Objects

Vocabulary

- **Class**: A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable**: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables aren't used as frequently as instance variables are.
- **Instance variable**: A variable that is defined inside a method and belongs only to the current instance of a class.
- **Method**: A special kind of function that is defined in a class definition.
- **Instance**: An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation**: The creation of an instance of a class.

Class definition

- Creating a new class:

object is the root class of all python classes

```
class MyClass(object):
      pass
```

- Instantiation:

```
class MyClass(ExistingClass):
      pass
```

```
class MyClass(object):
      pass

my_instance = MyClass()
```

Methods & constructors

- Defining and calling a method:

```
class MyClass(object):

      def my_method(self, a, b):
          return a + b

my_instance = MyClass()

print(my_instance.my_method(2, 3))              # will display 5

# or equivalently

print(MyClass.my_method(my_instance, 2, 3))    # will display 5
```

reserved words

```
class MyClass(object):

      def __init__(self, ...):
          ...

      def __del__(self, ...):
          ...
```

Class and instance variables

- Defining class and instance variables:

```
class MyClass(object):

   my_class_variable1 = 0
   my_class_variable2 = 'foo'

   def __init__(self, x):
          self.my_instance_variable1 = x
          slef.my_instance_variable2 = 'bar'

my_instance1 = MyClass(100)
my_instance2 = MyClass(200)

print(my_instance1.__class__.my_class_variable1)      # returns 0
my_instance1.__class__.my_class_variable1 = 999
print(my_instance2.__class__.my_class_variable1)      # returns 999

print(my_instance1.my_instance_variable1)          # returns 100
print(my_instance2.my_instance_variable2)          # returns 200
```

Inheriting

```
class MyClass1(object):

  def __init__(self, foo):
      self.foo = foo

  def print(self):
      print(self.foo)

  def hello(self):
      print('hello %s' % self.foo)

class MyClass2(MyClass1):

  def __init__(self, foo, bar):
      MyClass1.__init__(self, foo)     # call the parent constructor
      self.bar = bar

  def print(self):                     # the method is overloaded
      MyClass1.print(self)             # call the parent method
      print(self.bar)
```

Inheriting – alternative syntax

```
class MyClass1(object):

  def __init__(self, foo):
      self.foo = foo

  def print(self):
      print(self.foo)

  def hello(self):
      print('hello %s' % self.foo)

class MyClass2(MyClass1):

  def __init__(self, foo, bar):
      super(MyClass1, self).__init__(foo) # call the parent constructor
      self.bar = bar

  def print(self):                          # the method is overrideded
      super(MyClass1, self).print()    # call the parent method
      print(self.bar)
```

Static and class methods

- Defining a static or class method:

```
class Math(object):

      _pi = 3.14

      @staticmethod          # both instance and class not visible
      def add(x, y):
          return x + y

      @classmethod           # only class visible
      def pi(cls):
          return cls._pi

      def pi_in_a_instance_method(self):
          return self.__class__._pi
```

Properties

- Defining an property:

```
class MyClass(object):

      def __init__(self):
          self._foo = None

      @property
      def foo(self):
          return self._foo

      @rawdata.setter
      def foo(self, value):
          if value < 0:
              raise MyException(“foo must be > 0”)
          self._foo = value

my_instance = MyClass()
print(my_instance.foo)          # returns None
my_instance.foo = 256
print(my_instance.foo)          # returns 256
my_instance.foo = -1            # raise MyException
```

Operator overloading

- Just implement these methods:
  - __lt__(self, other)
  - __le__(self, other)
  - __eq__(self, other)
  - __ne__(self, other)
  - __gt__(self, other)
  - __ge__(self, other)
  - __cmp__(self, other)

```
L = [0, 1, 2, 3]
len(L)          # returns 4
```

[]

- __len__(self)
- __getitem__(self, idx)
- __setitem__(self, idx, val)
- __delitem__(self, idx)
- __contains__(self, item)
- Etc...

