Once you have the notebook open, type:

```
print("Hello, world!")
```

⊞ The message “Hello, world!” is then displayed...

To get yourself started, type the following stack of instructions

```
a = 3
b = 2*a
print(b)
```

⊞ should display 6….

Now try this and guess what the output will be?

```
print(a*b)
```

⊞ should display ….

Let’s work with some strings

```
b = 'hello'
print(b)
```

⊞ should display hello….

Now try this:

```
print(b+b)
```

⊞ What does it display?….

Notice something? Numbers and text are different.

```
type(b)
```

⊞ b is of the type string, meaning a string of letters .

Remember a? What type might a be?

```
type(a)
```

⊞ b is of the type int, meaning a whole number.

These few lines have covered two fundamental concepts: Firstly, variables, which are like little things the computer (more specifically the Python kernel) remembers. We can perform simple operations that change the value in the variable. This allows us to do basic calculations. The second bit is that variables have a certain type, here we learned the types *int* that can hold whole numbers and *string* that can store letter chains, aka text. Importantly, simple operators work differently for *int* and *string*. Let's demonstrate this:

Let's do simple things with *int*

```
b = 2
print(b+b)
```

⊞ should display 4….

Let's try this with a *string*

```
b = '2'
print(b+b)
```

⊞ should display 22
Do you understand why?

Discuss this with your friends in class!

There are many more types, you can read more at:
https://lectures.scientific-python.org/intro/language/basic_types.html

Let's try something more fancy

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
mu = 0
std = 1
x = np.linspace(start=-4, stop=4, num=100)
y = stats.norm.pdf(x, mu, std)
plt.plot(x, y)
plt.show()
```

⊞ This should show a nice plot with a normal distribution

Let's unpack this:

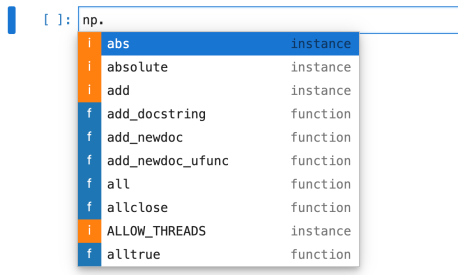
‘import’ loads fancy python libraries that do great stuff. Firstly, we load the library numpy a very very useful one for math stuff. I personally think python without numpy is broken. You can end the line after ‘import numpy’ but nobody does that for numpy, instead they say ‘as np’ because they are too lazy to type numpy. np is nothing but a short alias for numpy. If we want to use the fancy things numpy can do, in the code example above we use the function:

```
x = np.linspace(start=-4, stop=4, num=100)
print(x)
```

The function linspace creates a set of numbers with linear spacing. Try:

```
print(np.linspace(start=0, stop=1, num=11))
```

if you type np. and then press the tab key, python will give you a drop down list of functions that np contains.



```
[ ]: np.
i abs               instance
i absolute          instance
i add               instance
f add_docstring     function
f add_newdoc        function
f add_newdoc_ufunc  function
f all               function
f allclose          function
i ALLOW_THREADS     instance
f alltrue           function
```

There are many many things in np. Such as np.sin, calculating the sine operation of the input. I.e. try:

```
print(np.sin(3.142))
```

or

```
print(np.cos(3.142))
```

Going back to the little demo script, the next line

```
y = stats.norm.pdf(x, mu, std)
```

uses the stats library we loaded and calculates a normal distribution, or more precisely the probability for a given value. In the example above, we feed a list of numbers (x) to the function, and we get a list of numbers back. Essentially for every x from -4 to 4, the function returns the height of the pdf (with mu=0 and std=1).

Note: This function is called the pdf (probability density function) of the normal distribution.

Finally,

```
plt.plot(x, y)
plt.show()
```

put’s it into a nice little graph.

We can also use plt.plot in a more simple way:

```
plt.plot([0, 0.5, 1],[0, 1, 3])
plt.show()
```

which plots the three dots: 0,0 0.5,1 and 1,3. Notice for the plot command all the x cords are in the first list, while the ys are in the second list. Whereas in my text I paired the x and y for each dot. (see colour code. Oh and no you can't copy that colour code to Python…)
By default plot draws a line from dot to dot, that worked well for the pdf function above, but it is not so great for the three dot graph as it implies a continuous dataset. So try this:

```
plt.plot([0, 0.5, 1],[0, 1, 3], 'xr')
plt.show()
```

There are a number of different flags one can use for plotting. 'xr' stands for use x as marker and in colour red. 'oy' stands for yellow circles, 'gd' for green diamonds…
If you want to dig into the depth of plotting options:
https://docs.xarray.dev/en/stable/user-guide/plotting.html

Of course, we want to label the axes
(https://xkcd.com/833/)

```
plt.xlabel("Time (s)")
```

Task 1:
Can you use the 'xr' flag for plotting the pdf of the normal distribution?

Decay function:

Now, next let's have a look at the decay function.
The function we want to work with is

$$V(t) = V_\infty * (1 - e^{-\frac{t}{\tau}})$$

So how do we approach that? You may want start a new Jupyter book, but if you like you can continue the old one. Your choice. First make some groundwork, import some libraries and define basic parameters.

