In [1]:
```
using PyPlot
using LinearAlgebra
```

Problem 1 - Computational geometry

Given a function $f(x)$ defined on $[0,1]$, and its derivatives $f'(x)$, $f''(x)$ draw the osculating circle of at a point $x_0 \in (0,1)$.

An osculating circle at x_0 is the circle that passes through the point and has the same tangent and curvature of the curve at $f(x_0)$.

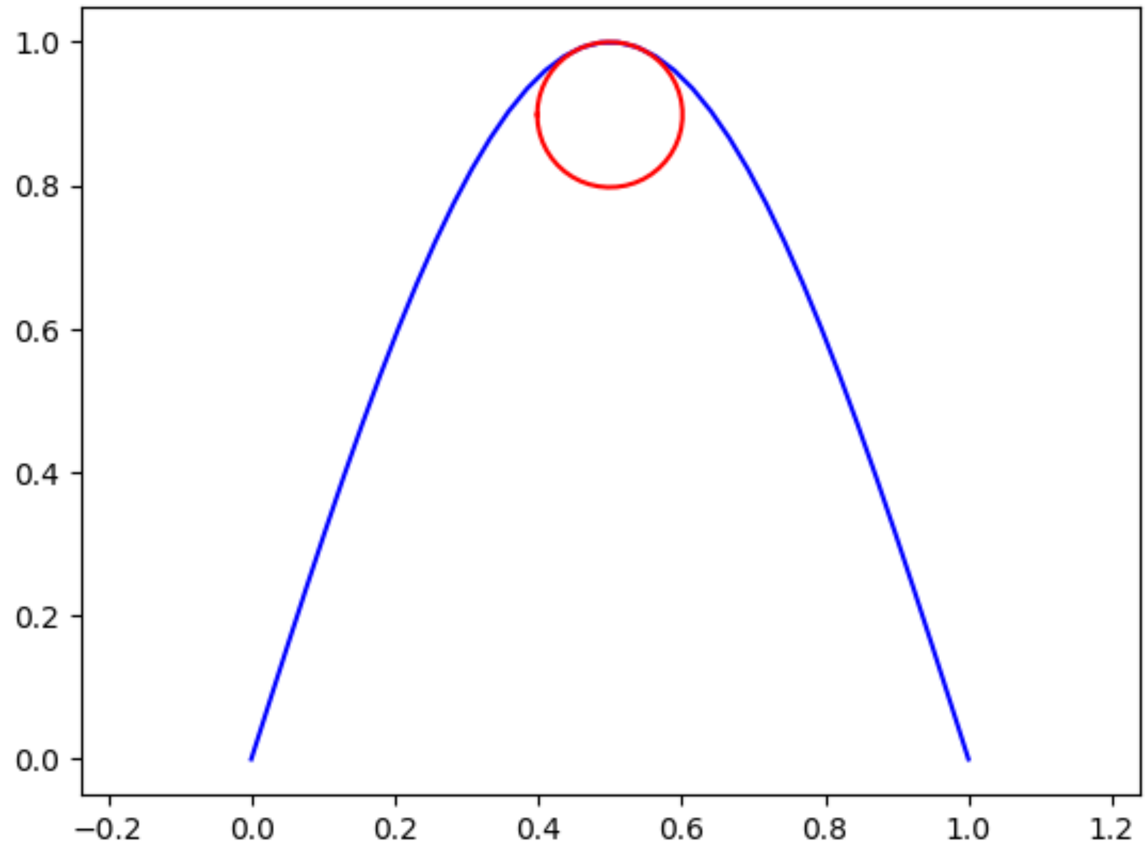
Reminder that curvature can be written as $\frac{f''(x)}{(1+f'(x))^{\frac{3}{2}}}$

Solution:

