geometry2d.py

```
from math import pi,sqrt,atan2;
from enum import Enum;

#--------------------------------------------------------------------------------#

twicePi=2.0*pi;
halfPi=0.5*pi;

r2d=180.0/pi;
d2r=pi/180.0;

#--------------------------------------------------------------------------------#

maxLength=10000.0; # area extension
epsilon=0.005
deltaAngles=epsilon/maxLength; # radian
deltaCoordinates=epsilon;

def areCoordinatesEqual(c1,c2:float)->bool:
    return abs(c2-c1)<deltaCoordinates;

def areLengthsEqual(l1,l2:float)->bool:
    return abs(l2-l1)<deltaCoordinates;

def areAnglesEqual(a1,a2:float)->bool:
    return abs(a2-a1)<deltaAngles;

#--------------------------------------------------------------------------------#

class BadError(Exception):
    def __init__(self,data):
        self.Data=data;

class InvalidDataError(Exception):
    def __init__(self,data):
        self.Data=data;

class InfinityPoints(Exception):
    def __init__(self): pass;

class NoPoint(Exception):
    def __init__(self): pass;

#--------------------------------------------------------------------------------#

class IntersectionKind(Enum):
    onePoint=1;
    noPoint=2;
    infinityPoints=3;
# end IntersectionKind

#--------------------------------------------------------------------------------#

class Vectors(object):

    def __init__(self,x=0.0,y=0.0):
        self.X=x; self.Y=y;

    def toClone(self)->'Vectors':
        return Vectors(self.X,self.Y);

    def __str__(self)->str:
        return str(self.X)+', '+str(self.Y);

    def __eq__(self,vc:'Vectors')->bool:
        return (areCoordinatesEqual(self.X,vc.X) and
                areCoordinatesEqual(self.Y,vc.Y));
```

```
    def lengthOf(self)->float:
        return sqrt(self.X*self.X+self.Y*self.Y);

    def toUnitVector(self)->'Vectors':
        lg=sqrt(self.X*self.X+self.Y*self.Y);
        return Vectors(self.X/lg,self.Y/lg);
    # end toUnitVector

    def __add__(self,vc:'Vectors')->'Vectors':
        return Vectors(self.X+vc.X,self.Y+vc.Y);

    def __sub__(self,vc:'Vectors')->'Vectors':
        return Vectors(self.X-vc.X,self.Y-vc.Y);

##    def __mul__(self,vc): # to float
##        return self.X*vc.X + self.Y*vc.Y;

    def __mul__(self,sc:float)->'Vectors': # with scalar
        return Vectors(self.X*sc,self.Y*sc);

    def smul(self,sc:float)->None: # with scalar to self
        self.X*=sc; self.Y*=sc;

    def __rmul__(self,sc:float)->'Vectors': # with scalar
        return Vectors(self.X*sc,self.Y*sc);

    def __truediv__(self,sc:float)->'Vectors': # with scalar
        return Vectors(self.X/sc,self.Y/sc);

    def sdiv(self,sc:float)->None: # with scalar to self
        self.X/=sc; self.Y/=sc;

    def dotOp(self,vc:'Vectors')->float:
        return self.X*vc.X + self.Y*vc.Y;

    def crossOp(self,vc:'Vectors')->float:
        return self.X*vc.Y - self.Y*vc.X;

    def dirAngleOf(self)->float:
        angle=atan2(self.Y,self.X);
        if angle<0.0:
            return angle+twicePi;
        else:
            return angle;
    # end directionAngle

    def sameDirectionOf(self,withV:'Vectors')->float:
        angleDiff=abs(withV.dirAngleOf()-self.dirAngleOf());
        if angleDiff<pi:
            return angleDiff<deltaAngles;
        else:
            return (twicePi-angleDiff)<deltaAngles;
    # end sameDirectionOf

# end Vectors

#-------------------------------------------------------------------------------#

class Points(object):

    def __init__(self,x=0.0,y=0.0):
        self.X=x; self.Y=y;

    def toClone(self)->'Points':
        return Points(self.X,self.Y);

    def updateOf(self,withP:'Points')->None:
        self.X=withP.X;
```

```
        self.Y=withP.Y;
    # end updateOf

    def __str__(self)->str:
        return str(self.X)+', '+str(self.Y);

    def __eq__(self,pt: 'Points')-> bool:
        return (areCoordinatesEqual(self.X,pt.X) and
                areCoordinatesEqual(self.Y,pt.Y));

    def __add__(self,vc:'Vectors')->'Points': # point with vector to point
        return Points(self.X+vc.X,self.Y+vc.Y);

    def __sub__(self,pt:'Points')->'Vectors': # point with point to vector
        return Vectors(self.X-pt.X,self.Y-pt.Y);

    def distanceOf(self,toPoint:'Points')->float:
        return Vectors.lengthOf(toPoint-self);

    def areaOf(self,p1,p2:'Points')->float: # of triangle
        return 0.5*Vectors.crossOp(p1-self,p2-self);

#end Points

#-----------------------------------------------------------------------------#

class Lines(object):
    # Implicite line equation is A*x+B*y=C with sqrt(A*A+B*B)=1

    def __init__(self,a,b,c:float):
        d=sqrt(a*a+b*b);
        self.A=a/d; self.B=b/d; self.C=c/d;
        if self.A==0.0: #!!!#
            xlp=0.0;
            ylp=self.C/self.B;
            xdir=1.0;
            ydir=0.0;
        elif self.B==0: #!!!#
            xlp=self.C/self.A;
            ylp=0.0;
            xdir=0.0;
            ydir=1.0;
        else:
            xlp=0.0;
            ylp=self.C/self.B;
            c=sqrt(self.A*self.A+self.B*self.B);
            xdir=self.B/c;
            ydir=-self.A/c;
        # end if
        self.LP=Points(xlp,ylp);
        self.DV=Vectors(xdir,ydir);
    # end __init__

    def toClone(self)->'Lines':
        return Lines(self.A,self.B,self.C);

    def __str__(self)->str:
        return ('ABC: '+str(self.A)+' '+str(self.B)+' '+str(self.C)+
                '\n\tLP: '+str(self.LP)+' | DV: '+str(self.DV));

    def __eq__(self,ln:'Lines')->bool:
        return (self.LP==ln.LP) and ((self.DV==ln.DV) or (self.DV==(-1.0*ln.DV)));

    @staticmethod
    def toLine(p1,p2:'Points')->'Lines':
        a=p2.Y-p1.Y;
        b=p1.X-p2.X;
        c=a*p1.X + b*p1.Y;
        return Lines(a,b,c);
```

```
    # end toLine

    @staticmethod
    def toLineWithDir(p1:'Points',dv:'Vectors')-> 'Lines':
        p2: Points=p1+dv;
        return Lines.toLine(p1,p2);
    # end toLineWithDir

    def distanceOf(self,pt:'Points')->float: # signed distance
        return self.A*pt.X + self.B*pt.Y - self.C;

    def nearestPointOf(self,fromP:'Points')->'Points':
        lP=self.LP;
        dV=self.DV;
        t=Vectors.dotOp(dV,fromP-lP);
        lP=lP+t*dV;
        return lP;
    # end nearestPointOf

    def isPointInside(self,pt:'Points')->bool:
        np=self.nearestPointOf(pt);
        return pt==np;
    # end isPointInside

    def areParallel(self,withLine:'Lines')->bool:
            # Vektorvergleich?
        an1=self.DV.dirAngleOf();
        an2=withLine.DV.dirAngleOf();
        sameDir=areAnglesEqual(an1,an2) or areAnglesEqual(an1,an2+pi);
        return sameDir and not (self.LP==withLine.LP);
    # end areParallel

    def intersectionOf(self,withLine:'Lines')->'Points':
        # Precondition: Lines are not parallel
        L1=self;
        L2=withLine;
        if L1==L2:
            raise InfinityPoints;
        if L1.areParallel(L2):
            raise NoPoint;
        lP1=L1.LP.toClone();
        lP2=L2.LP;
        dV1=L1.DV;
        dV2Ortho=Vectors(-L2.DV.Y,L2.DV.X);
        q=Vectors.dotOp(dV1,dV2Ortho);
        ts=Vectors.dotOp(dV2Ortho,lP2-lP1);
        ts/=q;
        lP1=lP1+ts*dV1;
        return lP1;
    # end intersectionOf

# end Lines

#-------------------------------------------------------------------------------#

class Rays(object):

    def __init__(self,startPoint:'Points',direction:'Vectors'):
        self.StartPoint=startPoint;
        self.Direction=direction.toUnitVector();
    # end __init__

    def toClone(self)->'Rays':
        return Rays(self.StartPoint,self.Direction);

    def __str__(self)->str:
        return 'SP: '+str(self.StartPoint)+' | DV: '+str(self.Direction);

    def __eq__(self,withRay:'Rays')->bool:
```

```
        return ((self.StartPoint==withRay.StartPoint) and
                (self.Direction==withRay.Direction));

    @staticmethod
    def toRay(pt1,pt2:'Points')->'Rays':
        return Rays(pt1,pt2-pt1);

    def toLine(self)->'Lines':
        return (Lines.toLine(self.StartPoint,
                self.StartPoint+self.Direction));

    def nearestPointOf(self,fromP:'Points')->'Points':
        lP=self.StartPoint.toClone();
        dV=self.Direction;
        t=Vectors.dotOp(dV,fromP-lP);
        if t>=0.0:
            lP=lP+t*dV;
        return lP;
    # end nearestPointOf

    def isPointInside(self,pt:'Points')->bool:
        np=self.nearestPointOf(pt);
        if np==self.StartPoint:
            return False;
        else:
            return np==pt;
    # end isPointInside

    def intersectionOf(self,withRay:'Rays')->'Points': # or exceptions - TBD
        ry1=self;
        ry2=withRay;
        ln1=ry1.toLine();
        ln2=ry2.toLine();
        if Lines.areParallel(ln1,ln2):
            raise NoPoint;
        if ln1==ln2:
            if (ry1.isPointInside(ry2.StartPoint) or
                ry2.isPointInside(ry1.StartPoint)):
                raise InfinityPoints;
            elif ((ry1.StartPoint==ry2.StartPoint) and
                  (ry1.Direction==ry2.Direction)):
                raise InfinityPoints;
            else:
                raise NoPoint;
        # end if
        ip=Lines.intersectionOf(ln1,ln2);
        if (ry1.isPointInside(ip) and
            ry2.isPointInside(ip)):
            return ip;
        else:
            raise NoPoint;
        # end if
    # end intersectionOf

# end Rays

#-----------------------------------------------------------------------------#

class Sectors(object):

    def __init__(self,startPoint:'Points',direction1,direction2:'Vectors'):
        self.StartPoint=startPoint;
        self.Direction1=direction1.toUnitVector();
        self.Direction2=direction2.toUnitVector();
    # end __init__

    def toClone(self)->'Sectors':
        return Sectors(self.StartPoint,self.Direction1,self.Direction2);
```

```
    def __str__(self)->str:
        return ('SP: '+str(self.StartPoint)+
                ' | DV1: '+str(self.Direction1)+
                ' | DV2: '+str(self.Direction2));

    def __eq__(self,withSector:'Sectors')->bool:
        return ((self.StartPoint==withSector.StartPoint) and
                 ((self.Direction1==withSector.Direction1) and
                 (self.Direction2==withSector.Direction2)) or
                 ((self.Direction1==withSector.Direction2) and
                 (self.Direction2==withSector.Direction1)));

    @staticmethod
    def toSector(pt0,pt1,pt2:'Points')->'Sectors':
        return Sectors(pt0,pt1-pt0,pt2-pt0);