```
import numpy as np
import matplotlib.pyplot as plt
Vinf=1
tau=5
```

Next we want to actually write the function:

```
t=0
Vt=Vinf*(1-(np.e**(-t/tau)))
print(Vt)
```

```
0.0
```

Note the ** operator does the power. i.e. 3**2 is three squared, i.e. 9.

So the function starts at zero, cool cool.
What after a short while, i.e. t = tau/2?

```
Vt=Vinf*(1-(np.e**(-2.5/tau)))
print(Vt)
```

```
0.3934693402873666
```

O.K. so it has risen a bit. Notice, I have entered 2.5 directly into the equation instead of defining it in a previous line. You can do whatever you prefer.

So what after t=tau?

```
Vt=Vinf*(1-(np.e**(-tau/tau)))
Vt
```

```
0.6321205588285577
```

Remember this number. 63% is the key value you will see in decay functions all over the place!

Now feeding one value after the other into the function is a bit, … tedious. So, let’s try something better:

```
t = np.arange(0, 15, 0.5)
print(t)
```

```
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5  6.   6.5
  7.   7.5  8.   8.5  9.   9.5 10.  10.5 11.  11.5 12.  12.5 13.  13.5
 14.  14.5]
```

Now, t is not a single value, but a list of values (timepoints if you like) from 0 to almost 15. Specifically, we asked for a list of values from 0 to 15, in steps of 0.5. Don’t ask me why 15 is not in that list. It’s a Python quirk. The same (well similar) function in matlab would include 15.

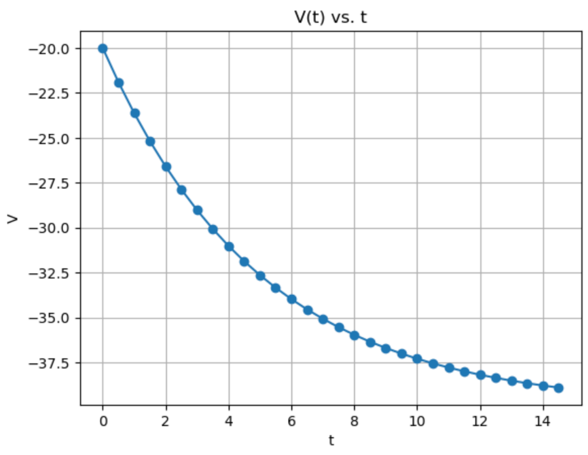
```
Vt=Vinf*(1-(np.e**(-t/tau)))
print (Vt)
```

and finally print it:

```
plt.plot(t, Vt, marker='o', linestyle='-')
plt.xlabel('t')
plt.ylabel('V')
plt.title('V(t) vs. t')
plt.grid(True)
plt.show()
```

Task 2

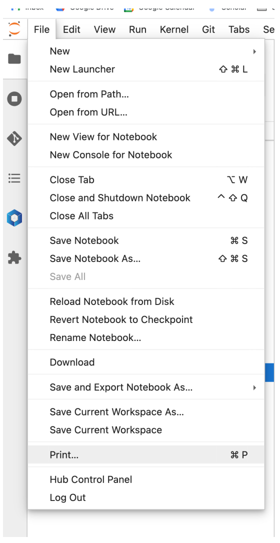
The function above starts at 0 and rises towards a final goal of 1. Modify the equation so that the output (plot) starts at -20 and then decays to -40. Imagine the intracellular potential after an EPSP, and it then returns to rest at around -40mV. It should look like this:



Making a Labbook (Assessment item!)

Now once you are done, please save the jupyterbook(s) you just made under Exerc_1_YourInitinals.ipynb. If you have part Task 1 and Task 2 in two independent Jupyterbooks, print them both. For the assessment compile all your Jupyterbooks into a single PDF….

For printing the jupyerbook into a pdf:


File
Edit
View
Run
Kernel
Git
Tabs
Se
New
New Launcher ⇧ ⌘ L
Open from Path…
Open from URL…
New View for Notebook
New Console for Notebook
Close Tab ⌥ W
Close and Shutdown Notebook ^ ⇧ Q
Close All Tabs
Save Notebook ⌘ S
Save Notebook As… ⇧ ⌘ S
Save All
Reload Notebook from Disk
Revert Notebook to Checkpoint
Rename Notebook…
Download
Save and Export Notebook As…
Save Current Workspace As…
Save Current Workspace
Print… ⌘ P
Hub Control Panel
Log Out