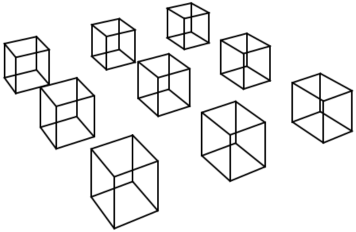
Object Oriented Programming in Python



Using Sage

Here's how you create a 3×3 grid of cubes in Sage

```
n=3
g = None
for x in range(0,2*n,2):
    for y in range(0,2*n,2):
        g = cube((x,y,0), color='blue') + g
g.show(aspect_ratio=[1,1,1])
```

- Sage has lots of functions ready imported such as `cube`
- `g` is an example of an object. It stores data and has functions such as `show`.
- We'll pretend in this lecture that 3d graphics classes don't exist.

Numpy

```
v1 = [0,1,2]
v2 = [3,4,5]
print( v1 + v2 )
```

Prints `[0,1,2,3,4,5]`. The fix is

```
import numpy
v1 = numpy.array([0,1,2])
v2 = numpy.array([3,4,5])
print( v1 + v2 )
```

Prints `[3 5 7]`.

Jupyter notebooks and matplotlib

We can create a new Jupyter notebook which is a bit like a Mathematica worksheet.

```
%matplotlib inline

import matplotlib.pyplot as plt

x = numpy.array( range(0,100))/10
y = numpy.vectorize( lambda x : x**2 )(x)

plt.plot(x,y,'b-.')
plt.show()
```

- Use `numpy.vectorize` to take a function that operates on scalars and turn it into one that operates on vectors.
- `matplotlib` is full of tools to generate plots

numpy and efficient numerics

- Use dot to multiply matrices (or in general tensor product).
- Use * for element wise multiplication.
- Python is rather slow. numpy calls low-level routines and so is very fast.
- For efficient use of numpy try to write your computations as manipulations of large arrays. This is called “array programming”.
- In array programming, you often want to perform component by component calculations. For example, element wise multiplication of matrices. This is what *

Example: The Mandelbrot set

```
import numpy
import matplotlib.pyplot as plt
numpy.seterr('ignore') # ignore overflow

n = 1000
nSteps = 100

xMin, xMax, yMin, yMax=(-2,0.8,-1.2,1.2)
rValues = numpy.linspace(xMin,xMax,n)
iValues = numpy.linspace(yMin,yMax,n)*1j
R,I = numpy.meshgrid(rValues,iValues)
c = R + I
z = numpy.zeros((n,n))

for i in range(0,nSteps):
    z = z**2 + c
mandelbrot = numpy.logical_or( numpy.isnan(z), abs(z)>=2)

plt.matshow(mandelbrot, extent=[xMin, xMax, yMin, yMax])
plt.show()
```

Objects and classes

- A class is a type of object. For example `Rectangle` is a class, a specific `Rectangle` is an object.
- An object has associated data and functions.
- The data available and the functions available are determined by the class.
- Example: a class `Circle` should say that all *instances* of this class have a center and a radius. All circles should also have functions area and circumference that return their area and circumference.

The circle class

```python
class Circle2D:
    """Represents a circle"""

    def __init__(self):
        self.center = numpy.array([0,0])
        self.r = 1

    def circumference(self):
        return self.r * 2* math.pi

    def area(self):
        return (self.r ** 2 ) *math.pi
```

```python
c = Circle2D();
c.r = 2
print( c.circumference())
```

Writing classes

- The first parameter of each function is called `self` and represents the object on which the method has been called.
- You use a `.` to access fields and functions of an object.
- There is a special function called `__init__` which initializes the object. The variables of the object are defined by assigning values in the `__init__` function.

