School of Mathematics and Physics – UNIVERSITY OF PORTSMOUTH

BASIC INTRODUCTION TO PYTHON PROGRAMMING

DR SERGI SIMON

Lecture Notes used in Semester 1 of the module INTRODUCTION TO COMPUTATIONAL PHYSICS (U24200, years 2019/2020, 2020/2021)

FURTHER READING:

Online courses (different platforms)
Hetland, M. L., *Beginning Python From Novice to Professional*
Norton, P. et al., *Beginning Python, Wiley Publishing Inc.*
MathWorks Inc, *The Student Edition of Matlab*
Wilson, H. B. et al, *Advanced Mathematics And Mechanics*

Portsmouth, July 15, 2021

I Introduction to Python

- Created by **Guido Van Rossum** (1991). OS *Amoeba* and user interface *Bourne shell* did not mesh well. He devised a programming language to communicate both more fluidly.
- Despite the logo, name stems from Van Rossum's predilection for *Monty Python*.
- Programming languages can be high-level or low-level:
  - **Low-level languages**: syntax is closer to machine language (0's and 1's).
  - **High-level languages**: their syntax is closer to a natural human language.

  Python is a **high-level language** – so much so, that an English speaker can understand what any Python routine is doing.
- **Clear and simple grammar**: no symbols, e.g. semicolon, used by other languages.
- **Strongly typed** (like Java, C++, unlike PHP) e.g. integer variables distinguishable from strings
- It is also **dynamically typed**, i.e. you can establish the type at any point – whereas in a **static** type system, e.g. Java or C++, you have to declare the variable type *before* using it.

- Python is an **object-oriented (OOP)** programming language. We will see what this means later.
- **Open source**.
- **Easy to learn**. Hence the usual choice to introduce neophytes to programming languages.
- **Large standard library**, i.e. multiple useful classes and libraries come by default.
- It is an **interpreted language**, i.e. does not need the intermediate step of *compilation*.
- **Versatile**: useful to create any type of application (same as Java).
- **Multi-platform**: can be executed in Linux, Windows, Mac, ...
- **Basic Installation and/or use on your own computer:**
  - Go to `https://www.python.org/downloads` and download latest version...
  - ... or download Anaconda Distribution from `https://www.anaconda.com/distribution`
  - ... or use `http://sciserver.org`
- **Basic use on University computers:**
  - AppsAnywhere + Anaconda distribution

How do we write and execute Python:

- With files having extension `.py`. For instance if we open a file called `hello.py` using an editor (the default IDLE Python editor, Spyder, Text Sublime, Notepad++, Eclipse, ...) and write in it:

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

  save it and run it on a command line writing `python hello.py` (using the IDLE shell or a Mac/Windows/Linux terminal) and our output will be

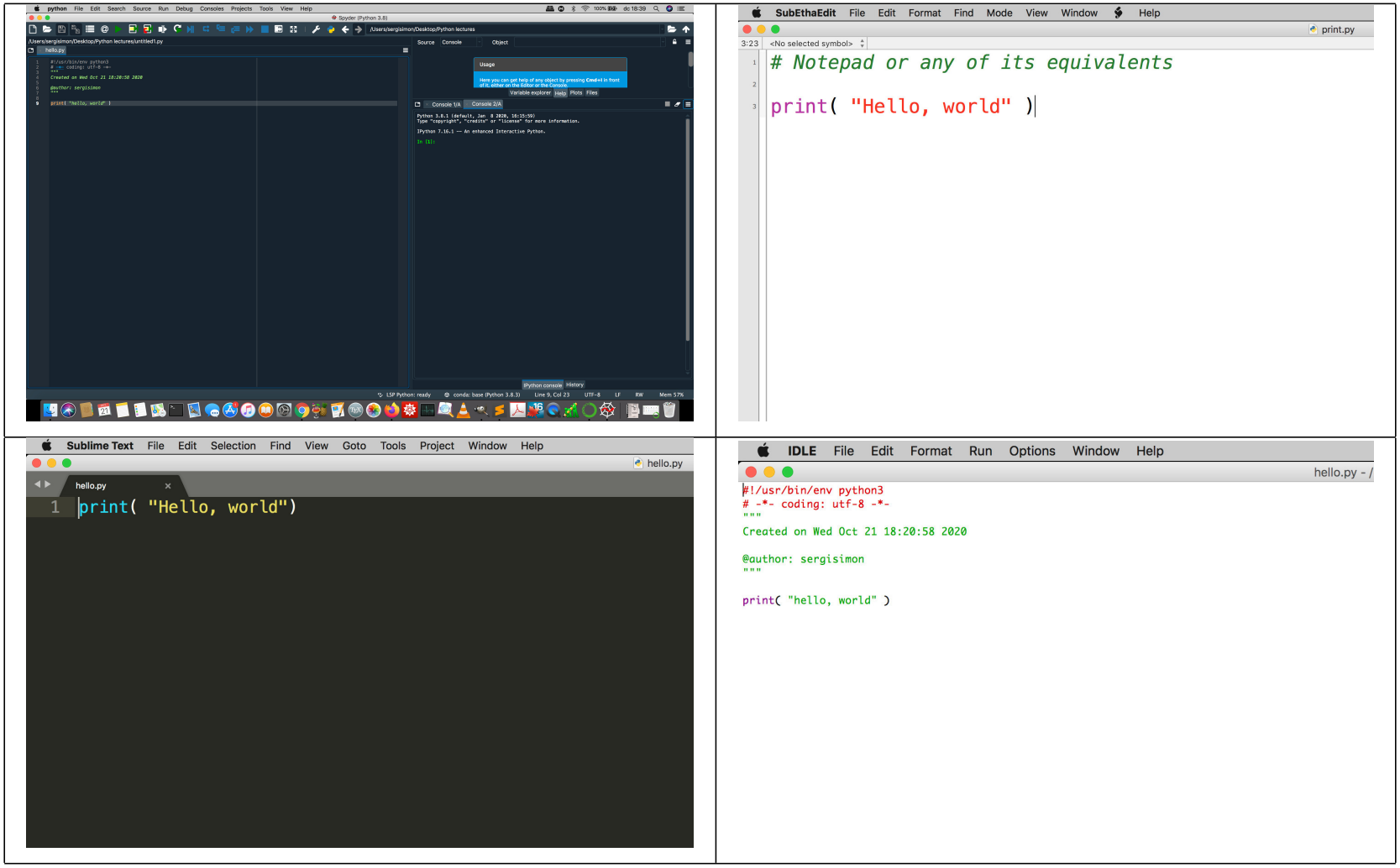
  ```
  Hello, world
  ```

- You can use any Python-adapted environment via Anaconda, e.g. Spyder itself to run the `.py` file.
- You can write command lines (using the IDLE shell or a Mac/Windows/Linux terminal), e.g.

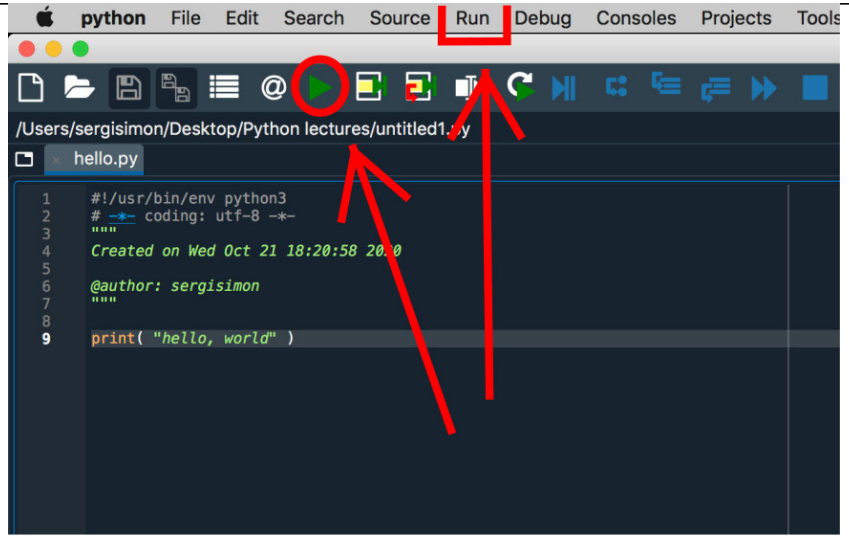
  `C:\Username>python` or `C:\Username>python3`

  which will allow you to test short amounts of code if you don't want to create a new `.py` file.
- You can use `Sciserver.org` or **Anaconda** to run **Jupyter notebooks** which combine text with code.
- You can use any other coding platform, e.g. `REPL.it`.
- See the next four pages for examples on how to edit and compile Python files.

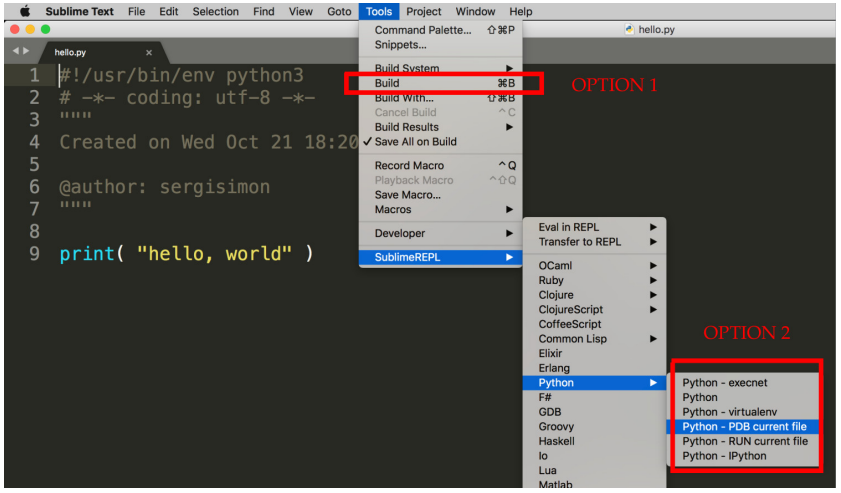
- Create the file (for instance `hello.py`), using Spyder, any basic text editor, e.g. Notepad or Sublime Text, or the default Python IDLE shell (if you first installed Python on your computer, see `https://www.python.org/downloads`):

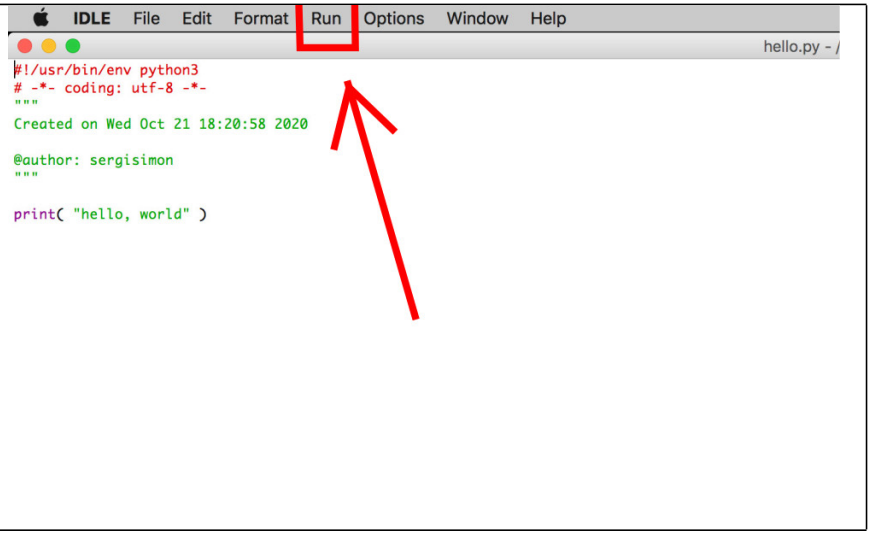


- Save your file. You can open it again with any editor as well as open a file created by someone else.
- You can then run the program; there are different options depending on your editor:

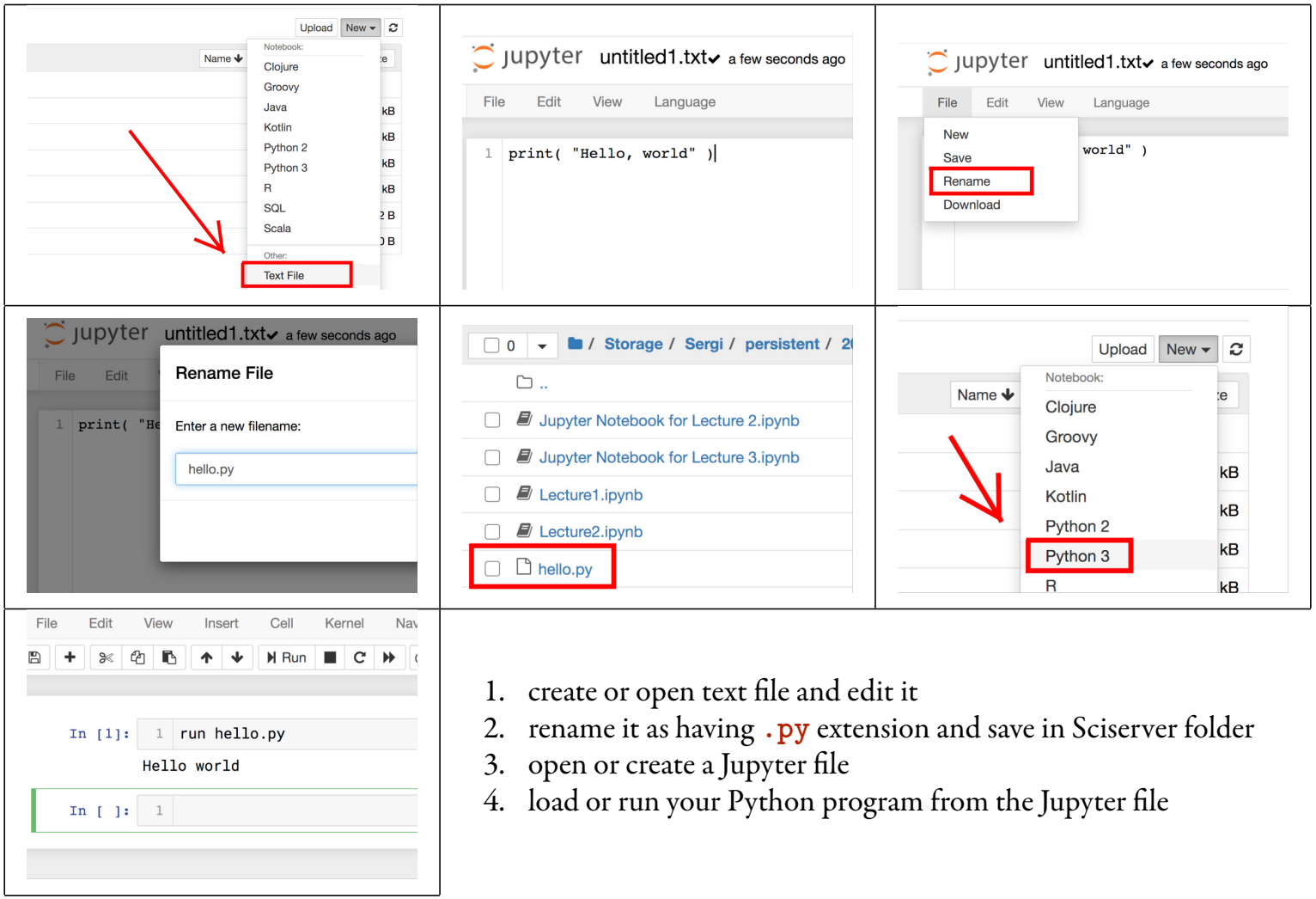


1. this list is **not exhaustive**; there are many editors that you can work with
2. most text editors, however, do not have a Python compiler
3. you can also search for online Python editors and compilers on the internet





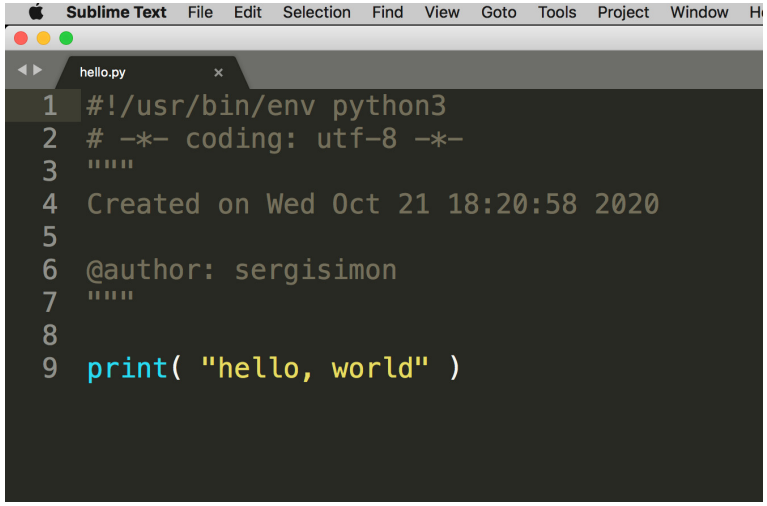
- You can use SciServer to create files and run them from a Jupyter notebook, but it is less practical:

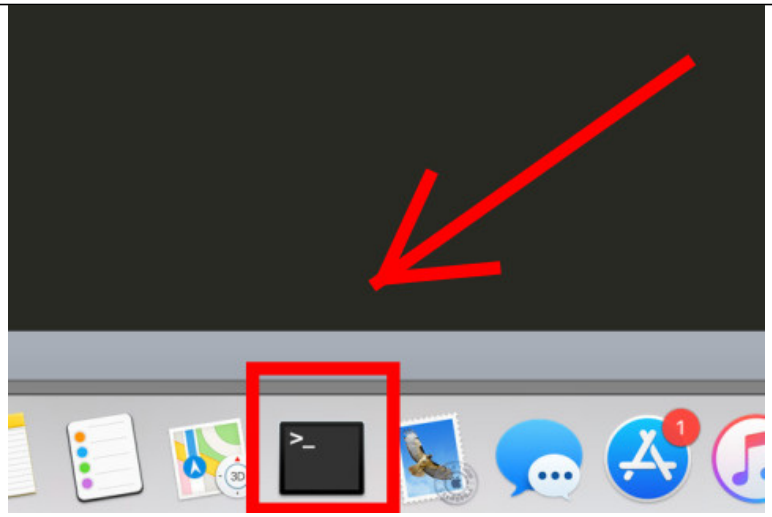


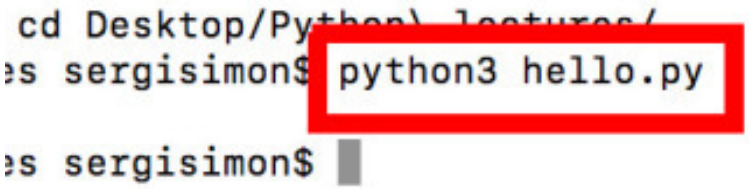
1. create or open text file and edit it
2. rename it as having `.py` extension and save in Sciserver folder
3. open or create a Jupyter file
4. load or run your Python program from the Jupyter file

Although if your `.py` file had been created with another editor (e.g. those in page 1), then uploading it to SciServer and running it on a Jupyter notebook is still a comfortable option.

- You can also use a command prompt to run Python programs. I will illustrate it on Mac (and it will be the same as in Linux); be sure to google images of Windows command prompts running Python if you are a Windows user.





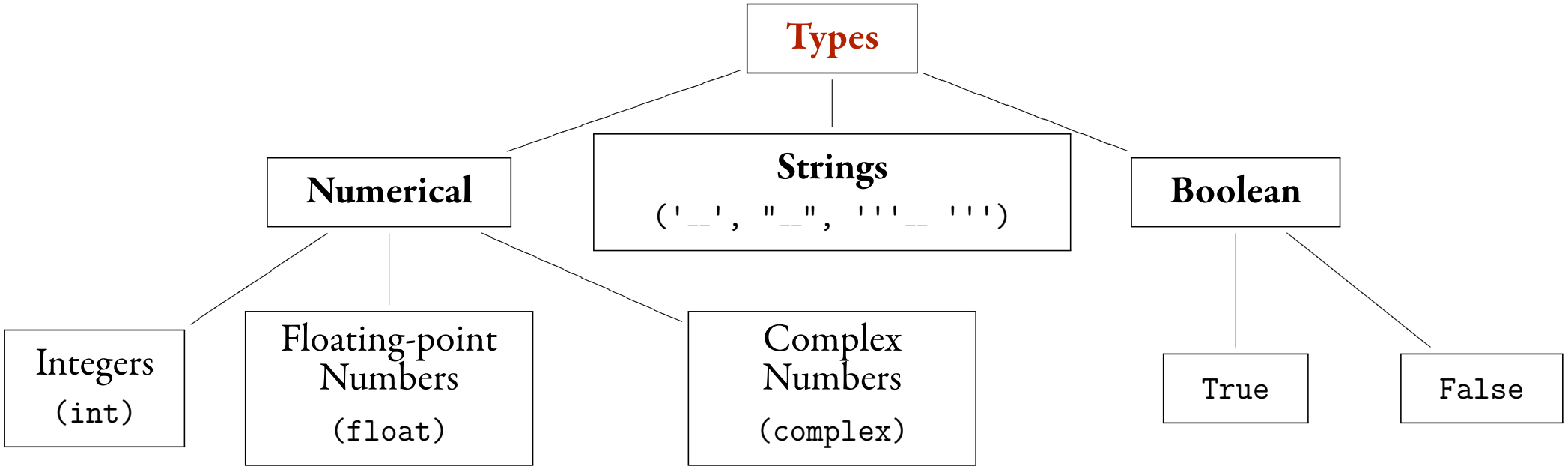


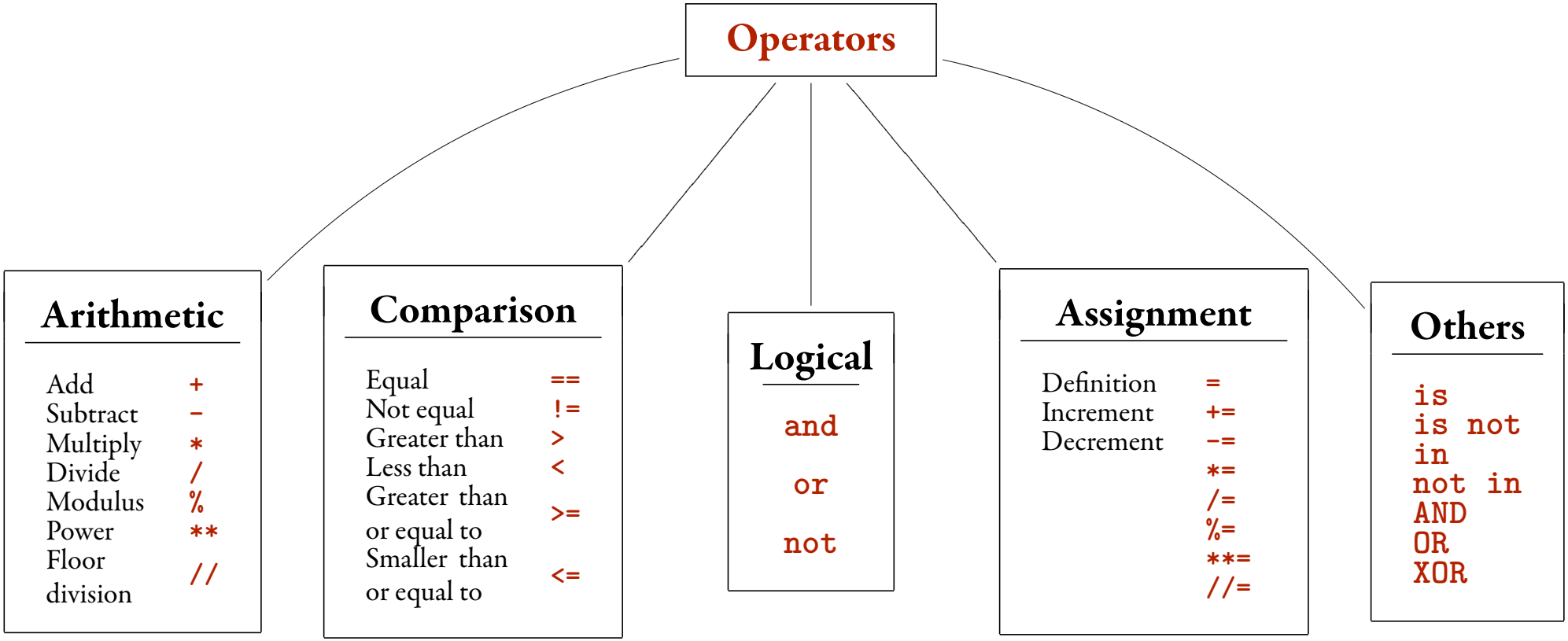
1. create/open/edit/save `.py` file with any editor (page 1)
2. open a terminal or command prompt and make sure your working directory is the one containing the file
3. write `python` or `python3` followed by file name
4. if a written output is expected, it will appear on the same terminal; other outputs (modifying files, etc) will be visible elsewhere, more on this in a few weeks

Remarks

- We will be using mostly Spyder in class because it has the advantage of combining Python source code and output in the same console, and because it is available for you to use in University computers, but this is only one out of many possible methods.
- There are plenty of ways to edit, run, and edit and run Python files, e.g. Visual Studio, and we strongly encourage you to search them on the internet.
- The one thing you must always remember, is that Python files have extension `.py`.
- Python files can import or "call" other Python files, and we will be studying this in a few weeks.

