In [16]:
```
%matplotlib notebook
%pylab

# For comments on the use of "%pylab", see the bottom of this notebook.
```

```
Using matplotlib backend: nbAgg
Populating the interactive namespace from numpy and matplotlib
```

Using Numpy arrays

NumPy arrays are a special class of objects loaded from the NumPy . Working with NumPy arrays in Python is very much like using arrays in Matlab.

Here are a few key differences between Matlab arrays and NumPy arrays:

- Indexing into NumPy arrays uses square brackets `[]` rather than parenthesis `()`.
- Indexing in NumPy arrays always starts with 0. The first entry in an array is `x[0]`. If the array has `N` entries, the last entry in the array is `x[N-1]` or just `x[-1]`.
- To extract a subset of indices starting with `I` and ending with `J`, use the colon operator with indices `I:J+1'.

In [17]:
```
# Indexing into arrays
N = 10
x = arange(N)  # array of numbers 0,1,2,3,4,..,N-1
print(x)
print(x[0])
print(x[N-1])   # last entry
print(x[-1])    # shorthand for last entry (like Matlab x(end))
```

```
[0 1 2 3 4 5 6 7 8 9]
0
9
9
```

In [18]:
```
# extracting a subset of indices I, I+1, I+2, ... J
x = arange(20)  # x = [0,1,2,...,19]
print("x = ",x)

# Subset of entres x[5], x[6], x[7], ..., x[11]
print("")
I = 5
J = 11
print("x[5:12] = ", x[I:J+1])
```

```
x =  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]

x[5:12] =  [ 5  6  7  8  9 10 11]
```

In [19]:
```
# To get all entries from index I=13 to the end : x[13],x[14], ...., x[N-1]
I = 13
print("x[13:] = ", x[I:])
```

```
x[13:] =  [13 14 15 16 17 18 19]
```

In [20]:
```
# To get entries starting from the first entry to the second to last entry :
print("x[:-1] = ", x[:-1])
```

```
x[:-1] =  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18]
```