A line class

```
class Line2D:
    """Represents a single line segment joining two points"""

    def __init__(self, p1=numpy.array([0,0]), p2=numpy.array([1,1])):
        self.p1 = p1
        self.p2 = p2

    def draw(self):
        xVals = [self.p1[0],self.p2[0]]
        yVals = [self.p1[1],self.p2[1]]
        lines = plt.plot(xVals, yVals, 'k-')
        plt.setp(lines, linewidth=2)

    def length(self):
        p1 = self.p1;
        p2 = self.p2;
        return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)

    def __str__(self):
        return 'A line from {} to {}'.format(self.p1, self.p2)
```

- Our line now knows how to draw itself using `pyplot`.
- It is drawn as black and solid `k-`.
- We can set additional properties of the lines in our plot using `setp`.
- To make lines easy to work with we add a `__str__` function.

Examples of usage

```
l1 = Line2D()
print( str(l1) )
print('The length of the line is {}'.format(l1.length()))

l2 = Line2D()
l2.p1 = numpy.array([1,0])
l2.p2 = numpy.array([1,0])
print( str(l2))
print('The length of the line is {}'.format(l2.length()))

l1.draw()
plt.show()
```

Enhancing the circle class

```
class Circle2D:
    # ... code from earlier
    def __str__(self):
        return 'A circle centre {} and radius {}'.format(self.center, self.r)

    def length(self):
        return self.circumference()

    def _pointAtAngle(self, theta):
        return [self.center[0] + math.cos(theta)*self.r,
                self.center[1] + math.sin(theta)*self.r]

    def draw(self):
        n = 100;
        theta = 0;
        last_point = self._pointAtAngle(theta)
        step = 2*math.pi/n;
        while theta< 2*math.pi:
            theta += step
            point = self._pointAtAngle(theta)
            line = Line2D()
            line.p1 = last_point
            line.p2 = point
            line.draw()
            last_point = point
```

A Graphics2D class

```
class Graphics2D:
    """An object that groups together shapes that can then be displayed using matplotlib"""

    def __init__(self):
        self.elements = []

    def add(self, element):
        self.elements.append(element)

    def show(self, large=False):
        plt.axis('off')
        plt.axes().set_aspect('equal', 'datalim')
        for element in self.elements:
            element.draw()
        if large:
            # get current figure and set its size
            fig = plt.gcf()
            fig.set_size_inches(10, 10)
        plt.show()

    def length(self):
        ret = 0.0
        for element in self.elements:
            ret+=element.length()
        return ret
```

Discussion

- The Graphics2D simply gathers together a list of objects that we can draw.
- We assume that all the objects added have a `draw` method which we call in the `show` function.
- We assume that all the objects have a `length` method. We call this in the `length` function to compute the total length of all the lines in our image and hence estimate the total cost of ink required to display the object on screen.

Usage

```
g = Graphics2D()
g.add(l1)
g.add(l2)

c = Circle2D()
g.add(c)
print('Cost of ink {}'.format(g.length()))
g.show()
```

- We have the makings of a useful 2-d geometry library.
- Let's try to design a 3d geometry library...

Isometry class

```
class Isometry:
    """Isometry of R^3"""

    def __init__(self):
        self._u = numpy.array([[1,0,0],[0,1,0],[0,0,1]])
        self._v = numpy.array([0,0,0])

    def transform_point(self, point):
        # Apply the transformation to the point
        return numpy.dot(self._u,point) + self._v;

    def transform_vector(self, vector):
        # Apply the transformation to a vector
        return numpy.dot(self._u, vector);
```

Properties

```
class Isometry:

    # ... all the code above ...

    @property
    def u(self):
        """Unitary matrix"""
        return self._u

    @u.setter
    def u(self, value):
        value = numpy.array(value)
        product = numpy.dot(value,value.transpose())
        assert numpy.linalg.norm(product - numpy.identity(3))<0.0001, \
            "u must be a unitary matrix"
        self._u = value

    @property
    def v(self):
        """Offset vector"""
        return self._v

    @v.setter
    def v(self, value):
        self._v = numpy.array(value)
```

Properties discussion

- By convention a variable or method beginning with an underscore is considered "private". This means that the programmer of the class reserves the right to change the name or get rid of the variable without notice.
- To generate a property type `prop` and tab or `props` and tab in Pycharm
- We have used properties to make sure that the matrix is unitary and that numpy is used.
- Functions can have attributes marked with the `@` notation. Some attributes are understood by Python, but you can add your own if you want too.

