Computational Physics

Object-Oriented Programing

Prof. Paul Eugenio
Department of Physics
Florida State University
26 Feb 2019

http://hadron.physics.fsu.edu/~eugenio/comphy/

Announcements

- **Mid-Term 1**
  - **Will hand out next Tuesday, due by Friday**

Poor programming practices

```
def stateEquation(theta):
    """ **** """"
    value = np.tan(theta) - np.sin(theta) - m*g/(2.0*k*L)
    return value



# main program
#

# create data points with radians but plot using degrees
theta = np.linspace(0,np.pi/2,1000)

m = 5
L = 0.3
k = 1000
g = 9.81
sPoints = s(theta)
plt.plot(theta,sPoints)
```

values rely on later and non-local declarations

better programming practice

```
def stateEquation(theta):
    """ [function documentation here] """

    m = 5           # [kg]    mass of object
    L = 0.3         # [m]     half distance between support
    k = 1000        # [N/m]   spring constant
    g = 9.81        # [N/kg]  acceleration due to gravity

    value = np.tan(theta) - np.sin(theta) - m*g/(2.0*k*L)
    return value

# main program
#

# create data points with degrees but pass data as radians
theta = np.linspace(0, 89, 90)

sPoints = s( np.radians( theta ) )
plt.plot( theta, sPoints )
plt.show()
```

better to have variables in local scope

Even better programming practice

```
class stateParameters:
    """ parameter constants for the two springs equation """
    m = 5           # [kg]     mass of object
    L = 0.3         # [m]      half distance between support
    k = 1000        # [N/m]    spring constant
    g = 9.81        # [N/kg]   acceleration due to gravity

def stateEquation(theta, par):
    """ plotting function: tan(x) - sin(x) - mg/(2kL) """
    value = np.tan(theta) - np.sin(theta) - par.m*par.g/(2.0*par.k*par.L)
    return value

# ************main program**********
# create data points with degrees but pass data as radians to function
theta = np.linspace(0, 89, 90)

statePoints = stateEquation( np.radians(theta), stateParameters )
plt.plot(theta, statePoints)
plt.show()
```

better to have variables wrapped in an object container

Procedural Programing

```
#! /usr/bin/env python
"""
 *  factorial
 *  PHY4151
 *
 *  Created by Paul Eugenio
 *
 *  This program was created to illustrate the basic
 *  components of procedural-type programming
 *
 """
from __future__ import division, print_function
```

Post processing statements

```
#
# main program

number = int(raw_input("Enter value for factorial "))
n, result = number, 1

# calculate factorial of n
while n > 0:
    result = n * result
    n = n - 1


print("The factorial of the number", number, "is", result)
```