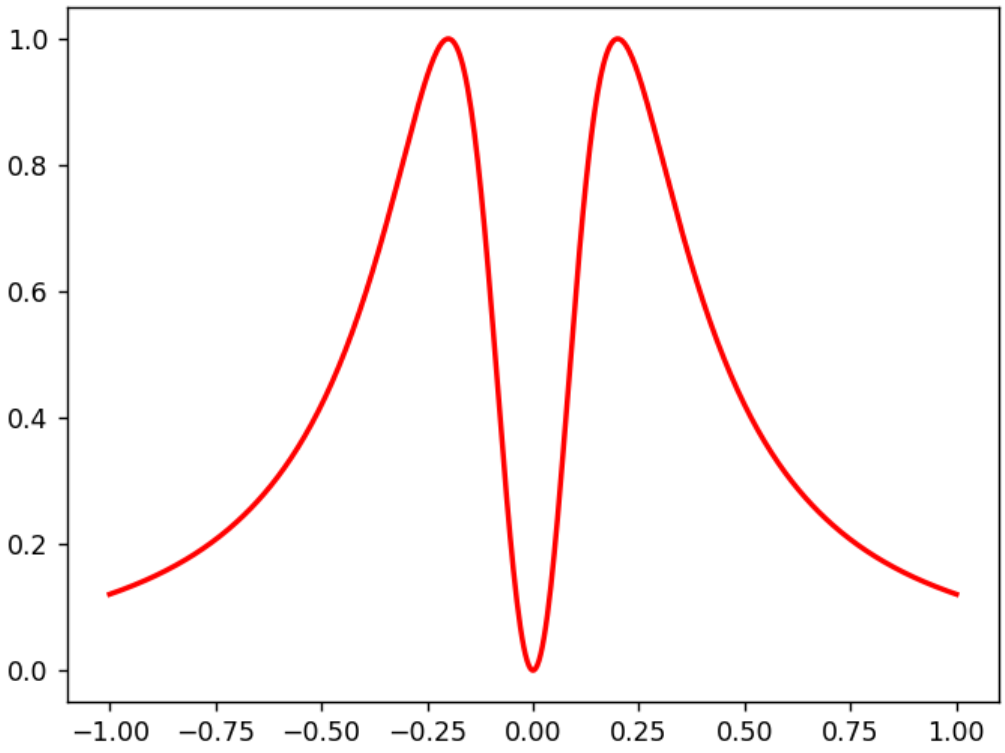
Arithmetic operations on NumPy arrays

NumPy arrays can be used in most mathematical calculations involving arithmetic operations elementary functions. For example, to plot the function

$$f(x) = \sin\left(\frac{\pi}{1+25x^2}\right) \tag{1}$$

over the interval $[-1, 1]$, we use the `linspace` command to create an array of equally spaced values in $[-1, 1]$. The plot commands are very similar to those used in Matlab

In [21]:
```
figure(1)
clf()

x = linspace(-1,1,500)

plot(x,sin(pi/(1 + 25*x**2)),'r',linewidth=2)
```



Out[21]:
```
[<matplotlib.lines.Line2D at 0x7fc80a7ac650>]
```

Unlike in Matlab, NumPy does not require the use of a "dot" `.` to indicate that operations are all "elementwise" operations.

Note on the use of the magic command `%pylab`

If you google "pylab", you will find that its use is discouraged. The reason that is given is that the magic command %pylab at the top of this notebook imports several different packages, listed below and the namespace becomes polluted. Here is some commentary

Slightly more worrisome is the fact that pylab (which is a module within matplotlib) is soon to be **deprecated**. In up-to-date documentation of Matplotlib, `pylab` is not mentioned.

Here is what the magic command `%pylab`, at the top of this notebook, does:

```
# Here is what %pylab does
import numpy
import matplotlib
from matplotlib import pylab, mlab, pyplot
np = numpy
plt = pyplot

from IPython.core.pylabtools import figsize, getfigs

from pylab import *
from numpy import *
```

The advantage of the importing pylab is that we have access to all numerical functions (e.g. `sin`, `cos`, `sqrt` and so on) without having to prefix these commands as `np.sqrt`, or worse `np.pi`. Of course, these prefixes can be helpful in telling us where different functions and methods are coming from. But with some care, it is easy to avoid any possible naming collisions.

That said, a cleaner namespace is

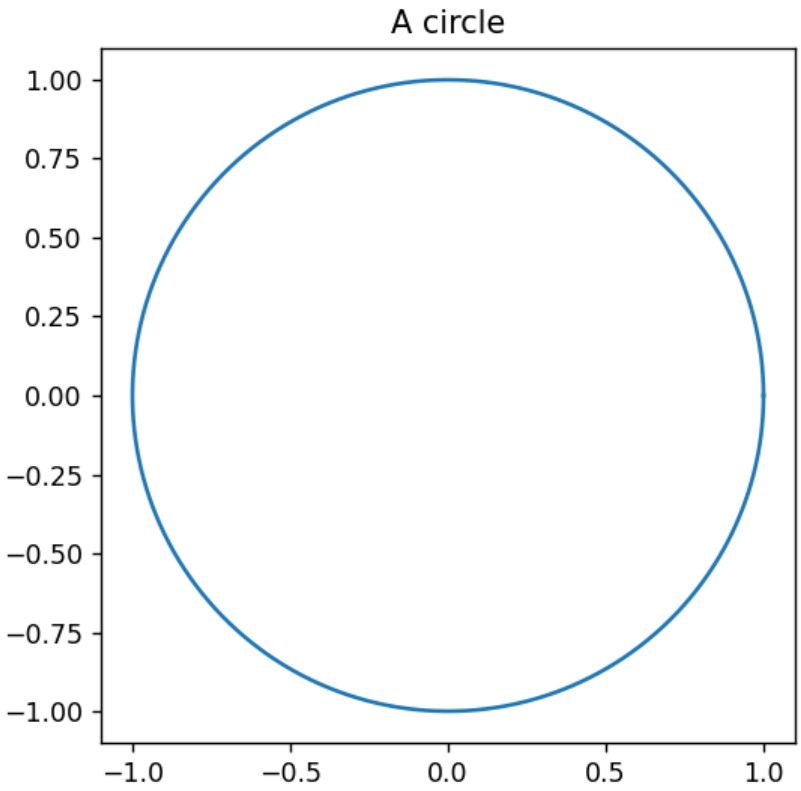
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

