OBJECT ORIENTED PROGRAMMING IN PYTHON

Eric Prebys

Definition of Object-Oriented Programming

- From the web:
  - Object-oriented programming (OOP) is a method of programming in which programs are structured as collections of interacting objects, each representing an instance of a class that defines its **data** and the **operations** that can be performed on that data.
- Lexicon:
  - “class”: This is a template for the object we want to create
  - “data”: Data associated with the class
  - “method”: This is a routine defined within the class
  - “instance”: This is an object that is created based on a defined class. I can create many objects from one class definition and organize them in any way I like, including as part of other classes.
  - ”inheritance”: Creating a new class based on one or more existing classes, generally to increase its functionality.

Simple Example: 2D Point class

- "self" is a pointer to the current instance.
- It's automatically added at the beginning when you call a method.
- You use it to point to the data associate with the current instance

```
# Define a class called Point
class Point:
    # text in triple quotes will be accessed with __doc__
    """Simple class to demonstrate OO programming in Python"""

    # The __init__ method will be called when the class is created
    # Equivalent to the C++ "constructor".
    # Can't be overloaded, but can have defaults
    def __init__(self,x=0.,y=0.):  # "self" is a pointer to a particular object
        # Load these into internal variables, which will be stored along with the object
        self.x = x
        self.y = y

    # Define a "radius" method
    def rad(self):
        """Method to determine the radius"""
        r = math.sqrt(self.x**2+self.y**2)
        return r

    # The __str__ method is what get's called when you "print" the object
    def __str__(self):
       return f"Point: x={self.x}, y={self.y}"
```

(go to "Lecture 6 – OO.ipynb")

Inheritance

- We can create a new class based on an old class. In this case, we create a Point3D class

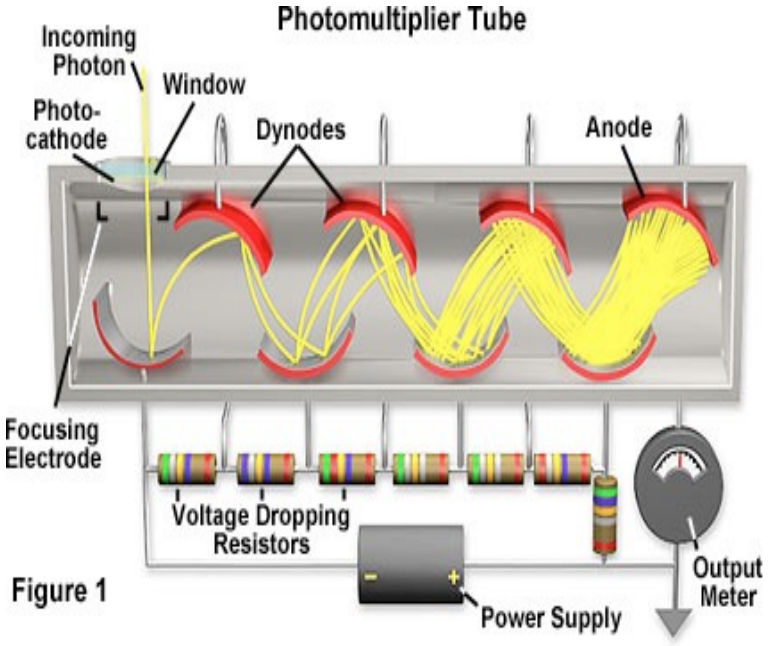
```
# A better way to do this would be to create a new class derived from the old one.
# This is called inheritance
class Point3D(Point):     # This tells me that Point3D is based on Point
    """3D Point"""
    def __init__(self,x=0.,y=0.,z=0.):
        # Call the orignal initialization with x and y
        super().__init__(x,y)
        # add z
        self.z = z

    # We'll override the original definition of rad
    def rad(self):
        """Calculate 3D radius"""
        r = math.sqrt(self.x**2+self.y**2+self.z**2)
        return(r)

    # New __str__ method uses old method
    def __str__(self):
        # Build a new string from the old string.
        s = super().__str__()
        s += f", z={self.z}"
        return s
```

Digression: Photomultiplier Tubes

- Photomultiplier tubes are one of the oldest types of particle detectors.
- They use a series of “dynodes” to amplify the photoelectron ejected by a single photon until it’s a measurable signal.



- If I want precision, I need to calibrate each PMT and correct the ”raw” data as

$$\text{calibrated\_value} = \text{offset} + \text{gain} * \text{raw\_value}$$

Our Example Project

- We're going to define a "PMT" class, which stores the following for each PMT
  - Unique ID number
  - Location:
  - Offset and Gain
- It will include methods to
  - Load raw readout data
  - Check to see of readout data is there
  - Convert raw data to calibrated data
  - Clear the readout data once I'm done.
- We're then going to use the class to generate a 10x10 array of PMTs and do a simulated readout.

```python
#
# Define a class for a photomultiplier tube.
#
class PMT:
    """Class defining a photomultiplier tube (PMT)"""
    def __init__(self, ID, position, offset=0.0, gain=1.0, threshold=100.):
        self.ID = ID                # Unique ID number
        self.position = position    # Position [x,y,z]
        self.offset = offset        # Offset
        self.gain = gain            # gain
        self._raw = -1.             # raw value (-1 means "no data")

    # Dummy routine.  This takes the place of raw data readout
    def read_raw(self, raw_value):
        """Simulate reading a raw value from the PMT."""
        self._raw = raw_value

    # This returns the calibrated data
    def get_calibrated(self):
        """Return the calibrated sensor reading."""
        return self.gain * self._raw + self.offset

    # This will tell if the signal is above threshold
    def is_hit(self):
        """Return the hit status"""
        return (self._raw>=0.)

    # This will clear the data for the net readout
    def clear(self):
        """Clears raw data"""
        self._raw = -1.

    # The __str__ method is
    def __str__(self):
        return f"Sensor(ID={self.ID}, position=[{self.position[0]:.2f},{self.position[1]:.2f},{self.position[2]:.2f}], gain={
```

(go to ”Lecture 6 – PMT Example.ipynb”)