Classes and Objects
Object Oriented Programming

Genome 559: Introduction to Statistical and Computational Genomics

Elhanan Borenstein

A quick review

- Returning multiple values from a function

```
return [sum, prod]
```

- Pass-by-reference vs. pass-by-value
  - Python passes arguments by reference
  - Can be used (carefully) to edit arguments “in-place”
- Default Arguments

```
def printMulti(text, n=3):
```

- Keyword Arguments

```
runBlast(“my_fasta.txt”, matrix=“PAM40”)
```

A quick review – cont’

- **Modules:**
  - A module is a file containing a set of related functions
  - Python has numerous standard modules
- It is easy to create and use your own modules:
  - Just put your functions in a separate file
- To use a module, you first have to import it:

  ```
  import utils
  ```
- Use the dot notation:

  ```
  utils.makeDict()
  ```

utils.py

```
# This function makes a dictionary
def makeDict(fileName):
   myFile = open(fileName, "r")
   myDict = {}
   for line in myFile:
      fields = line.strip().split("\t")
      myDict[fields[0]] = float(fields[1])
   myFile.close()
   return myDict

# This function reads a 2D matrix
def makeMatrix(fileName):
    < ... >
```

my_prog.py

```
import utils
import sys

Dict1 = utils.makeDict(sys.argv[1])
Dict2 = utils.makeDict(sys.argv[2])

Mtrx = utils.makeMatrix(“blsm.txt”)

…
```

Classes and Objects

What is a class?

What is an object?

Why do we need them?

How do we use them?

How do we define new classes?

Classes

- A class defines the “type” of variables:
  1. **What kind of data is stored**
  2. **What are the available functions**

- Python includes (and you used) several built-in classes:
  - String
  - Dictionary
  - Number

What kind of data do these “classes” store?

What kind of functions do they provide?

- Modules may provide additional classes …

Objects

- 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`.
- You can only have one class named "string"
- But .. You can have many string objects
  - `my_str = "AGGCGT"`
  - `your_str = "Jim Thomas"`

Using objects

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

```
>>> my_str = "ATCCGCG“
>>> your_str = “Jim Thomas”
>>> print my_str.find(“h")
6
>>> print your_str.count(“m")
2
```

Objects

Object methods

This is useful …

But … why stop with built-in classes?

Wouldn’t it be great if we could have many more classes?

Genome

Person

Student

Gene

Date

Book

GO Function

Organism

DNA

PhyloTree

Course

Chair

This approach is known as

Object Oriented Programming (OOP)

(P.S. not supported in all programming languages)

Why classes?

- **Bundle together data and operations on data**
  - Keep related data together
  - Keep functions connected to the data they work on
- **Allow special operations appropriate to data**
  - “count” or “split” on a string;
  - “square root” on numbers
- **Allow context-specific meaning for common operations**
  - `x = ‘a’; x*4` vs. `x = 42; x*4`
- **Help organize your code and facilitates modular design**
  - Large programs aren’t just small programs on steroids

Why classes? The more profound answer

Why functions?

Technical factor

Allow to **reuse** your code
Help **simplify** & **organize** your code
Help to avoid **duplication** of code

Human factor

Human approach to problem solving:
Divide the task into smaller tasks
Hierarchical and **modular** solution

Why classes?

Technical factor

Bundle together data and operations
Allow context-specific operations
Help to **organize** your code

Human factor

Human representation of the world:
Classify objects into categories
Each category/class is associated
with unique data/functions

Defining our first new class

- As an example, let's build a ***Date*** class

Defining our first new class

- As an example, let's build a ***Date*** class
- The "dream" *Date* class should …
  - **store** day, month, and year
  - provide functions that **print** the date in different formats
  - provide functions to **add** or **subtract** a number of days from the date
  - provide a way to **find** the difference (in days) between 2 dates
  - **check** for errors:
    - Setting month to "Jamuary"
    - Copying the month without the associated day
    - 14 days after Feb 18 probably shouldn't be Feb 32

Data (members)

Functions (methods)

A very, very simple *Date* class

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

Define the class *Date*

Create and initialize class members (not mandatory!!!)

Note the Format

A very, very simple *Date* class

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

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

print mydate.day, mydate.month
15 Jan

yourdate = mydate
```

Define the class *Date*

Create and initialize class members (not mandatory!!!)

Note the Format

Create a new Date object (instance of the class Date)

Access and change object members

Print object members

Copy the object into another object

Hmmm… a good start

- What do we have so far:
  - **Date data are bundled together (sort of …)**
  - **Copying the whole thing at once is very handy**

- 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.)**

A slightly better *Date* class

```
mydate = Date()
mydate.day = 15
mydate.month= "Jan"
mydate.printUS()
Jan / 15
mydate.printUK()
15 . Jan
```

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
```

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

class functions **(methods)**

Call method functions of this Date object

Where did the argument go? Answer to come .

We’re getting there …

- 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**

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

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

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> …
```

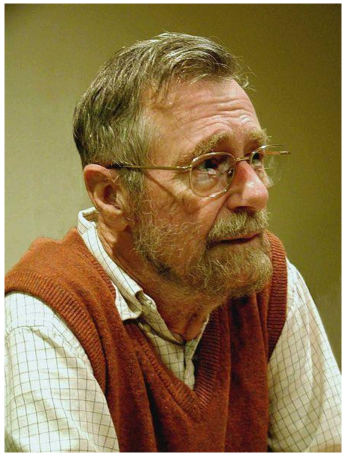
  - Remember the colon and indentation

- 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!**

Code like a pro …

TIP OF THE DAY


Edsger Wybe Dijkstra
1930 –2002

"Testing shows the presence,
not the absence of bugs."

- Code running ≠ code is correct or bug-free
- Be much more concerned about the bugs you don't see than the ones you do!!
- **Especially true in bioinformatics, high-throughput data analysis, and simulations**

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
```

Challenge Problem

1. Add the function addDays(n) to the class *Date.* This function should add n days to the current date. Make sure to correctly handle transitions across months AND across years (when necessary). Take into account the different number of days in each month.

2. Revise the Date class such that it will again work with the month’s name (rather than its number), while preserving the functionality of the addMonths and addDays functions.