And then to use Numpy functions, or to plot, we have to prefix our functions with `np` or `plt`. Below, we give an example

In [22]:
```
%reset -f

import numpy as np
```

```
import matplotlib.pyplot as plt
%matplotlib notebook
```

In [23]:
```
plt.figure(2)
plt.clf()

th = np.linspace(0,2*np.pi,200)
x = np.cos(th)
y = np.sin(th)

plt.plot(x,y)

plt.title('A circle')

plt.gca().set_aspect('equal')

plt.show()
```



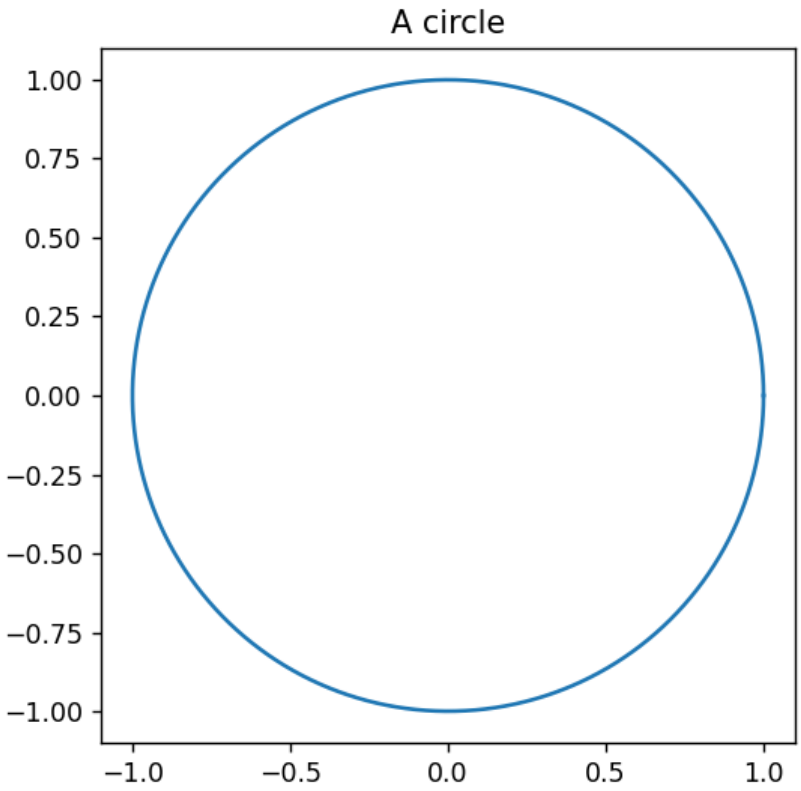
Some commentary on the above style can be found here.

> *So, why all the extra typing instead of the MATLAB-style (which relies on global state and a flat namespace)? For very simple things like this example, the only advantage is academic: the wordier styles are more explicit, more clear as to where things come from and what is going on. For more complicated applications, this explicitness and clarity becomes increasingly valuable, and the richer and more complete object-oriented interface will likely make the program easier to write and maintain.*

If you find that type `np.XXX` and `plt.XXX` tedious, but do not want to rely on the magic command `%pylab`, the following should work in most cases.

In [9]:
```
%reset -f

from numpy import *
from matplotlib.pyplot import *
%matplotlib notebook
```

In [10]:
```
figure(3)
clf()

th = linspace(0,2*np.pi,200)
x = cos(th)
y = sin(th)

plot(x,y)

title('A circle')

gca().set_aspect('equal')

show()
```



In [ ]:
```

```