Example

```
i = Isometry();
i.v = [1,2,3] # converted to numpy automatically
i.u = [[2,0,0],[0,2,0],[0,0,2]] # causes error
```

- Using properties and private variables makes your class easier to use
- Goal of object-oriented programming: write classes that are easy for other people to use.

A static method

```
    @staticmethod
    def rotation(x_angle=0.0, y_angle=0.0, z_angle=0.0):
        """Creates a transformation representing a rotation through
           the given angle about each axis in turn"""
        i = Isometry()
        r_z = numpy.array([[math.cos(z_angle), math.sin(z_angle), 0],
                           [-math.sin(z_angle), math.cos(z_angle), 0],
                            [0, 0, 1]])
        r_x = numpy.array([[1, 0, 0],
                          [0, math.cos(x_angle), math.sin(x_angle)],
                          [0, -math.sin(x_angle), math.cos(x_angle)]])
        r_y = numpy.array([[math.cos(y_angle), 0, -math.sin(y_angle)],
                          [0, 1, 0],
                          [math.sin(y_angle), 0, math.cos(y_angle)]])
        i.u = numpy.dot(numpy.dot(r_z, r_y), r_x)
        return i
```

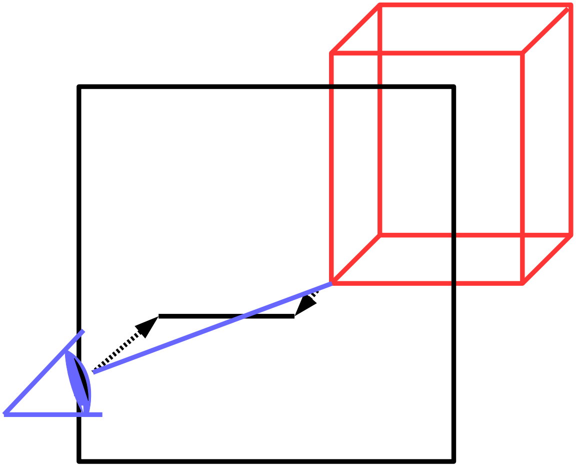
Static methods

- Static methods are associated with the class itself and not with any instance.
- They don't have `self` as the first parameter
- A static method that is designed to help create instances of a class is called a factory method.

The Viewpoint class

```
class Viewpoint:
    """Represents the position of an artist's eye and the
       coordinate system they are using for the canvas"""

    def __init__(self):
        self.eye = numpy.array([0,-1,0])
        self.canvasOrigin = numpy.array([0,0,0])
        self.canvasX = numpy.array([1,0,0])
        self.canvasY = numpy.array([0,0,1])

    def _perp(self):
        unnormalized = numpy.cross(self.canvasX, self.canvasY)
        return unnormalized/numpy.linalg.norm(unnormalized)
```

Perspective drawing 101



The Viewpoint class, continued

```
    def _project(self, coords_3d):
        """Project the point orthogonally onto the canvas,
           returns canvas coordinates of point and signed distance"""
        offset = coords_3d-self.canvasOrigin
        x_vec = self.canvasX
        y_vec = self.canvasY
        p = self._perp()
        distance = numpy.dot(offset, p)
        canvas_offset = offset - distance*p
        canvas_x = numpy.dot( canvas_offset, x_vec )/numpy.dot( x_vec, x_vec)
        canvas_y = numpy.dot( canvas_offset, y_vec )/numpy.dot( y_vec, y_vec)
        return numpy.array([canvas_x,canvas_y]), distance

    def coords_2d( self, coords_3d ):
        eye_proj, eye_distance = self._project( self.eye )
        point_proj, point_distance = self._project( coords_3d )
        total_distance = eye_distance - point_distance
        canvas_point = -eye_proj * point_distance/total_distance + \
                       point_proj*eye_distance/total_distance
        return canvas_point
```

```
class Line3D:

    def __init__(self, p1=numpy.array([0,0,0]), p2=numpy.array([1,0,0])):
        self.p1 = p1
        self.p2 = p2

    def draw(self, viewpoint : Viewpoint, graphics: Graphics2D):
        c1 = viewpoint.coords_2d(self.p1)
        c2 = viewpoint.coords_2d(self.p2)
        l2 = Line2D( c1, c2)
        graphics.add(l2)

    def transform(self, transformation: Isometry):
        other = copy.deepcopy(self)
        other.p1 = transformation.transform_point(self.p1)
        other.p2 = transformation.transform_point(self.p2)
        return other
```

