Using DSolve

In this lesson, we will use the `sympy` package and in particular, the `dsolve` function. The dsolve function will take a differential equation as input, and the output will be an equation involving the unknown function `x(t)` but none of it derivatives. In the best case scenario, the equation will explicitly give `x(t)` though sometimes it might only be implicit.

To begin with, we do `from sympy import *`. This imports all functions from `sympy`.

In [1]:
```
from sympy import *
```

The important thing about `dsolve` is that it does not find a solution *numerically* but finds a solution *symbolically* (i.e. it "knows" the methods that we will be learning about in class.) Therefore, we must make an equation that tells `sympy` that certain things are symbols and should be treated that way.

There are a few things going on, and we will go step-by-step looking at some more basic examples (that have nothing to do with differential equation) first.

In [2]:
```
#eq is an equation
eq = Eq(symbols('y')**2, 2)
print(eq)
pprint(eq)
```

```
Eq(y**2, 2)
 2
y  = 2
```

The first line encodes the equation $y^2 = 2$ in a format that `sympy` can understand. The `Eq` (equation) class is what `sympy` uses to represent equations. The part involving `symbols` tells `sympy` that `y` is a symbolic variable (instead of a normal programming variable that contains a literal value). So the expression `Eq(symbols('y')**2, 2)` stores the equation $y^2 = 2$ in a way that `sympy` understands. Next, we issu the `print` command so we can see what `Eq` is. The `pprint` command means "pretty print" and prints the equation in a slightly more readable format.

Below, we see what happens if we don't tell `sympy` that `x` is a `symbol`. If `x=1` (as we have below), then the line `eq = Eq(x**2, 2)` means "$1^2 = 2$" and this is of course not true, so the output given when we print (using either `pprint` or `print`) is `False`.

In [3]:
```
x = 1
eq = Eq(x**2,2)
print(eq)
pprint(eq)
```

```
False
False
```

The package `sympy` has a function called `solve` that can solve this equation. It is pretty straight forward. The `solve` function returns the list of solutions to the equation:

In [4]:
```
#eq is an equation
eq = Eq(symbols('y')**2, 2)
```

```
# soln will be the solution
soln = solve(eq)
print(soln)
pprint(soln)
```

```
[-sqrt(2), sqrt(2)]
[-√2, √2]
```

The block directly below solves the equation $\sin y + \cos(x^2) = 1$ for y. The expression

```
soln = solve(eq, symbols('y'))
```

says to solve `eq` for the variable `y`. In other words, it says that the solutions to this equation are:

$$y = \arcsin(\cos(x^2) - 1) + \pi$$
$$y = -\arcsin(\cos(x^2) - 1).$$

In [5]:
```
eq = Eq(sin(symbols('y')) + cos(symbols('x')**2), 1)
pprint(eq)
soln = solve(eq, symbols('y'))
#print(soln)
print("")
print("")
pprint(soln)
```

```
               ⎛ 2⎞
sin(y) + cos⎝x ⎠ = 1

⎡    ⎛   ⎛ 2⎞    ⎞          ⎛   ⎛ 2⎞    ⎞⎤
⎣asin⎝cos⎝x ⎠ - 1⎠ + π, -asin⎝cos⎝x ⎠ - 1⎠⎦
```

The block below is similar, but instead of solving for `y`, it solves for `sin y`.

In [6]:
```
eq = Eq(sin(symbols('y')) + cos(symbols('x')**2), 1)
pprint(eq)
print("")
print("")
soln = solve(eq, sin(symbols('y')))
print(soln)
print("")
print("")
pprint(soln)
```

```
               ⎛ 2⎞
sin(y) + cos⎝x ⎠ = 1


[1 - cos(x**2)]

⎡        ⎛ 2⎞⎤
⎣1 - cos⎝x ⎠⎦
```

In other words, this says:

$$\sin y = 1 - \cos(x^2).$$

To add an assumption, like the variables are real, add the " `real = True` " expression. The block below asks `solve` to find a real solution to the equation $y^2 = -2$. Since there isn't a solution to

this equation among the real numbers, an empty solution set ([]) is returned.