In [2]:
```
function osculatingCircle( f, df, d2f, x0 )

    #Calculate function and derivatives
    fx0 = f(x0);
    dfx0 = df(x0);
    d2fx0 = d2f(x0);

    #Calculate circle quantities
    kappa = d2fx0 / abs( 1.0+dfx0 )^(3.2);
    radius = 1/kappa;
    vec = [-dfx0,1];
    vec /= norm(vec)
    centre = [x0, fx0] + vec*radius;

    #Draw function
    xpts = LinRange(0,1,40);
    ypts = [f(x) for x in xpts];
    plot(xpts, ypts, "b-")

    #Draw circle
    theta = LinRange( 0, 2*pi, 50 )
    xpts = [ radius*cos(t)+centre[1] for t in theta ]
    ypts = [ radius*sin(t)+centre[2] for t in theta ]
    plot(xpts, ypts, "r-")

end
```

Out[2]:
```
osculatingCircle (generic function with 1 method)
```

In [3]:
```
g(x) = sin(x*pi)
dg(x) = cos(x*pi)*pi
d2g(x) = -sin(x*pi)*(pi)^2
x0 = 0.5

osculatingCircle( g, dg, d2g, x0 );
axis("equal");
```



Problem 2 - Optimisation

Part 1: Write out the Taylor Series approximation for $f(x_0 + h)$ centred at the point x_0 up to second order terms.

Part 2: Write out the Taylor Series approximation for $f(x_0 - h)$ centred at the point x_0 up to second order terms.

Part 3: Combine these two to get an approximation for the first derivative $f'(x_0)$.

Part 4: Use this approximate derivative to implement gradient descent to minimise some function $f(x)$ (100 maximum iterations, h = 1e-4, α=0.1).

Solution

1) $f(x_0 + h) = f(x_0) + hf(x_0) + \frac{h^2}{2} f''(x_0)$

2) $f(x_0 - h) = f(x_0) - hf(x_0) + \frac{h^2}{2} f''(x_0)$

3) $\frac{f(x_0+h)-f(x_0-h)}{2h} \approx f'(x_0)$

Code for part 4:

In [4]:
```
function approxGradientDescent( f, x0 )

    maxiters = 100;
    prevx = 0.0;
    currx = x0;
    h = 1e-4;
    alpha = 0.1;

    for iter = 1:maxiters

        dfx = ( f( currx+h ) - f( currx-h ) ) / ( 2*h );
        prevx = currx + 0.0;
        currx -= alpha * dfx;
        if abs( prevx-currx ) < 1e-8
            return currx
        end

    end

    return currx

end
```

Out[4]:
```
approxGradientDescent (generic function with 1 method)
```

In [5]:
```
f1(x) = cos(x)^2

approxGradientDescent( f1,0.1 )
```

Out[5]:
```
1.5707962946425253
```

Problem 3 - Graphs

A bipartite graph is one where you can separate nodes into two groups A, B such that each edge of the graph connects a vertex of A to one of B.

Edit BFS to determine whether a graph is bipartite.

In [6]:

```
struct Vertex
    neighbors::Vector{Int}       # Indices of neighbors of this Vertex
    coordinates::Vector{Float64} # 2D coordinates of this Vertex - onl
    Vertex(neighbors; coordinates=[0,0]) = new(neighbors, coordinates)
end

function Base.show(io::IO, v::Vertex)
    print(io, "Neighbors = ", v.neighbors)
end
```

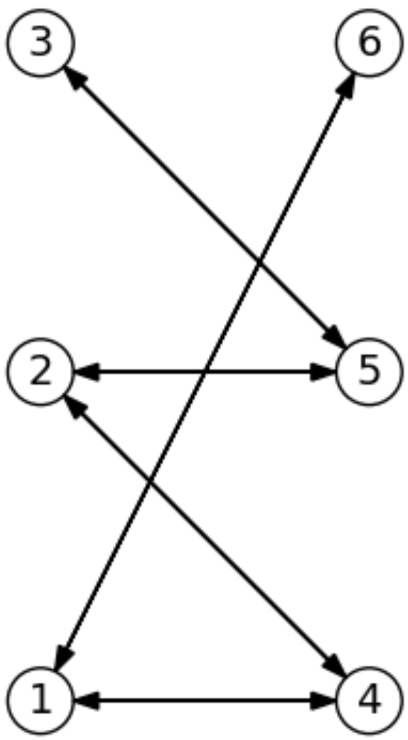
In [7]:

```
struct Graph
    vertices::Vector{Vertex}
end

function Base.show(io::IO, g::Graph)
    for i = 1:length(g.vertices)
        println(io, "Vertex $i, ", g.vertices[i])
    end
end
```