To help the auto-complete we are telling Python that the parameter named `viewpoint` is of class `Viewpoint` etc. That's what the colon is doing in the function parameters of `draw`. This is optional.

Defining a cube class

First compute all the edges of the unit cube

```
def _compute_unit_cube_edges():
    """Compute a list of pairs representing the edges
       in the unit cube """
    ret = []
    offsets = [numpy.array([i, j, k]) for i in range(0, 2) for j in range(0, 2) for k in range(0, 2)];
    for i1 in range(0, len(offsets)):
        for i2 in range(i1 + 1, len(offsets)):
            p1 = offsets[i1]
            p2 = offsets[i2]
            diff = p1 - p2
            if abs(numpy.dot(diff, diff) - 1.0) < 0.001:
                ret.append((p1, p2))
    return ret
```

```
class Cube:
    """A cube with the given side length and edges
    in the directions v1, v2 and v1xv2"""

    _unit_cube_edges = _compute_unit_cube_edges()
    """Pairs of triples indicating the edges of a cube"""

    def __init__(self):
        self._origin = numpy.array([0,0,0])
        self._v1 = numpy.array([1,0,0])
        self._v2 = numpy.array([0,1,0])
        self._v3 = numpy.array([0,0,1])
        self.side_length = 1

    def _vertex(self, offsets):
        return self._origin + self.side_length*(\
            offsets[0]*self._v1 + offsets[1]*self._v2\
            + offsets[2]*self._v3)
```

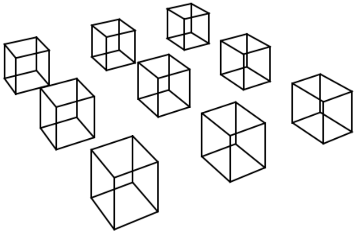
`_unit_cube_edges` is called a static variable because there is only one variable associated with the whole class. Add static variables at the top of your class declaration.

Cube class continued...

```
    def _edges(self):
        edges = []
        for p1,p2 in Cube._unit_cube_edges:
            v1 = self._vertex(p1)
            v2 = self._vertex(p2)
            line3d = Line3D(v1,v2)
            edges.append(line3d)
        return edges

    def draw(self, viewpoint: Viewpoint, graphics: Graphics2D):
        for edge in self._edges():
            edge.draw(viewpoint, graphics)

    def transform(self, transformation : Isometry):
        other = copy.deepcopy(self)
        other._v1 = transformation.transform_vector( self._v1)
        other._v2 = transformation.transform_vector( self._v2)
        other._v3 = transformation.transform_vector( self._v3)
        other._origin = transformation.transform_point(self._origin);
        return other
```

```
cubes = []
r = Isometry.rotation(z_angle=-0.5)
n = 3
for x in range(0,n):
    for y in range(0,n):
        cube = Cube()
        cube.side_length = 0.35
        t = Isometry.translation([x,y,0])
        c1 = cube.transform(t)
        c2 = c1.transform(r);
        cubes.append( c2 )

g = Graphics2D()
# set the viewpoint quite far back and a little above ground
v.eye = numpy.array([0,-4,3])
for cube in cubes:
    cube.draw(v,g)
g.show( large=True)
```

The Result



Diffie Hellman Key Exchange

- Diffie Hellman Key Exchange allows two people to devise a shared secret that can be used for encrypting messages.
- Alice and Bob agree on an Abelian group G with multiplicative notation and a group element g.
- Alice picks a random integer a.
- Bob picks a random integer b.
- Alice tells Bob g^a.
- Bob tells Alice g^b.
- They both now compute g^{ab}, their shared secret.