In [7]:
```
#eq is an equation
eq = Eq(symbols('y', real = True)**2, -2)

# soln will be the solution
soln = solve(eq)
print(soln)
pprint(soln)
```

```
[]
[]
```

But if we don't specify real solutions, it will give solutions over complex numbers:

In [8]:
```
#eq is an equation
eq = Eq(symbols('y')**2, -2)

# soln will be the solution
soln = solve(eq)
print(soln)
pprint(soln)
```

```
[-sqrt(2)*I, sqrt(2)*I]
[-√2⋅ⅈ, √2⋅ⅈ]
```

As you might have noticed already, typing " `symbols('y')` " and such gets annoying. So, just like any data type, we can store these as python variables. In the code below, we first assign to the variable `x` the symolic variable `symbols('x')` and similarly to the variable `y` , we assign the symbolic variable `symbols('y')` . That way, when the code runs, `sympy` interprets the line:

```
eq = Eq(sin(y) + cos(x**2), 1)
```

as

```
eq = Eq(sin(symbols('y')) + cos(symbols('x')**2), 1).
```

In [9]:
```
x = symbols('x')
y = symbols('y')
eq = Eq(sin(y) + cos(x**2), 1)
pprint(eq)
print("")
print("")
soln = solve(eq, y)
print(soln)
print("")
print("")
pprint(soln)
```

```
             ⎛ 2⎞    
sin(y) + cos⎝x ⎠ = 1


[asin(cos(x**2) - 1) + pi, -asin(cos(x**2) - 1)]

⎡    ⎛   ⎛ 2⎞    ⎞           ⎛   ⎛ 2⎞    ⎞⎤
⎣asin⎝cos⎝x ⎠ - 1⎠ + π, -asin⎝cos⎝x ⎠ - 1⎠⎦
```

In principle, the names of the variables don't have to have any connection with their symbolic

representation. For example, the code below does the same thing as above (notice that the output is exactly the same), but note what names we've given the symbolic variables:

In [10]:
```
rob = symbols('x')
cara = symbols('y')
eq = Eq(sin(cara) + cos(rob**2), 1)
pprint(eq)
print("")
print("")
soln = solve(eq, cara)
print(soln)
print("")
print("")
pprint(soln)
```

```
             ⎛ 2⎞
sin(y) + cos⎝x ⎠ = 1


[asin(cos(x**2) - 1) + pi, -asin(cos(x**2) - 1)]

⎡    ⎛   ⎛ 2⎞    ⎞           ⎛   ⎛ 2⎞    ⎞⎤
⎣asin⎝cos⎝x ⎠ - 1⎠ + π, -asin⎝cos⎝x ⎠ - 1⎠⎦
```

It would be incredibly unwise to do so, but you can also do this (note that we are setting `x` to be `symbols('y')`):

In [11]:
```
y = symbols('x')
x = symbols('y')
eq = Eq(sin(x) + cos(y**2), 1)
pprint(eq)
print("")
print("")
soln = solve(eq, x)
print(soln)
print("")
print("")
pprint(soln)
```

```
             ⎛ 2⎞
sin(y) + cos⎝x ⎠ = 1


[asin(cos(x**2) - 1) + pi, -asin(cos(x**2) - 1)]

⎡    ⎛   ⎛ 2⎞    ⎞           ⎛   ⎛ 2⎞    ⎞⎤
⎣asin⎝cos⎝x ⎠ - 1⎠ + π, -asin⎝cos⎝x ⎠ - 1⎠⎦
```

Again, we get the same exact output, but the naming convention is very very bad.

Now that we have some basic understanding about `sympy` and the `Eq` class, we now turn to using `sympy` to solve some ODEs.

Below, we solve the equation: $x'(t) = x(t)$. First, we set the symolic variable `t`. The next line, `x = Function('x')` tells `sympy` that `x` represents the function whose name is `x`. (If, for example, we had done `x = Function('z')` then `x` would represent the function whose name is `z`. It's a good idea to alter the code to see what I am talking about.)

The expression `diff(x(t), t)` is the derivative of `x` with respect to `t`. So, the line:

```
deq = Eq(diff(x(t),t), x(t))
```

is the sympy `Eq` for $x'(t) = x(t)$. Then `xsoln = dsolve(deq, x(t))` says to solve the differential equation for the function `x(t)`. If you do `dsolve(deq)`, this will work on most examples (bascially, `dsolve` guesses at what we want to do) but it's probably best to be in the habit of telling `dsolve` exactly what we want.

