# PART 1: PYTHON - A (VERY) SHORT INTRODUCTION

Today you'll learn the basics of programming in Python. We use a so-called *Python notebook*. the notebook consists of two types of cells, those that contain text, like the one you're reading right now, and those that contain python code.
Below is an example of a cell with Python code. When we click on a cell with code and either press [SHIFT] + [ENTER] or click on the (RUN) button above, we tell the computer to interpret the commands which means that it turns the Python code into actions we want it to execute.

In [1]:

```
# this is a comment. Everything preceeded by a hash symbol (#) is ignored by the
# computer. This allows you to add comments/notes/documentation to your python p
rogramme.
print("hello world")
```

```
hello world
```

As you can see, we instructed the computer to "print" a string of letters. Don't worry too much about the syntax right now, later we'll go through this step by step!

## 1.1 VARIABLES

Variables are used to store information that can later be referenced and manipulated by your computer program. It's helpful to think of variables as containers for information. For example, imagine you build a drawer and put socks in it. In this case, the drawer would be a variable and the socks we put in would correspond to its value(s). When you reference the variable, i.e. open the drawer, you'll find socks in there.

In [2]:

```
# let's create a drawer with socks
drawer = "socks"

# let's have a look what's inside the drawer
drawer
```

Out[2]:

```
'socks'
```

### 1.1.1 Assign Values to Variables

Variables would be useless if you couldn't assign values to them. Python is able to handle all sorts of different datatypes, such as numbers, strings (=sequences of letters and symbols) and lists of numbers/letters. Below are a few examples

In [3]:

```
# this is an integer (a number without any decimal points)
variable1 = 5
# this is another integer
variable2 = 7
# this is a float (a number with decimal points)
variable3 = -2.563
# this is a string (a sequences of characters)
variable4 = 'Hello World'
# this is a list (a set of different items)
variable5 = [1,2, 'Tomato',0,'???']
```

Let's check out what's inside these variables

In [4]:

```
variable5
```

Out[4]:

```
[1, 2, 'Tomato', 0, '???']
```

**Important!!** Variable names are somewhat arbitrary. But try to give your variables sensible names that explain what kind of information they hold (remember the drawer example above). This prevents a lot of confusion when you come back to code you've written weeks/months ago!

## 1.1.2 Operations using defined variables

What makes programming languages so powerful is their ability to perform various operations on the variables you have defined. In this section, you'll learn a few of those!

#### Basic addition & subtraction

Let's write a program that adds two of the variables we have defined above:

In [5]:

```
variable1 + variable2
```

Out[5]:

```
12
```

You can conduct basic math directly in-line, but if you need more than one operation to run, you'll want to assign them to variables:

In [6]:

```
sum_of_two = variable1 + variable2
difference = variable1 - variable2
difference2 = difference - variable3
```

Notice how the results of these operations were not automatically displayed in the command line. This is desired behaviour of any programming language. Imagine that you have a very long and complicated script full of mathematical operations, but are only interested in the end result. It would be a nightmare if each and every intermediate result was displayed in the command line..

To display specific variables (and some text) in the command line, use the **print** statement.
The print statement is a **function**. As in maths, functions take some value as input, apply a transformation and return the result of this transformation as output. In case of the print statement, it takes strings of characters and variables as input, and - as the name suggests - "prints" the result to the command line (in our case the space below a cell with Python code). We'll learn more about functions later in the course.
As you can see in the example below, the print function accepts all sorts of different input formats. Choose the one that works best for you!

In [7]:

```
# one variable
print(difference2)
# multiple variables can be separated by a comma
print(sum_of_two, difference)

# we can also construct a string by concatenating text and numbers (that we have
turned into strings)
print('Sum of int1 and int2 is ' + str(sum_of_two) + ' and difference between th
em is ' + str(difference))

# you can also use {} as placeholders and use the format method to feed values i
nto the placeholders
print('Sum of int1 and int2 is {}, and differene between them is {}'.format(sum_
of_two, difference))
```

```
0.5630000000000002
12 -2
Sum of int1 and int2 is 12 and difference between them is -2
Sum of int1 and int2 is 12, and differene between them is -2
```