This is a “secret” because although exponentiation can be performed quickly, finding the logarithm to the base g in a group can be hard.

Multiplicative Group Elements

```
class ModInteger:
    """An integer mod p"""

    def __init__(self, p, value):
        self.p = p # the associated multiplicative group
        self.value = value % p
        if self.value<0:
            self.value += p

    def multiplicative_identity(self):
        return ModInteger(self.p,1)

    def __mul__(self, other):
        new_value = self.value*other.value
        return ModInteger(self.p,new_value)
```

Remarks

- This is called operator overloading.
- I have implemented lots of operators in the ModInteger class, but `__mul__` is all that's important no
- We now see how `sympy` works. It has overloaded + and $*$ etc. for symbolic objects.
- Recall we typed `x,y=var('x y')`
- This returns objects of type `Symbol`

Efficient exponentiation

```
def exponentiate(g, exponent):
    """Raise a multiplicative group element to the given power"""
    result = g.multiplicative_identity()
    base = g
    while exponent > 0:
        if exponent % 2 == 1:
            result *= base
        exponent = exponent >> 1  # bitwise shift left
        base = g*g
    return result
```

The cool thing is this will work for different groups. This is called *polymorphism*. All we need is a `multiplicative_identity` function and a $*$ function (i.e. `__mul__`)

Random Number Generation

```
class PoorRandom:
    """A pseudo random number generator, awful for cryptography"""

    def generate(self, n_bits):
        """Generate a random integer with the given number of bits"""
        return random.randint(0,2**n_bits-1)


class BetterRandom:
    """This is securely random"""

    def generate(self, n_bits):
        n_bytes = int(math.ceil(n_bits/8))
        random_bytes = os.urandom(n_bytes)
        int_value = int.from_bytes( random_bytes, byteorder='big')
        return int_value % (2**n_bits)
```

They both have the same `generate` function. They can be used polymorphically.

```
class DiffieHelmanExchanger:

    def __init__( self, base, random=BetterRandom(), n_bits=512 ):
        self.__secret = random.generate( n_bits )+1
        self.__public = exponentiate( base, self.__secret )

    def shared_secret(self, other_public ):
        return exponentiate( other_public, self.__secret )

    @property
    def public(self):
        return self.__public
```

Usage

```
def test_diffiehelman():
    p = 982451653
    base = ModInteger(p,126363)

    alice = DiffieHelmanExchanger( base )
    bob = DiffieHelmanExchanger( base )
    shared_secret1 = alice.shared_secret( bob.public )
    shared_secret2 = bob.shared_secret( alice.public )

    assert shared_secret1==shared_secret2
```

Remarks on what we have achieved

- Objects allow you to collect together data and functionality in a convenient way
- By creating objects that have methods in common we can use *polymorphism*. This allows us to write functions which can act equally well on different kinds of input.
- You can overload operators to make your objects really easy to use. Don't go crazy though, it normally only makes sense for maths objects.

Implementing elliptic curves

```
class EllipticCurve(object):
   def __init__(self, a, b):
      self.a = a
      self.b = b
      self.discriminant = -16 * (4 * a*a*a + 27 * b * b)
      assert self.discriminant!=0

   def contains(self, x, y):
      return y*y == x*x*x + self.a * x + self.b

   def __str__(self):
      return 'y^2 = x^3 + %Gx + %G' % (self.a, self.b)

   def __eq__(self, other):
      return (self.a, self.b) == (other.a, other.b)
```

Implementing elliptic curves

```
class EllipticCurve:
    """An elliptic curve and its associated group"""

    def __init__(self, a, b, n):
        assert n!=2
        assert n!=3
        self.n = n
        a = self.to_ring_element(a)
        b = self.to_ring_element(b)
        self.a = a
        self.b = b
        self.n = n
        self.discriminant = -self.to_ring_element(16) * \
            (self.to_ring_element(4) * a * a * a + self.to_ring_element(27) * b * b)
        assert self.discriminant!=self.to_ring_element(0)  , 'Curve is not smooth'

    def to_ring_element(self, x):
        return ModInteger(self.n,x)

    def contains(self, x, y):
        return y*y  == x*x*x + self.a * x + self.b
```