- See:
  - http://docs.python.org/2/reference/datamodel.html

Operator overloading

- __add__(self, other)
- __sub__(self, other)
- __mul__(self, other)

- __neg__(self)
- __pos__(self)
- __abs__(self)
- __invert__(self)

- __repr__(self) # returns a string, goal is to be unambiguous
- __str__(self) # returns a string, goal is to be readable

Modules

Importing class/functions/etc... and aliases

- Importing a class/function/etc... and makes it directly visible without module prefix:

```
from my_module import MyClass [, MyOtherClass, my_function, etc...]

my_instance = MyClass()
```

- Importing all the class/function/etc... and make them directly visible without module prefix:

```
from my_module import *
```

- Importing sub-module and make them directly visible:

```
from my_module import sub_module
```

- An import can be aliased:

```
import my_module as my_alias1

from foo import bar as my_alias2
```

Creating a module

- Creating a module:
  - A module is just a directory with a file __init__.py

```
$ mkdir my_module

$ touch my_module/__init__.py
```

- All the definitions in __init__.py become available.
- A module can contain other files in addition to __init__.py

```
$ touch my_module/foo.py

$ touch my_module/bar.py
```

- A module can contain sub-modules, etc…

```
import my_module
import my_module.foo
import my_module.bar
```

Example

my_module/__init__.py

```
def my_function():
      print('Hello World !')
```

my_module/foo.py

```
PI = 3.14
```

my_module/my_submodule/__init__.py

```
E = 2.71
```

my_program.py

```
import my_module
import my_module.foo
from my_module.my_submodule import *

my_module.my_function()                 # returns 'Hello World !'
print(my_module.foo.PI)                 # returns 3.14
print(E)                                # returns 2.71
```

PYTHONPATH

- Modules must be:
  - In the same directory than the application
  - In a directory registered in the environment variable PYTHONPATH:

```
$ export PYTHONPATH=/path/of/my/module1:/path/of/my/module2
```

Exercise (1h)

- Try to write a vector class (arbitrary dimension)
  - Create the module “vector”
  - Create the class “Vector”
  - Write the constructor [def __init__(self, dim)]
  - Overload operators
  - Display your vectors

Importing a module

- A module is a collection of classes, functions, etc...
- Modules can be shared between applications.
- Importing a module:

```
import my_module
```

- A module can have sub-modules:

```
import my_module.sub_module
```

```
import my_module

my_instance = my_module.MyClass()
```

```
import os
import my_module, my_module.sub_module
```

The Python standard library
&
useful libraries for physicists

Miscellaneous operating system interfaces

- import os
  - http://docs.python.org/2/library/os.html
  - Contains lot of functions of the POSIX standard.
  - Files and directories:
    - mkdir, rmdir, remove, chmod, stat
  - Environment variables:
    - environ[], getenv, putenv
  - Shell command and program execution:
  - system, popen
  - Etc...

Common pathname manipulations

- import os.ptah
  - http://docs.python.org/2/library/os.path.html
  - Common functions:
    - basename, dirname, abspath, join
    - exists, is_dir, is_file, is_link
    - Etc...

System-specific parameters and functions

- import sys
  - Common variables:
    - argv[], platform
    - stdin, stdout, stderr (see builtin function open)
    - Etc...
  - Common functions:
    - exit
    - path (modifies PYTHONPATH at runtime)
    - Etc...