In [8]:

```
# in case you were confused: Python distinguished between different types of var
iables.
# If we assign a number to a variable, Python will assume that it's a number.
# If we put the number into quotation marks, it will assume it's a character (=
 letter on your keyboard) and
# can't do maths with it.
# "+" between numbers adds them up. "+" between strings/characters concatenates
 them
five_num = 5
five_str = '5' # this is equal to str(five_num)
one_num = 1
one_str = '1'
print(five_num + one_num)
print(five_str + one_str)
print(five_num + five_str)
```

```
6
51

---------------------------------------------------------------------
--------
TypeError                                 Traceback (most recent ca
ll last)
<ipython-input-8-d51f6ffb130c> in <module>()
     10 print(five_num + one_num)
     11 print(five_str + one_str)
---> 12 print(five_num + five_str)

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

#### Basic Multiplication and Division

Imagine how limited a programming language would be, if you couldn't perform multiplication and division. Luckily, Python has you covered!

In [9]:

```
int1 = 20
int2 = 6
product_of_two = int1 * int2
quotient_of_two = int1 / int2
modulo_of_two = int1 % int2
print('Product of int1 and int2 is {}'.format(product_of_two))
print('Dividing int1 by int2 yields {} '.format(quotient_of_two))
print('The remainder after dividing int1 by int2 is {} '.format(modulo_of_two))
```

```
Product of int1 and int2 is 120
Dividing int1 by int2 yields 3.3333333333333335
The remainder after dividing int1 by int2 is 2
```

#### Exponentiation

And of course, you can also perform exponentiation!

In [10]:

```
base_1 = 2
base_2 = 9
exponent_1 = 4
exponent_2 = -1
exponent_3 = 0.5
result_1 = base_1**exponent_1
result_2 = base_1**exponent_2
result_3 = base_2**exponent_3
print('2^4= ' + str(result_1))
print('2^-1 = 1/2 = ' + str(result_2))
print('9^0.5 = square root of 9 =  ' + str(result_3))
```

```
2^4= 16
2^-1 = 1/2 = 0.5
9^0.5 = square root of 9 =  3.0
```

**Logical Operators**

In programming you very often need to know whether a statement is true or false. This can be achieved with Boolean variables, which you'll find become very handy over time:

In [11]:

```
bool1 = True
bool2 = False
print(bool1)
print(bool2)
print(bool1 and bool2)
print(bool1 or bool2)
print(bool1 and not bool2)
```

```
True
False
False
True
True
```

More frequently, you will be able to check whether relationships between variables are true or false or do something conditionally based on whether a statement is true (see below):

In [12]:

```
print(5 < 6)
print(6 >= 5)
print(5 != 10/2)
print('hello' != 'world')
```

```
True
True
False
True
```

Logical operators are very powerful. For instance, you can test whether a variable lies within a certain interval:

In [13]:

```
int1 = 10
print(int1 < 12 and int1 > 9)
print( 9 < int1 < 12)
```

```
True
True
```

### TASK: Define variables and write code to...

1. Calculate square root of 25, 144 and 289
2. Translate the following into Python
   - assign numbers between 1 and 100 to 3 different variables
   - add up your 3 variables and print whether the sum is bigger than 100

**Hints**:

- You can either use a built-in function (sqrt(x)), or perform clever exponentiation (what does x^0.5 do to x?)
- look at the section on logical operators if you're stuck!

In [14]:

```
# Your code for example 1 here
```

In [15]:

```
# Your code for example 2 here
```

# 1.2 LOOPS and STATEMENTS

Loops allow you to execute the same operation multiple times, but with a different type of input on each iteration.
Statement allow you to structure your code, and have the computer care about certain bits only if conditions you specify are met.
In this section, we'll learn more about both

### 1.2.1 While Loops

Use while loops whenever you need to apply an operation multiple times, but don't know exactly how many iterations are needed:

In [16]:

```
# WHILE loops
x = 500
while x > 10:
    print(str(x))
    x = x // 2
```

```
500
250
125
62
31
15
```

### 1.2.2 For Loops

For loops come in handy whenever you need to apply a function to each item of a (potentially very long) list, and you know exactly how many iterations you need: (note for the pros: all for loops in python are actually for-each loops)

In [17]:

```
# FOR loops

modules = ["cognitive", "developmental", "neuro", "clinical", "social", "computa
tional"]
for x in modules:
    print(x)
```

```
cognitive
developmental
neuro
clinical
social
computational
```

### 1.2.3 If...Then Conditionals

Before we talk about if..then conditionals, let's first learn something new! the len() command counts the number if symbols in a list. Example:

In [18]:

```
# let's count how many numbers are in this list
print(len([4,5,5,2,1]))
# Note that it also works with strings (which are essentially lists of symbols)
print(len('hello world'))
```

```
5
11
```

Programming is all about case-specific processing. Use if..then statements to process inputs depending on conditions you have specified:

In [19]:

```
# IF...THEN statements


favoriteModule = "computational"
firstModule = "social"

if len(firstModule) < len(favoriteModule):
    print('These modules are of different lengths!')
elif len(firstModule) == len(favoriteModule):
    print('These modules are of same lengths!')
```

```
These modules are of different lengths!
```

Very commonly in coding practice you'll find that you want to check multiple conditions before doing something with your variables. Python is easy when it comes to this - just tell it whether you want to use 'and' or 'or' or a different logical connector.

Link to a good starting point on Boolean operators in Python (https://https://docs.python.org/2.0/ref/lambda.html)

In [20]:

```
a = 200
b = 33
c = 500
if a > b and c > a:
    print('Both conditions are true!')
else:
    print('Both conditions are not true!')


if a > b or a > c:
    print('At least one of the conditions is true!')
```

```
Both conditions are true!
At least one of the conditions is true!
```

## TASK: Write your own for loop and if/then statement

Given

```
modules = ["cognitive", "developmental", "neuro", "clinical", "social",
"computational"]
comparisonVariable = "introductory"
```

Determine whether any of the modules have names that are longer than the comparison variable.

In [21]:

```
modules = ["cognitive", "developmental", "neuro", "clinical", "social", "computa
tional"]
comparisonVariable = "introductory"

# Your code here
```

# 1.3 FUNCTIONS

## 1.3.1 Introduction to Functions

Functions are nothing new. You have already worked with them in the previous exercises!

Here are some examples...

In [22]:

```
print("hello world")
print("2*3:",2*3)
print(len("hello world"))
```

```
hello world
2*3: 6
11
```

What makes coding so great, is that you're able to write your own functions! Whenever you identify a recurring task in your workflow, consider writing a function instead of copying & pasting code over and over again:

In [23]:

```
# naive approach for testing parity of several numbers:
if 3 % 2 == 0:
    print('{} is even'.format(3))
else:
    print('{} is odd'.format(3))


if 2 % 2 == 0:
    print('{} is even'.format(2))
else:
    print('{} is odd'.format(2))


if 11 % 2 == 0:
    print('{} is even'.format(11))
else:
    print('{} is odd'.format(11))
```

```
3 is odd
2 is even
11 is odd
```

In the example above, we tested whether several numbers are odd or even. Instead of repeating the same if..then construct over and over again, we can put it in a neat little function:

In [24]:

```
# let's write a function that does the work for us!
def return_parity(n):
    result = str(n) + ' is'
    if n % 2 == 0:
        return result + ' even'
    else:
        return result + ' odd'
```

Now we can simply apply the function to each of the numbers we're interested in

In [25]:

```
print(return_parity(3))
print(return_parity(2))
print(return_parity(11))
```

```
3 is odd
2 is even
11 is odd
```

By the way, if..then statements can be compressed into one-liners:

In [26]:

```
# let's make this shorter
def return_parity(n):
    result = str(n) + ' is '
    return result + 'even' if n % 2 == 0 else result + 'odd'
```

Did you notice something? We applied the function to several different numbers. Why not put all numbers in a list and use a loop to iterate over them?

In [27]:

```
# let's combine this with loops
numbers = [3, 2, 11]
for ii in numbers:
    print(return_parity(ii))
```

```
3 is odd
2 is even
11 is odd
```

Functions can be put inside other functions. Let's write a wrapper that iterates over a list!

In [28]:

```
#let's write a wrapper!
def check_item_parity(arr):
    for ii in arr:
        print(return_parity(ii))
```

In [29]:

```
check_item_parity(numbers)
```

```
3 is odd
2 is even
11 is odd
```

Now if you define a different list of numbers you want to check parity for, you can simply use your wrapped function instead of calling check_parity over and over again. See below:

In [30]:

```
longerArray = [2, 3, 4, 5, 6, 7, 8, 9, 10, 23, 25, 72, 43, 52, 26, 24, 23, 16]
check_item_parity(longerArray)
```

```
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
23 is odd
25 is odd
72 is even
43 is odd
52 is even
26 is even
24 is even
23 is odd
16 is even
```

Functions are treated as so-called first class objects - you can assign them to variables...

In [31]:

```
my_len = len
print(my_len("hello world"))

# (don't ask why you would want to do this. but it works x) )
```

```
11
```

.. and you can pass them as inputs to other functions

In [32]:

```
def do_sth_with_number(number, something):
    print(something(number))
```

In [33]:

```
do_sth_with_number(-2, abs)
do_sth_with_number(9.34,round)
do_sth_with_number(9,return_parity)
```

```
2
9
9 is odd
```

### 1.3.2 Docstrings

Programming can be a very complicated endeavour. And who is able to remember for each and every function how to use it? Nobody. In fact, in real life, whenever you need to write a program, you'll notice that you spend most of the time googling stuff and scrolling through online forums. That's absolutely normal and even the biggest coding pros do this all the time.
But even if you don't have internet, there's still hope: Python provides documentation for all of its built-in functions. These brief info snippets are called *docstrings* and we'll show you two ways to access them.
First, you could simply add a question mark to the function you're interested in, and execute the line of code:

In [34]:

```
print?
```

Docstring: print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default. Optional keyword arguments: file: a file-like object (stream); defaults to the current sys.stdout. sep: string inserted between values, default a space. end: string appended after the last value, default a newline. flush: whether to forcibly flush the stream. Type: builtin_function_or_method

Alternatively, you pass it as argument to the help function:

In [35]:

```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=Fal
se)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sy
s.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

When you write your own functions, don't forget to add documentation on how to use them. This saves a lot of time and avoids nervous breakdowns when you come back to your code a few months later. In fact, you can easily write your own docstrings! Here's an example
Docstrings are indicated by """ yourtextgoeshere """

In [36]:

```
def return_parity(n):
    """
      This function tests whether its input is odd or even.
      INPUT: (int) an arbitrary number
      OUTPUT: (str) 'even' or 'odd'
    """
    result = str(n) + ' is '
    return result + 'even' if n % 2 == 0 else result + 'odd'
```

In [37]:

```
help(return_parity)
```

```
Help on function return_parity in module __main__:

return_parity(n)
    This function tests whether its input is odd or even.
    INPUT: (int) an arbitrary number
    OUTPUT: (str) 'even' or 'odd'
```

## Task: Write your own function!

Write a function that takes a list of numbers as inputs and returns the mean of these numbers!
Remember, given a vector of numbers

$$\mathbf{x} = [x_1, x_2, x_3, x_4, \ldots, x_n]$$

the mean is defined as:

$$mean(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} x_i = \frac{1}{N}(x_1 + x_2 + x_3 + \ldots + x_n)$$

In [38]:

```
numbers = [2,5,8,7,1,4,4,9]
# your code goes here
```

Now write a function that computes the variance of a list of numbers. Remember, the variance is defined as

> $var(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$, where $mean(\mathbf{x}) = \bar{x}$
>
> or alternatively:
>
> $var(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N}(x_i^2) - \bar{x}^2$ ("expectation of square minus square of expectation")

In [39]:

```
numbers = [2,5,8,7,1,4,4,9]
# your code goes here
```

Now add docstrings to your functions to explain how to use them

In [40]:

```
# your code goes here
```

## BONUS: Advanced Programming Principle: Recursion

Functions are very versatile! They can even call themselves with a subset of of the original problem. This is called recursion. Below is a naive solution to find the factorial of a number. As reminder, a factorial of a number n is just a product of all integers from 1 to n:
**n! = 1*2*3*.....*n**

In [41]:

```
def iterative_factorial(n):
    fact = 1
    for ii in range(n):
        fact *= ii+1
    return fact
```

In [42]:

```
print("4!: ",iterative_factorial(4))
```

```
4!:  24
```

In [43]:

```
def recursive_factorial(n):
    if n >1:
        fact = n*recursive_factorial(n-1)
        return fact
    else:
        return n
```

In [44]:

```
print("4!: ",recursive_factorial(4))
```

```
4!:  24
```

# 1.4 .METHODS()

Python is an object oriented language. An object is an abstract entity that contains some data and functions that can be applied to the data (Imagine a drawer (container) for socks (data) that has a built-in sock sorting machine (function)). Functions that are applied directly to objects are called **methods**. In this section we'll learn how to apply methods to objects.
Many entities in Python are objects. For example, strings are objects as they contain some data, as for example "hello world", and they come with many functions that can be applied to the data. Let's have a look at an example:

## 1.4.1 Example: methods to manipulate strings

In [45]:

```
# here we create a simple variable that contains a sequence of symbols
string = "hello world"
# Python understands that this is what it would call a string, and allows you to
apply methods
# that are purpose-built for string manipulation:
print(string.upper())
print(string.capitalize())
print(string.replace(' world',', nice to meet you'))
listOfWords = string.split(' ')
print(listOfWords)
newString = ', '.join(listOfWords)
print(newString)
```

```
HELLO WORLD
Hello world
hello, nice to meet you
['hello', 'world']
hello, world
```

in contrast to functions, for which the syntax is *function(argument)*, methods are bound to objects. Thus, the syntax is *object.method(argument)*

## 1.4.2 Methods are object specific!

As methods are bound to a particular object type, you can't apply them to other objects. For instance, you can't capitalise numbers:

In [46]:

```
# let's create a variable that contains a number
number = 0.5
# Python realises that this is not a string, and therefore you're not able to ca
pitalise it!
number.capitalize()
```

```
---------------------------------------------------------------------------
--------
AttributeError                            Traceback (most recent ca
ll last)
<ipython-input-46-93d524e9ed5f> in <module>()
      2 number = 0.5
      3 # Python realises that this is not a string, and therefore
 you're not able to capitalise it!
----> 4 number.capitalize()

AttributeError: 'float' object has no attribute 'capitalize'
```

### 1.4.3 How to figure out which methods are provided for a given object

to see a list of all available methods, either press [TAB] after placing a dot behind an object, or use the following syntax

In [47]:

```
dir(str)
```

Out[47]:

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
```

```
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

In [48]:

```
help(str.split)
```

```
Help on method_descriptor:

split(...)
    S.split(sep=None, maxsplit=-1) -> list of strings

    Return a list of the words in S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are
    removed from the result.
```

## Task: String Manipulation

You're given the following two strings:

```
gibberish_1 = 'is much I can''t enough.'
gibberish_2 = 'cOdinG sO FuN. JuST gEt '
```

- turn the strings into lists of words (use the .split method)
- make sure that gibberish_2 contains only lower case letters (.lower() method)
- create a new empty list ( assign [] to a variable)
- loop through 5 steps and for each iteration, add pairs of words to your new empty list
- turn the list into a string (use the ' '.join(list) method)
- print the result

In [49]:

```
gibberish_1 = 'is much I can''t enough.'
gibberish_2 = 'cOdinG sO FuN. JuST gEt '

# Your code goes here
```

# PART2: PYTHON FOR SCIENTISTS: PACKAGES/LIBRARIES

Some of the most powerful tools you'll use in Python are packages and libraries. These contain useful functions - from quite simple to more advanced ones. We'll go through a few basic examples today, but leave you with a list of libraries and packages to check out:

- numpy
- matplotlib
- seaborn
- scipy

In [50]:

```
#You'll need to import every package you want to use.
# This is not a one-time thing - you have to do it in each new Python notebook.
# With time you'll have a pretty good sense of what packages you need
#(and Stackoverflow to the rescue, always!).
import numpy
```

In [51]:

```
array = [2, 4, 6, 8, 10]

averageOfArray = numpy.mean(array)
print(averageOfArray)
```

```
6.0
```

In [52]:

```
# Alternatively, you can use abbreviations - this is really handy.
import numpy as np

averageOfArray = np.mean(array)
print(averageOfArray)
```

```
6.0
```

In [53]:

```
# And, of course, you can print or use any of these directly, such as:
print(np.mean(array))
```

```
6.0
```

In [54]:

```
# Numpy is also powerful in prepopulating your arrays
a = np.zeros((2))
print(a)

print('') # space to separate outputs

# or even something like if you need multiple dimensions
b = np.ones((2, 2))
print(b)

print('')

# you can also create an array filled with the same number of choice, something
 like
x = 27.34
c = np.full((5), x)
print(c)
```

```
[0. 0.]

[[1. 1.]
 [1. 1.]]

[27.34 27.34 27.34 27.34 27.34]
```

A few ways to visualize your data and important things to note

In [55]:

```
# Let's start by plotting something quite simple using a line plot in matplotlib
(remember to import this in every notebook)
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [2, 5, 6, 11, 14]

plt.plot(x, y, linestyle='--')
```

Out[55]:

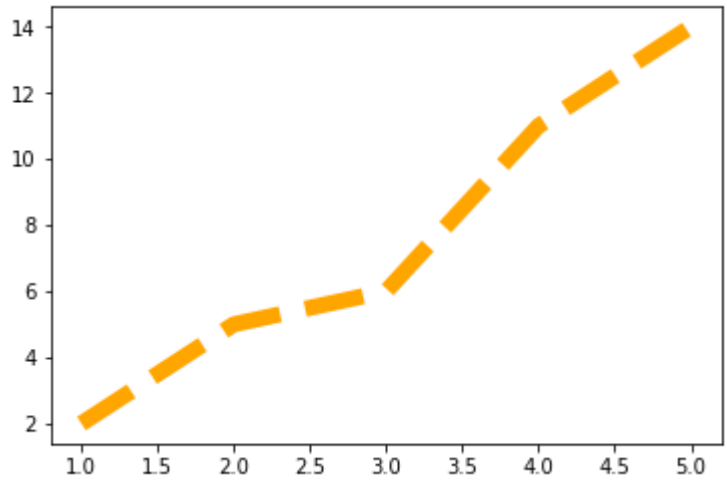
```
[<matplotlib.lines.Line2D at 0x7f89d1957be0>]
```

In [56]:

```
# Alternatively, styles can be spelled out as well (this is true for colors, et
c.)
plt.plot(x, y, linestyle='dashed', color = 'orange', linewidth = 8)
```

Out[56]:

```
[<matplotlib.lines.Line2D at 0x7f89d1767ef0>]
```

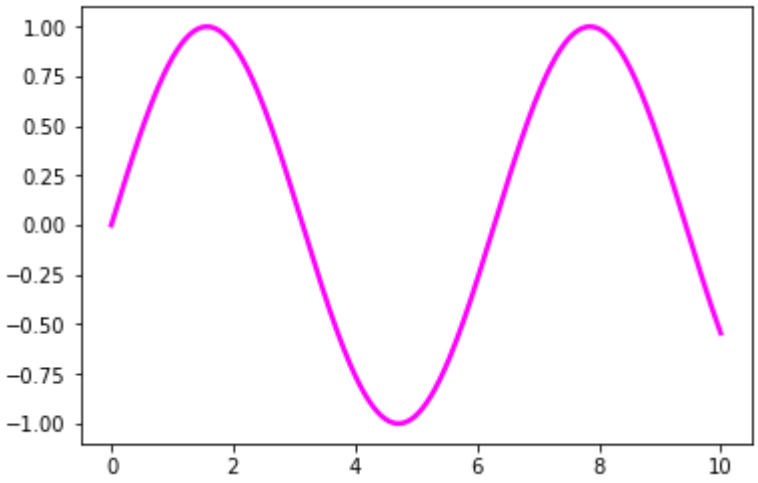


In [57]:

```
# You will rarely work with variables to plot that are this simple.
# Instead, it makes sense to learn how to define your x axis as continuously
# increasing so that you can plot your variable of interest on y-axis

# Here we take advantage of numpy's functions linspace and sin.

import numpy as np

x = np.linspace(0, 10, 1000)

plt.plot(x, np.sin(x), c = 'magenta', linewidth = 2.5)
```

Out[57]:

```
[<matplotlib.lines.Line2D at 0x7f89d16e8518>]
```

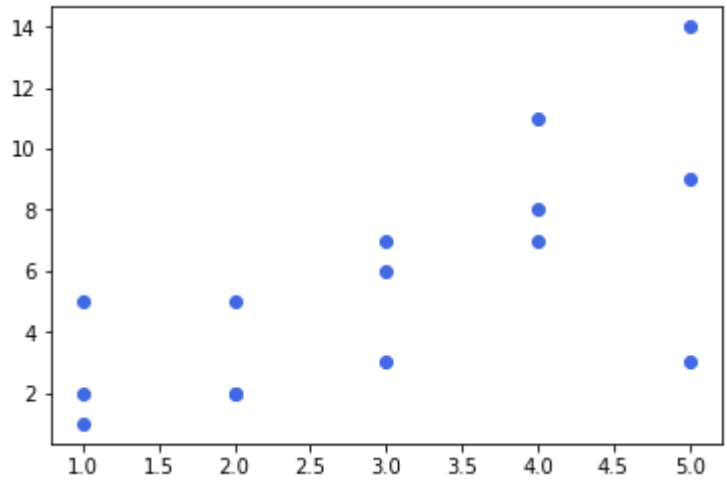


In [58]:

```
# It's likely you'll often want to look at a relationship between two variables.
# You can do this simply by scattering them in matplotlib...

x = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
y = [2, 5, 6, 11, 14, 5, 2, 7, 8, 9, 1, 2, 3, 7, 3]

plt.scatter(x, y, color = 'royalblue')
```

