()
22 . 12 . 76
>>> mydate.addMonths(3)
>>> mydate.printUK()
22 . 3 . 77
>>> mydate.addMonths(25)
>>> mydate.printUK()
22 . 4 . 79
```

Solution #2

```
class Date:
    def __init__(self, day, month, year=2018):
        self.day = day
        self.mon = month
        self.year = year

    def printUS(self):
        print self.mon , "/" , self.day , "/" , self.year

    def printUK(self):
        print self.day , "." , self.mon , "." , str(self.year)[2:]

    def addMonths(self, n=1):
        new_mon = self.mon + n
        self.year += (new_mon-1) / 12
        self.mon = (new_mon-1) % 12 + 1
```

Sample problem #3

- 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 #3

```
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'
```

Challenge Problem

- Overload the operator + for the Date class.

- Now try to overload the operator – for the Data 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?