The main body of the program

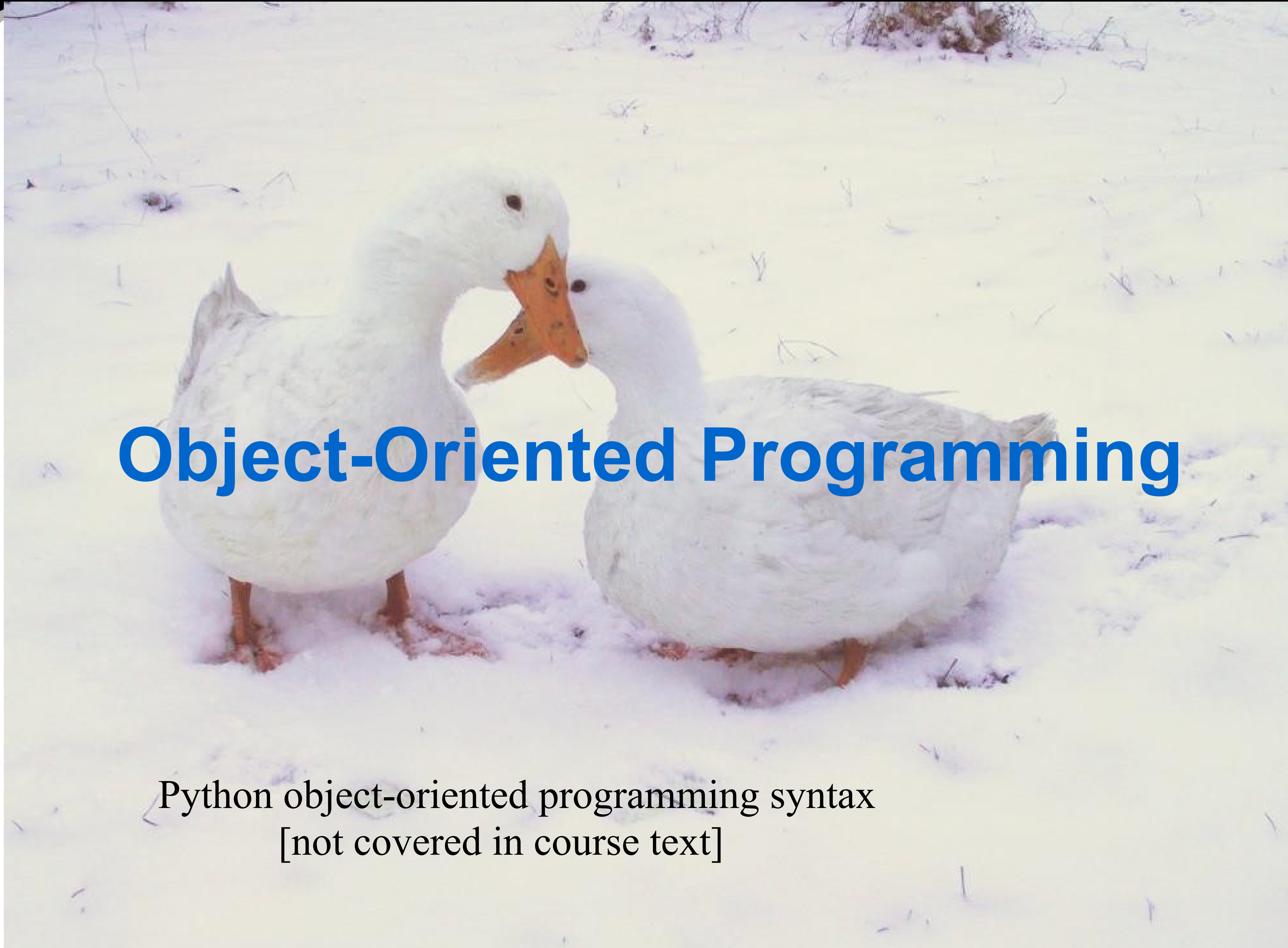
A Better Procedural Program

```
#! /usr/bin/env python
"""
 *  factorial
 *  PHY4151
 *
 *  Created by Paul Eugenio
 *
 *  This program was created to illustrate the basic
 *  components of procedural-type programming
 *
 """
from __future__ import division, print_function



def factorial(aNumber):
    result = 1
    if aNumber > 0:
        result = aNumber* factorial(aNumber - 1)



#
# main program

number = int(raw_input("Enter value for factorial "))
result = factorial( number )

print("The factorial of the number", number, "is", result)
```

Post processing statements

declaration of the factorial() function

The main body of the program



Object-Oriented Programming

Python object-oriented programming syntax
[not covered in course text]

Defining Object via a Class Declaration

- A class is a template for containing
  - Data/objects/variables
    - often hidden from user
  - Member functions/Methods
    - Constructors, Destructors, Getters, & Setters
  - Scope
    - keeping it all local

Python has class

Python is an object-oriented programming language

```
# Defining a class
class class_name:
    [statement 1]
    [statement 2]
    [statement 3]
    [etc.]
```

A "class" defines an object as a container for data (variables) and methods (functions)

