Week 2: “Nature does not care about our mathematical difficulties; She integrates empirically.”*

Object oriented programming and numerical solution of ODEs

Dr K Clough, Topics in Scientific computing, Autumn term 2023

*Albert Einstein (paraphrased!)

Plan for today

1. Object oriented programming in python - class based approach rather than functional approach

2. Revision of ordinary differential equations (ODEs)

3. How to solve ODEs numerically - explicit methods - Euler's method, solve_ivp() method in scipy

4. Tutorial: Classes for shapes and predator-prey equations

Classes

Classes are a way of "packaging things up" in a very satisfying way so things that are related are kept together in a neat way.
If your cupboards look like this, you will like classes.

Functional coding style

You have probably coded in this way up until now, and take it for granted that this is the right way to do it

```
def move_point(a_point,dx,dy) :
    xnew = a_point[0] + dx
    ynew = a_point[1] + dy
    return np.array([xnew,ynew])

def calculate_distance_between_two_points(A, B) :
    return np.sqrt((A[0] - B[0])**2.0 + (A[1] - B[1])**2.0)

# Create variables
point_P = np.array([1.0, 2.0])
print(point_P)
point_Q = np.array([3.0, 5.5])
point_Q = move_point(point_Q, 1.0, 2.0)
print(point_Q)

plt.plot(point_P[0], point_P[1], 'o', label="P")
plt.plot(point_Q[0], point_Q[1], 'o', label="Q")
plt.grid()
plt.legend();
```

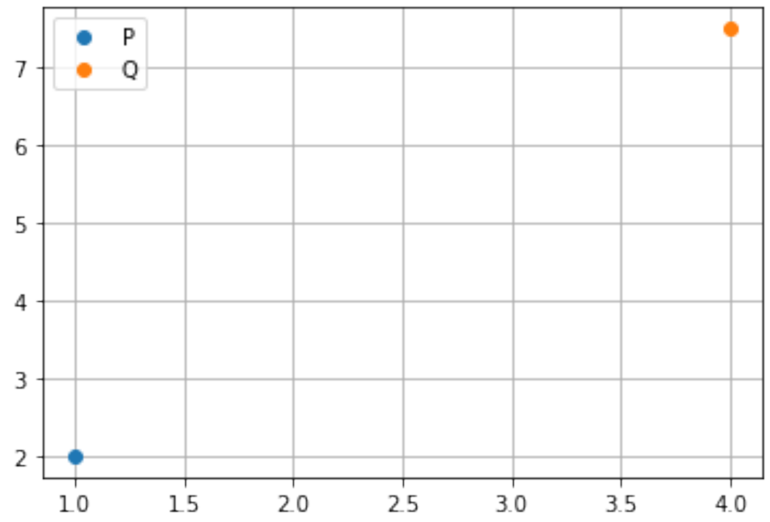
What are the values of Point P and Point Q?

Functional coding style

You have probably coded in this way up until now, and take it for granted that this is the right way to do it

```
def move_point(a_point,dx,dy) :
    xnew = a_point[0] + dx
    ynew = a_point[1] + dy
    return np.array([xnew,ynew])

def calculate_distance_between_two_points(A, B) :
    return np.sqrt((A[0] - B[0])**2.0 + (A[1] - B[1])**2.0)

# Create variables
point_P = np.array([1.0, 2.0])
print(point_P)
point_Q = np.array([3.0, 5.5])
point_Q = move_point(point_Q, 1.0, 2.0)
print(point_Q)

plt.plot(point_P[0], point_P[1], 'o', label="P")
plt.plot(point_Q[0], point_Q[1], 'o', label="Q")
plt.grid()
plt.legend();
```

```
[1. 2.]
[4.  7.5]
```



Points represented as variables
Functions act on variables

Object oriented programming - Classes

```python
# Points class

class Point :
    """
    Represents a point in a 2D space

    attributes: x, y, name

    """

    # constructor function
    # The double underscores indicate a private method or variable
    # not to be accessed outside the class (in principle)
    def __init__(self, x=0.0, y=0.0, name = ""):
        self.x = x
        self.y = y
        self.name = name

    def print_point(self) :
        print("Point ", self.name, "is", self.x, self.y)

    # Note that we don't use self here so don't need to pass it in
    # (This is a static function - it does not require an instance of the class)
    def calculate_distance_between_two_points(A, B) :
        return np.sqrt((A.x - B.x)**2.0 + (A.y - B.y)**2.0)

    def move_point(self,dx,dy) :
        self.x += dx
        self.y += dy

    def plot_point(self, ax) :
        ax.plot(self.x, self.y, 'o', label=self.name)
```

Here instead is a Point class.

Now the functions live within the class:
We call them **methods**

Now the values of the variables live within the class:
We call them **attributes**

Naming:
Classes are nouns
Classes are named in CamelCase
e.g. FluffyCat, Point, Rectangle etc

Classes - how to define a class

```
# Points class

class Point :
    """
    Represents a point in a 2D space

    attributes: x, y, name

    """

    # constructor function
    # The double underscores indicate a private method or variable
    # not to be accessed outside the class (in principle)
    def __init__(self, x=0.0, y=0.0, name = ""):
        self.x = x
        self.y = y
        self.name = name

    def print_point(self) :
        print("Point ", self.name, "is", self.x, self.y)

    # Note that we don't use self here so don't need to pass it in
    # (This is a static function - it does not require an instance of the class)
    def calculate_distance_between_two_points(A, B) :
        return np.sqrt((A.x - B.x)**2.0 + (A.y - B.y)**2.0)

    def move_point(self,dx,dy) :
        self.x += dx
        self.y += dy

    def plot_point(self, ax) :
        ax.plot(self.x, self.y, 'o', label=self.name)
```

Classes have an initialiser or "**constructor**" function that sets the key attributes

The methods can act on the attributes (but they don't have to, the main thing is that they are somehow related to how the object the class represents *behaves*)

Classes - how to use a class

```
# Some examples of using the Points class
first_point = Point(1.0,2.0,"P")
first_point.print_point()

second_point = Point(3.0,5.5,"Q")
second_point.move_point(1.0, 2.0)
second_point.print_point()
```

We make an **instance** of the class (we "instantiate" it), which we refer to as an **object**

```
# Use the static function
distance = Point.calculate_distance_between_two_points(first_point,second_point)
print("Distance is ", distance)

plt.plot()
plt.grid()
plt.xlabel("x position")
plt.ylabel("y position")
ax = plt.gcf().gca()
first_point.plot_point(ax)
second_point.plot_point(ax)
third_point.plot_point(ax)
plt.legend()
```

Note that the static method is accessed using Point.method() not object.method()

Think of this as "Hey, first point, go and print yourself!"

Classes

```
# Cat class

class FluffyCat :

    """
    Represents a fluffy cat

    Attribute: ???

    Methods: ???

    """
```

What could the Cat class attributes and methods be?

Classes

```
# Cat class

class FluffyCat :

    """
    Represents a fluffy cat

    attributes: fluffiness (int, scale of 1 to 10),
                hungriness (int, scale of 1 to 10),
                colour (string)
                name (string)
                ...

    methods:
             feed_cat()
             change_cat_name()
             brush_cat()
             ...

    """
```

Classes - a cat with a colour

```python
# Cat class

class FluffyCat :

    """
    Represents a fluffy cat

    Attribute: colour

    Methods: print the colour of the cat

    """

    cat_colours = ["black", "ginger", "pink"]

    # constructor function
    def __init__(self, colour = cat_colours[0]):
        self.colour = colour

    def print_colour(self) :
        print(self.colour)
```

```python
my_cat = FluffyCat()
my_cat.print_colour()
```

```
black
```

Classes

```
# Cat class

class FluffyCat :

    """
    Represents a fluffy cat

    Attribute: colour

    Methods: print the colour of the cat

    """

    cat_colours = ["black", "ginger", "pink"]

    # constructor function
    def __init__(self, colour = cat_colours[0]):
        self.colour = colour

    def print_colour(self) :
        print(self.colour)
```

```
my_cat = FluffyCat()
my_cat.print_colour()
```

```
black
```

```
print(cat_colours[0])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/var/folders/p9/hydj_8nx5w3c8rkwjmgvty5r0000gp/T/ipykernel_20835/868534374.py in <module>
----> 1 print(cat_colours[0])

NameError: name 'cat_colours' is not defined
```

What is wrong?

Classes

```python
# Cat class

class FluffyCat :

    """
    Represents a fluffy cat

    """

    cat_colours = ["black", "ginger", "pink"]

    # constructor function
    def __init__(self, fluffiness=10, hungriness=0, name = "", colour = cat_colours[0]):
        self.fluffiness = fluffiness
        self.hungriness = hungriness
        self.name = name
        self.colour = colour

    def print_colour(self) :
        print(self.colour)
```

```python
my_cat = FluffyCat()
my_cat.print_colour()
```

```
black
```

```python
print(FluffyCat.cat_colours[0]) # Access via the class definition without an instance of the class
print(my_cat.cat_colours[0])    # Access via the class object that has been created
```

```
black
black
```

For something common to ALL cats, I would favour the first option

Classes

```python
# Cat class

class FluffyCat :

    """
    Represents a fluffy cat

    Attribute: colour

    Methods: print the colour of the cat

    """

    cat_colours = ["black", "ginger", "pink"]

    # constructor function
    def __init__(self, colour = cat_colours[0]):
        self.colour = colour

    def print_colour(self) :
        print(self.colour)
```

```python
my_cat = FluffyCat()
my_cat.print_colour()
```

```
black
```

```python
my_cat.init(FluffyCat.cat_colours[1])
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/var/folders/p9/hydj_8nx5w3c8rkwjmgvty5r0000gp/T/ipykernel_20835/3253530536.py in <module>
----> 1 my_cat.init(FluffyCat.cat_colours[1])

AttributeError: 'FluffyCat' object has no attribute 'init'
```

What is wrong?

Classes

```
# Cat class

class FluffyCat :

    """
    Represents a fluffy cat

    Attribute: colour

    Methods: print the colour of the cat

    """

    cat_colours = ["black", "ginger", "pink"]

    # constructor function
    def __init__(self, colour = cat_colours[0]):
        self.colour = colour

    def print_colour(self) :
        print(self.colour)
```

```
my_cat = FluffyCat()
my_cat.print_colour()
```

```
black
```

```
my_cat.colour = FluffyCat.cat_colours[1]
my_cat.print_colour()
```

```
ginger
```

The underscores __ mean that init is a *private* method that should not be accessed outside the class. Instead we can modify directly the attribute.

Classes

An even better solution is to write a *modifier* function that allows you to change the colour, to which you can add asserts and other conditions to make sure it is sensible and prevent user error

```
# Cat class

class FluffyCat :

    """
    Represents a fluffy cat

    Attribute: colour

    Methods: print the colour of the cat, change colour of cat

    """

    cat_colours = ["black", "ginger", "pink"]

    # constructor function
    def __init__(self, colour = cat_colours[0]):
        self.colour = colour

    def print_colour(self) :
        print(self.colour)

    def change_colour(self, new_colour) :
        assert new_colour in self.cat_colours, 'Need to specify one of the allowed cat colours'
        self.colour = new_colour
```

```
my_cat = FluffyCat()
my_cat.change_colour(FluffyCat.cat_colours[2])
my_cat.print_colour()

my_cat.change_colour("green") #Returns an error
```

```
pink
```

Classes

TOP TIP:
A useful strategy for more complicated classes is to keep the init minimal and let the user add attributes later

```python
# CatHome class

class CatHome :

    """

    Attributes : some number of cats >=2

    Methods: write the cat colours

    """

    # constructor function
    def __init__(self):
        self.list_of_cats = []
        self.num_cats = 0
        self._is_defined = False
        print("Class requires addition of at least 2 cats to be defined")

    def add_cat(self, new_cat) :

        #Add new cat to the list of cats
        self.list_of_cats.append(new_cat)
        self.num_cats += 1

        if(self.num_cats >= 2) :
            self._is_defined = True
            print("Cat Home definition complete!")
        else :
            print("Need to add another cat to make a cat home")


    def print_cats_colours(self) :
        assert self._is_defined, "Insufficient cats added, Cat Home not defined!"

        for cat in self.list_of_cats :
            cat.print_colour()
```

Classes

```
ginger_cat = FluffyCat(FluffyCat.cat_colours[1])
pink_cat = FluffyCat(FluffyCat.cat_colours[2])
new_cat = ginger_cat + pink_cat
new_cat.print_colour()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/p9/hydj_8nx5w3c8rkwjmgvty5r0000gp/T/ipykernel_21077/3432849571.py in <module>
      1 ginger_cat = FluffyCat(FluffyCat.cat_colours[1])
      2 pink_cat = FluffyCat(FluffyCat.cat_colours[2])
----> 3 new_cat = ginger_cat + pink_cat
      4 new_cat.print_colour()

TypeError: unsupported operand type(s) for +: 'FluffyCat' and 'FluffyCat'
```

What is wrong?

Classes

Need to tell python how to add cats

There is a specific syntax for each of the arithmetic and logical operators +, -, >, and, or etc to allow you to override them for your new type (ie, your class)

```
# Cat class

class FluffyCat :

    """
    Represents a fluffy cat

    Attribute: colour

    Methods: print the colour of the cat

    """

    cat_colours = ["black", "ginger", "pink"]

    # constructor function
    def __init__(self, colour = cat_colours[0]):
        self.colour = colour

    def __add__(self, other) :
        min_colour_index = min(self.cat_colours.index(self.colour),
                               self.cat_colours.index(other.colour))
        baby_cat_colour = self.cat_colours[min_colour_index]
        baby_cat = FluffyCat(baby_cat_colour)
        return baby_cat

    def print_colour(self) :
        print(self.colour)
```

```
ginger_cat = FluffyCat(FluffyCat.cat_colours[1])
pink_cat = FluffyCat(FluffyCat.cat_colours[2])
new_cat = ginger_cat + pink_cat
new_cat.print_colour()
```

```
ginger
```

Inheritance

A Lion ***is a*** Cat

Therefore we want to inherit the Cat properties into the Lion class

What is printed here?

```python
# Lion class

class Lion(FluffyCat):

    """
    Represents a Lion, which is a FluffyCat

    Attribute: colour, strength

    Methods: increase the strength of the Lion
    (plus inherit all of the FluffyCat methods)

    """

    def __init__(self, colour = FluffyCat.cat_colours[0], strength=10) :
        self.colour = colour
        self.strength = strength

    def increase_strength(self, increment) :
        self.strength += increment

my_lion = Lion()
my_lion.print_colour()
my_lion.increase_strength(20)
print(my_lion.strength)
```

Inheritance

A Lion ***is a*** Cat

Therefore we want to inherit the Cat properties into the Lion class

```
# Lion class

class Lion(FluffyCat):

    """
    Represents a Lion, which is a FluffyCat

    Attribute: colour, strength

    Methods: increase the strength of the Lion
    (plus inherit all of the FluffyCat methods)

    """

    def __init__(self, colour = FluffyCat.cat_colours[0], strength=10) :
        self.colour = colour
        self.strength = strength

    def increase_strength(self, increment) :
        self.strength += increment

my_lion = Lion()
my_lion.print_colour()
my_lion.increase_strength(20)
print(my_lion.strength)
```

```
black
30
```

Plan for today

1. ~~Object oriented programming in python - class based approach rather than functional approach~~

2. Revision of ordinary differential equations (ODEs)

3. How to solve ODEs numerically - explicit methods - Euler's method, solve_ivp() method in scipy

4. Tutorial: Classes for shapes and predator-prey equations

Ordinary differential equations

$$\frac{d^2x}{dt^2} = x^2 - x - 1$$

$$\frac{dx}{dt} = x^2 - xy - 1\ , \quad \frac{dy}{dt} = 2x + y$$

What is:

1. The dependent variable(s)?
2. The independent variable(s)?
3. The order?
4. The dimension?

Ordinary differential equations

$$\frac{d^2x}{dt^2} = x^2 - x - 1$$

$$\frac{dx}{dt} = x^2 - xy - 1\ , \quad \frac{dy}{dt} = 2x + y$$

1. The dependent variables are x and y
2. The independent variable is t (only **one** in an ODE)
3. The first is second order, the second is first order (look at the highest derivative order)
4. The dimension of the first is one (only one dependent variable x) and the second is dimension two (x and y)

Ordinary differential equations

$$\frac{d^2x}{dt^2} = x^2 - x - 1$$

$$\frac{dx}{dt} = x^2 - xy - 1 \ , \quad \frac{dy}{dt} = 2x + y$$

How do we know:

1. If it is autonomous?
2. If it is linear?

Ordinary differential equations

$$\frac{d^2x}{dt^2} = x^2 - x - 1$$

$$\frac{dx}{dt} = x^2 - xy - 1 \ , \quad \frac{dy}{dt} = 2x + y$$

1. It is autonomous if the functions and coefficients do not have a dependence on t (except in the derivatives)

2. It is linear if the coefficients of x and y and their derivatives are constants

Ordinary differential equations

What about this one?

$$\frac{d^2x}{dt^2} + \frac{dx}{dt} + x^2 + x - 1 = \sin(t)$$

Ordinary differential equations

One independent variable t so ODE not PDE

One dependent variable x so dimension 1

$$\frac{d^2x}{dt^2} + \frac{dx}{dt} + x^2 + x - 1 = \sin(t)$$

Second order

Non linear

Not autonomous

Ordinary differential equations

What do these things tell us PHYSICALLY?

$$\frac{d^2x}{dt^2} + \frac{dx}{dt} + x^2 + x - 1 = \sin(t)$$

Ordinary differential equations

One independent variable t - evolution depends on time only, not (e.g.) space and time

Only one variable describes the system, e.g. the x position rather than x and y position

$$\frac{d^2x}{dt^2} + \frac{dx}{dt} + x^2 + x - 1 = \sin(t)$$

Second order - need 2 boundary conditions to solve system / know full state

Non linear - solutions cannot be superposed - small changes in the variable may have large effects

Some kind of forcing function - the physical scenario is changing over time

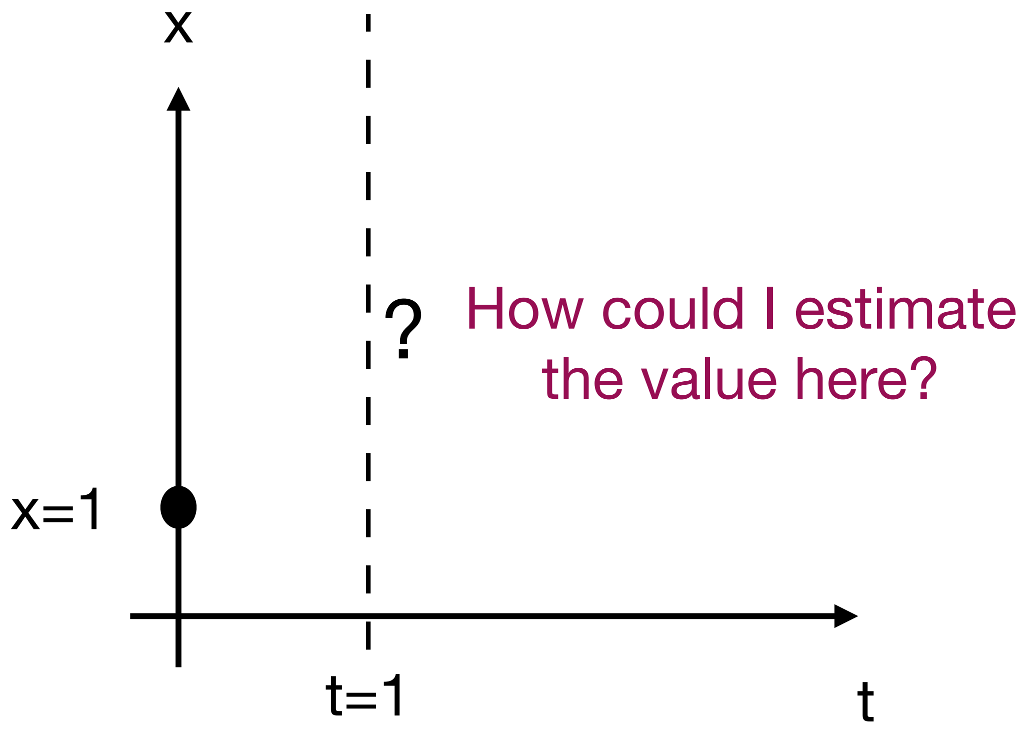
Plan for today

1. ~~Object oriented programming in python - class based approach rather than functional approach~~
2. ~~Revision of ordinary differential equations (ODEs)~~
3. How to solve ODEs numerically - explicit methods - Euler's method, solve_ivp() method in scipy
4. Tutorial: Classes for shapes and predator-prey equations

How to solve ODEs numerically?

$$\frac{dx}{dt} = x^2 + x - 1$$

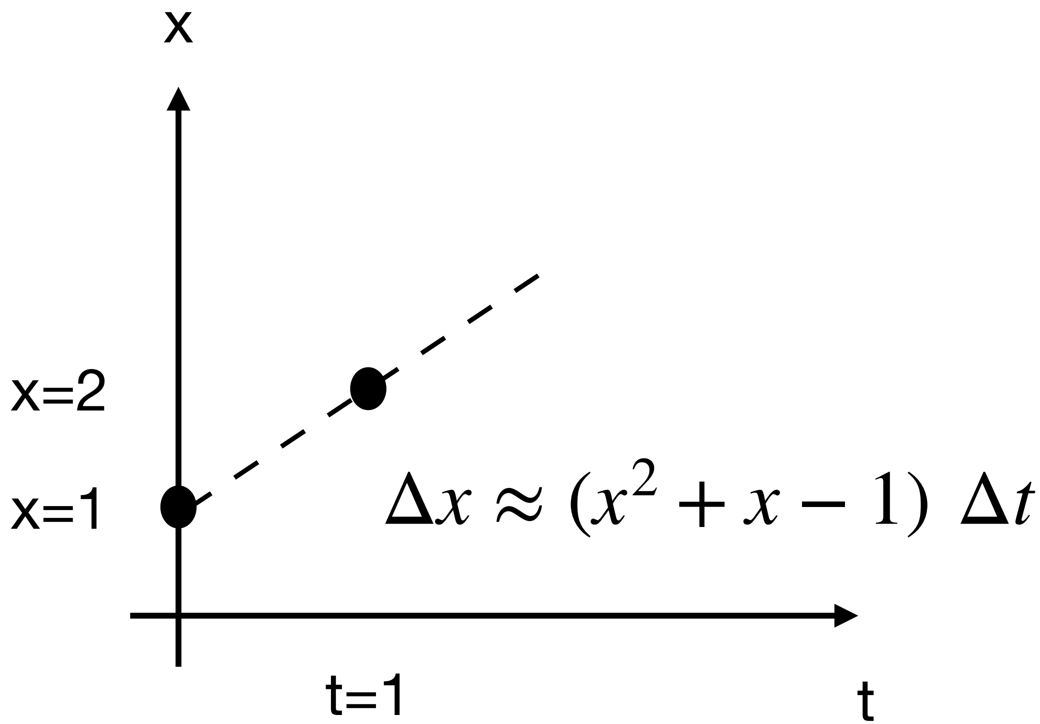
$$x(t=0) = 1$$



Euler's method

$$\frac{dx}{dt} = x^2 + x - 1$$

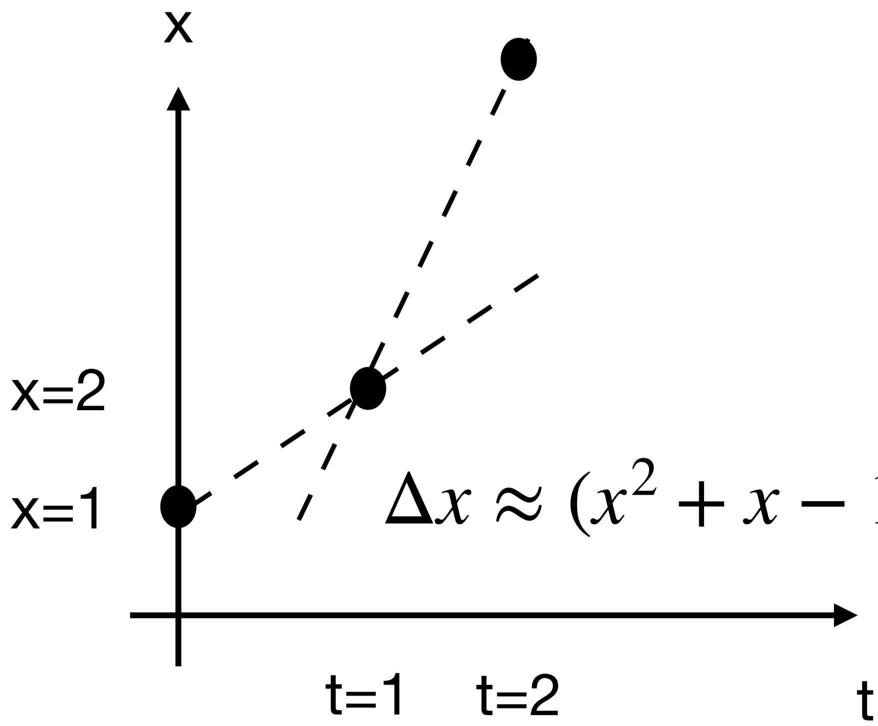
$$x(t = 0) = 1$$



Euler's method

$$\frac{dx}{dt} = x^2 + x - 1$$

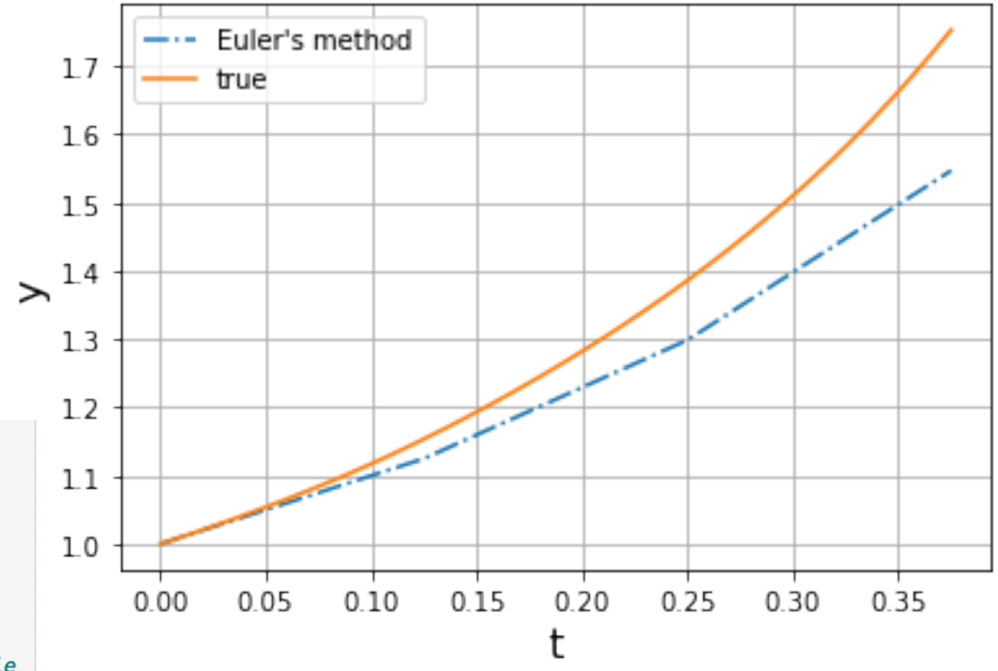
$$x(t = 0) = 1$$



$$\Delta x \approx (x^2 + x - 1)\ \Delta t$$

Euler’s method

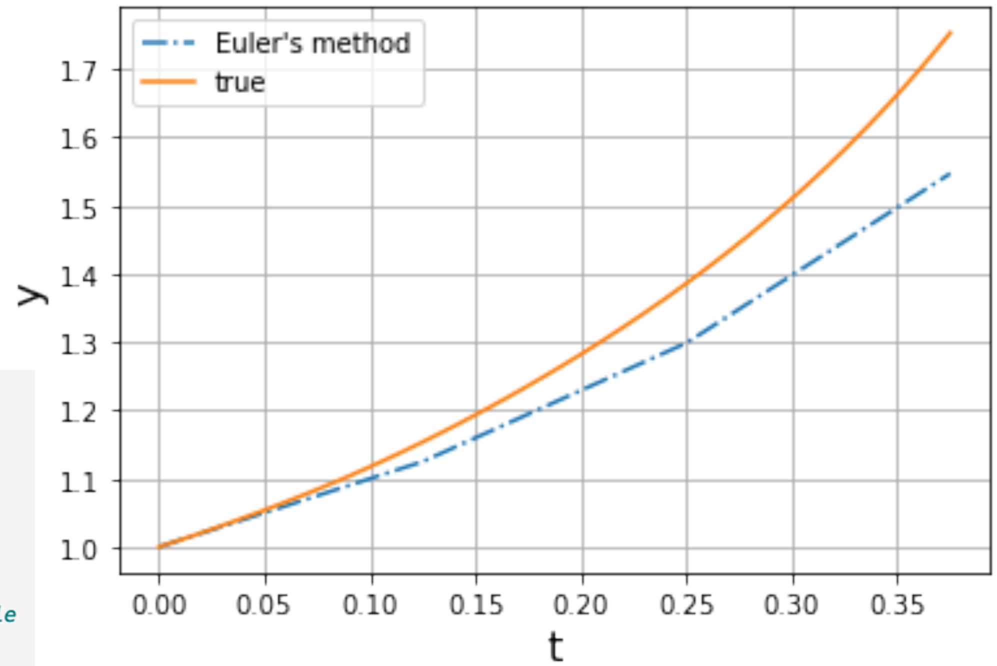
```
# Note that the function has to take t as the first argument and y as the second
def calculate_dydt(t, y):
    """Returns the gradient dy/dt for the given function"""
    dydt = y*y + y - 1
    return dydt

max_time = 0.5
N_time_steps = 4
delta_t = max_time / N_time_steps
t_solution = np.linspace(0.0, max_time, N_time_steps+1) # values of independent variable
y0 = np.array([1.0]) # an initial condition, y(0) = y0

# Euler's method
# increase the number of steps to see how the solution changes
y_solution = np.zeros_like(t_solution)
y_solution[0] = y0
for itime, time in enumerate(t_solution) :
    if itime > 0 :
        dydt = calculate_logistic_dydt(time, y_solution[itime-1])
        y_solution[itime] = y_solution[itime-1] + dydt * delta_t

plt.plot(t_solution, y_solution, '-.',label="Euler's method")
```



How can I reduce the error here?

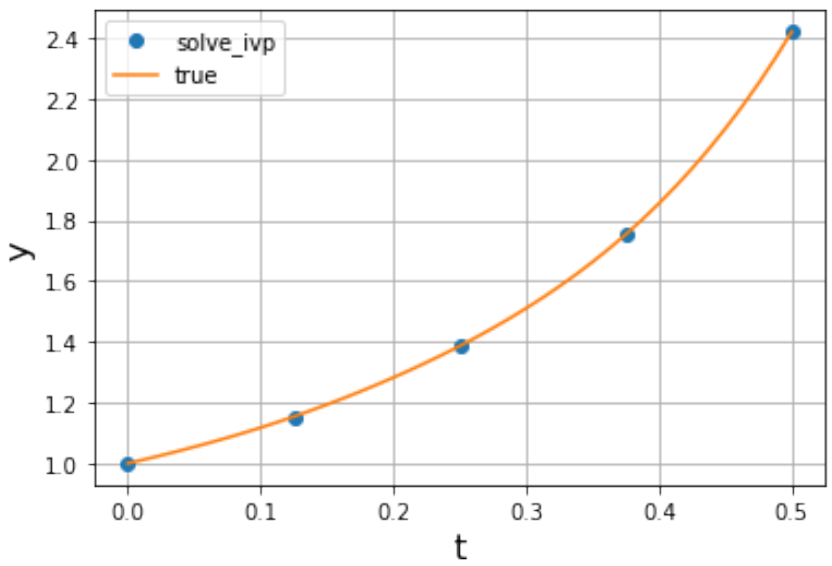
Euler’s method

```
# Note that the function has to take t as the first argument and y as the second
def calculate_dydt(t, y):
    """Returns the gradient dy/dt for the given function"""
    dydt = y*y + y - 1
    return dydt

max_time = 0.5
N_time_steps = 4
delta_t = max_time / N_time_steps
t_solution = np.linspace(0.0, max_time, N_time_steps+1) # values of independent variable
y0 = np.array([1.0]) # an initial condition, y(0) = y0

# Euler's method
# increase the number of steps to see how the solution changes
y_solution = np.zeros_like(t_solution)
y_solution[0] = y0
for itime, time in enumerate(t_solution) :
    if itime > 0 :
        dydt = calculate_dydt(time, y_solution[itime-1])
        y_solution[itime] = y_solution[itime-1] + dydt * delta_t

plt.plot(t_solution, y_solution, '-.',label="Euler's method")
```



The global error is related to the step size delta_t, so can reduce it, or use a better method to estimate the gradient (more next week)

Integration with scipy solve_ivp()



Syntax:

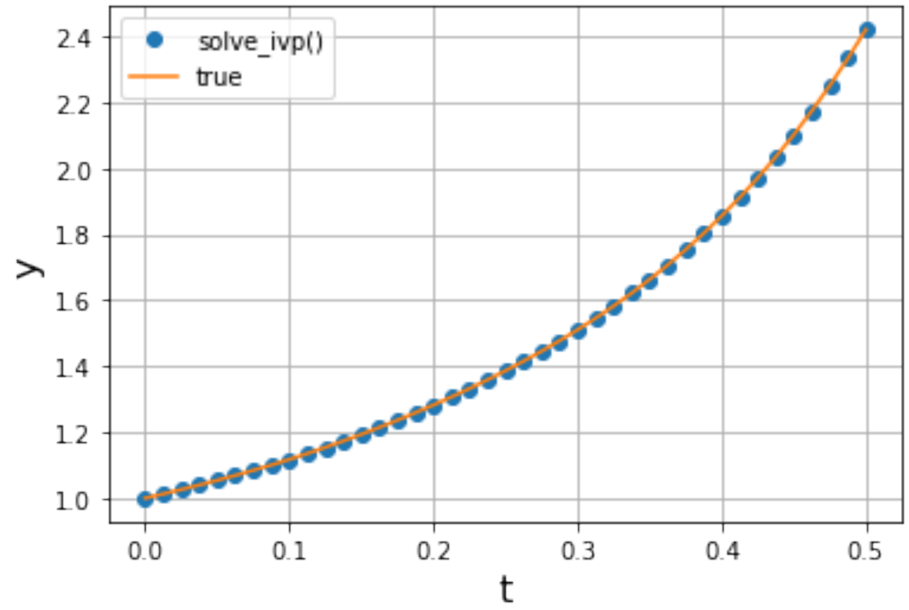
solve_ivp(function_for_derivative,
independent_var_range
dependent_var_initial_condition)

```
# Now solve using solve_ivp()
max_time = 0.5
N_time_steps = 4
delta_t = max_time / N_time_steps
t_solution = np.linspace(0.0, max_time, N_time_steps+1) # values of independent variable
y0 = np.array([1.0]) # an initial condition, y(0) = y0

solution = solve_ivp(calculate_dydt, [0,max_time], y0, t_eval=t_solution)
plt.plot(solution.t, solution.y[0], 'o', label="solve_ivp")
```

I asked for the solution at only 5 points - this is set by “t_eval”

Integration with scipy solve_ivp()

If I ask for more points things look better.
The step size we ask for in t_eval does NOT determine the step size used for solving the ODE



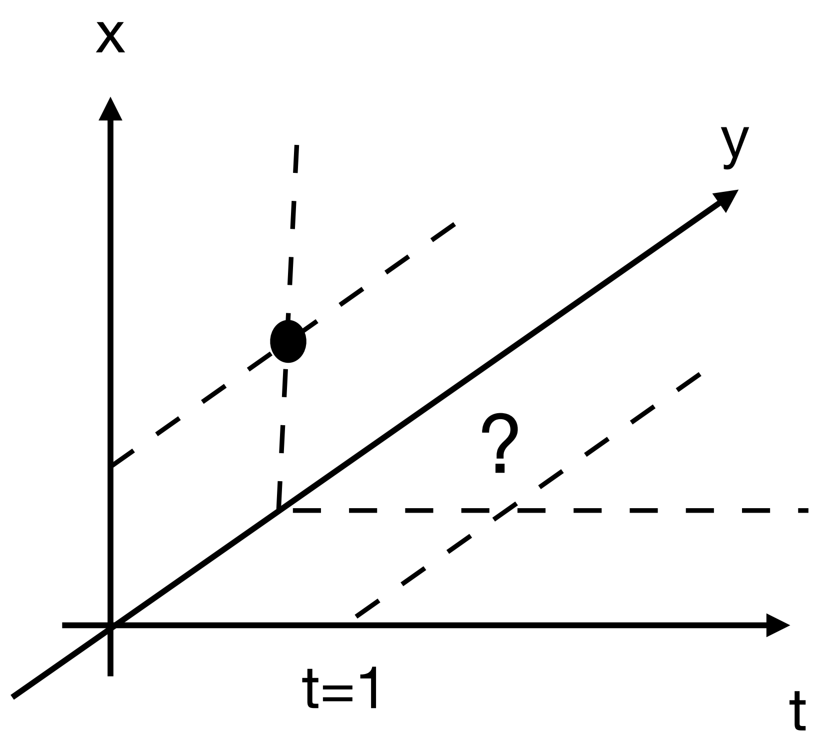

```
# Now ask for a higher number of points
t_solution = np.linspace(0.0, max_time, 10*N_time_steps+1)
solution = solve_ivp(calculate_dydt, [0,max_time], y0, t_eval=t_solution)
plt.plot(solution.t, solution.y[0], 'o', label="solve_ivp()")
```

What about dimension greater than 1?