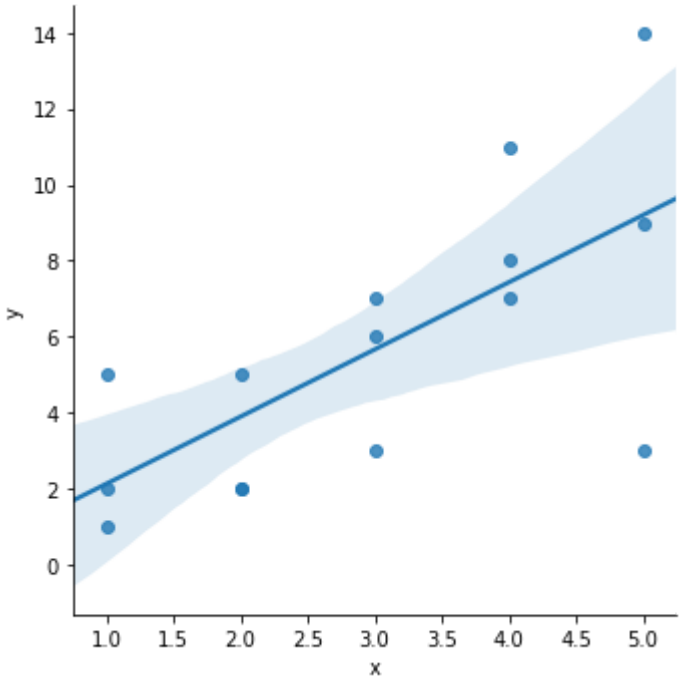
Out[58]:

```
<matplotlib.collections.PathCollection at 0x7f89d16c9ac8>
```



In [59]:

```
# Alternatively, some more powerful visualizations are available in seaborn.
# Here, we plot the exact same as in the figure above, but denoting the line of
 best
# fit for this data.

import pandas as pd
dataframe = pd.DataFrame({'x': x,
                          'y': y})


import seaborn as sns
g = sns.lmplot('x', 'y', data = dataframe)
```



In [60]:

```
# Want to know how related your variables are?
# There are always multiple ways to do things in Python. Here is an example of
# how even something as simple as a correlation between two variables can be cal
culated
# using multiple packages.

x = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
y = [2, 5, 6, 11, 14, 5, 2, 7, 8, 9, 1, 2, 3, 7, 3]

print('Correlation using numpy: ', np.corrcoef(x, y)[0][1])

from scipy.stats import linregress
print('Correlation using scipy: ', linregress(x, y)[2])
```

```
Correlation using numpy:  0.6923521761619424
Correlation using scipy:  0.6923521761619424
```

# Task: Some visualization exercises

So far in this class we've avoided using external datasets, but big data are hard to visualize with simulated sets. So, for th next few examples we'll ask you to play with an existing dataset for Python called Iris, which contains some flower feature data.

This data is in a pandas dataframe, which means some of the functions that are most useful to work with it will be other pandas functions. Working with a dataset like this is a great example of a task that will require a bit of googling and pasting together bits of code other people have written or reading some documentation for functions you are interested in.

In [61]:

```
from sklearn import datasets
import seaborn as sns

iris = sns.load_dataset("iris") #load the available data set
iris.describe() #useful method for taking a look at your data
```

Out[61]:

| | sepal_length | sepal_width | petal_length | petal_width |
|---|---|---|---|---|
| **count** | 150.000000 | 150.000000 | 150.000000 | 150.000000 |
| **mean** | 5.843333 | 3.057333 | 3.758000 | 1.199333 |
| **std** | 0.828066 | 0.435866 | 1.765298 | 0.762238 |
| **min** | 4.300000 | 2.000000 | 1.000000 | 0.100000 |
| **25%** | 5.100000 | 2.800000 | 1.600000 | 0.300000 |
| **50%** | 5.800000 | 3.000000 | 4.350000 | 1.300000 |
| **75%** | 6.400000 | 3.300000 | 5.100000 | 1.800000 |
| **max** | 7.900000 | 4.400000 | 6.900000 | 2.500000 |

In [62]:

```
# Let's have a look at the data. for 150 flowers, we have their sepal length/wid
th, petal length/width and species. Neat!
print(iris.shape)
iris.head()
```

```
(150, 5)
```

Out[62]:

| | sepal_length | sepal_width | petal_length | petal_width | species |
|---|---|---|---|---|---|
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 | setosa |

## TASK: DATA PLOTTING

Try to plot lengths against widths of the petals for all flower types. The kind of plot you'll want to use is most likely a scatterplot, so see if you can google your way to syntax that will take petal length and width as inputs for each datapoint.