Common libraries

- import glob
  - File Wildcards

```
>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

- import re
  - Regular expression for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions.
- import math
  - Gives access to the underlying C library functions for floating point math.
- import random
  - random generators

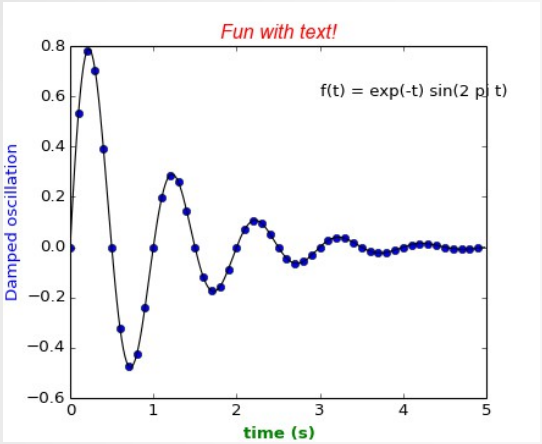
Common libraries

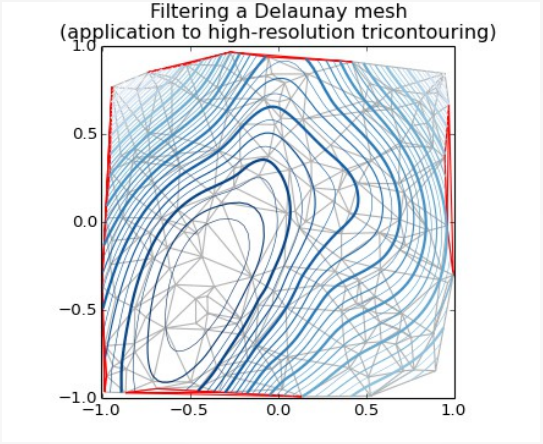
- import urllib2, smtplib, ...
  - There are a number of modules for accessing the internet and processing internet protocols.
- import time, datetime
  - Supplies classes for manipulating dates and times in both simple and complex ways.
- import zlib
  - Common data archiving and compression formats are directly supported by modules including: zlib, gzip, bz2, zipfile and tarfile.
- See:
  - http://docs.python.org/2/index.html

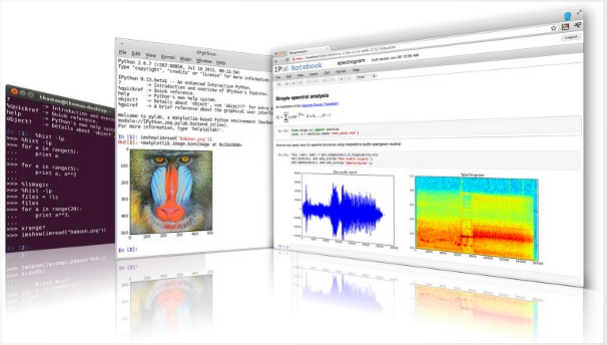
For physicists

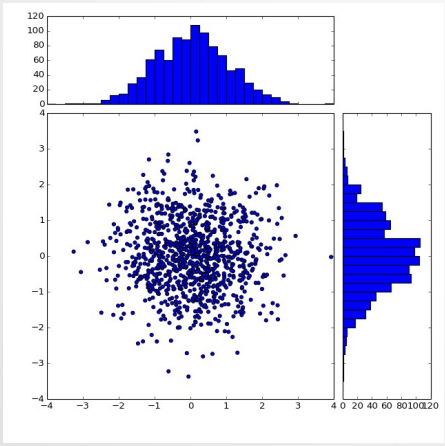
- pyROOT
  - The CERN's data analysis framework
  - http://root.cern.ch
- ROOT allows to:
  - Save and load data
  - Process data
  - Show results
  - Develop interactive or built applications

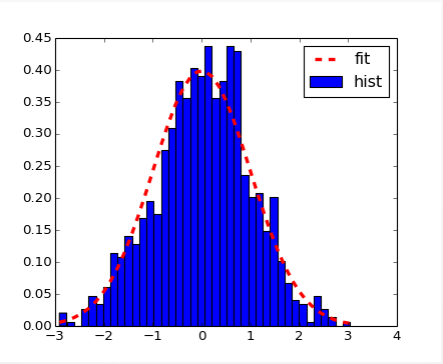
SciPy


Fun with text!
f(t) = exp(-t) sin(2 p t)
Damped oscillation
time (s)


Filtering a Delaunay mesh
(application to high-resolution tricontouring)






fit
hist

The end