In [8]:
```
function PyPlot.plot(g::Graph; scale=1.0)
    fig, ax = subplots()
    ax.set_aspect("equal")

    xmin = minimum(v.coordinates[1] for v in g.vertices)
    xmax = maximum(v.coordinates[1] for v in g.vertices)
    ymin = minimum(v.coordinates[2] for v in g.vertices)
    ymax = maximum(v.coordinates[2] for v in g.vertices)
    sz = max(xmax-xmin, ymax-ymin)
    cr = scale*0.05sz
    hw = cr/2
    axis([xmin-2cr,xmax+2cr,ymin-2cr,ymax+2cr])
    axis("off")

    for i in 1:length(g.vertices)
        c = g.vertices[i].coordinates
        ax.add_artist(matplotlib.patches.Circle(c, cr, facecolor="none
        ax.text(c[1], c[2], string(i),
                horizontalalignment="center", verticalalignment="cente
        for nb in g.vertices[i].neighbors
            cnb = g.vertices[nb].coordinates
            dc = cnb .- c
            L = sqrt(sum(dc.^2))
            c1 = c .+ cr/L * dc
            c2 = cnb .- cr/L * dc
            arrow(c1[1], c1[2], c2[1]-c1[1], c2[2]-c1[2],
                  head_width=hw, length_includes_head=true, facecolor=
        end
    end
end
```

Here is an example of a bipartite graph:

In [9]:
```
v1 = Vertex([4,6], coordinates=[0,-1])
v2 = Vertex([5,4], coordinates=[0,0])
v3 = Vertex([5], coordinates=[0,1])
v4 = Vertex([1,2], coordinates=[1,-1])
v5 = Vertex([2,3], coordinates=[1,0])
v6 = Vertex([1], coordinates=[1,1])
g1 = Graph([v1,v2,v3,v4,v5,v6]);

plot(g1)
```



Here is BFS:

In [10]:
```
function bfs(g::Graph, start)
    visited = falses(length(g.vertices))
    S = [start]
    visited[start] = true
    while !isempty(S)
        ivertex = popfirst!(S)
        println("Visiting vertex #$ivertex")
        for nb in g.vertices[ivertex].neighbors
            if !visited[nb]
                visited[nb] = true
                push!(S, nb)
            end
        end
    end
end
```

Out[10]:
```
bfs (generic function with 1 method)
```

Solution:

In [11]:
```
function isBipartite(g::Graph, start)
    isAB = -ones(Int, length(g.vertices))
    isAB[start] = 0

    visited = falses(length(g.vertices))
    S = [start]
    visited[start] = true
    while !isempty(S)
        ivertex = popfirst!(S)
        println("Visiting vertex #$ivertex")

        for nb in g.vertices[ivertex].neighbors
            if !visited[nb]
                visited[nb] = true
                isAB[nb] = 1-isAB[ivertex];
                push!(S, nb)
            else
                if isAB[nb] == isAB[ivertex]
                    return false
                end
            end
        end
    end

    return true
end
```

Out[11]:
```
isBipartite (generic function with 1 method)
```

In [12]:
```
isBipartite(g1,1)
```

```
Visiting vertex #1
Visiting vertex #4
Visiting vertex #6
Visiting vertex #2
Visiting vertex #5
Visiting vertex #3
```

Out[12]:
```
true
```

In [ ]: