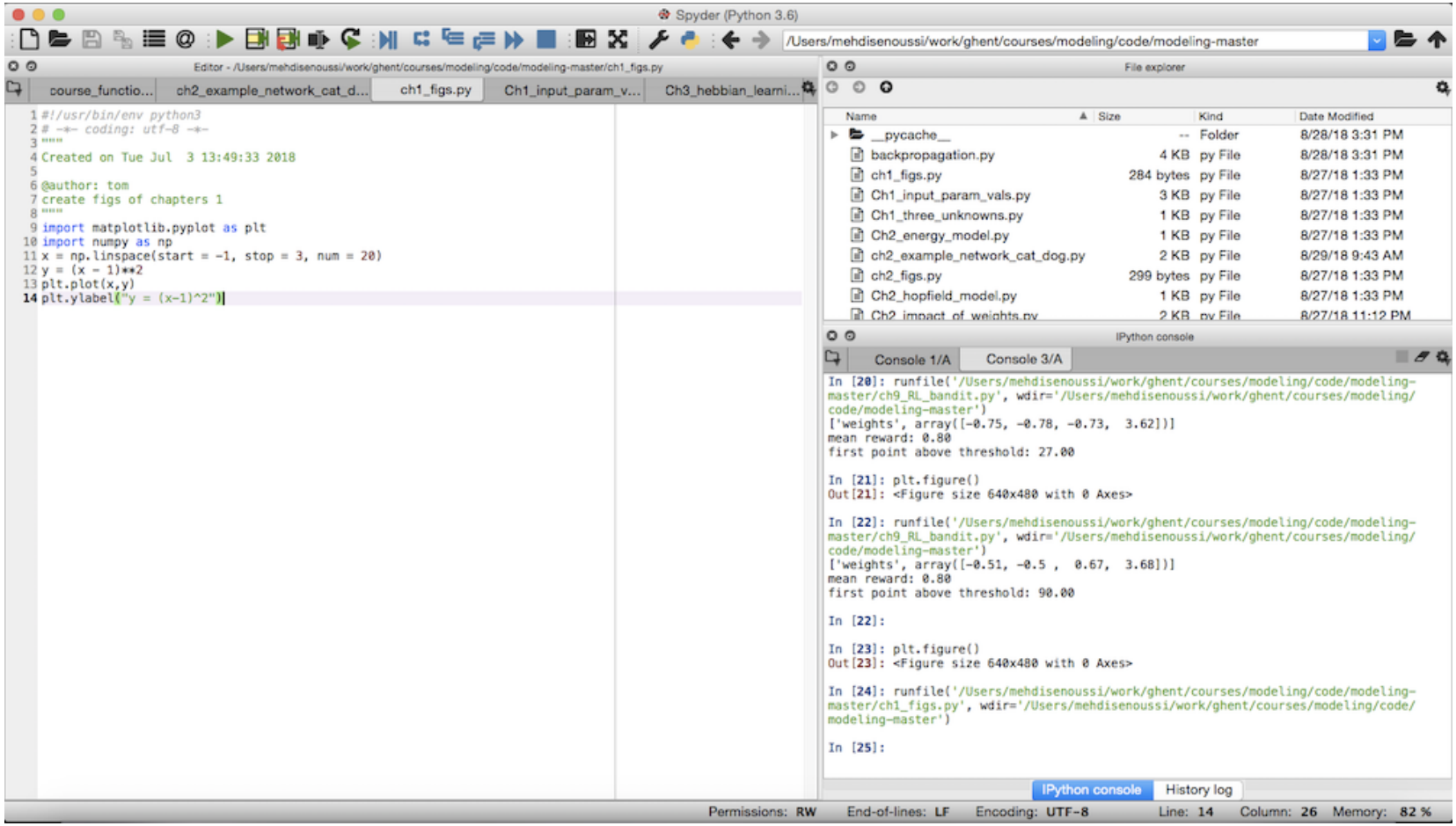
Modelling of Cognitive Processes

Intro to practical sessions

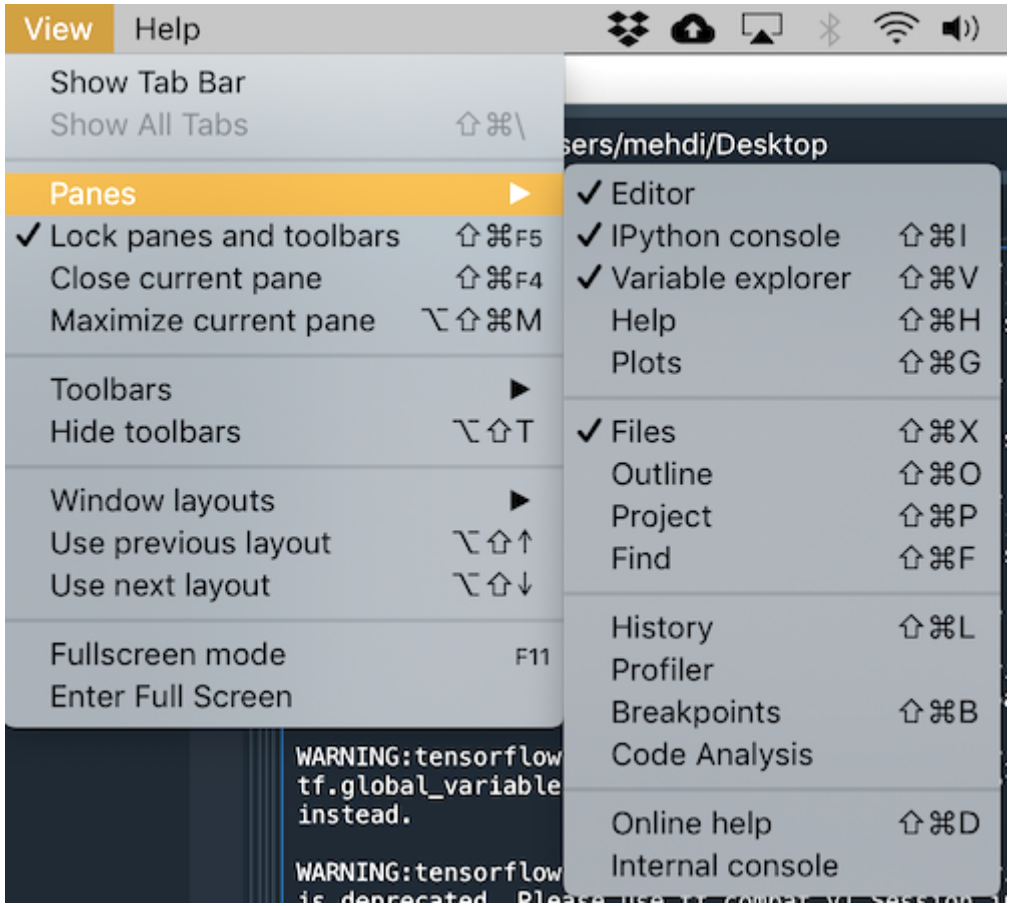
Lesson 02
29/09/2020 Mehdi Senoussi

Introduction

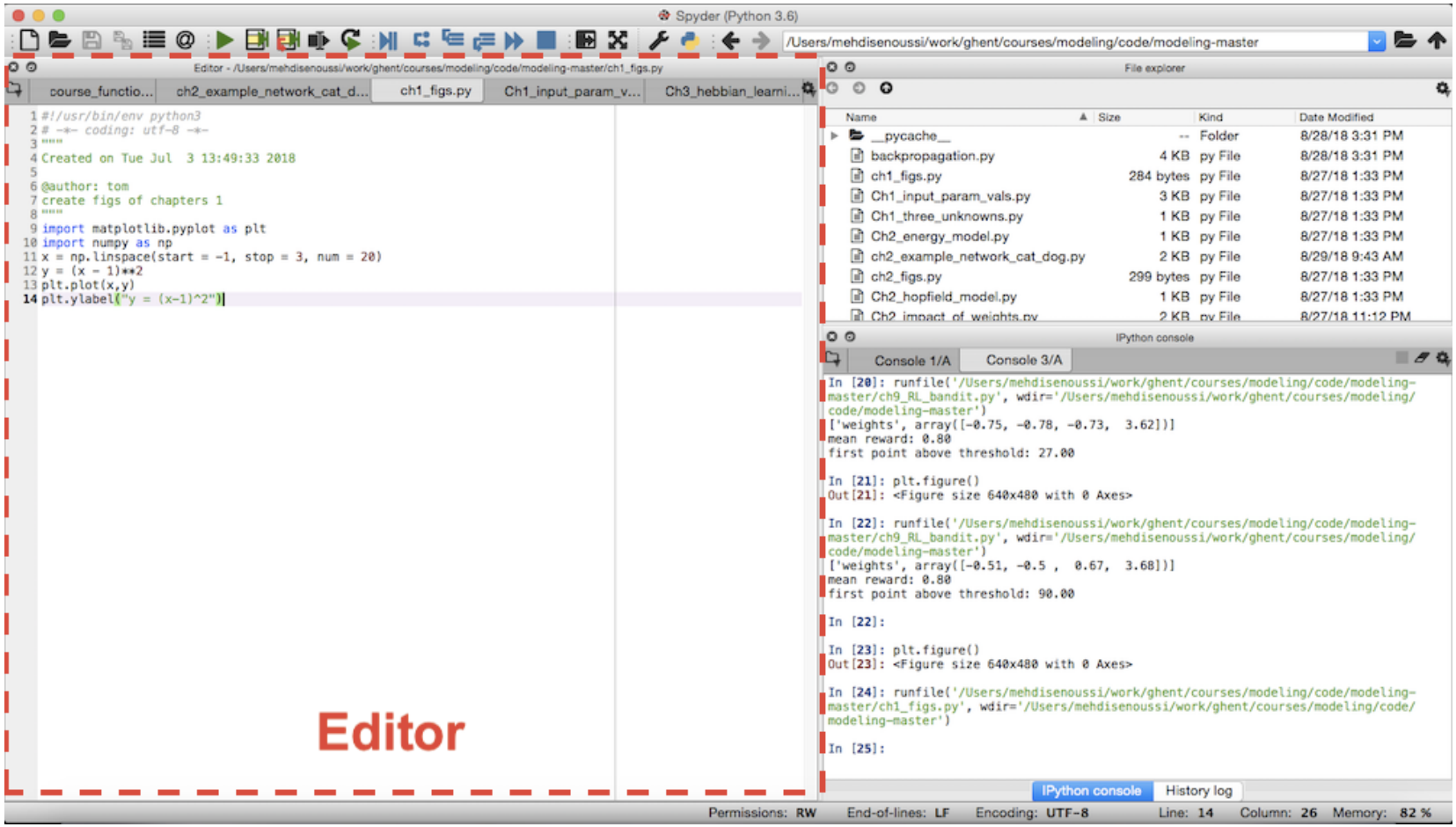
Introduction: Spyder



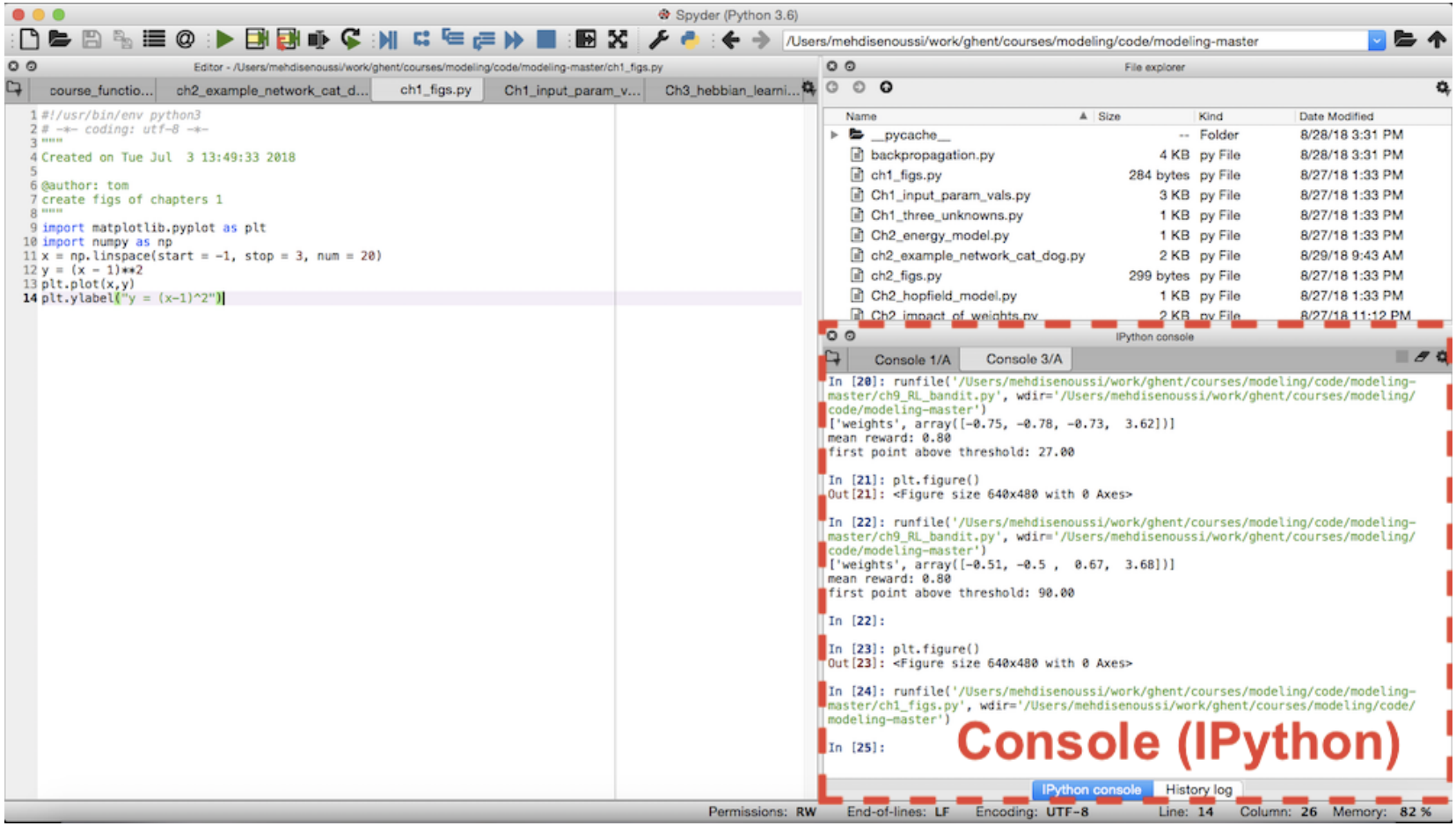
Introduction: Spyder



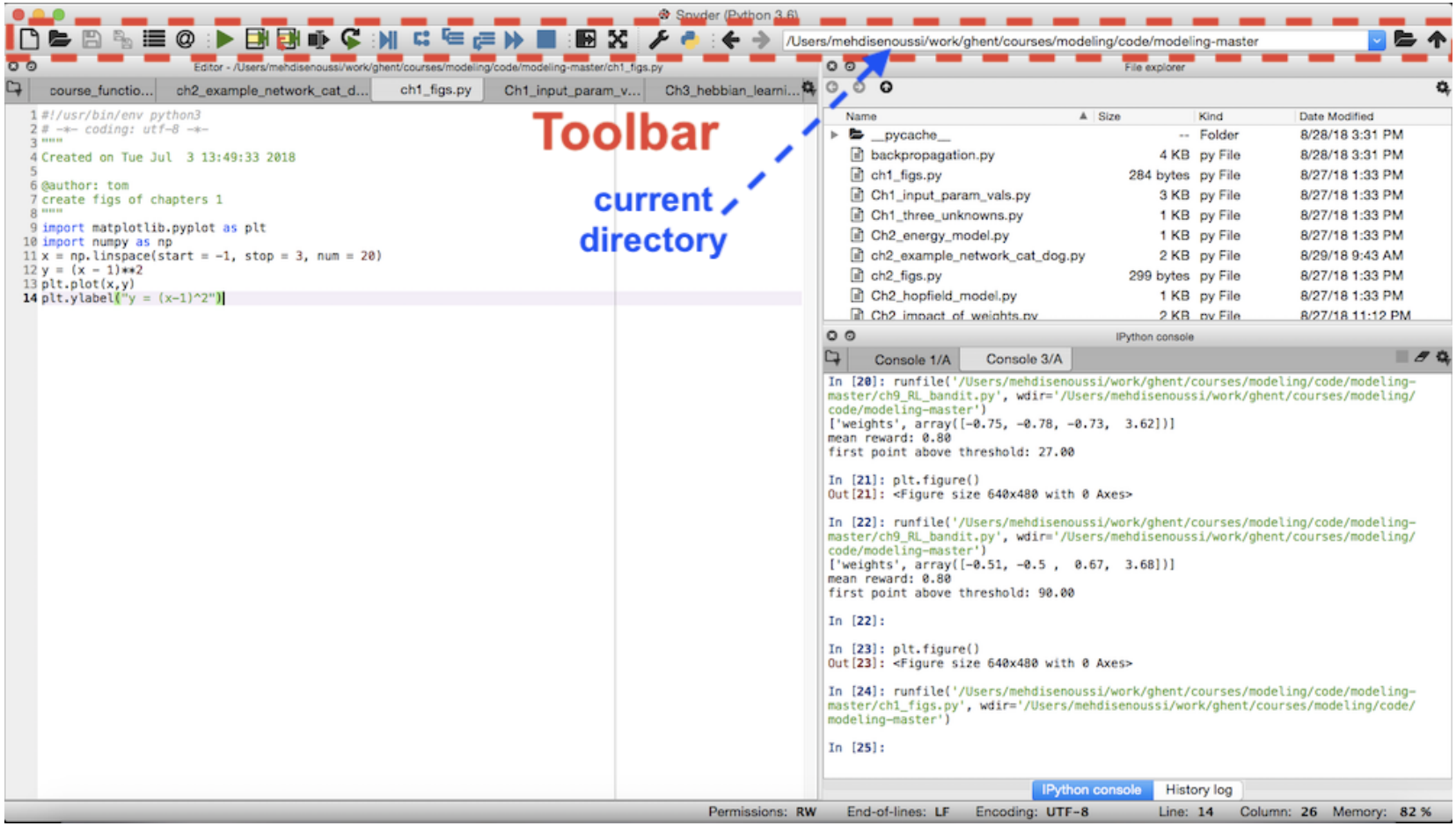
Introduction: Spyder



Introduction: Spyder



Introduction: Spyder



Introduction: Python (refresher)

Some useful function

`(from ...) import ... (as ...)`

Imports a module (optionally from a package/library) to use its functions
Can also give it a name to easily use it in your code (e.g. **import numpy as np**)

`np.arange(x, (y, step))`

Generates array of numbers from 0 to (**x**-1). If **x** **and** **y** are specified generates array of numbers between **x** and (**y**-1). **step** is the spacing between values.

`np.linspace(start, stop, num)`

Generates an array of **num** numbers between **start** and **stop**

`np.sum(x (, axis))`

Sums all the values in **x**. If you want to sum only one dimension, use **axis**

```
for i in np.arange(x): / while i < x:
    ...
```

Introduction: Python (refresher)

Some useful function

`np.load/np.save/np.savez()`

Loads/saves some variables from/to a file

```
def ...():
    ...
```

Create a function

`arr = np.array([...])`

Create an array

`my_list = [] / my_list.append(...)`

Create a list / add an element to the list

Introduction: Python (refresher)

Using `numpy` and `scipy`

Some useful function

`np.exp(x, (y, step))` – Computes the exponential of **x**

`np.log(x)/np.log10(x)` – Computes the logarithm (at the specified base) of **x**

`x**Pow` – Computes **x** to the power **Pow**

`np.random.random/randint/randn/…()`

Some random number generating functions (e.g. **randint** generates random integers)

Introduction: How to represent data graphically (Matplotlib)

Useful explanation on what are Matplotlib and PyPlot:

```
from matplotlib import pyplot as plt
```

Some useful function

`plt.figure()` – Creates a figure to plot stuff in

`plt.plot(x, y)` – Plots simple stuff (points, curves)

`plt.imshow(x)` – Plots 2-D arrays (e.g. correlation or covariance matrix)

`plt.title('This is my title')` – Puts a title on your figure

`plt.xlabel/plt.ylabel('label')` – Label your axes

`plt.xlim/plt.ylim(min, max)` – Manages the range shown in your figure

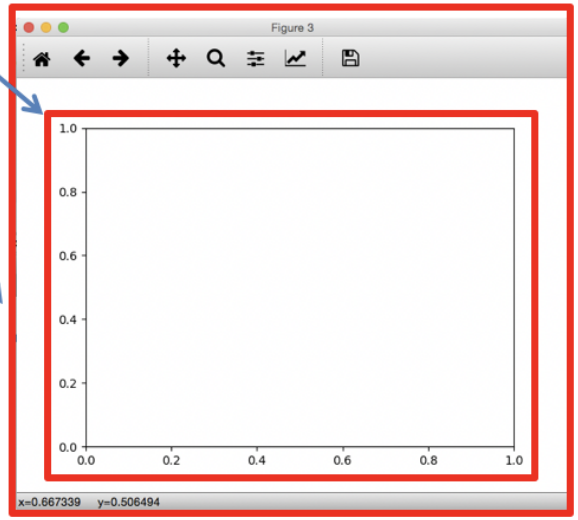
Introduction: How to represent data graphically (Matplotlib)

Useful explanation on what are Matplotlib and PyPlot:

```
from matplotlib import pyplot as plt
```

Some useful function

```
fig, axes = plt.subplots(nrows=1, ncol=1)
```
– Creates figure and plot(s)



Derivative

Derivative

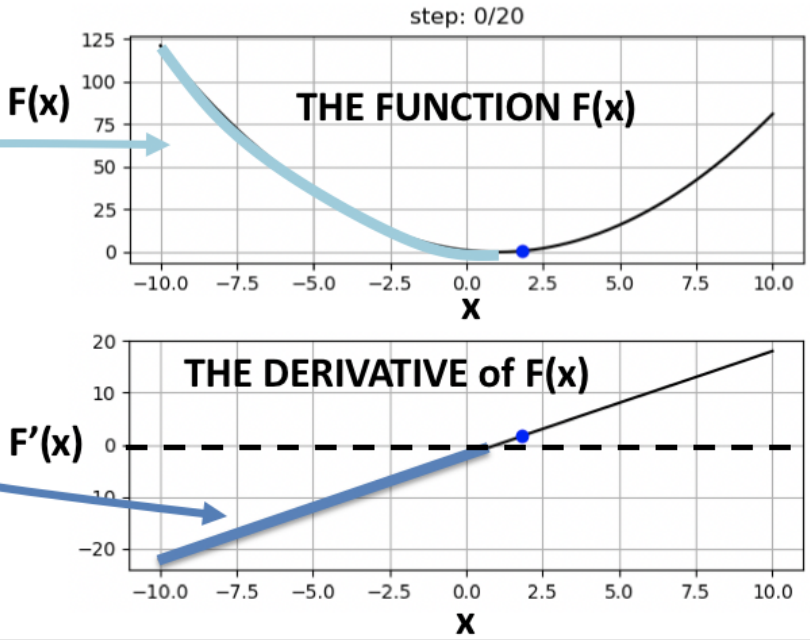
Implement the function from chapter 1

- Open the script `ch1_gradient_descent_der_solution.py`
- Run it using the “play” button on Spyder

- A plot will appear and you can go through the optimization by hitting a key or clicking

Reminder on derivatives:

This is negative because **this** goes down



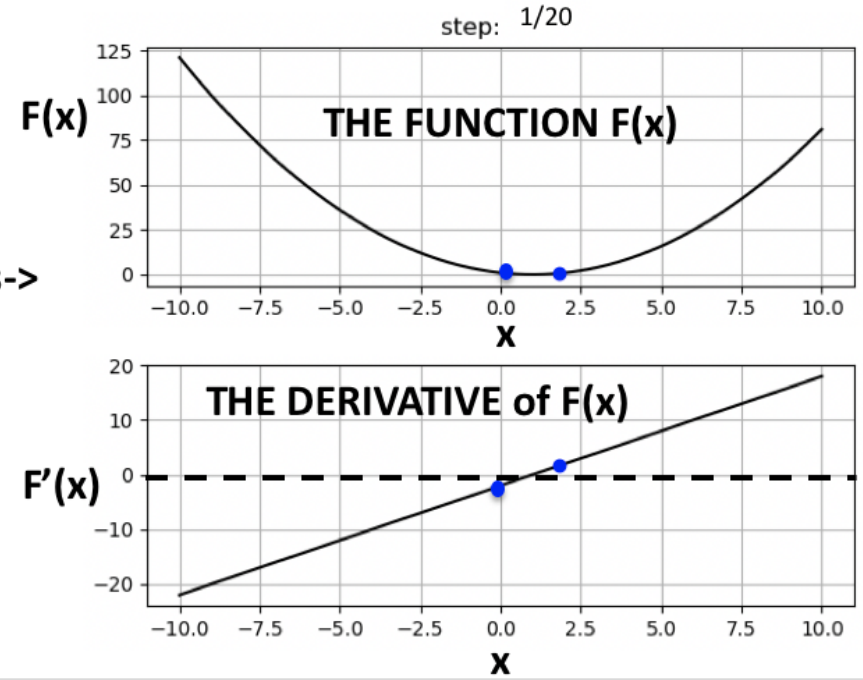
Derivative

Implement the function from chapter 1

- Open the script `ch1_gradient_descent_der_solution.py`
- Run it using the “play” button on Spyder

- A plot will appear and you can go through the optimization by hitting a key or clicking

Click/key press shows next optimization point on both plots->



Let's descend the gradient

Let's descend the gradient

Implement the function from chapter 1

- Open the script `ch1_gradient_descent_der_solution.py`

- **Exercise 1**:
  - **A.** How does the starting point affect the optimization?
    - **Line 48**: change it to see what happens in different starting situations
      - What happens when it is very far from the minima?
      - What happens when it is very close to the minima?

  - **B.** Where is the step size parameter set?
    - **Line 41**: change it to see what happens
      - What happens when there is a very very small step size?
      - What happens when the step size is very large?

Let's descend the gradient

Going further!

- Open the script `ch1_gradient_descent_der_solution.py`

- **Exercise 2**:
  Change to a function of **your choice** (AND ITS DERIVATIVE!) and optimize again.
  - Let's start with our function right now: $f(x) = (x - 1)^2$ **->** $f(x) = x^2 + 2x + 1$

  - For any polynomial f(x), i.e. something that has the form:

$$P(X) = \sum_{0 \leq i \leq n} a_i X^i + \boldsymbol{c}$$ (e.g.: $0.6*x^3 + 5*x^2 + 2$)

Its derivative is:

$$\mathrm{d}P(X) = \sum_{1 \leq i \leq n} i a_i X^{i-1}$$ (e.g.: $0.6*3*x^2 + 5*2*x^1$)

Our function was: $x^2 + 2x + 1$, so for this example $a_1 = 1$, $a_2 = 2$ and we had a constant ***c*** (=1) that disappeared in the derivative

-> find a new function and try to optimize it!

Let's descend the gradient

TO YOUR KEYBOARDS!

Let's descend the gradient

Going further!

- Open the script `ch1_gradient_descent_der_solution.py`

- **Exercise 3**: Implement a bimodal function (functions that have 2 peaks/throughs):
  - Use this polynomial: `f(x) = 2*x^4 - 0.3*x^3 - 2.5*x^2`
    - **CHANGE THE ARRAY x TO SMALLER VALUES: from -1.3 to 1.3 (otherwise you won't see the interesting stuff)**
  - Compute its derivative, remember for any polynomial f(x), i.e. something that has the form: $P(X) = \sum_{0 \leq i \leq n} a_i X^i + \boldsymbol{c}$

Its derivative is: $\mathrm{d}P(X) = \sum_{1 \leq i \leq n} i a_i X^{i-1}$

- Launch it multiple times to have different starting point
  - What happens when you start at 0? Why?
  - What happens when you start at - 0.001? Why?
  - What happens when you start at + 0.001? Why?

Some more info on derivatives: https://www.mathsisfun.com/calculus/derivatives-rules.html

Let's descend the gradient

TO YOUR KEYBOARDS!

Let's descend the gradient

Going further!

- **Exercise 3**: Implement a bimodal function (functions that have 2 peaks/throughs):
  - Use this polynomial: $f(x) = 2*x^4 - 0.3*x^3 - 2.5*x^2$

- Launch it multiple times to have different starting point
  - What happens when you start at 0? Why?
  - What happens when you start at - 0.001? Why?
  - What happens when you start at + 0.001? Why?

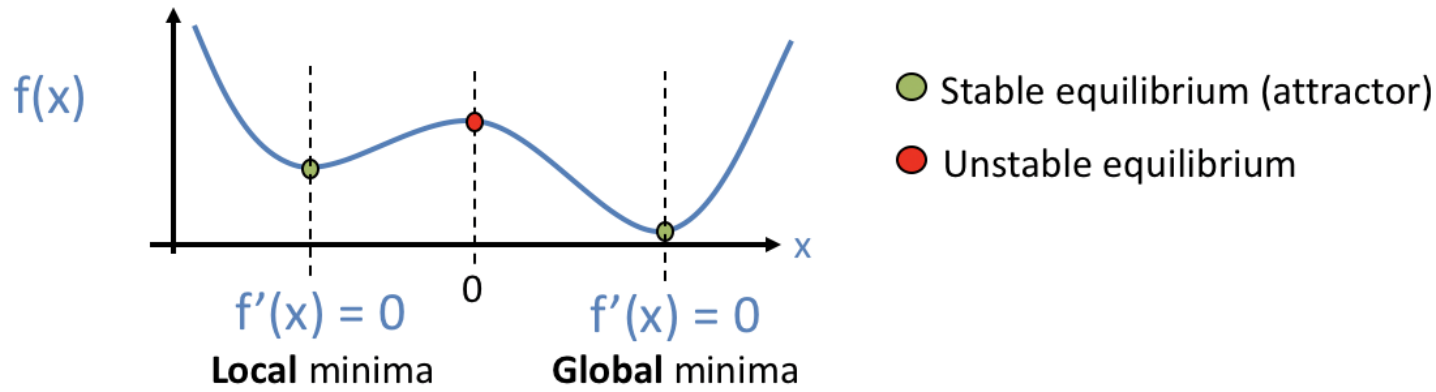
Some more info on derivatives: https://www.mathsisfun.com/calculus/derivatives-rules.html

Let's descend the gradient

Going further!

- **Exercise 3**: Implement a bimodal function (functions that have 2 peaks/throughs):
  - Use this polynomial: $f(x) = 2*x^4 - 0.3*x^3 - 2.5*x^2$

- Launch it multiple times to have different starting point
  - What happens when you start at 0? Why?
  - What happens when you start at - 0.001? Why?
  - What happens when you start at + 0.001? Why?



Some more info on derivatives: https://www.mathsisfun.com/calculus/derivatives-rules.html

Let's descend the gradient

Going further!

- **Advanced exercise for home**:
  1. Calculate the gradient WITHOUT the derivative: using finite differences
     - **Hint 1:** The basic concept behind a derivative is simply: $\frac{f(x_2) - f(x_1)}{x_2 - x_1}$

       So as long as you have two "x" to compare you can decide in which "direction" the next step should be, just like with the derivative function we had before.
     - **Hint 2**: Because you need to get a difference to start you'll have to get two **random** steps before going into the optimization loop
  2. Open the script `ch1_gradient_descent_unknown.py`
     - In the for loop at the end of the script implement the finite difference method to find the minimum of the function
     - Try the two functions (by commenting one or the other and running the script)
     - Try different values of the scaling parameter alpha, e.g. 0.001, 0.3, 0.99, 2.0, 10.0..)

     -> How does it affect the optimization?
  3. Extend the gradient descent algorithm to a 2-dimensional function of your choice

Let's model!

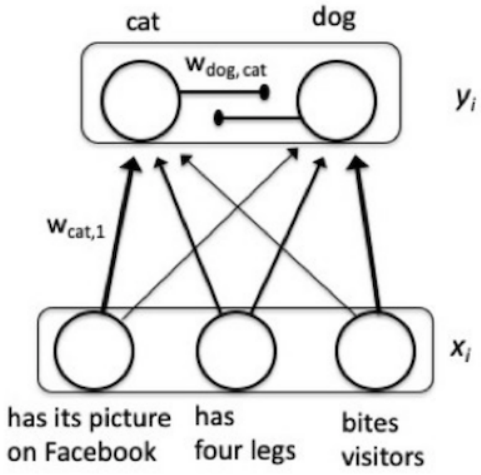
Let's model!

To build models we will use a Python package called TensorFlow (TF).

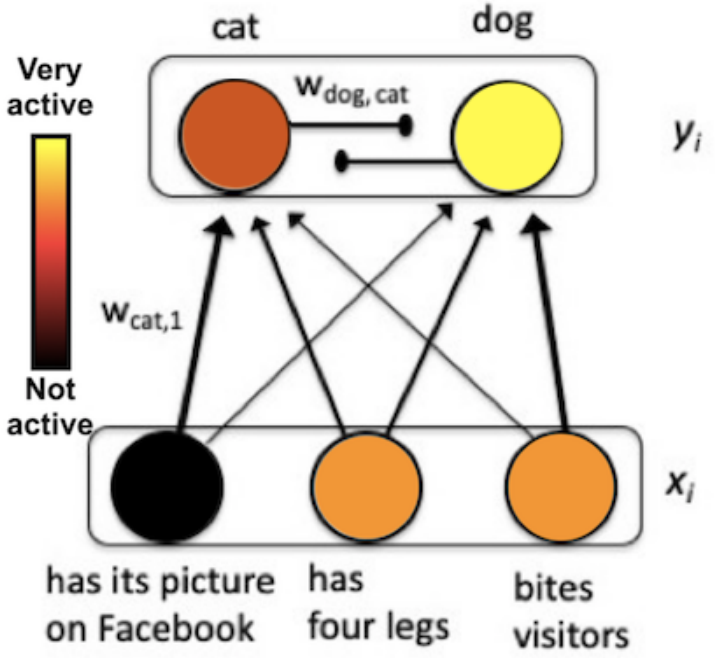
In principle, you can do modelling with a pen and a piece of paper (you'll just need to compute everything by hand at every stage...) or in "plain" Python (but you'll need to build all the tools/functions you need to optimize or train your model).

For this course, we will use TF because it allows to easily create neural networks, optimize them, train them, and build very large networks (e.g. used by google/facebook/etc.) and use lots of different types of models (e.g. reinforcemnt learning (chap. 9))

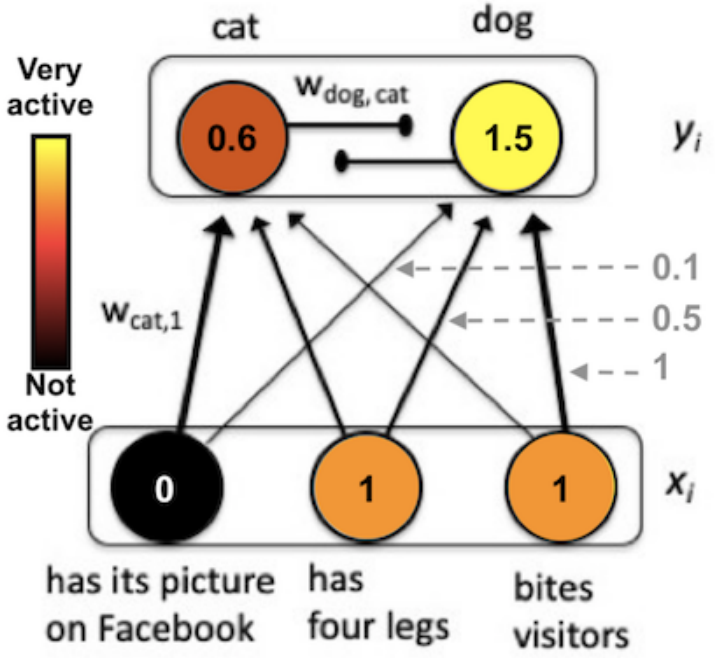
But before we get to TF, let's think about how information is represented in a model.



Let's model!


cat
dog
Very active
Not active
$w_{dog, cat}$
$w_{cat,1}$
y_i
x_i
has its picture on Facebook
has four legs
bites visitors

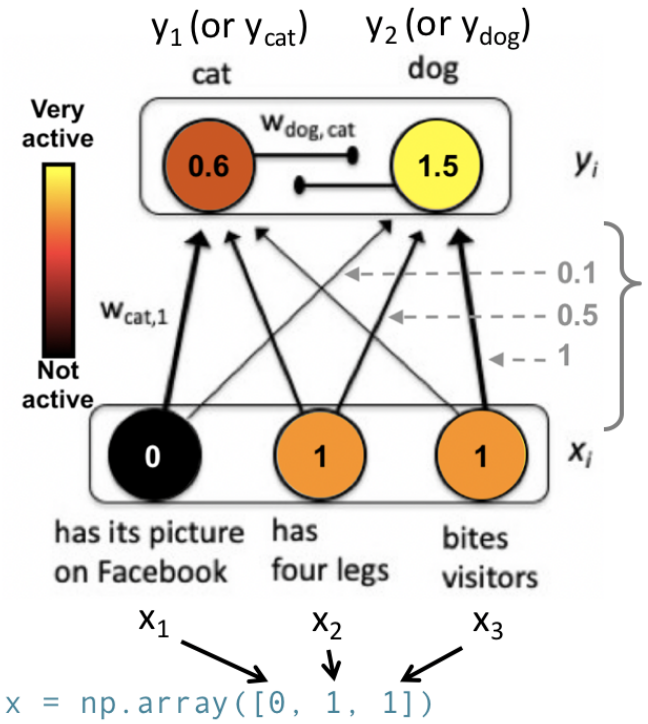
Let's model!


cat
dog
Very active
Not active
$w_{dog,cat}$
0.6
1.5
y_i
$w_{cat,1}$
0.1
0.5
1
0
1
1
x_i
has its picture on Facebook
has four legs
bites visitors

Let's model!

To compute y_1 and y_2 we need in_{cat} and in_{dog}

$in_{cat}, in_{dog} = ?$



Weight matrix

(i.e. math notation as in chapters)

FROM

TO		x_1	x_2	x_3
		1	2	3
y_1	1	w_{11},	w_{12},	w_{13}
y_2	2	w_{21},	w_{22},	w_{23}

rows columns

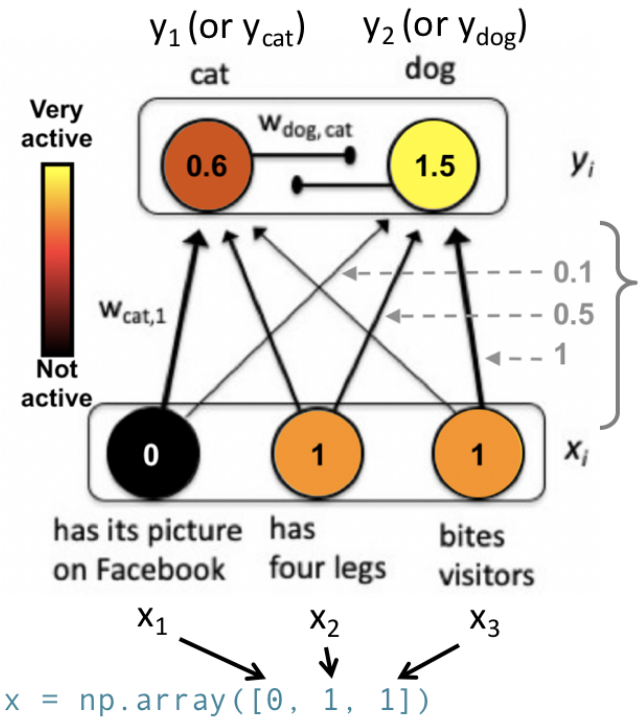
2-by-3 matrix

```
W = np.array([[w11, w12, w13], [w21, w22, w23]])
```

Let's model!

To compute y_1 and y_2 we need in_{cat} and in_{dog}

$in_{cat}, in_{dog} = (?)$




```
x = np.array([0, 1, 1])
```

Weight matrix

(i.e. math notation as in chapters)

FROM

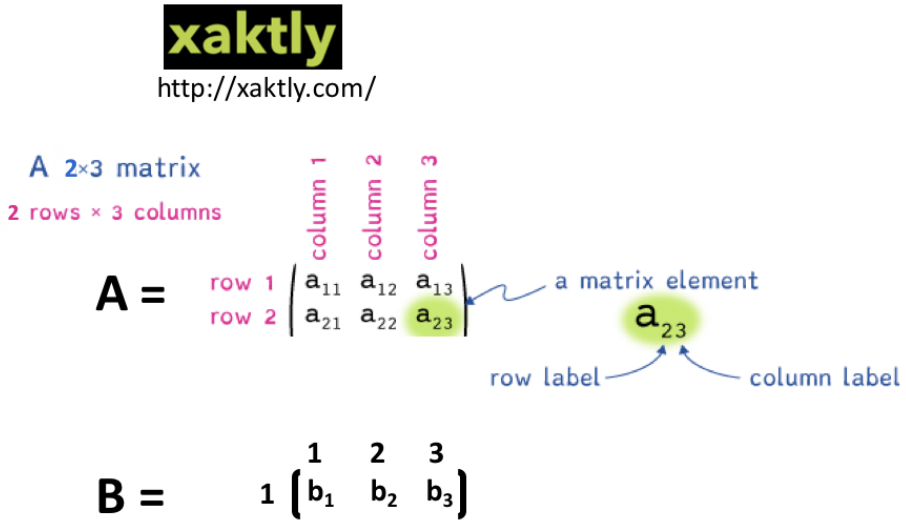
TO		x_1	x_2	x_3
		1	2	3
y_1	1	1,	0.5,	0.1
y_2	2	0.1,	0.5,	1

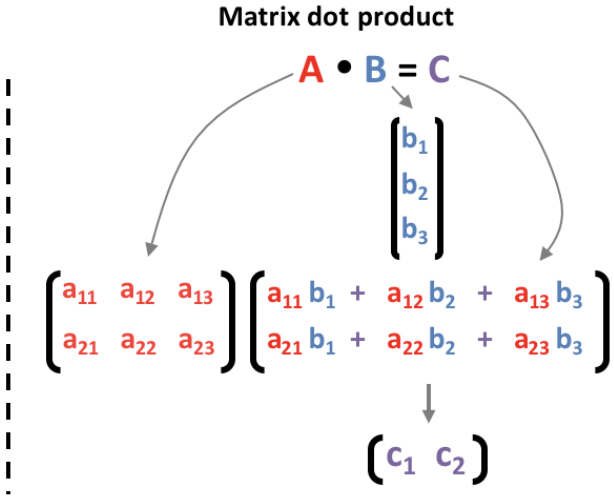
rows columns

2-by-3 matrix

```
W = np.array([[1, 0.5, 0.1], [0.1, 0.5, 1]])
```

Let's model!





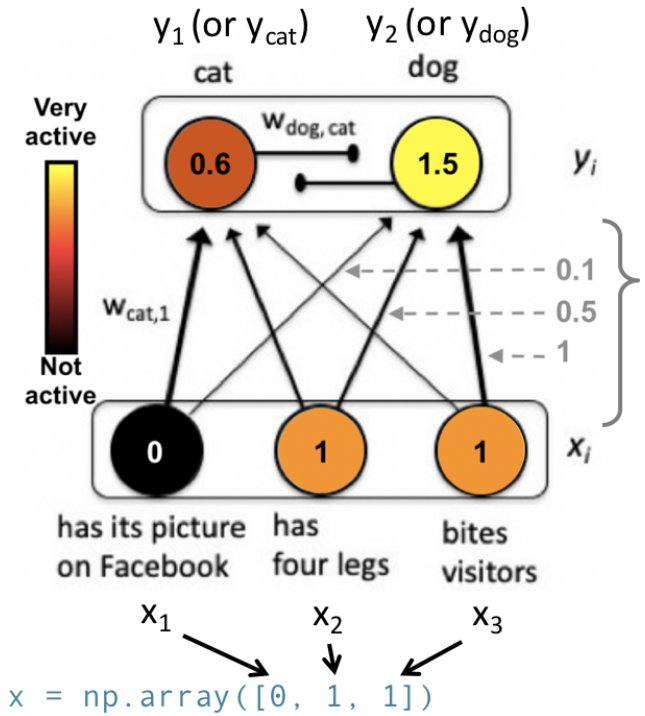
I highly encourage you to check these very didactic "tutorials" on matrices (and other things on maths but also physics, etc.) on the http://xaktly.com (http://xaktly.com) website:

- Intro into matrices (http://xaktly.com/MatrixDefinitions.html)
- Matrix operations (e.g. dot product) (http://xaktly.com/MatrixOperations.html)

Let's model!

To compute y_1 and y_2 we need in_{cat} and in_{dog}

$in_{cat}, in_{dog} = (?)$



Weight matrix

(i.e. math notation as in chapters)

FROM

TO		x_1 1	x_2 2	x_3 3
y_1	1	w_{11},	w_{12},	w_{13}
y_2	2	w_{21},	w_{22},	w_{23}

rows columns

2-by-3 matrix

```
W = np.array([[1, 0.5, 0.1], [0.1, 0.5, 1]])
```

$$W \bullet x = ?$$

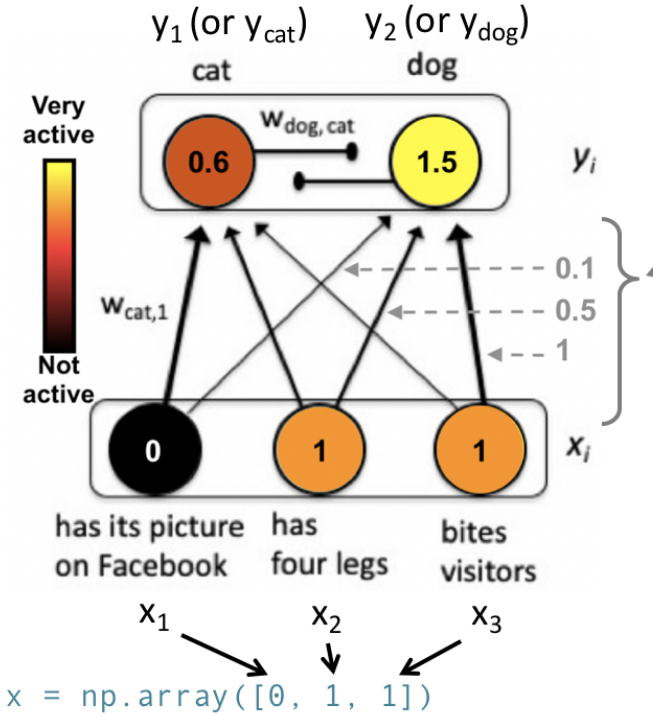
$$\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \\ w_{21}x_1 + w_{22}x_2 + w_{23}x_3 \end{pmatrix} = \begin{pmatrix} ?_1 \\ ?_2 \end{pmatrix}$$

Let's model!

To compute y_1 and y_2 we need in_{cat} and in_{dog}

```
in_cat, in_dog = np.matmul(W, x)
```

Dot product




```
x = np.array([0, 1, 1])
```

Weight matrix

(i.e. math notation as in chapters)

FROM x_1 x_2 x_3 (1 2 3)

TO y_1 1, y_2 2

$$\begin{bmatrix} w_{11}, & w_{12}, & w_{13} \\ w_{21}, & w_{22}, & w_{23} \end{bmatrix}$$

rows, columns → *2-by-3 matrix*

```
W = np.array([[1, 0.5, 0.1], [0.1, 0.5, 1]])
```

$W \bullet x = ?$

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \\ w_{21}x_1 + w_{22}x_2 + w_{23}x_3 \end{bmatrix}$$

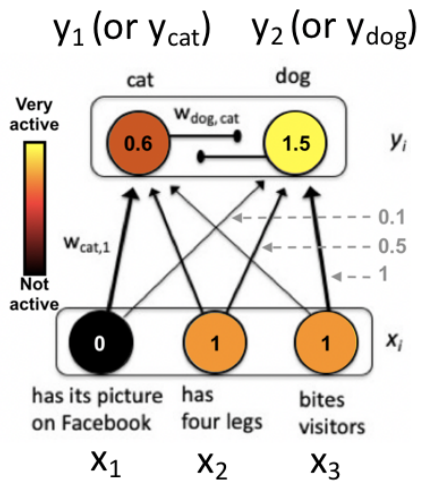
$$\begin{bmatrix} in_{cat} & in_{dog} \end{bmatrix}$$

This is in_{cat} and in_{dog}!

(see eq. 2.1 & 2.2)

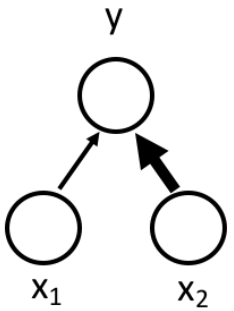
Let's model!

We want to build this:



BUT...

Let's start with this:



Let's model!

1. Open ch2_tf_simple_model.py
2. Run the script, what is y's value?

3. Change the input values to

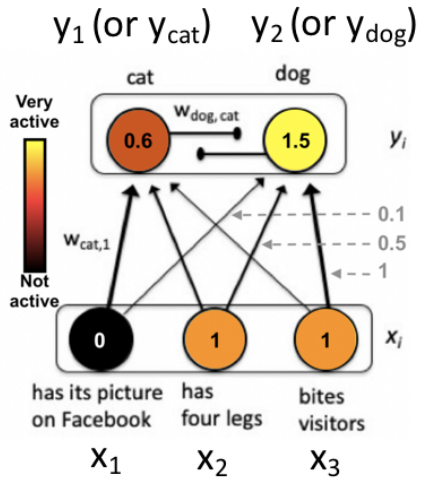
   x_1 = 3, x_2 = 0.3

   Then re-run the script and check that y's value is still correct.

Let's model!

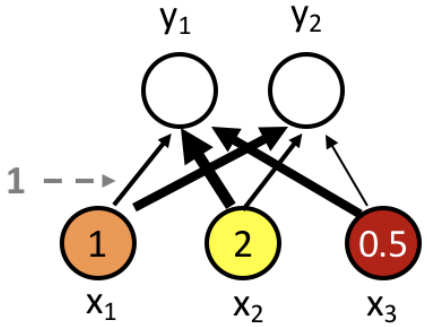
Exercise 4: extend the simple model

We want to build this:



BUT…

Let's move on to a larger model:



Input

x = ?

Weights

		x_1	x_2	x_3
		1	2	3
y_1	1	1	3	2
y_2	2	2	1	0.5

W = ?

rows col.

2-by-3 matrix

Let's model!

Correction of Exercise 4.

Let's model!

Now let's tackle the cat and dog model.

Open ch2_tf_cats_dogs.py

1. Run the script multiple times, does the correct output unit always wins?

Let's model!

Now let's tackle the cat and dog model: open ch2_tf_cats_dogs.py

Exercise 5

1. Increase the noise standard deviation to 2, does the correct output unit always wins?
2. Put the noise std back to 0.5 and make the optimization stop when any of the two output units reaches an activation of 8, i.e. a threshold. (hint: we're using a for loop now, but what kind of loop do we need to iterate **until** a certain criterion?)
3. Run the script 20 times for each input example (i.e. cat or dog, line 15 and 16), (or even better, make a loop to run it 20 times) and store whether the correct output unit was the most activated (model accuracy) and the number of loop iterations needed to reach the threshold for each input example (model reaction time). Calculate the mean accuracy, and median and standard deviation of the model reaction time, for the cat and dog examples.
4. Change the noise std to 1 and the weights to:

$$\begin{array}{cccc} & & \text{FROM} & \\ & & x_1 \quad x_2 \quad x_3 & \\ & & 1 \quad 2 \quad 3 & \\ \text{TO} & y_1 \; 1 & \begin{pmatrix} 0.5, & 0.8, & 0.2 \end{pmatrix} \\ & y_2 \; 2 & \begin{pmatrix} 0.05, & 0.6, & 2 \end{pmatrix} \end{array}$$

   Re-run the script 20 times for each example. Do the model reaction times change? Do the model accuracy change?

The end!