1 Types, operators, variables and syntax basics




Operators
Arithmetic
Add +
Subtract -
Multiply *
Divide /
Modulus %
Power **
Floor division //
Comparison
Equal ==
Not equal !=
Greater than >
Less than <
Greater than or equal to >=
Smaller than or equal to <=
Logical
and
or
not
Assignment
Definition =
Increment +=
Decrement -=
*=
/=
%=
**=
//=
Others
is
is not
in
not in
AND
OR
XOR

- **Variables** correspond to spaces in the memory of a computer (containers) wherein a data value (numerical, string, etc) is stored, and can be changed during execution (hence the name).
- Unlike other languages, Python has no command for declaring a variable. A variable is created the moment you first assign a value to it.
- Variable names are **case-sensitive** (i.e. `name` and `Name` are different) and:
  - cannot start with a number, e.g. `2n` is not a valid variable name.
  - must start with a letter or with the underscore _ character
  - can only contain alpha-numerics and underscores (A-z, 0-9, and _), e.g. `my_number`
  - **optional**: `i`, `t` usually reserved for loop indices,
  - **optional**: `n`, `m` traditionally reserved for integer variables.
- The **type** of a variable is not specified by the container but by the **content**. For instance, in other languages such as Java or C++, the type is defined by the container, i.e. the variable itself:

```
int number = 5;
```

  whereas if in Python we write `number = 5`, this is automatically a numerical and integer variable because it is given by the *content* (i.e. the value), not by the container.
- Python is 100% object oriented, thus everything (including variables) are **objects**.

- If we open a file (we can call it `variable.py`) and write the following:

```
name=1
print ( type( name ) )
```

  save the program and run it, we will obtain the output `<class 'int'>`.
- If we repeat the process replacing `name=1` by `name=2.4`, we obtain `<class 'float'>`
- Replacing it by `name="hello"` or `name='hello'`, we obtain `<class 'str'>`. Strings are surrounded either by single or by double quotation marks, i.e. 'hello' is the same as "hello"
- We can also use triple quotes to allow multiple lines for a single string, e.g.

```
name= '''This is
a sample message'''
print ( type( name ) )
print ( name )
```

or

```
name= """This is
a sample message"""
print ( type( name ) )
print ( name )
```

we obtain

```
<class 'str'>
This is
a sample message
```

- **Comments** are used to explain Python code. Commented lines are not executed.
- Everything in the same line and after # will be commented and thus ignored by Python:

```
# The print function yields
# the quoted output onscreen
print ( "hello world" ) # this is it
```

produces output `hello world`.

- Triple quotes serve the same purpose but can comment multiple lines at once:

```
'''The print function yields
the quoted output onscreen'''
print ( "hello world" )
```

or

```
"""The print function yields
the quoted output onscreen"""
print ( "hello world" )
```

produce the same output `hello world` as above.

- `\n` and `\t`: newline and tabulation; `print("Hello \t Goodbye \n hello again")` produces

```
Hello   Goodbye
hello   again
```

You will often combine strings and numbers, e.g.

```
number1 , number2 = 1,2
print("The sum of ",number1," and ",number2," is ",number1+number2)
```

You can create more examples yourselves in this week's Jupyter notebook.

2 Functions

- A **function** is a set of lines (i.e. a block of code), working as a single unit and carrying out a specific task **only when it is called**. A function:
  - can **return data**
  - can **have data (known as parameters or arguments) passed into it**
- Functions can be:
  - **built-in** or **predefined**, e.g. `print` and `type` which we have already seen, and many others (`abs`, `max`, ...), see e.g. `https://docs.python.org/2/library/functions.html`.
  - created by the user (meaning: you), which we will learn how to do these next weeks.
- **Advantage**: the ability to reuse code (whenever it is necessary)
- **Syntax**: depending on whether or not arguments are passed into the function,

```
def function_name ():
    instructions...
    return... (optional)
```

or

```
def function_name (parameters):
    instructions...
    return... (optional)
```

every single line after `def` must be indented

- To call the function, we use the function name followed by brackets: `function_name ()` in the first case, `function_name (parameters)` in the second one.
- Functions are called **methods** if they are defined in a **class** (which we will explain in the future).

- First example: write a program named `conversion.py` as follows:

```
def miles_to_km ( miles ):
    return miles*1.6
print(miles_to_km(7.1))
```

  Run the file: you should obtain `11.36` as an output. Change `7.1` and you will change the output.
- Alternatively, you could have used a variable to store the product before returning it:

```
def miles_to_km ( miles ):
    x=miles*1.6
    return x
print(miles_to_km(7.1))
```

- Variable `x` above was **local** after the assignment, i.e. recognised only by the function it was first defined in. This is in contrast to other languages where variables are **global** (their range encompasses the entire program) unless specified otherwise. For instance, in the following piece of code

```
y=1.6
def miles_to_km ( miles ):
    x=miles*y
    return x
print(miles_to_km(7.1))
```

  `y` is *global* with respect to `miles_to_km` (or local to a wider function whose boundaries are not visible here), whereas `x` is *local* to `miles_to_km` because it was first defined therein.

- Let us see this with an incorrect example. Assume our entire program reads as follows:

```
def miles_to_km ( miles ):
    x=miles*1.6
    return x
print (x)
```

The output will be an error message containing the following:

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

This is because `x` was first *defined* (not just *referenced*) within function `miles_to_km`, hence local to it. The rest of this program, therefore, does not acknowledge `x`.

- However, if we had first defined `x` outside of the function then called the function,

```
x=1
def miles_to_km ( miles ):
    x=miles*1.6
    return x
print (miles_to_km ( 3 ))
print (x)
```

we would have obtained `1` as the output for `x` (and `4.8` for the other `print` command). The program is technically correct, but `x` is reused in a misleading way.

- Even if we made no use of argument `miles` and operated only with *apparently* global values,

```
x=1
def miles_to_km ( ):
    x=3*1.6
    return x
print (miles_to_km ( ))
print (x)
```

  The outputs will still be `4.8` and `1` respectively. `x` would remain unchanged and it would also be one variable too many for the program to be optimal.

- We said before that defining a variable in a function immediately rendered it local. **If no value is assigned to it in that function, however, it is global**. Hence if a variable is only *referenced* inside a function, it is global. However if we assign any value to a variable inside a function, its scope becomes local to that unless explicitly declared global. This will also yield an error message:

```
x=1
def miles_to_km ( ):
    x=x*1.6
    return x
print (miles_to_km ( ))
print (x)
```

  because `x` had been defined outside the scope of the function, thus `x=x*1.6` requests overriding a variable (the right-hand side `x`) that had not been defined inside the function.

- Way to solve the problem, although the function will still not be optimal:

```
x=1
def miles_to_km ( ):
    y=x*1.6
    return y
print (x)
print(miles_to_km ( ))
```

  `x` remains *global* because we did not re-assign values to it inside the function. The function uses `x` correctly and returns adequate value, but still depends on defining `x` before calling the function.

- This, however, is better:

```
x=1
def miles_to_km ( miles ):
    y=miles*1.6
    return y
print (x)
print(miles_to_km ( x ))
```

  `x` has not changed thus is the value of the original variable. The value of the function of `x`, namely `1.6`, is printed correctly in the last line. There is clear distinction between the original variable `x` and the final value `miles_to_km ( x )`.

- Besides, in the above program we can call the function for any argument, e.g. `miles_to_km ( 4 )` will yield `6.4`.

- Hence these are correct definitions (and call examples) for this function

```
def miles_to_km ( miles ):
    y=miles*1.6
    return y
print(miles_to_km ( 4.7 ))     #or x=4.7 followed by print(miles_to_km(x))
```

```
def miles_to_km ( miles ):
    return miles*1.6
print(miles_to_km ( 4.7 ))     #or x=4.7 followed by print(miles_to_km(x))
```

- The local variable `y` in the first example could have been replaced by any other variable, e.g.

```
def miles_to_km ( miles ):
    whatever=miles*1.6
    return whatever
print(miles_to_km ( 4.7 ))     #or x=4.7 followed by print(miles_to_km(x))
```

`x` would do as well, even if you defined `x=4.7` outside of the function – but we do not encourage the use of the same variable name for both local and global variables.

- You can practice by modifying this week's Jupyter notebook, see what you get by playing with the variables.

- What if we wanted to keep `x` global inside the function? This is not necessary or advisable in this example, but it could be elsewhere. We use the keyword `global` in order to tell Python that, contrary to its usual practice, this variable that is being changed within the function is not local:

```
# This function modifies global variable x
def miles_to_km ( ):
    global x
    x=x*1.6
    return x
x=1
miles_to_km()
print(x)
miles_to_km()
print(x)
```

Calling the function twice changes the value of global variable `x` twice. Our printed output will be `1.6` followed by `2.56`.

- Passing arguments into the function, however, will often be the advisable path. For instance:

```
def add ( ):
    n1=3
    n2=2
    return n1+n2

print(add())
print(add())
print(add())
```

will return `5` thrice because the variables involved in the sum could not be modified. However,

```
def add ( n1, n2 ):
    return n1+n2

print(add(3, 2))
print(add(3,2.2))
print(add(0,-1))
```

and generally speaking any call of this `add` will work correctly if we pass *any* two arguments to it.

- You can pass any number of arguments. Assume we want to check the *distributive property* of integers, $a \cdot (b_1 + b_2) = a \cdot b_1 + a \cdot b_2$:

```
def dist_lhs ( a, b1, b2 ):
    return a*(b1+b2)
def dist_rhs ( a, b1, b2 ):
    return (a*b1)+(a*b2)
print(dist_lhs(1,2,4) == dist_rhs(1,2,4))
```

Forms enclosed in brackets, i.e. parentheses are treated as atoms and their content is computed before releasing it as part of a larger operation

will return `True` (and will remain so if you change `1,2,4` by any other three integers on both sides).

- Proper programming habits entail using ancillary local variables to store intermediate steps in functions and general calculations. For instance this:

```
def dist_lhs ( a, b1, b2 ):
    var1=b1+b2
    var2=a*var1
    return var2
def dist_rhs ( a, b1, b2 ):
    var1=a*b1
    var2=a*b2
    return var1+var2
print(dist_lhs(1,2,4) == dist_rhs(1,2,4))
```

In this case the number of operations was the same, but in other functions this will not be the case

yields the same output as the above, but abides by a way of writing that will be useful to minimise the number of calculations in future programs.

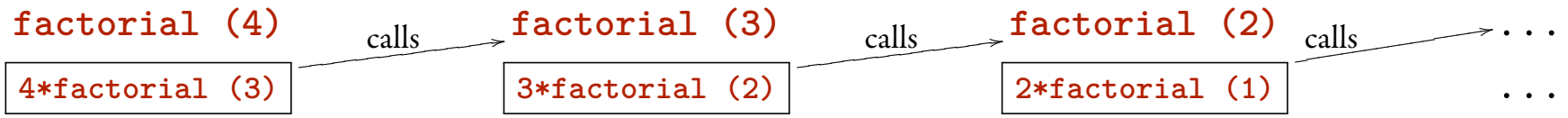
- A function that calls itself is **recursive**. For instance, let us compute the *factorial* of n:

$$\boxed{n! := n \cdot (n-1) \cdots 2 \cdot 1} \qquad n \text{ being a non-negative integer}$$

Notice that $n! = n(n-1)!$. Using this, consider the following recursive function:

```
def factorial ( n ):
    return n*factorial(n-1)
```

If this function works, then calling `factorial (4)` will trigger the following:



An *error message* ensues because the function needs a **base** (or **terminating**) **case**, i.e. a condition breaking the recursion chain. Otherwise it keeps calling itself: `factorial(0)`, `factorial(-1)`, ...

- This is done as follows (remember we will study conditionals more in detail later on):

```
def factorial ( n ):
    if n==0:
        var=1
    else:
        var=n*factorial(n-1)
    return var
```

based on the convention that $0! = 1! = 1$:

then calling `factorial (4)` will produce `24`, calling `factorial (5)` will produce `120`, etc.