In [63]:

```
# Your code for task 1 goes here:

sns.scatterplot(iris['petal_length'],iris['petal_width'])
```

```
---------------------------------------------------------------------
--------
AttributeError                            Traceback (most recent ca
ll last)
<ipython-input-63-02d091fd63ff> in <module>()
      1 # Your code for task 1 goes here:
      2
----> 3 sns.scatterplot(iris['petal_length'],iris['petal_width'])

AttributeError: module 'seaborn' has no attribute 'scatterplot'
```

Now try to change your scatterplot code from the cell above to color different species of flowers differently. Hint: Seaborn is fancy, so it will call color "hue"... look for hue designations in scatterplot documentation for help.

In [ ]:

```
# Your code goes here


import seaborn as sns
sns.scatterplot(iris['petal_length'],iris['petal_width'],hue=iris['species'])
```

### TASK: DATA VISUALIZATIONS

1. Try to plot all datapoints of petal lenghts by species in this dataset. What you want are little clusters of petal length values distributed by type of flower in your dataset. Hint: you might want to look at **swarmplots** using seaborn for this.

In [ ]:

```
# Your code for task 1 goes here:

sns.swarmplot(x=iris['species'],y=iris['petal_length'])
```

1. When you are done with 1, try to plot means and standard deviations of petal lengths by species. Essentially, you are trying to create the exact same visualization as in the previous task, but now instead of seeing every data point you are interested in seeing the key values of your set - mean and stdev. Hint: you might want to look at **boxplots** using seaborn for this.

In [ ]:

```
# Your code for task 2 goes here:

sns.boxplot(x=iris['species'],y=iris['petal_length'])
```

# PART3: OUTLOOK

We're now at the end of our little taster session. Good job!

Most importantly: Don't worry if you feel a little bit overwhelmed. We covered quite a lot of material that is usually taught over several weeks. But this notebook is self-contained and will remain available online. Feel free to revisit it whenever you feel like learning more about Python or brushing up your skills!

Also, there is no need to remember every single nitty gritty bit of a programming language. Even experts spend most of their time googling and searching stackoverflow.com for solutions to their problems. That's one of the great benefits of working with computers and being connected to the internet! A solution is always only a few clicks away.

Below we've collected a few ressources that we think come in handy if you'd like to learn more. Thanks a lot for your attention!

## Resources

**Learn coding!**

1. A complete course https://www.learnpython.org (https://www.learnpython.org)
2. Ditto https://www.w3schools.com/python/ (https://www.w3schools.com/python/)
3. Advanced Course https://automatetheboringstuff.com (https://automatetheboringstuff.com)

**How to install Python**

1. Just python https://www.codecademy.com/articles/install-python (https://www.codecademy.com/articles/install-python)
2. The Jupyter notebook (the thing you've been working with today) https://jupyter.org/install (https://jupyter.org/install)
3. Anaconda (a collection of useful packages and other software for data scientists) https://www.anaconda.com/distribution/ (https://www.anaconda.com/distribution/)

**Text Editors**

Note: If you don't use notebooks, you'll need a text editor to writer your python code. Whatever you do, don't use Microsoft Word! You want something that automatically highlights code (makes it easier to distinguish variables, functions and the like), and provides autocompletion (saves a lot of time!). Below are a few free and popular choices:

1. atom editor https://atom.io/ (https://atom.io/)
2. vscode https://code.visualstudio.com/ (https://code.visualstudio.com/)
3. Sublime https://www.sublimetext.com/ (https://www.sublimetext.com/) All thes editors support multiple programming languages. Thus no need to switch between programmes. I write my Python, Matlab, Javascript code and documentation in Markdown or Latex in Atom. Saves so much time!

**All in One Solutions**

You can also use an all in one solution that provides you with an editor, code interpreter (the thing that allows the computer to read and execute python code) and file managers. A bit like the Matlab software or R-Studio, which you might have heard of, but for python:

1. Spyder (free) https://www.spyder-ide.org/ (https://www.spyder-ide.org/)
2. Pycharm (free basic and commercial pro version) https://www.jetbrains.com/pycharm/ (https://www.jetbrains.com/pycharm/)

**Coding Challenges**

These challenges are a great way to test your understanding of computer science concepts and improve your logical thinking / problem solving skills. Also good to know: Almost every company uses these to assess candidates for software engineering / data science positions

1. Hackerrank https://www.hackerrank.com/ (https://www.hackerrank.com/)
2. Leetcode https://leetcode.com (https://leetcode.com)

**Python for Psychologists**

You can use python to run your own experiments, analyse behavioural, eye-tracking and neuroimaging data and to make beautiful visualisations of your results. Below are a few pointers https://www.marsja.se/best-python-libraries-psychology/ (https://www.marsja.se/best-python-libraries-psychology/)