    def openingAngleOf(self)->float: # no orientation, Öffnungswinkel
        return abs(self.Direction2.dirAngleOf()-self.Direction1.dirAngleOf());

# end Sectors

#--------------------------------------------------------------------------#

class Segments(object):

    def __init__(self,p1,p2:'Points'):
        self.P1=p1; self.P2=p2;

    @staticmethod
    def toSegmentWithDir(mp:'Points',dv:'Vectors',L:float)-> 'Segments':
        Lh=0.5*L;
        p1: Points=mp+(+Lh*dv);
        p2: Points=mp+(-Lh*dv);
        return Segments(p1,p2);
    # end toSegmentWithDir

    def toClone(self)->'Segments':
        return Segments(self.P1,self.P2);

    def update(self,withSegment:'Segments')->None:
        self.P1=withSegment.P1;
        self.P2=withSegment.P2;
    # end update

    def __str__(self)->str:
        return str(self.P1)+' | '+str(self.P2);

    def toLine(self)->'Lines':
        return Lines.toLine(self.P1,self.P2);

    def isPointInside(self,pt:'Points')->bool:
        ps1=self.P1;
        ps2=self.P2;
        rs1=Rays.toRay(ps1,ps2);
        rs2=Rays.toRay(ps2,ps1);
        return rs1.isPointInside(pt) and rs2.isPointInside(pt);
    # end isPointInside

    def distanceOf(self,pt:'Points')->float: # signed distance
        From=pt; To=self;
        if (Vectors.dotOp(From-To.P2,To.P2-To.P1)>0.0):
            return From.distanceOf(To.P2);
        elif (Vectors.dotOp(From-To.P1,To.P1-To.P2)>0.0):
            return From.distanceOf(To.P1);
        else:
            ln=To.toLine();
            return ln.distanceOf(From);
        # end if
    # end distanceOf
```

```
def isIntersectionOf_withLine(self,withLine: 'Lines') -> bool:
    # or exceptions TBD; Abstand = 0 TBD
    ps1=self.P1;
    ps2=self.P2;
    # Orientierter (vorzeichenbehafteter) Abstand der beiden Endpunkte
    # der Nadel von der Dielengerade
    d1=withLine.distanceOf(ps1); assert(d1!=0.0);
    d2=withLine.distanceOf(ps2); assert(d2!=0.0);
    # Prüfen, ob die Nadel die Dielengerade kreuzt
    return ((d1>=0.0>=d2) or (d1<0.0<d2));
# end isIntersectionOf_withLine


def intersectionOf_withRay(self,withRay:'Rays')->'Points': # or exceptions - TBD
    ps1=self.P1;
    ps2=self.P2;
    rs1=Rays.toRay(ps1,ps2);
    rs2=Rays.toRay(ps2,ps1);

    intersectionKind1=IntersectionKind.onePoint;
    intersectionKind2=IntersectionKind.onePoint;

    try:
        ip1=withRay.intersectionOf(rs1);
        if (ip1==ps1) or (ip1==ps2):
            raise NoPoint;
    except NoPoint:
        intersectionKind1=IntersectionKind.noPoint;
    except InfinityPoints:
        intersectionKind1=IntersectionKind.infinityPoints;
    # end try

    try:
        ip2=withRay.intersectionOf(rs2);
        if (ip2==ps1) or (ip2==ps2):
            raise NoPoint;
    except NoPoint:
        intersectionKind2=IntersectionKind.noPoint;
    except InfinityPoints:
        intersectionKind1=IntersectionKind.infinityPoints;
    # end try

    if ((intersectionKind1==IntersectionKind.onePoint) and
        (intersectionKind2==IntersectionKind.onePoint)):
        assert(ip1==ip2); # TBD
        return ip1;
    elif ((intersectionKind1==IntersectionKind.noPoint) and
          (intersectionKind2==IntersectionKind.noPoint)):
        raise NoPoint;
    elif ((intersectionKind1==IntersectionKind.noPoint) and
          (intersectionKind2==IntersectionKind.onePoint)):
        raise NoPoint;
    elif ((intersectionKind2==IntersectionKind.noPoint) and
          (intersectionKind1==IntersectionKind.onePoint)):
        raise NoPoint;
    else:
        raise InfinityPoints; # TBD
    # end if
# end intersectionOf_withRay

def intersectionOf(self,withSegment:'Segments')->'Points': # or exceptions - TBD
    ps1=self.P1;
    ps2=self.P2;
    rs1=Rays.toRay(ps1,ps2);
    rs2=Rays.toRay(ps2,ps1);
    try:
        ip1=withSegment.intersectionOf_withRay(rs1);
        ip2=withSegment.intersectionOf_withRay(rs2);
```

```
        except InfinityPoints:
            if (withSegment.isPointInside(ps1) or
                withSegment.isPointInside(ps2)):
                raise InfinityPoints;
            else:
                raise NoPoint;