- A function performs no further action after a `return` line, hence equally correct ways are:

```
def factorial ( n ):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

```
def factorial ( n ):
    if n==0:
        return 1
    return n*factorial(n-1)
```

thus `else` is not necessary – think why!

- **Example of functions calling other functions**: once you have defined `factorial` in any of the above ways, you can compute the *binomial coefficient* $\binom{n}{m}$, i.e. the number of ways to choose m elements from a set of n elements. This is given by the following:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \qquad \text{(other notations: } C(n,m),\ {}_nC_m,\ {}_nC^m,\ \ldots\ldots)$$

Our program `binomial.py` would read, for example,

```
def factorial ( n ):
    if n==0:
        return 1
    return n*factorial(n-1)

def binomial ( n, k ):
    var1=factorial(n)
    var2=factorial(k)
    var3=factorial(n-k)
    return int(var1/(var2*var3))

n=int(input("Write number n: "))
k=int(input("Write number k: "))
print( "The binomial coefficient is ", binomial(n,k))
```

Conversion to `int` type is necessary because `input` processes data as strings by default.

Exercises

- We like to avoid error messages, but we still have not explained **exceptions**. For the time being, adapt the above function `binomial` to return customised messages if the numbers `n,k` we feed the function make the computation of $\binom{n}{k}$ impossible as a positive integer.
- Write a program in a file `exponential.py` which takes variables `x` (a float) and `k` (an integer) as inputs, and returns the following approximation of the *exponential function* e^x:

$$f(x,k) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots + \frac{x^k}{k!} = \sum_{i=0}^{k} \frac{x^i}{i!} \qquad \text{(remember } x^0 = 0! = 1\text{)}$$

  Compare the output of this function for any `x` and for large values of `k`, with the numerical value $\boxed{e^x}$ or $\boxed{\exp}$ at `x` produced by any calculator. *We have not seen loops yet so think of a way to write this function without them.*
- Using the previous item, write a program called `hyperbolic.py` which approximates, for any given float variable `x`, the *hyperbolic sine and cosine of* `x`:

$$\cosh x = \frac{e^x + e^{-x}}{2}, \qquad \sinh x = \frac{e^x - e^{-x}}{2}$$

- The *Fibonacci numbers* are defined as follows:

$$F_0 = 0, \quad F_1 = 1, \qquad \boxed{F_n = F_{n-1} + F_{n-2}, \quad n > 1.}$$

  Thus F_0, F_1 are the initial conditions or values for sequence $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

  – Write a program `fibonacci1.py` taking any input `n` and returning the Fibonacci number F_n according to the above construction.

  – Another way of obtaining F_n is as follows:

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}, \qquad \text{where} \begin{cases} \varphi = \frac{1+\sqrt{5}}{2} \simeq 1.618033\ldots \text{ is the } \textbf{golden ratio} \\ \psi = -\varphi^{-1} = -\frac{1}{\varphi} \simeq -0.618033\ldots \end{cases}$$

  Write another program `fibonacci2.py` to compute F_n in this manner for any `n` obtained by input. You will have to import the `math` module in order to compute square roots. The way to do so is to write `import math` at the very beginning of the program, and then any time you need to compute a square root $\sqrt{y}$ all you have to do is write `math.sqrt( y )`.

- Write a program `temperature.py` that converts degrees Fahrenheit to degrees Celsius and vice versa. The program should give the user all the options you deem reasonable and should take these options (and the numbers) as input.

3. Collection data types

- Collection types or *arrays* can be divided in four categories in Python:
  - **Lists**, written with square brackets `[...]`
  - **Tuples**, written with round brackets `(...)`
  - **Sets**, written with curly brackets `{...}`
  - **Dictionaries**, with curly brackets and double-barreled entries `{⋆:⋆,...,⋆:⋆}`
- Their differences are based on different criteria:
  - Whether they are *ordered* or *unordered*:
    - in an **ordered** collection, order is relevant, for instance `[a,b,c]!=[b,a,c]`
    - in an **unordered** collection, order is irrelevant, for instance `{a,b,c}=={b,a,c}`
  - Whether they are *indexed* or *unindexed*:
    - in an **indexed** array, an item can be accessed by referring to its index number, e.g. `x[0]` will be the first element in `x`, `x[1]` the second, etc.
    - in an **unindexed** array, this is impossible
  - Whether they are **changeable** or **unchangeable** after their first definition.
  - Whether or not they **allow duplicate members**.

Lists

- Lists `[ ... ]` are data structures allowing us to store large amounts of values. They:
  - are **ordered**, i.e. if you change the order you change the list;
  - are **indexed**, i.e. its elements can be located by their position;
  - are **dynamically changeable** (both in length and in content) during the program;
  - **allow member duplication**.
  - allow membership checks with `in`.
- In Python (unlike in other languages) lists allow the storage of different types of values, e.g.

  ```
  x=[1,"hello",2.0]
  ```

  even lists can be members of other lists:

  ```
  y=[1,"goodbye",x]
  ```

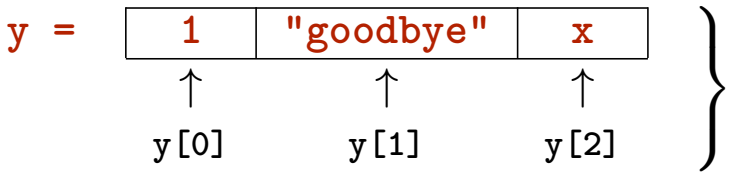
- To check whether an element is a member of a list, we use `in`:

```
mylist=[1,2,3,-1,1,0,0,"u"]
print(3 in mylist)
print(1 in mylist)
print("a" in mylist)
```

output →

```
True
True
False
```

- The **index** of an element is the position it occupies in the list minus one (indices start in 0), e.g.



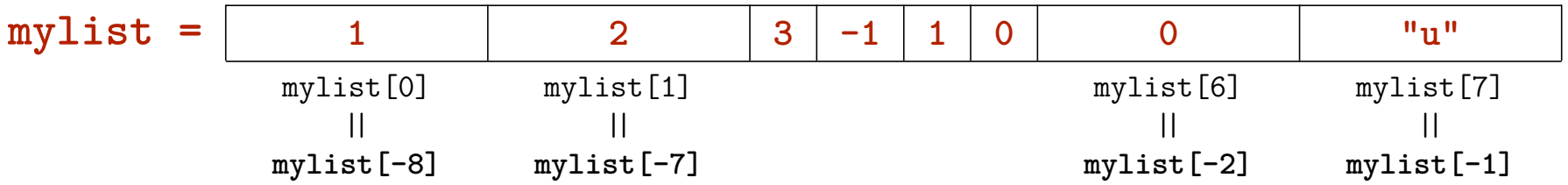
You can check this yourselves writing `print (y[0])`, `print(y[1])`, or `print(y[2])`.

- List members can be changed by reference to index:

```
x=[1,2,3,"name",3.2,2]
x[2]="something else"
print ( x )
```

output → `[1,2,"something else","name",3.2,2]`

- Negative indexing starts from the opposite end. For instance, `print(mylist[-4])` yields `1`.



- The length of a list is denoted by `len`. Thus the indices of a list `L` are within the range `0`, ..., `len(L)-1`.

```
x=[1,-4.1,"a","c",3,56.3,-10.1]
print ( len(x) )
```

output → 7

- Assignment affects members if done properly:

```
[a,b,c]=[1,0.2,3]
print(a)
print(b)
```

output →
```
1
0.2
```

- Index ranges retrieve list fragments: `x[j:k]` = elements in `x` having index in $\{j, \ldots, k-1\}$.
  - Negative ranges work the same way: `x[-j:-k]` stands for indices from $-j$ (incl.) to $-k$ (excl.).
  - Open ranges work intuitively as well.

```
x=[1, -4.1, "a", "c", 3, 56.3, -10.1]
print ( x[2:3] )
print ( x[2:5] )
print ( x[-4:-1] )
print ( x[:2] )
print ( x[4:] )
```

output →

```
["a"]
["a", "c", 3]
["c",3,56.3]
[1, -4.1]
[3, 56.3, -10.1]
```

- We can also append elements (add them to the end of a list):

```
list1=[-1,2,3,0.3,"a"]
list1.append(34)
print ( list1 )
```

output → `[-1,2,3,0.3,'a',34]`.

- `list.insert(k,member)` places member in index `k` (thereby raising `len(list)` by one):

```
list1=[-1,2,3,0.3,"a"]
list1.insert(1,"b")
print ( list1 )
```

output → `[-1,'b',2,3,0.3,'a']`

because `"b"` was placed in index `1` (hence pushing the rest of the list one place to the right).

- Needless to say, `list.insert(0,···)` is the same as *prepending*, i.e. placing `···` at the beginning.
- `extend()` concatenates lists:

```
list=[1,2,3,4,5,6]
list.extend([1,7,8])
print ( list )
```

output → `[1,2,3,4,5,6,1,7,8]`.

- Deletion can be done as follows:
  - `remove` eliminates the first occurrence of a specified member:

```
list1=[1,2,3,"a",5,"b","a"]
list1.remove("a")
print(list1)
```

output → `[1, 2, 3, 5, 'b', 'a']`

    and we would have to call `remove()` again to eliminate all occurrences of `"a"`.

  - `pop()` removes a specified index or removes the last item if no index is specified:

```
list.pop(1)
print(list)
list.pop()
print(list)
```

output →
```
[1, 3, 5, 'b', 'a']
[1, 3, 5, 'b']
```

  - `del` removes a specified index:

```
del list[2]
print(list)
```

output → `[1, 3, 'b']`

    or deletes a whole list by writing, e.g. `del list`.

  - `clear` does precisely what it name indicates:

```
list.clear()
print(list)
```

output → `[]`

- `clear` is equivalent to successive applications of `pop`, `remove` or `del`:

```
list=[1,2,3,4]
print(list)
list.pop()
print(list)
list.pop()
print(list)
list.pop()
print(list)
list.pop()
print(list)
print(len(list))
```

output

```
[1,2,3,4]
[1,2,3]
[1,2]
[1]
[]
0
```

```
list=[1,2,3,4]
print(list)
list.clear()
print(list)
print(len(list))
```

output

```
[1,2,3,4]
[]
0
```

Thus an empty list `[ ]` has length zero but is still a list, and can be replenished again:

```
list.append(1)
print(list)
```

output

```
[1]
```

- `sort` does what its name implies to numerical lists:

```
list1=[1,2,3,4]*3
list1.sort()
print(list1)
```

output

```
[1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4]
```

- Assignment, e.g. `list2=list1` translates in passing a *reference* (a place within memory) from `list1` to `list2`. This means that any change made to `list1` is automatically made to `list2`:

```
list1=[1,2,3,4]
list2=list1
list1.append(5)
print(list1)
print(list2)
```

output

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

- In order to copy a list to another list that will be henceforth independent of it, use `copy()`:

```
list1=[1,2,3,4]
list2=list1.copy()
list1.append(5)
print(list1)
print(list2)
```

output

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4]
```

- Operator `+` concatenates the given lists (same as `extend` explained before):

```
list1=[1,2,3,4]+[-1,-2,-3]
print(list1)
```

output

```
[1, 2, 3, 4, -1, -2, -3]
```

- And by this logic, "multiplying" a list by an integer concatenates copies of it:

```
list1=[1,2,3,4]*3
print(list1)
```

output

```
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

- `reverse` does what it name implies to any list. Check this week's Jupyter notebook.

Tuples

- Tuples `(...)` are **unchangeable** arrays, i.e. cannot be modified after their creation. Hence
  - adding (`append`, `extend`, `insert`...),
  - removing (`remove`, `pop`, `del`, `clear`...),
  - or changing individual members,

  are not possible in any given tuple.
- They do allow extraction of portions, but the extracted output is a new tuple.
- They **allow duplicates**, e.g. `(a,a,b)` makes sense and is different from `(a,b)`.
- From Python 2.6 onward, they are **indexed**, i.e. members can be located with an index; index ranges, negative indexing etc works just like in lists.
- Membership of a given element in a tuple can be checked (with `in`).
- Why are they useful and when are they advantageous?
  - Faster to process and occupy less memory space than lists (they are more *optimal*).
  - They allow us to format strings.
  - They can be used as keys in dictionaries (unlike lists).

- Tuples are written:
  - with round brackets,

```
tuple1 = ("a",1,2,3,0,"hello there")
print(type(tuple1))
```

output

```
<class 'tuple'>
```

  - or with no brackets at all (this is specific to tuples and does not apply to lists):