In [12]:
```
#deq is the differential equation
t = symbols('t')
x = Function('z')

deq = Eq(diff(x(t),t), x(t))
xsoln = dsolve(deq, x(t))
pprint(xsoln)
```

```
           t
z(t) = C₁⋅e
```

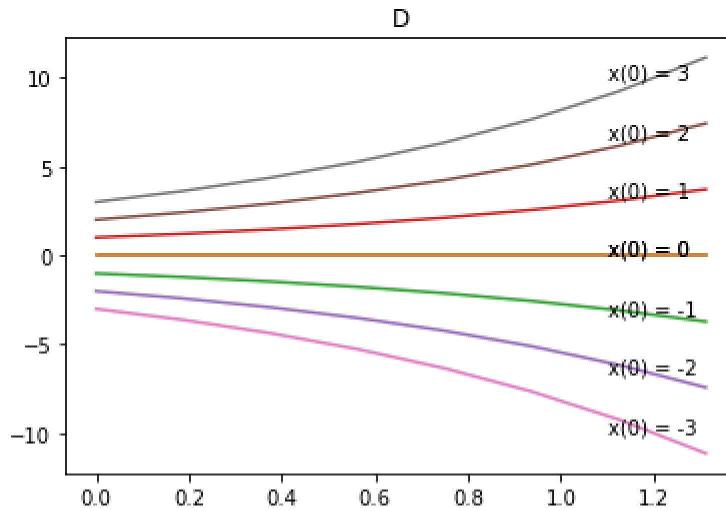
We can now solve a slightly more complicated equation: $x'(t) = ax(t)$:

In [13]:
```
#deq is the differential equation
t = symbols('t')
a = symbols('a')
x = Function('x')

deq = Eq(diff(x(t),t), a*x(t))
xsoln = dsolve(deq, x(t))
pprint(xsoln)
```

```
           a⋅t
x(t) = C₁⋅e
```

Now let's graph some of these solution curves.

In [14]:
```
import numpy as np
import matplotlib.pyplot as plt

def deqsoln(t,C):
    return C*np.exp(t)

T = np.arange(0,1.5,1.5/8)
plt.figure()
plt.title('D')
for k in range(4):
    plt.plot(T, deqsoln(T, -k))
    plt.annotate(f"x(0) = {-k}", xy = (1.1, deqsoln(1.2,-k)) )
    plt.plot(T, deqsoln(T, k))
    plt.annotate(f"x(0) = {k}", xy = (1.1, deqsoln(1.2,k)) )
```



The field mouse example we saw in class was:

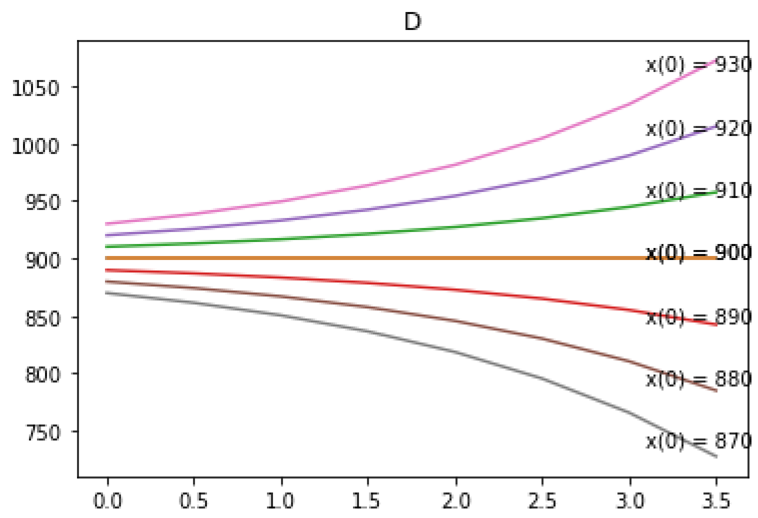
$$\frac{dp}{dt} = \frac{p - 900}{2}.$$

We also solved this in class. We will solve this now and plot some solution curves.

In [15]:
```
import numpy as np
import matplotlib.pyplot as plt
import sympy

t = symbols('t')
x = Function('x')

deq = Eq(diff(x(t),t), (x(t) - 900) / 2)
xsoln = dsolve(deq, x(t))
pprint(xsoln)
```

```
           t
           ─
           2
x(t) = C₁⋅ℯ  + 900
```

In [16]:
```
def deqsoln(t,C):
    return C*np.exp(t/2) + 900

T = np.arange(0,4, .5)
plt.figure()
plt.title('D')
for k in range(4):
    plt.plot(T, deqsoln(T, 10*k))
    plt.annotate(f"x(0) = {10*k + 900}", xy = (3.1, deqsoln(3.4,10*k)) )
    plt.plot(T, deqsoln(T, -10*k))
    plt.annotate(f"x(0) = {-10*k + 900}", xy = (3.1, deqsoln(3.4,-10*k)) )
```


D
x(0) = 930
x(0) = 920
x(0) = 910
x(0) = 900
x(0) = 890
x(0) = 880
x(0) = 870
1050
1000
950
900
850
800
750
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5