This represents the curve $y^2 = x^3 + ax + b$ over the integers mod n.

```python
class EllipticCurvePoint:
    """A point on an elliptic curve"""

    def __init__(self, curve, x=0, y=0, point_at_infinity=False):
        self.curve = curve
        self.x = curve.to_ring_element(x)
        self.y = curve.to_ring_element(y)
        self.point_at_infinity = point_at_infinity
        if not point_at_infinity:
            assert curve.contains(self.x,self.y)

    def multiplicative_identity(self):
        return EllipticCurvePoint(self.curve,point_at_infinity=True)

    def __eq__(self, other):
        if self.point_at_infinity:
            return other.point_at_infinity
        else:
            return self.x == other.x and self.y == other.y

    def __str__(self):
        return "(" + str(self.x.value) + "," + str(self.y.value) + ")"
```

```python
def __mul__(self, other):
    """This isn't computationally the most efficient multiplication method as it
        involves division"""
    if self.point_at_infinity:
        return other
    if other.point_at_infinity:
        return self

    x_1, y_1, x_2, y_2 = self.x, self.y, other.x, other.y

    if (x_1, y_1) == (x_2, y_2):
        if y_1 == self.curve.to_ring_element(0):
            return EllipticCurvePoint(self.curve,point_at_infinity=True)

        # slope of the tangent line
        m = (self.curve.to_ring_element(3) * x_1 * x_1 + self.curve.a) / \
                (self.curve.to_ring_element(2) * y_1)
    else:
        if x_1 == x_2:
            return EllipticCurvePoint(self.curve,point_at_infinity=True)

        # slope of the secant line
        m = (y_2 - y_1)/(x_2 - x_1)

    x_3 = m * m - x_2 - x_1
    y_3 = m * (x_3 - x_1) + y_1

    return EllipticCurvePoint(self.curve, x=x_3.value, y=-y_3.value)
```

```
class ModInteger:

    #  ... functions defined above ...

    def __add__(self, other):
        new_value = self.value+other.value
        return ModInteger(self.p,new_value)

    def __sub__(self, other):
        new_value = self.value-other.value
        return ModInteger(self.p,new_value)

    def __neg__(self):
        return ModInteger(self.p,-self.value)

    def __truediv__(self, y):
        '''Modular division, can be avoided'''
        g, a, b = mymath.euclidean_algorithm(y.value, self.p)
        return self * ModInteger(self.p,a)

    def __str__(self):
        return str(self.value) + " mod " + str(self.p)

    def __eq__(self, other):
        return other.value==self.value and other.p == self.p
```

Remarks on elliptic curve cryptography

- You might think that Abelian groups are simple, but in fact the isomorphism between an Abelian group and the "canonical" group in its isomorphism class may be very hard to compute efficiently.
- This particular use of elliptic curves is susceptible to quantum computing attacks. However, elliptic curves are used in more modern and sophisticated schemes that should hopefully be able to withstand quantum computing.
- We have used polymorphism to implement a scheme that is "pluggable" we can choose the group and the random number generator. A realistic cryptography scheme must be pluggable so it can be upgraded when needed.

Exercises

- How you would compute the greatest common divisor of two polynomials in Python? The code will take too long to write, sketch the idea.
- Change the `area` and `circumference` functions of circle so that they are properties. In other words you should be able to type `circle.circumference`.
- Write a Cylinder class. Just draw (say) a 20 sided polygonal prism.
- Create a stylised 3D image of Battersea Power station (a rectangular box with four cylinders on each corner).
- Write an AdditiveGroup class. This should represent the integers under addition with a given modulus but using multiplicative notation. Check that Diffie Hellman key exchange works. Note that this is not very useful!