            # end if
        except NoPoint:
            raise NoPoint;
        else:
            assert ip1==ip2;
            return ip1;
        # end try
    # end intersectionOf

    def bisectorOf(self)->'Lines':
            # Return the bisector to self,
            # i.e. the line that is perpendicular
            # to self and goes through its middle
        ln=self.toLine();
        xMid=(self.P1.X+self.P2.X)/2.0;
        yMid=(self.P1.Y+self.P2.Y)/2.0;
        return Lines(-ln.B,
                     ln.A,
                    -ln.B*xMid + ln.A*yMid);
    # end bisectorOf

# end Segments

#-------------------------------------------------------------------------------#

class Triangles(object):

    def __init__(self,p0,p1,p2:'Points'):
        self.P0=p0; self.P1=p1; self.P2=p2;

    def __str__(self)->str:
        return str(self.P0)+' | '+str(self.P1)+' | '+str(self.P2);

    def areaOf(self)->float:
        return Points.areaOf(self.P0,self.P1,self.P2);

    def isPointInside(self,pt:'Points')->bool:
        areaT=Points.areaOf(self.P0,self.P1,self.P2);
            # Barycentric coordinates with (alpha+beta+gamma)=1
            # alpha=Points.areaOf(pt,self.P1,self.P2)/areaT;
        beta =Points.areaOf(pt,self.P2,self.P0)/areaT;
        gamma=Points.areaOf(pt,self.P0,self.P1)/areaT;

        return (0.0<beta<1.0) and (0.0<gamma<1.0);
            #! No use of some delta
    # end isPointInside

# end Triangles

#-------------------------------------------------------------------------------#

class Polygons(object):

    def __init__(self,pointList: list['Points']):
        self.NbPoints=len(pointList);
        self.Points=[];
        self.Points+=pointList;
    # end __init__

    def toClone(self)->'Polygons':
        return Polygons(self.Points);

    def __str__(self)->str:
```

```
        output='';
        for pt in self.Points:
            output+=str(pt)+' | ';

        return(output);
    # end __str__

    def distanceOf(self,pt:'Points')->float:
        dist=-1;
        for Ix in range(0,self.NbPoints-1):
            p1=self.Points[Ix];
            p2=self.Points[Ix+1];
            sgm=Segments(p1,p2);
            dist=min(maxLength,sgm.distanceOf(pt));
        # end for
        p1=self.Points[1];
        p2=self.Points[self.NbPoints-1];
        sgm=Segments(p1,p2);
        dist=min(dist,sgm.distanceOf(pt));
        return dist;
    # end distanceOf

    def areaOf(self)->float: # signed area
        # Precondition: convex polygons only
        p0=self.Points[0];
        area=0.0;
        for Ix in range(1,self.NbPoints-1):
            p1=self.Points[Ix];
            p2=self.Points[Ix+1];
            area+=Vectors.crossOp(p1-p0,p2-p0);
        # end for
        return 0.5*area;
    # end areaOf

# end Polygons

#-----------------------------------------------------------------------------#

class Circles(object):

    class NoCircle(Exception):
        def __init__(self): pass;

    def __init__(self,point:'Points',radius:float):
        self.C=point;
        self.R=radius;
    # end def

    def toClone(self)->'Circles':
        return Circles(self.C,self.R);

    def __str__(self)->str:
        return str(self.C)+' | '+str(self.R);

    @staticmethod
    def toCircle(pt1,pt2,pt3:'Points')->'Circles': # or exceptions - TBD
        sgm1=Segments(pt1,pt2);
        sgm2=Segments(pt2,pt3);
        bis1=sgm1.bisectorOf();
        bis2=sgm2.bisectorOf();
        try:
            center=Lines.intersectionOf(bis1,bis2);
        except (NoPoint,InfinityPoints):
            raise Circles.NoCircle;
        # end try
        radius=pt1.distanceOf(center);
        return Circles(center,radius);
    # end toCircle
```

```
    def areaOf(self)->float:
        return pi*self.R*self.R;

    def distanceOf(self,pt:'Points')->float: # signed distance
        d=Points.distanceOf(self.C,pt);
        r=self.R;
        return d-r;
    # end distanceOf

    def isPointInside(self,pt:'Points')->bool:
        dist=self.distanceOf(pt);
        return (dist<0) and not areLengthsEqual(dist,0);