Example: A Circle

Class Definition of Simple Circle

```
class Circle:
    def __init__(self, aRadius=1):
        self.radius = aRadius

    def area(self):
        return np.pi*self.radius**2

    def circumference(self):
        return 2*np.pi*self.radius

    def __add__(self, other):
        return Circle(self.radius + other.radius)

    def print(self):
        print("Hello, I am a circle")
        print("my radius is",self.radius)
        print("My area is", self.area())
        print("My circumference is", self.circumference())
        print()
```

Special Methods

constructor

```
def __init__(self, aRadius=1):
    """ Default constructor sets unit radius """
    self.radius = aRadius
```

operator overloading

```
def __add__(self, other):
    """
     Add circles where C = A + B equals a circle
     with radius equal to the sum of the two radii
    """
    return Circle(self.radius + other.radius)
```

```
A = Circle(3)
B = Circle()
C = A + B       # C is a circle of radius 4
 ...
```

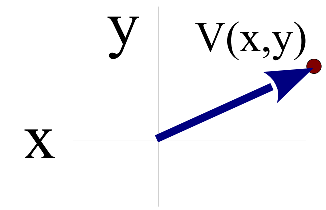
Utilizing a Circle Object

```
# main

A = Circle()
B = Circle(3)

A.print()
B.print()

C = A + B
print("\n Area of circle C is", C.area())

(A+C).print()
```

```
hpc-login 213%  circle.py
Hello, I am a circle
my radius is 1
My area is 3.14159265359
My circumference is 6.28318530718

Hello, I am a circle
my radius is 3
My area is 28.2743338823
My circumference is 18.8495559215

 Area of circle C is 50.2654824574

Hello, I am a circle
my radius is 5
My area is 78.5398163397
My circumference is 31.4159265359
```

OOP Promotes Bottom-up Programming

2D Vector Procedural Approach




```
""""" Simple  2D vector  """""

# data list for x,y
Vector = [0.0, 0.0]

# modify data value
Vector[0] = 1.0                # x value
Vector[1] = 2.0                # y value

def addVect2D(V1, V2):
    Vsum = []
    Vsum += V1[0] + V2[0]       # add x's
    Vsum += V1[1] + V2[1]       # add y's
    return Vsum

def multVect2D(V1, V2):
    return sqrt(V1[0]*V2[0] + V1[1]*V2[1])

def printVect2D( V ):
    print(“vector(x,y) is (”, V[0], “,”,V[1],“)”)
```

```
# main

P1 = [1.0, 1.0]
P2 = [2.0,2.0]

P = addVect2D( P1, P2 )
Q = multVect2D( P, P2 )

printVect2D(P1)
printVect2D(P2)
printVect2D(P)
printVect2D(Q)
```

OOP Example: Vector2D

Member functions
& variable data

```
class Vector2D:
  """ Vector2D Object declaration """

  def __init__(self, aX=0.0, aY=0.0):
    """ Default constructor"""
     self.x, self.y  =  aX, aY

  def r(self) :
    """ Polar coordinate magnitude """
     return numpy.sqrt( self.x()**2 + self.y()**2 )

  def theta(self) :
     """ Polar coordinate angle"""
     return numpy.atan2( self.y(), self.x() )

  def __add__(self, other):
    """ Operator overload for vector addition """
     return Vector2D(self.x() + other.x(), self.y() + other.y())

  def __mul__(self, other):
    """ Operator overload multiplication for dot product """
     return numpy.sqrt( self.x() * other.x()  +  self.y() * other.y()   )

  def print(self):
    """ Vector2D print vector information """
     print(“vector(x,y) is (”, self.x(), “,”,  self.y(),“)”)
```

OOP Example: Vector2D

```
# main

P1 = Vector2D()
P2 = Vector2D(1.0,1.0)
P3 = Vector2D(2.0,2.0)

P1.print()
P2.print()
P3.print()
P1 = P2 + P3
P1.print()
```

```
 hpc-login 55% myvectors.py
vector(x,y) is (0,0)
vector(x,y) is (1,1)
vector(x,y) is (2,2)
vector(x,y) is (3,3)
 hpc-login 56%
```