$$\frac{dx}{dt} = 2x - xy$$

$$\frac{dy}{dt} = -y + 3xy$$

$$x(t=0) = 1$$
$$y(t=0) = 2$$

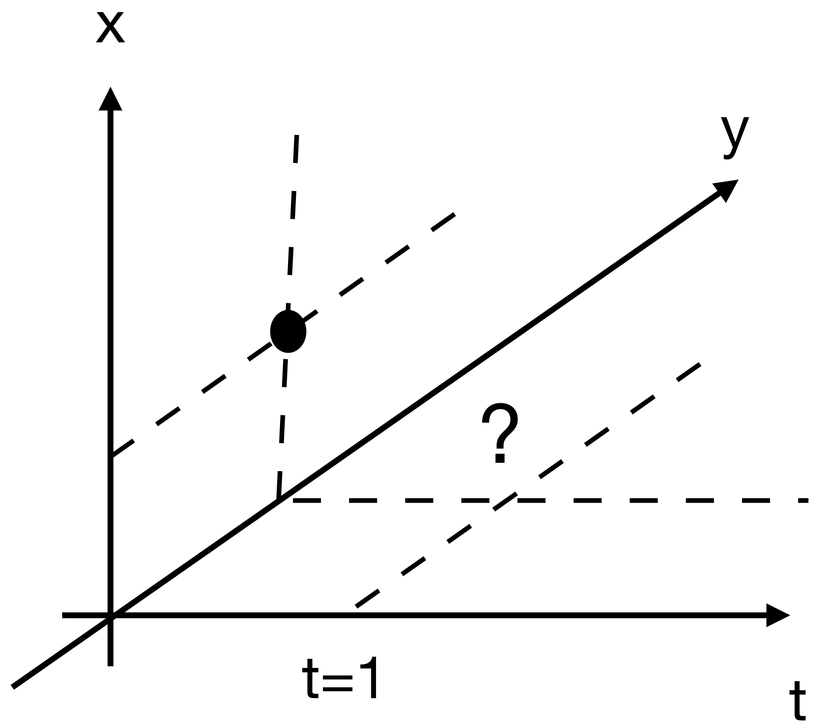


Think of it as evolution of a vector

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)$$

What is f?

$$\begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$



solve_ivp() needs to output and input vectors

```
# Note that the function has to take t as the first argument and y as the second
def calculate_predator_prey_dydt(t, y):
    """Returns the gradient dy/dt for the predator-prey equations"""

    # Set the values of the ecosystem
    a = 0.1
    b = 0.2
    c = 0.1
    d = 0.2

    #Just for readability
    predators = y[0]
    prey = y[1]

    dydt = np.zeros_like(y)
    dydt[0] = a * predators - b * predators * prey #This is the equation for x, predators
    dydt[1] = -c * prey + d * predators * prey #This is the equation for y, prey

    return dydt
```

Plan for today

1. ~~Object oriented programming in python - class based approach rather than functional approach~~

2. ~~Revision of ordinary differential equations (ODEs)~~

3. ~~How to solve ODEs numerically - explicit methods - Euler's method, solve_ivp() method in scipy~~

4. Tutorial: Classes for shapes and predator-prey equations

This week's tutorial - part 1

```python
class Point :
    """
    Represents a point in a 2D space

    attributes: x, y, name

    """

    # constructor function
    # The double underscores indicate a private method or variable
    # not to be accessed outside the class (in principle)
    def __init__(self, x=0.0, y=0.0, name = ""):
        self.x = x
        self.y = y
        self.name = name
        self.__private_variable = 42

    def __add__(self, other) :
        new_point = Point(self.x+other.x, self.y+other.y)
        return new_point

    def print_point(self) :
        print("Point ", self.name, "is", self.x, self.y)

    # Note that we don't use self here so don't need to pass it in
    # (This is a static function - it does not require an instance of the class)
    def calculate_distance_between_two_points(A, B) :
        return np.sqrt((A.x - B.x)**2.0 + (A.y - B.y)**2.0)

    def move_point(self,dx,dy) :
        self.x += dx
        self.y += dy

    def plot_point(self, ax) :
        ax.plot(self.x, self.y, 'o', label=self.name)

    def update_name(self, new_name) :
        self.name = new_name
```

I have given you a Point class, you need to make a Rectangle class

This week's tutorial - part 2 (to continue next week)

```python
# Solve the 1d logistic equation from class
from scipy.integrate import solve_ivp

# Note that the function has to take t as the first argument and y as the second
def calculate_logistic_dydt(t, y):
    """Returns the gradient dx/dt for the logistic equation"""
    dydt = y*(1 - y)
    return dydt

max_time = 10.0
N_time_steps = 25
delta_t = max_time / N_time_steps
t_solution = np.linspace(0.0, max_time-delta_t, N_time_steps) # values of independent variable
y0 = np.array([0.5]) # an initial condition, y(0) = y0, note it needs to be an array

solution = solve_ivp(calculate_logistic_dydt, [0,max_time], y0,
                        method='RK45', t_eval=t_solution)

plt.grid()
plt.xlabel("t", fontsize=16)
plt.ylabel("y", fontsize=16)
plt.plot(solution.t, solution.y[0], 'o', label="solve_ivp")

# Now do it the "cheap" way
# increase the number of steps to see how the solution changes
y_solution = np.zeros_like(t_solution)
y_solution[0] = y0
for itime, time in enumerate(t_solution) :
    if itime > 0 :
        dydt = calculate_logistic_dydt(time, y_solution[itime-1])
        y_solution[itime] = y_solution[itime-1] + dydt * delta_t

plt.plot(t_solution, y_solution, '-.',label="cheap method")

# Now plot the true solution
A = 1.0/y0 - 1.0
y_true = 1.0 / (1.0 + A * np.exp(-t_solution))
plt.plot(t_solution, y_true, '-', label="true solution")
plt.legend(loc='best');
```

I have given you a dimension 1 ODE, you need to solve a dimension 2 ODE (the predator-prey equations)