    # end isPointInside

# end Circles

#-------------------------------------------------------------------------------#
#-------------------------------------------------------------------------------#

if __name__ == '__main__':
    print('\n# Beginning Test ...');

#-------------------------------------------------------------------------------#
#---Vectors---------------------------------------------------------------------#

    if True:
        print('\nVectors ...');
        vc1=Vectors(1,2);
        assert str(vc1)=='1, 2';
        assert vc1==Vectors(1,2);
        assert vc1==Vectors(1,2.001);
        assert not vc1==Vectors(1,2.01);
        assert not vc1==Vectors(1,3);
        vc2=Vectors(2,4);
        assert vc1+vc1==vc2;
        assert vc1*2==Vectors(2,4);
        assert 2*vc1==Vectors(2,4);
        vc3=vc2/2.0;
        assert vc3==vc1;
        vc1.smul(2);
        assert vc1==vc2;
        vc1.sdiv(2);
        assert vc1==vc1;
        assert Vectors.dotOp(vc1,vc1)==5;
        assert Vectors.dotOp(Vectors(1,1),Vectors(-1,1))==0;
        assert Vectors.crossOp(vc1,vc1)==0;
        assert Vectors.crossOp(vc1,Vectors(1,3))==1;
        vc3=vc1.toUnitVector();
        assert not vc3.lengthOf()==1;
        assert abs(vc3.lengthOf()-1) <epsilon;
        vc4=Vectors(1,1);
        assert vc4.dirAngleOf()*r2d==45;
        vc4=Vectors(-1,1);
        assert vc4.dirAngleOf()*r2d==135;
        vc4=Vectors(-1,-1);
        assert vc4.dirAngleOf()*r2d==225;
        assert vc1.sameDirectionOf(vc1);
        assert vc1.sameDirectionOf(Vectors(1,2.000001));
        assert not vc1.sameDirectionOf(Vectors(1,2.00001));
    # end if

#---Points----------------------------------------------------------------------#

    if True:
        print('\nPoints ...');
        pt1=Points(1,2);
        assert pt1==pt1;
        assert str(pt1)=='1, 2';
        assert pt1.distanceOf(pt1)==0.0;
```

```
        pt2=Points(3,2);
        assert pt1.distanceOf(pt2)==2.0;
        assert pt2.distanceOf(pt1)==2.0;
        vc1=Vectors(1,1);
        assert (pt2-pt1)==Vectors(2,0);
        assert (pt1+vc1)==Points(2,3);
        pt3=Points(0,0); pt4=Points(2,0); pt5=Points(1,1);
        assert pt3.areaOf(pt4,pt5)==1.0;
        assert pt3.areaOf(pt5,pt4)==-1.0;
    # end if

#---Lines-------------------------------------------------------------------#

    if True:
        print('\nLines ...');
        pt1=Points(0,0); pt2=Points(2,2);
        pt3=Points(0,2); pt4=Points(2,0);
        L12=Lines.toLine(pt1,pt2);
        L34=Lines.toLine(pt3,pt4);
        assert L12==L12;
        assert not L12==L34;
        ipt=Lines.intersectionOf(L12,L34);
        pt5=Points(1,1);
        assert ipt==pt5;
        try:
            ipt=Lines.intersectionOf(L12,L12);
        except InfinityPoints:
            pass;
        assert L12.nearestPointOf(pt5)==pt5;
        assert L12.nearestPointOf(Points(0,2))==pt5;
        pt6=Points(3,3);
        assert Lines.isPointInside(L12,pt6);
        assert not Lines.isPointInside(L12,pt3);
        assert abs(abs(Lines.distanceOf(L12,pt3))-sqrt(2))<epsilon;
        assert not abs(Lines.distanceOf(L12,pt3)-sqrt(2))<epsilon;
        assert not Lines.areParallel(L12,L12);
        pt7=Points(1,0); pt8=Points(3,2);
        L78=Lines.toLine(pt7,pt8);
        assert Lines.areParallel(L12,L78);
        try:
            ipt=Lines.intersectionOf(L12,L78);
        except NoPoint:
            pass;
    # end if

#---Rays--------------------------------------------------------------------#

    if True:
        print('\nRays ...');
        sp1=Points(1,1); dir1=Vectors(1,1);
        ray1=Rays(sp1,dir1);
        assert ray1==ray1;
        assert str(ray1)=='SP: 1, 1 | DV: 0.7071067811865475, 0.7071067811865475';
        ln1=ray1.toLine();
        ln2=Lines(0.707106781186545,-0.7071067811865475,0.0);
        assert ln1==ln2;
        sp2=Points(3,1); dir2=Vectors(-1,1);
        ray2=Rays(sp2,dir2);
        ip12=Rays.intersectionOf(ray1,ray2);
        assert ip12==Points(2,2);
        sp3=Points(1,1); dir3=Vectors(-1,1);
        ray3=Rays(sp3,dir3);