Data Encapsulation

```
class Vector2D:
   """ Vector2D Object declaration """

   def __init__(self, aX=0.0, aY=0.0):
      self.__x = aX
      self.__y = aY


   def x(self):
      return self.__x

   def y(self) :
      return self.__x

   def r(self) :
      return math.sqrt( self.x()**2 + self.y()**2 )

def theta(self) :
      return math.atan2( self.y(), self.x() )
```

using two "_" makes the data private

this is data encapsulation

```
def __add__(self, other):
  """ ** """
   return Vector2D(self.x() + other.x(), \
             self.y() + other.y())

def __mul__(self, other):
  """ ** """
   return sqrt( self.x() * other.x()  +      \
                self.y() * other.y()   )

def print(self):
  """ ** """
   print("vector(x,y) is (", self.x(), ",",  \
          self.y(),")")
```

** for brevity docstrings have been omitted

Data Encapsulation

If we want to represent the Vector2D data by (r, θ) rather than by (x, y) then very little changes are need in the class definition and **NO** changes will be needed in the user code.

Vector2D with encapsulated polar data

```
class Vector2D:
    """ Vector2D Object declaration """

    def __init__(self, aX=0.0, aY=0.0):
        self.setR( numpy.sqrt(aX**2 + aY**2) )
        self.setTheta( numpy.arctan2(aY, aX) )

    def setR(self, aR):
        self.__r = aR

    def setTheta(self, aTheta):
        self.__theta = aTheta

    def x(self):
        return self.r() * numpy.cos( self.theta() )

    def y(self) :
        return  self.r() * numpy.sin( self.theta() )

    def r(self) :
        return self.__r

    def theta(self) :
        return self.__theta
```

using two "_" makes the data private

this is data encapsulation

```
    def __add__(self, other):
        """ ** """
        return Vector2D(self.x() + other.x(), \
                        self.y() + other.y())

    def __mul__(self, other):
        """ ** """
        return sqrt( self.x() * other.x()  +      \
                     self.y() * other.y()   )

    def print(self):
        """ ** """
        print("vector(x,y) is (", self.x(), ",",  \
              self.y(),")")
```

** for brevity docstrings have been omitted

program operates as before

```
# main

P1 = Vector2D()
P2 = Vector2D(1.0,1.0)
P3 = Vector2D(2.0,2.0)

P1.print()
P2.print()
P3.print()
P1 = P2 + P3
P1.print()
```

```
 hpc-login 55% myvectors.py
vector(x,y) is (0,0)
vector(x,y) is (1,1)
vector(x,y) is (2,2)
vector(x,y) is (3,3)
 hpc-login 56%
```

Extending Objects: Vector3D

```
class Vector3D:
    def __init__(self, aX, aY, aZ):
        self.__V2D = Vector2D(aX, aY)
        self.__z = aZ




    def __add__(self, other):
            ...
```

Vector3D *<u>has a</u>* Vector2D

Inheritance: Vector3D

```
class Vector3D(Vector2D):
    def __init__(self, aX, aY, aZ):
        Vector2D.__init__(self, aX, aY)
        self.__z = aZ




    def z(self):
        return self.__z

    def __add__(self, other):
            ...
```

Point3D *is a* Point2D

Other Special Methods

Construction	Meaning
a.__init__(self, args)	constructor: a = A(args)
a.__del__(self)	destructor: del a
a.__call__(self, args)	call as function: a(args)
a.__str__(self)	pretty print: print a, str(a)
a.__add__(self, b)	a + b
a.__sub__(self, b)	a - b
a.__mul__(self, b)	a*b
a.__truediv__(self, b)	a/b
a.__pow__(self, b)	a**p
a.__lt__(self, b)	a < b
a.__gt__(self, b)	a > b
a.__le__(self, b)	a <= b
a.__ge__(self, b)	a => b
a.__eq__(self, b)	a == b
a.__ne__(self, b)	a != b
a.__bool__(self)	boolean expression, as in if a:
a.__len__(self)	length of a: len(a)
a.__abs__(self)	abs(a)
…	

Let's get working