        try:
            ip13=Rays.intersectionOf(ray1,ray3);
        except NoPoint:
            pass;
        try:
            ip13=Rays.intersectionOf(ray1,ray1);
        except InfinityPoints:
```

```
            pass;
        sp4=Points(1,1); dir4=Vectors(-1,-1);
        ray4=Rays(sp4,dir4);
        try:
            ip44=Rays.intersectionOf(ray4,ray1);
        except NoPoint:
            pass;
    # end if

#---Sectors ----------------------------------------------------------------#

    if True:
        print('\nSectors ...');
        pt0=Points(0,0);
        pt1=Points(1,1);
        pt2=Points(-1,1);
        sec=Sectors.toSector(pt0,pt1,pt2);
        assert sec==sec;
        sp1=Points(2,0);
        dv1=Vectors(-0.7071067811865475,0.7071067811865475);
        dv3=Vectors(-0.7071067811865476,0.7071067811865476);
        dv2=Vectors(0.4472135954999579, 0.8944271909999159);
        dv4=Vectors(0.447213595499958,  0.894427190999916);
        sc1=Sectors(sp1,dv1,dv2);
        sc2=Sectors(sp1,dv3,dv4);
        assert sc1==sc2;
        sc3=Sectors(sp1,dv1,dv2);
        sc4=Sectors(sp1,dv4,dv3);
        assert sc3==sc4;
    # end if

#---Segments----------------------------------------------------------------#

    if True:
        print('\nSegments ...');
        pt1=Points(1,1);
        pt2=Points(2,1);
        sgm=Segments(pt1,pt2);
        assert sgm==sgm;
        assert str(sgm)=='1, 1 | 2, 1';
        sp1=Points(2.5,0); dir1=Vectors(-1,1);
        ray1=Rays(sp1,dir1);
        ip1=sgm.intersectionOf_withRay(ray1);
        assert ip1==Points(1.5,1.0);
        assert sgm.distanceOf(pt2)==0.0;
        try:
            pt0=Points(0,0); dir0=Vectors(1,1);
            ray0=Rays(pt0,dir0);
            ip0=sgm.intersectionOf_withRay(ray0);
        except NoPoint:
            pass;
        else:
            raise BadError('### in intersectionOf_withRay ###');
        # end try
        pt3=Points(1.5,2);
        assert sgm.distanceOf(pt3)==-1.0;
        pt3=Points(1.5,0);
        assert sgm.distanceOf(pt3)==1.0;
        pt1a=Points(1.4,0);
        pt2a=Points(1.4,2);
        sgma=Segments(pt1a,pt2a);
        ipa=sgm.intersectionOf(sgma);
        assert ipa==Points(1.4,1);
        pt1b=Points(1.3,0);
        pt2b=Points(4,3);
        sgmb=Segments(pt1a,pt2b);
        try:
            ipb=sgm.intersectionOf(sgmb);
        except NoPoint: pass;
```

```
    # end if

#---Polygons-------------------------------------------------------------------#

    if True:
        print('\nPolygons ...');
        pt1=Points(1,0); pt2=Points(2,1); pt3=Points(0,1);
        pts1=[pt1,pt2,pt3];
        pg1=Polygons(pts1);
        assert pg1.areaOf()==1.0;
        pts2=[pt1,pt3,pt2];
        pg2=Polygons(pts2);
        assert pg2.areaOf()==-1.0;
        pt4=Points(2,2);
        assert pg1.distanceOf(pt4)==1.0;
    # end if

#---Triangles------------------------------------------------------------------#

    if True:
        print('\nTriangles ...');
        pt1=Points(1,0); pt2=Points(2,1); pt3=Points(0,1);
        tri=Triangles(pt1,pt2,pt3);
        assert tri.areaOf()==1.0;
        p4=Points(3,3);
        assert not tri.isPointInside(p4);
        p5=Points(1,0.5);
        assert tri.isPointInside(p5);
    # end if

#---Circles---------------------  ----------------------------------------#

    if True:
        print('\nCircles ...');
        pt1=Points(2,2);
        cc1=Circles(pt1,1);
        assert not cc1.isPointInside(Points(1,1));
        assert cc1.isPointInside(Points(1.5,2));
        pt2=Points(1,1); pt3=Points(2,0); pt4=Points(2,2);
        cc2=Circles.toCircle(pt2,pt3,pt4);
        assert str(cc2)=='2.0, 1.0 | 1.0';
        pt5=Points(1,1); pt6=Points(2,1); pt7=Points(3,1);
        try:
            cc3=Circles.toCircle(pt5,pt6,pt7);
        except Circles.NoCircle:
            pass;
    # end if

#-------------------------------------------------------------------------------#
#-------------------------------------------------------------------------------#
    print('\n# Finished Test.\n');
# end if main
```