```
tuple2 = "a",1,2,3,0,"hello there"
print(type(tuple2))
```

output

```
<class 'tuple'>
```

- You can create any array with only one item. However, in such conditions a tuple needs a comma at the end to be recognised as a tuple (whereas a list does not):

```
list_1=[1]
maybe_tuple_1=(1)
maybe_tuple_2=(1,)
print(type(list_1))
print(type(maybe_tuple_1))
print(type(maybe_tuple_2))
```

output

```
<class 'list'>
<class 'int'>
<class 'tuple'>
```

Needless to say, the aforementioned item implies you can write `(1,)` as `1,` and it will still be recognised as a tuple.

- You try to change the length or the contents of a tuple, and you get an error:

```
tuple1.insert(1,"a")      ——output——>  AttributeError:
                                       'tuple' object has no attribute 'insert'

tuple1.append("a")        ——output——>  AttributeError:
                                       'tuple' object has no attribute 'append'

tuple1.pop()              ——output——>  AttributeError:
                                       'tuple' object has no attribute 'pop'

tuple1[0]=3               ——output——>  TypeError:
                                       'tuple' object does not support item assignment

del tuple1[0]             ——output——>  TypeError:
                                       'tuple' object doesn't support item deletion

tuple1.remove("a")        ——output——>  AttributeError:
                                       'tuple' object has no attribute 'remove'
```

  and same goes for **`extend`**, **`clear`**, etc.

- But using function **`list`** we can copy the tuple to a list and perform the above on the latter:

```
list1=list(tuple1)
print(list1)
```

——output——> `['a', 1, 2, 3, 0, 'hello there']`

- The same goes the other way with function **`tuple`** (albeit losing of course the ability to modify):

```
list1=[1,2,3]
mytuple=tuple(list1)
print(mytuple)
```

——output——> `(1, 2, 3)`

- Operator `+` (hence "multiplication" by integers) still works for concatenation, just like in lists:

```
tuple1=(1,2,3)
tuple2="a","b",4
tuple3=tuple1+tuple2
print(tuple3)
print(2*tuple2)
```

output →

```
(1, 2, 3, 'a', 'b', 4)
('a', 'b', 4, 'a', 'b', 4)
```

- Membership is still checked easily:

```
if 1 in tuple1:
    print("1 is in this tuple")
print("a" in tuple1)
```

output →

```
1 is in this tuple
False
```

- Indexing is the same as in lists in recent versions of Python, e.g. `print(tuple1[0])` yields `1`.
- There is a function useful to tuples and lists, namely `count`:

```
tuple_1="a","b",1,2,3,44,3
print(tuple_1.count("a"))
print(tuple_1.count(3))
```

output →

```
1
2
```

- Multiple member assignment works just like in lists:

```
u1,u2,u3,v,w,x,y=tuple_1
print(u1)
print(v)
```

output →

```
a
2
```

- Index ranges and `len` work exactly the same as in lists:

```
print(tuple_1[1:4])
print(len(tuple_1))
```

output →

```
('b', 1, 2)
7
```

Sets

- Sets `{ ... }` are **changeable**, **unordered** and **unindexed** arrays, i.e. can be modified, have no fixed order in which the items appear and items cannot be accessed with an index.
- Duplication is not allowed, i.e. every member is counted once:

```
set_1 = {1,2,"a",2}
print(set_1)
```

output

```
{1, 2, 'a'}
```

- Addition of one new element is carried out with `add`:

```
set_1.add(3)
print(set_1)
```

output

```
{3, 1, 2, 'a'}
```

- Addition of more than one element is carried out with `update` and either curly or square brackets:

```
set_1.update({3,4,5})
print(set_1)
set_1.update([4,5,6,"u"])
print(set_1)
```

output

```
{1, 2, 3, 'a', 4, 5}
{1, 2, 3, 'a', 4, 5, 6, 'u'}
```

- `len` and `remove` work same as in lists, as long as we are aware that duplication is not possible here:

```
set_1.remove(1)
print(set_1)
print(len(set 1))
```

output

```
{2, 3, 'a', 4, 5, 6, 'u'}
7
```

- Just like in lists, `clear` empties and `del` deletes completely. Same syntax (`set.clear()`, `del set`).

- `discard` works like `remove`, but if the element was originally absent, there is no error message:

```
set_1.discard(1000)
print(set_1)
```

output

```
{2, 3, 'a', 4, 5, 6, 'u'}
```

- Union and intersection work just like one would expect in any two sets:

```
set_1={1,2,3,"a",4}
set_2={2,4,"a","b",-1}
print(set_1.union(set_2))
print(set_1.intersection(set_2))
```

output

```
{1, 2, 3, 4, 'a', 'b', -1}
{2, 'a', 4}
```

- Functions `max` and `min` are shared by *numerical* lists, tuples and sets:

```
set_1={1,2,3,-4}
list_1=[1,2,7,-5]
tuple_1=(0.1,1,2,11)

print(max(set_1))
print(max(list_1))
print(max(tuple_1))
print(min(set_1))
print(min(list_1))
print(min(tuple_1))
```

output

```
3
7
11
-4
-5
0.1
```

- Other functions you can work on (see the Jupyter notebook for this week):

`copy()`	`difference()`	`difference_update()`	`isdisjoint()`
`issubset()`	`issuperset()`	`update()`	`symmetric_difference()`
`symmetric_difference_update()`	`pop()`	`intersection_update()`	

Dictionaries (short introduction, year 20/21)

Older, fuller version in the next three pages

- A fourth data array type, written with curly brackets `{ key_1:value_1,key_2:value_2,...}`, and
  - **indexed**, i.e. there exists a construct aimed at locating elements;
  - **unordered**, i.e. order is irrelevant (despite being indexed);
  - **changeable**, i.e. can have elements modified, added or removed;
- Allows us to store values of different types (`str`, `int`, `float`, arrays, even other dictionaries).
- Data are stored linked to a key; an association `key : value` is created for every stored element.
- Example: assume we want to keep tabs of famous films

these are the keys

```
mydict_1 = {
        "title"   :  "The Grapes of Wrath",
     "director"   :  "John Ford",
         "year"   :  1938
}
```

these are the values

- We will not study these in detail this year. You can try these separate lines of code yourselves:

```
mydict_1["year"]=1940
print(mydict_1)
print(mydict_1["director"])
```

```
x=mydict_1.get("year")
print(x)
y=mydict_1.get("country","nothing")
print(y)
print("country" in mydict_1.keys())
```

```
del mydict_1["director"]
print (mydict_1)
print (len( mydict_1 ))
print (mydict_1.keys())
print (mydict_1.values())
```

Dictionaries

- So far we have seen three types of data arrays: lists, tuples and sets. There is a fourth type, namely **dictionaries**, written with curly brackets `{ key_1:value_1,key_2:value_2,...}`, and
  - **unordered**, i.e. order is irrelevant;
  - **changeable**, i.e. can have elements modified, added or removed;
  - **indexed**, i.e. there exists a construct aimed at locating elements.
- Dictionaries are data structures allowing us to store values of different types (`str`, `int`, `float`, arrays, even other dictionaries).
- Data are stored linked to a key; an association `key : value` is created for every stored element.
- As said above, elements are not ordered, *yet* they can be located due to the above association.
- Example: assume we want to keep tabs of famous films

`title`, `director`, `year` are the **keys**

```
mydict_1 = {
        "title"   :   "The Grapes of Wrath",
     "director"   :   "John Ford",
         "year"   :   1938
}
```

these are the **values**

- The above data is wrong because the film was released in 1940. The way to correct this is the same we would used with indices in a list:

```
mydict_1["year"]=1940
print(mydict)
```

  yielding `{'title': 'The Grapes of Wrath', 'director': 'John Ford', 'year': 1940}`

- By the same principle,

```
mydict_2 = {
    "title": "Eraserhead",
    "director": "Steven Spielberg",
    "year": 1977
}
```

  would also need to be changed, using indexing `mydict_2["director"]="David Lynch"`.

- We access items by way of the same indexing principle or by using `get()`:

```
print(mydict_1["director"])
```
output → `'John Ford'`

```
x=mydict_1.get("year")
print(x)
```
output → `1940`

```
x=mydict_1.get("country","nothing")
print(x)
```
output → `"nothing"`

  where the second value in `get` is the value to be returned if the key is not found.

- `len( mydict_2 )`, `mydict_2.clear()` and `mydict_2.copy()` work just like they would in a list.

- We can decide we no longer want `mydict_2` to have a `"director"` slot:

```
del mydict_2["director"]
print (mydict_2)
```

output

```
{'title': 'Eraserhead', 'year': 1977}
```

This can also be done with `mydict_2.pop("director")` (check this yourselves!).

- We can check key membership with `in`:

```
if "year" in mydict_2:
    print("Indeed, 'year' a key in this dictionary")
```

output

```
Indeed, ...
```

- We can add a new member by adding a new key and linking it to a new value:

```
mydict = {"title": "Eraserhead", "director": "David Lynch", "year": 1977}
mydict["country"] = "US"
print (mydict)
```

output

```
{'title': 'Eraserhead', 'director': 'David Lynch', 'year': 1977, 'country': 'US'}
```

- The keys of `mydict` can be accessed via `mydict.keys()`. Ditto for `mydict.values()`

```
print(mydict.keys())
print(mydict.values())
print("year" in mydict.keys())
```

output

```
dict_keys(['title', 'director', 'year', 'country'])
dict_values(['Eraserhead', 'David Lynch', 1977, 'US'])
True
```

Exercises

- Build a function `reverse_tuple` that takes a tuple as an argument and returns the same tuple in reverse order. Test it in a file named `reverse_tuple.py`.
- In a file named `replace_array.py`, build functions taking any list (or tuple) and:
  - replacing a given member (provided via index) by a new one.
  - replacing all occurrences of a given member (provided via its value) by a new one.
  - returning the same array but with two indexed elements swapped.

  You can use `list.count ( member )` or `in` for the second item (or some alternative trick).
- Build a recursive function taking two numerical lists (or tuples) as inputs and returning their sum:

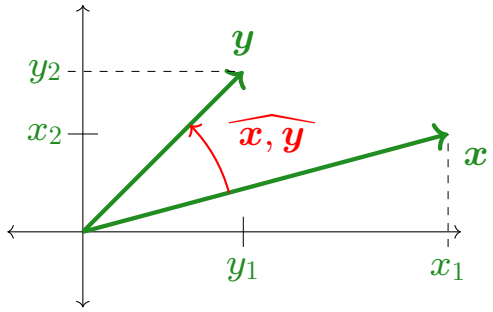
$$\left.\begin{array}{l} \boldsymbol{x} = (x_1, \ldots, x_n) \\ \boldsymbol{y} = (y_1, \ldots, y_n) \end{array}\right\} \quad \longmapsto \quad \boldsymbol{x} + \boldsymbol{y} = (x_1 + y_1, \ldots, x_n + y_n)$$

  Test this function in a file named `array_sum_1.py`.
- Build a function `complement` that computes the complement of a set S in a larger set X, i.e. the set of elements in X that are not in S. The function needs to check that S is indeed a subset of X. Use only the functions mentioned in this lecture.

- Write a program taking two arrays $\boldsymbol{x} = (x_1, \ldots, x_n)$, $\boldsymbol{y} = (y_1, \ldots, y_n)$ by keyboard input and returning their **dot product**: $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \sum_{i=1}^{n} x_i y_i$. We have not seen loops yet, so you need to program your function recursively. It will be much easier once we have seen loops.
- Given two vectors in the plane, $\boldsymbol{x} = (x_1, x_2)$, $\boldsymbol{y} = (y_1, y_2)$, the cosine of their angle is

$$\cos \widehat{\boldsymbol{x}, \boldsymbol{y}} = \frac{\langle \boldsymbol{x}, \boldsymbol{y} \rangle}{\|\boldsymbol{x}\| \, \|\boldsymbol{y}\|} \quad \text{where } \|(a,b)\| = \sqrt{(a,b)\,,(a,b)} \text{ is the } \textbf{norm} \text{ of } (a,b)\,,$$

and $\langle \cdot, \cdot \rangle$ is the dot product defined above.



Two vectors are perpendicular if the cosine of their angle is 0, and collinear if their cosine is ± 1. Write a program taking two vectors as input, returning their cosine as an output, and deciding whether they are perpendicular, collinear (specifying if they are so in the same direction or in opposite directions), or neither. Set a tolerance $< 10^{-14}$; anything having a smaller absolute value is considered to be zero.

4. Loops

- The purpose of **loops** is to repeat one or more lines of code, several times. Let us give an example of why they will become necessary. Suppose we are building the sum $\sum_{n=1}^{2} \frac{1}{n^2}$:

```
x=0
n=0

n = n+1
incr=1./(n**2)
x = x+incr

n = n+1
incr=1./(n**2)
x = x+incr
```

  writing these 3 lines twice is neither necessary nor advisable, but it is not too detrimental either. But what if we need $\sum_{n=1}^{20} \frac{1}{n^2}$, $\sum_{n=1}^{200} \frac{1}{n^2}$, $\sum_{n=1}^{2000} \frac{1}{n^2}$? Would we write those lines 2000 times?

- And what if we did not even know how many terms `k` of the sum $\sum_{n=1}^{k} \frac{1}{n^2}$ we needed, and this number `k` were to be given by the user who executes the program via `input`?
- And what if we had to keep asking the user for a reasonable (≥ 1, maybe not too large) value of `k` until the `input` command obtained a good value?

- Loops can be either:
  - **determinate**: commands contained in them are executed an *a priori* known number of times.
  - **indeterminate**: number of iterations is not known by the programmer while they are writing the code, and will depend on circumstances arising during the execution of the program.
- For example, a *determinate loop* can be used to compute $\sum_{n=1}^{k} \frac{1}{n^2}$, $k = 2, 20, 200, 2000$ in the previous page.
- However, if we need $\sum_{n=1}^{k} \frac{1}{n^2}$ after taking `k` via `input`, we will need an *indeterminate loop* because we might not know how many terms of the sum will be requested by the user.
- And if the user keeps making mistakes (e.g. choosing `k` to be negative or zero) or choosing values of `k` that we deem too large (imagine, for instance, we decide set a bound of $k_{\text{max}} = 5000$ for the user's choice of `k`) but we do not want to stop the program only on account of their bad choices, we need to keep asking the user for a value within the range $\{1, \ldots, k_{\text{max}}\}$. That, too, will need an indeterminate loop.
- We will see two loop statements: `for` and `while`. Both can be determinate or indeterminate.

The `for` loop

- Let `sequence` be any array, i.e.
  - a **list** (seen last week)
  - a **tuple** (seen last week)
  - a **set** (seen last week)
  - a **dictionary** (seen briefly last week)

  The syntax of a `for` loop iterating over `sequence` is as follows:

```
for member in sequence:
    # perform actions related to member
```

`member` can be replaced by any other variable. Unlike other languages, an indexing variable is not necessarily uniform in its type. For instance:

```
list1=[1, 3.4, "a"]
for x in list1:
    print (x)
```

output

```
1
3.4
'a'
```

```
list2=[1, 3.4, "a","b",-0.2,[1,2]]
for u in list2:
    if type(u)==str:
        print (u)
```

output

```
'a'
'b'
```

- The sequence can also be a tuple:

```
tuple_1=1, 2, 3, 4, 10, 20, 30, 40
for i in tuple_1:
    if i%3==1:
        print (i)
```

output

```
1
4
10
40
```

- or a set:

```
set_1={1,3,7,17}
for j in set_1:
    print (j//2)
```

output

```
0
1
8
3
```

```
for j in {1,2,"u"}:
    print ("hello")
```

output

```
'hello'
'hello'
'hello'
```

- or even a string:

```
email = False
for i in "mymail@gmail.com":
    if i=="@":
        email = True
if email:  # Another way of writing if email == True
    print("This seems to be an email address")
```

yields the output `This seems to be an email address`.

- or a dictionary.

- The `range` function saves us the effort of having to write a sequence from `0` to a given number:

```
for i in range(4):
    print (i)
```

output

```
0
1
2
3
```

hence `range(4)` stands for "the first four non-negative integers, starting at `0`".

- And `range(k,n)` stands for "the first n non-negative integers after and including `k`", i.e. `range(n)==range(0,n)`:

```
for var in range(4,7):
    print(var)
```

output

```
4
5
6
```

- And `range(k,n,step)` stands for "the `n` non-negative integers after and including `k`, with increments of `step` at a time", i.e. `range(n)==range(0,n,1)` :

```
for var in range(4,11,3):
    print(var)
```

output

```
4
7
10
```

- Nested loops are also easy to program:

```
for i in ["Mary","John"]:
    for j in ("Ann","David","Robert"):
        print (i," knows ",j)
```

output

```
Mary knows Ann
Mary knows David
Mary knows Robert
John knows Ann
John knows David
John knows Robert
```

- All of the above were **determinate** loops because we knew the loop-defining sequence (and thus its length). However, sometimes we will be faced with the need to program **indeterminate** loops:

```
k=int(input("Write a length for the range "))
for x in range(k):
    print(x)
```

  will have an output that will depend on what we write. *Test it yourselves*.
- Alternatively `k` could come from another function or computation. Think of possible examples.
- You will find an example in this week's Jupyter notebook approximating the *cosine* function with a finite sum whose number of summands can be decided by the user.
- The routine below allows us to decide how many terms of a certain sum we wish to compute:

```
s=0 # We need to initialise the sum to zero
k=int(input("Write the maximum value of your summand: "))
step=int(input("Write the increment: "))
for x in range(0,k,step):
    s += x # which is the same as s = s+x
print(s)
```

  for instance choosing `k=10` and `step=3` yields `0+3+6+9=18`. Again, these two parameters (`k` and `step`) could be the output of earlier functions instead of keyboard input.

- The `break` statement allows us to stop the loop whenever a condition is met, regardless of whether all the elements in the sequence have been browsed:

```
for x in (0,1,2,3,"a","b","c",8,9):
    print (x)
    if type(x)==str:
        break
print("x is now ",x)
```

output

```
0
1
2
3
a
x is now a
```

- Here we also store loop variables in a list:

```
x=[]
for j in range(10):
    if j % 8 == 7:
        break
    x.append(j)
print(x)
```

output

```
[0, 1, 2, 3, 4, 5, 6]
```

- The `continue` statement does not stop the entire loop; it only stops the iteration affected:

```
x=[]
for j in range(10):
    if j % 8 == 7:
        continue
    x.append(j)
print(x)
```

output

```
[0, 1, 2, 3, 4, 5, 6, 8, 9]
```

it only failed to append `7` to `x` but the loop went on for its entire range.

The `while` loop

- Assume we have a `condition`. The `while` loop linked to it will be iterated as long as the condition holds. The syntax of a `while` loop iterating over `sequence` is as follows:

```
while condition:
    # perform actions
```

- For instance:

```
i=1
while i <= 4:
    print("hello")
```

  is an example of an **infinite loop** because the condition will *always* be met.

- This loop, on the other hand, will never be executed because the condition will *never* be met:

```
i=1
while i > 4:
    print("hello")
```

- This one, on the other hand, will produce a finite output:

```
i=1
while i <= 4:
    print("hello")
    i=i+1
```

output

```
'hello'
'hello'
'hello'
'hello'
```

The `i++` present in other languages instead of `i=i+1` is not valid in Python. `i+=1` is, though.

- Example of **indeterminate** `while` loop:

```
age = int( input( "Write your age please "))
while age <0:
    print ( "When did ages become negative? ")
    age = int( input( "Write a correct age please "))
```

  if we write a proper age in the first instance, the `while` loop will not be entered.
- We can use operator `or` in the condition:

```
age = int( input( "Write your age please "))
while age <0 or age >= 200:
    print ( "Ages are neither negative nor unrealistically large ")
    age = int( input( "Write a correct age please "))
```

  the loop will be exited when the condition ceases to be met.
- We can also use operator `and` in the condition; pay attention to parentheses:

```
age = int( input( "Write your age please "))
k=0
while (age <0 or age >= 200) and k<5:
    print ( "Ages are neither negative nor unrealistically large ")
    age = int( input( "Write a correct age please "))
    k+=1
```

  this routine will be less patient and will only ask for a valid age an additional five times.

- What transpires from the above is what is already known (*De Morgan's laws*) about Boolean operators and negation of multiple conditions:

$$\text{not}\,(C_1 \textbf{ and } C_2 \textbf{ and } \ldots \textbf{ and } C_n) = (\text{not}\,C_1) \textbf{ or } (\text{not}\,C_2) \textbf{ or } \ldots \textbf{ or } (\text{not}\,C_n)$$
$$\text{not}\,(C_1 \textbf{ or } C_2 \textbf{ or } \ldots \textbf{ or } C_n) = (\text{not}\,C_1) \textbf{ and } (\text{not}\,C_2) \textbf{ and } \ldots \textbf{ and } (\text{not}\,C_n)$$

  thus condition `(age <0 or age >= 200) and k<5` would cease to be met when either

  - `age < 0` `or` `age >= 200` ceased to be met, i.e. `age >= 0` `and` `age < 200`,
  - or `k<5` ceased to be met, i.e. `k>=5`.

- `break` works just like in `for`:

```
age = int( input( "Write your age please "))
k=0
while age <0 or age >= 200:
    print ( "Ages are neither negative nor unrealistically large ")
    age = int( input( "Write a correct age please "))
    k+=1
    if k>=5:
        break
```

  is an alternative to the earlier double condition.

- `else` (which we will also see attached to conditional `if`) can be used in `while` to run a block of code whenever the condition ceases to be met:

```
age = int( input( "Write your age please "))
while age <0 or age >= 200:
    print ( "Ages are neither negative nor unrealistically large ")
    age = int( input( "Write a correct age please "))
else:
    print ( "Finally!")
```

The last message will only be shown if the user writes a proper age. Attempts are not counted here so this will last as long as the user wants it to.

Exercises

- Some exercises in the past two weeks could have been solved with loops instead of recursive functions. Identify them and rewrite them using loops instead.
- Write a program `sum_1.py` doing precisely what was described at the beginning of the lecture: take `k` as keyboard input, expect it to be within a valid range (larger than `0` and smaller than a bound pre-fixed by you) and compute $\sum_{n=1}^{k} \frac{1}{n^2}$. The larger the `k`, the closer your returned sum should be to $\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6} \simeq 1.6449340668482262$.
- In a file named `look_for_type.py`, build a function counting the elements of a list (or tuple) until it finds a member of a given type, in which case your function should return its index.
- Write a routine `email_input.py` that takes an email address by keyboard input and decides whether or not it is a valid address, based on three facts:
  - the presence of at least one alphabetic character at either side of the `@` sign;
  - the presence of exactly one `@` sign;
  - the presence of at least one dot (`.`) in the address.

  HINT: strings can be indexed and have their length measured, just like lists or tuples.

- The `for` loops shown in this lecture can be easily converted to `while` loops. Use a file called `for_to_while.py` to convert a few of them.
- Let $a > 0$ be a real number. Start with an initial condition $x_0 > 0$. The following sequence

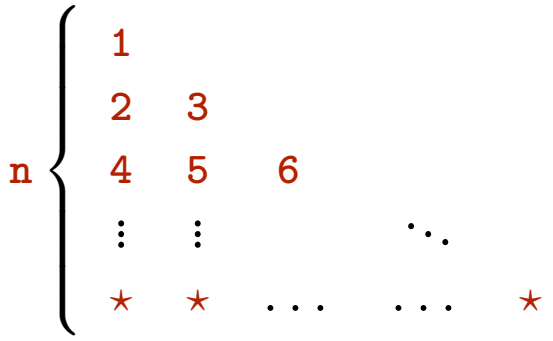
$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right), \qquad n \geq 0 \tag{1}$$

converges to $\sqrt{a}$, i.e. the larger the value of n, the closer x_n is to the square root of a. Write a function `Sqrt` that is fed the following data by keyboard input:

  - `a` of course;
  - a maximal amount `nmax` of iterations for (1);
  - a tolerance `tol` such, that any number below `tol` in absolute value will be considered zero.

  Your program will start on an initial condition x_0 and will perform (1) until either the number of iterations exceeds `nmax`, or the difference in absolute value $|x_k - x_{k+1}|$ between two consecutive iterations is smaller than `tol`. It will then return the last iteration performed.
- Write a function removing all occurrences of a given element from a tuple, set or list.

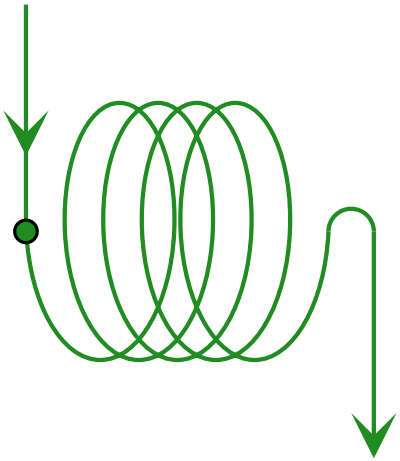
- Write a program that asks for a positive integer $n > 2$ and decides whether it is prime or not.
- Write a program that asks for a positive integer $n > 2$ and returns its decomposition as a product of prime numbers. For instance, 24 should yield a list `[2,2,2,3]`.
- Write a program asking for a positive integer `n` and returning a right triangle such as the one below,



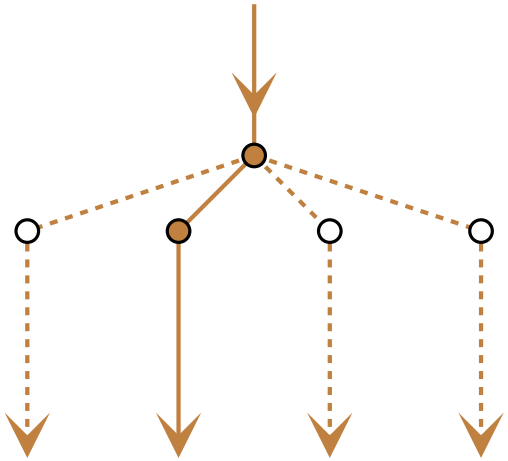
- Write a program storing course subjects (e.g. Maths, Quantum Physics, Chemistry, etc) in a list, asks the user for the marks they obtained in each subject, and shows the subjects that the user needs to resit.

5. Conditionals

- The **control flow** of a program is the order in which its individual instructions are executed.
- Control flow is usually directed *downwards vertically*, but sometimes this needs to be altered–notably, by **control structures** or **control flow structures** allowing us to modify the control flow *without breaking it or bifurcating it*. These control structures can be:
  - **loops**: `for` and `while`, which we have already seen;
  - **conditionals**: `if` - `else` / `elif` and **exception handling**: `try` - `except`.



Loop:
control flow is unidirectional and unbroken but it may become infinite



Conditional:
control flow will only break or bifurcate if options fail to be mutually exclusive and all-emcompassing

- The Python `if` conditional is similar to that of other languages (C++, etc). Its syntax is:

```
if condition:
    # set of instructions in case condition is met
```

- `elif` tries to catch the flow if it did not satisfy the conditions in the previous `if` or `elif` keywords:

```
if condition_1:
    # set of instructions in case condition_1 is met
elif condition_2:
    # set of instructions if condition_2 is met but condition_1 is not
elif condition_3:
    # set of instructions if condition_3 is met but condition_1,2 are not
...
```

- The `else` keyword catches the flow if none of the preceding conditions were met:

```
if condition_1:
    # set of instructions in case condition_1 is met
elif condition_2:
    # set of instructions if condition_2 is met but condition_1 is not
...
else:
    # set of instructions if none of the preceding condition_1,2,... are met
```

- The cumulative nature of `elif` and `else` with respect to previous conditions, means for instance that the following block of code:

```
x=int(input("Write an integer x"))
y=int(input("Write another integer y"))
if x<y:
    print("x is smaller than y")
elif x==y:
    print("they are both equal")
elif x>y:
    print("x is larger than y")
```

(`block 1`)

yields the same output as

```
x=int(input("Write an integer x"))
y=int(input("Write another integer y"))
if x<y:
    print("x is smaller than y")
elif x==y:
    print("they are both equal")
else:
    print("x is larger than y")
```

(`block 2`)

because in (`block 1`) we carefully chose conditions in the `if` and `elif` sub-blocks in such a way, that the final `elif` caught the only remaining potential option (just like `else` did in (`block 2`)).

- Try erasing the first `elif` in (`block 1`) and you will realise the program does nothing in the event of having `x=1` and `y=1`.

- And if we had programmed our second block differently, say only with one `if` and one `else`, our options would be less exhaustive:

```
x=int(input("Write an integer x"))
y=int(input("Write another integer y"))
if x<y:
    print("x is smaller than y")
else:
    print("x is larger than or equal to y")
```

- If we only want to use else in this program while keeping all options (`x<y`, `x>y`, `x==y`) detectable, then we can nest `if - else` blocks:

```
x=int(input("Write an integer x"))
y=int(input("Write another integer y"))
if x<y:
    print("x is smaller than y")
else:
    if x == y:
        print("they are both equal")
    else:
        print("x is larger than y")
```

- Logical clauses attached to the `if` and `elif` keywords are amenable to the same `and`, `or` operators we saw in `while` loops last week, hence to the same rules (De Morgan's laws) seen in page 49:

$$\text{not } (P_1 \text{ and } P_2) = (\text{not } P_1) \text{ or } (\text{not } P_2)\,, \qquad \text{not } (P_1 \text{ or } P_2) = (\text{not } P_1) \text{ and } (\text{not } P_2)\,.$$

- For instance

```
x=int(input("Write an integer x>1"))
y=int(input("Write another integer y<-2"))
if x <= 1 or y >= -2:
    print("Wrong numbers")
```

x=2, y=0 $\longrightarrow$ output: `'Wrong numbers'`

and any "good" choice, e.g. `x=2`, `y=-3`, yields no output and keeps the control flow going down.

- And

```
x=int(input("Write an integer x>1"))
y=int(input("Write another integer y<-2"))
if (x <= 1 and y < -2) or (x > 1 and y >= -2):
    print("One of the numbers is wrong")
```

would still be non-exhaustive with regards to the options (you can find examples of this).

- This program recognises *palindromes*, i.e. words with symmetric spelling. Note that strings can be indexed (`[ ]`), measured (`len`) and concatenated (`+`) just like tuples or lists:

```
def inverse_str ( string ):
    inv =""
    counter = len(string)
    index = -1
    while (counter >=1 ):
        inv += str(string[index])
        index -= 1
        counter -= 1
    return inv
```

```
def is_palindrome ( string ):
    inverse_word = inverse_str( string )
    index = 1
    for i in range( len( string ) ):
        if inverse_word[i] != string[i]:
            break
        index += 1
    if index < len( string ):
        return False
    return True
```

For instance, `is_palindrome ( "MADAM" )` should yield `True` and `is_palindrome ( "ADAM" )` should yield `False`. Also in the presence of the `return` statement no `else` is needed in `is_palindrome` because a `False` output would leave the function altogether and the other option would not be explored.

6. Exception handling

- Assume control flow is interrupted by an unexpected **error**. For instance,

```
# other lines of code
40 n = int(input("Write the dividend"))
41 m = int(input("Write the divisor"))
42 print ("The quotient n/m equals: ", n/m)
# other lines of code
```

n=2, m=0 → output

```
ZeroDivisionError:
division by zero
```

  if the rest of program can function at least partially without the need for `n/m`, we would like it to. However, if we executed the program as it is, it would end in line 42 and return the **exception**, i.e. the error type `ZeroDivisionError` described above.

- A way to do so is by means of **exception catching**. If we know division by zero is a serious risk, we can amend the program to "catch" this potential malfunction and move on in case it happens:

```
# other lines of code
n = int(input("Write the dividend"))
m = int(input("Write the divisor"))
try:
    print ("The quotient n/m equals: ", n/m)
except ZeroDivisionError:
    print ("We cannot divide by 0")
# other lines of code
```

n=2, m=0 → output

```
We cannot divide by 0
```

  and the program moves on, taking the potential exception into account if well programmed.

- We can also leave it as a generic exception (i.e. remove the `ZeroDivisionError` from the above block) if we are not sure what potential error can arise, e.g.

```
# other lines of code
n = int(input("Write the dividend"))
m = int(input("Write the divisor"))
try:
    print ("The quotient n/m equals: ", n/m)
except:
    print ("We cannot divide ", n, " by ", m)
# other lines of code
```

$\xrightarrow[\text{output}]{\texttt{n=2, m=0}}$ `We cannot divide 2 by 0`

but the ideal scenario is being always in control of the potential exceptions.

- Same way this would be correct:

```
try:
    print(z)
except:
    print("We could not print z")
```

but this is better (see `https://docs.python.org/3/library/exceptions.html` for a list):

```
try:
    print(z)
except NameError:
    print("We could not print z because you had not defined it first")
```

- The following block of code keeps asking for two valid numbers until both of them are the right value (i.e. can be converted to `int` automatically):

```
while True:
    try:
        x1 = int(input( "write the first number" ))
        x2 = int(input( "write the second number" ))
        break
    except ValueError:
        print("The values are not numerical, please try again")
```

When the flow arrives to `while True:` it prepares for a (potentially) infinite loop. If it gets to `break`, that means `x1` and `x2` were read correctly, hence it will exit without reading the `except`. Thus, if we input something that is not numerical in the first number, `x1=a`, the flow will step directly to the `except` message asking us to try again. If we also want to divide `x1` by `x2`, we can add another `except`:

```
while True:
    try:
        x1 = int(input( "write the first number" ))
        x2 = int(input( "write the second number" ))
        print("the division of x1 by x2 is"+str(x1/x2))
        break
    except ValueError:
        print("The values are not numerical, please try again")
    except ZeroDivisionError:
        print("You're asking me to divide by zero")
```

- We can make sure we exit the `try - except` clause regardless of whether results are satisfactory:

```
def divide ( ):
    try:
        x1 = int(input( "write the first number" ))
        x2 = int(input( "write the second number" ))
        print("the division of x1 by x2 is"+str(x1/x2))
    except ValueError:
        print("The values are not correct")
    except ZeroDivisionError:
        print("You're asking me to divide by zero")
    finally:
        print("we've finished our calculation")
divide( )
```

  regardless of whether or not the code catches any of the two exceptions, it will always execute the `finally` line. If anything goes wrong, it will not perform 100% of the try block, but it will always tell us `"we've finished our calculation"` and move on.

- `finally` is useful for instance when we need to work with external files that will have to be closed after reading or writing on them – this will be seen in a few weeks.

- `else` can be written instead of `finally` if there has been no exception; think of possible scenarios for this.

- We can also customise exceptions. Assume we want to compute square roots using the function `Sqrt ( a, nmax, tol )` programmed last week:

```
def sqrt ( number ):

    if number < 0:
        raise ValueError ("The number cannot be negative if you want to compute its square root")
    else:
        return Sqrt( number, 1000, 1.e-15)
```

```
num1 = float( input( "Write a number: " ))
try:
    print( "The square root of "+str(num1)+" is "+str(sqrt(num1)))
except ValueError as NegativeNumber:
    print (NegativeNumber)
```

```
print ("Program goes on")
```

What the above does, is throw a customised exception. Input `num1=-1` would return

```
The number cannot be negative if you want to compute its square root
```

because that is the label of the `ValueError` exception in our program.

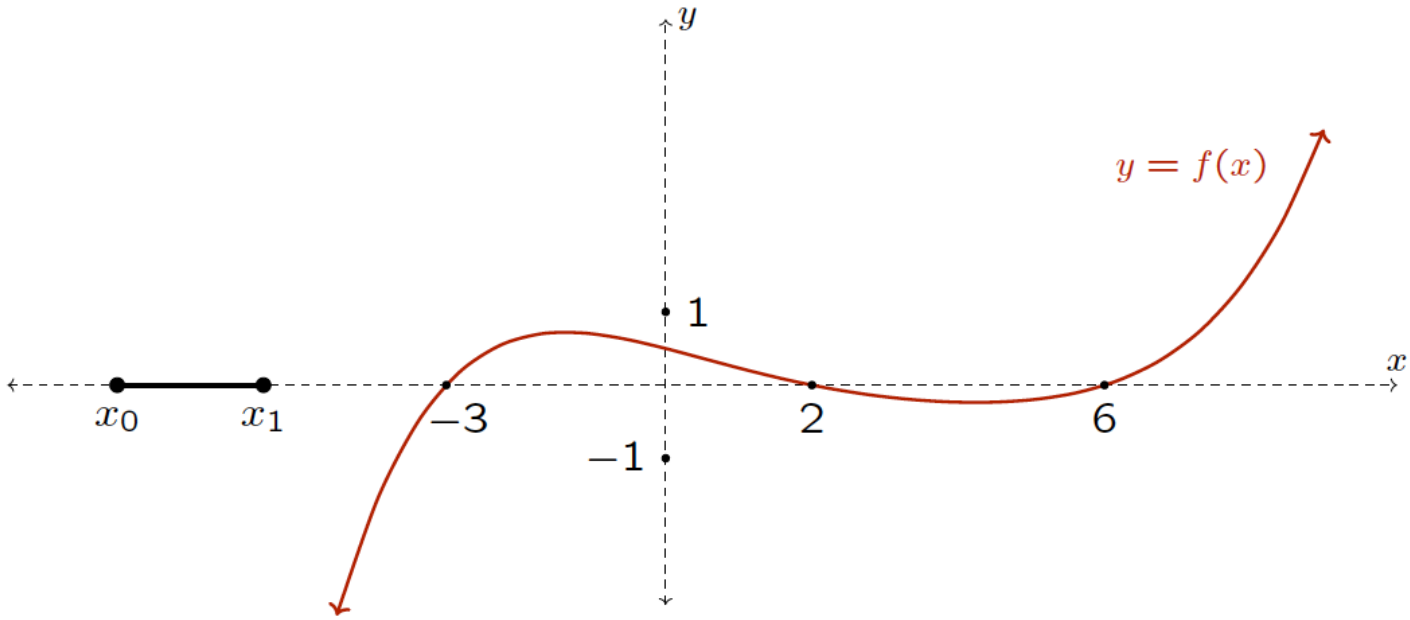
Exercises

- Some of the programs you have practiced on in the past three weeks could have been improved with a block of code asking for the user input up to a fixed number of times, or until the user writes something that makes sense. Implement that modification in some of those programs.
- This code is waiting for you to make it work properly:

```
x=int(input("Write an integer x>1"))
y=int(input("Write another integer y<-2"))
if x <= 1 or y >= -2:
   print("Both are wrong numbers")
elif x > 1:
   print("x is correct but y is not")
elif y <-2:
   print("y is correct but x is not")
else:
   print("both numbers are as requested")
```

- Define a function `max_out_of_three` expecting three numbers and returning their maximum.
- Define a function `vowel` taking a character and returning `True` if it is a vowel, and `False` otherwise.
- Write a single function that does the exact same task as the combination of `reverse_string` and `is_palindrome` in page 56, using only one loop and one conditional block.

- The following is the graph of a certain mathematical function $y = f(x)$:



it also contains an example of a closed interval $[x_0, x_1]$, i.e. a set of points in the x-axis such that $x_0 \leq x \leq x_1$ (extremes inclusive). Write a function `verify_signs` taking two extremes `x0`, `x1` of any closed interval and returning the number of changes in sign experienced by the function within that interval. For instance, the interval in the picture would return `0` because the function does not cross the x axis even once within the interval. `x0=-10`, `x1=10` would return `3`, etc.
Make sure your function asks for `x0` and `x1` until they are both numerical and ordered: `x0<x1`.

7. Inputting, outputting, storing and retrieving data

- We have already seen two built-in functions: `print` and `input`:

```
print ("Hello")
```
output → `'Hello'`

```
x=[1,2,"a"]
print (x)
```
output → `[1,2,"a"]`

```
x=float(input("Write a number"))
```
output →
```
Write a number
___
```

```
n=int(input())
```
output →
```
___
```

- We also know the meaning of `\n` (line feed) and `\t` (horizontal tab).
- An option in `print` is `end`, which by default is `\n` but can be changed at will:

```
print("Hello and", end = " ")
print("goodbye")
```
output → `Hello and goodbye`

```
print("Hello and")
print("goodbye")
```
output →
```
Hello and
goodbye
```

- We can concatenate and index strings just like we did with other types of arrays,

```
x="Hello "
y="World"
z=x+y
print(z)
```

output → `Hello World`

```
x = "Hello World"
print(x[2:5])
```

output → `llo`

- We can split them,

```
x = "Hello World Today"
print(x.split())
```

output → `['Hello', 'World', 'Today']`

- Replace elements in them,

```
x = "Hello World Today"
print(x.replace(" ", ", "))
```

output → `Hello, World, Today`

- Or check membership:

```
string = "Mary had a little lamb"
fragment = "ad a l"
print (fragment in string)
print ("lamb" not in string)
```

output →
```
True
False
```

- We have already seen in examples that we can convert other types (e.g. numbers) to string form,

```
frag = str(14//3)
print("Integer division of 14 by 3 is "+frag)
```

$\xrightarrow[\text{output}]{}$ `Integer division of 14 by 3 is 4`

- or print string blocks separated by commas; any of the following has the same output as above:

```
print ("Integer division of 14 by 3", "is", 14//3)
```

```
print ("Integer division of 14 by 3", "is", str(14//3))
```

- Another way to add "mobile" or initially unknown parts of text is by **formatting**:

```
frag = str(14//3)
txt = "Integer division of {} by {} is {}"
print(txt.format(14,3,frag))
```

and

```
frag = str(14//3)
txt = "Integer division of 14 by 3 is {}"
print(txt.format(frag))
```

have the same output as the above blocks. The block below allows for digit control:

```
frag = 14//3
txt = "Int. div. of {} by {:f} is {:.2f}"
print(txt.format(14,3,frag))
```

$\xrightarrow[\text{output}]{}$ `Int. div. of 14 by 3.000000 is 4.00`

note the difference between the default number of floating-point decimal digits (chosen for 3) and the two decimal points specifically chosen for 4.

- You can also ensure values are placed where they belong by indexing within the curly brackets:

```
frag = 14//3
txt = "Int. div. of {0} by {1:f} is {2:.2f}"
print(txt.format(14,3,frag))
```

output → `Int. div. of 14 by 3.000000 is 4.00`

- and index repetition has the obvious effect:

```
term = 2
txt = "{0} plus {0} is {1:.3f}"
print(txt.format(term,term+term))
```

output → `2 plus 2 is 4.000`

- We can customise indices in `{ }`, to the expense however of having to define them within `format`:

```
string = "{num1}+{num2} = {output}, {message}"
print(string.format(num1 = 2, num2=2, output=4,message="obviously"))
```

yields `2+2 = 4, obviously`. Replacing `{output}` by `{output:.2f}` yields `4.00`, etc.

- Formatting works well with loops and keyboard inputs:

```
string = "{n}+{m}={out:.2f}"
for i in range(3):
    for j in [0,1]:
        print(string.format(n=i,m=j,out=i+j))
```

output →

```
0+0=0.00
0+1=1.00
1+0=1.00
1+1=2.00
2+0=2.00
2+1=3.00
```

- Formatting can also be carried out with symbol **%**:

```
age = 30
print("John is %d years old" % age)
```

output → `John is 30 years old`

- Including multiple terms, same as before, and floating-point numbers operate as usual:

```
age1, age2 = 25, 30
name1, name2 = "Jane", "John"
print("%s is %d years old" % (name, 25))
print("Their age average is %.5f" % (0.5*(age1+age2)))
```

output →
```
Jane is 25 years old
Their age average is 27.50000
```

the latter of which is the same as below (bear in mind we are just storing values in a **`str`** variable):

```
age1 = 25
age2 = 30
string = "The age average is %.5f" % (0.5*(age1+age2))
print(string)
```

output → `Their age average is 27.50000`

- Most common types: **`%d`** (integers), **`%f`** (floating point numbers) and **`%s`** (strings).
- You can also use **formatted string literals** (also known as **f-strings**) to format strings in a more compact way. Check this week's Jupyter notebook for more examples:

```
name = "John"
age = 20
city = "Portsmouth"
print( f"Hello, {name}, you are {age} and you live in {city}" )
```

File handling

- Goal: **data persistence**, i.e. the need to store information produced during the execution of a program, in order to avoid its loss after the execution ends.
- We have two alternatives:
  - **external files**, which we will see today, and
  - **databases**, which will not be the focus of our course.
- ***External files***, i.e. files other than the `.py` program manipulating them, can be:
  (a) ***created*** from the program, i.e. your folder or directory will contain a new file after this step,
  (b) ***opened*** from the program, i.e. the external file will be ready to be read or written into,
  (c) ***read*** from the program, i.e. having their content stored or processed,
  (d) ***updated*** from the program, i.e. written into or having their content modified,
  (e) ***closed*** from the program, i.e. from here onwards the program will be unable to modify them,
  (f) ***deleted*** from the program, i.e. your folder will no longer contain the file after this step.
- They can take place several times in the same program. (c) and (d) can happen simultaneously.
- The tools to work with files are in the `io` module (`https://docs.python.org/3/library/io.html`)

- **Opening and creating a file**: we use the command `open`. Syntax:

```
variable = open ( "name.extension", options )
```

  - `options` can be:
    * `"x"`: ***creates a file***, returns an error message if it already exists.
    * `"w"` for **write**: opens file for writing, ***creates the file if it does not exist***
    * `"a"` for **append**: opens file for adding data, ***creates the file if it does not exist***
    * `"r"` for **read**: opens file for reading, returns an error message if it does not exist
  - as well as:
    * `"t"` for **text**, which will be the type you will use this year;
    * `"b"` for **binary**, e.g. images or video files.
  - Default values are `r` (read) and `t` (text). Thus,

```
file = open("samplefile.txt")
```
is the same as
```
file = open("samplefile.txt", "rt")
```

- **Closing a file**: we use `close`, e.g. `variable.close()`

- **Reading from a file**: can be done whenever we have opened it with `r`, e.g.
  - Create a new file, naming it `manually_created.txt`.
  - Open that file with an editor, and write anything you want on it.
  - Close file `manually_created.txt`.
  - Write the following code in a Python program:

```
myfile = open("manually_created.txt", "r")
u = myfile.read()
#sequence of commands
myfile.close()
```

  - if `#sequence of commands` includes `print(u[0])`, you obtain the first character in that file. `print(u[1])` will return the second character, etc.
  - `print(type(u))` yields `<type 'str'>`; file contents are retrieved in string format.
  - `print (u)` yields the entirety of the contents of myfile.
  - `u = myfile.read(k)`, if `k` is an integer, defines `u` as the first `k` characters written in the file.

- `readlines` reads the contents of a file and returns a list of its successive lines. For instance, assume the contents of your file `manually_created.txt` are

```
1 2
a b
1.2 hello goodbye
```

Then `readlines` reads those three lines and returns a line containing them:

```
file = open ("manually_created.txt")
f=file.readlines()
file.close()
print(f)
```

output

```
['1 2\n', 'a b \n', '1.2 hello goodbye ']
```

thus we can conduct ourselves the way we would in any list:

```
file = open ("manually_created.txt")
f=file.readlines()
file.close()
for x in f:
    print x
```

output

```
1 2

a b

1.2 hello goodbye
```

hence we can split these strings and (whenever possible) convert their parts to `float` or `int`:

```
u=f[0].split()
print(u)
x=float(u[0])
y=float(u[1])
print(x)
print(y)
```

output

```
['1', '2']
1.0
2.0
```

- **Writing on a file**: opening a file with `"w"` (i.e. preparing it to write data on it),
  - creates it if it did not exist, and
  - ***empties*** it if it already existed.

  For instance, if file `samplefile.txt` did not exist in our directory, then after the following

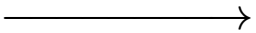
```
file = open("samplefile.txt", "w")
file.write("Hello world!")
file.close()
```

  a new file `samplefile.txt` appears in the same folder as the `.py` file, containing `"Hello world!"`.

- The same sequence, however, replacing `samplefile.txt` by the already-existing `manually_created.txt`, will erase its contents and replace them as follows:

```
1 2
a b
1.2 hello goodbye
```

old contents of `manually_created.txt`



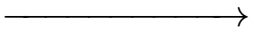
replace

```
Hello world!
```

new contents of `manually_created.txt`

- **Appending** writes on a file without erasing its previous content, and ***creates it if it did not exist***:

```
file = open("manually_created.txt", "a")
file.write("123")
file.close()
```



output

```
Hello world!123
```

new contents of `manually_created.txt`

- Conventions in **writing** and **appending** are the same as for `print`:

```
file = open("samplefile.txt", "w")
file.write("{:.5f} \t {:.10f} \n".format(1,2))
file.write("{:.10f} \t {:.2f} \n".format(.2,4))
file.write("%d, %f , %s\n" % (1,2,"whatever"))
file.close()
```

output

```
1.00000    2.0000000000
0.2000000000    4.00
1, 2.000000 , whatever
```

new contents of `samplefile.txt`

- With the additional function `writelines` which does exactly what it name says:

```
file = open("samplefile.txt", "a")
linearray= ["one\n","two", "\t and %.3f" % 3]
file.writelines(linearray)
file.close()
```

output

```
1.00000    2.0000000000
0.2000000000    4.00
1, 2.000000 , whatever
one
two     and 3.000
```

new contents of `samplefile.txt`

(and works equally well if `linearray` is a tuple instead of a list)

- And is amenable to everything we can do with a list, including shortcuts:

```
file = open("samplefile.txt", "w")
file.writelines(["%s\t" % item for item in range(5)])
file.close()
```

output

```
0    1    2    3    4
```

new contents of `samplefile.txt`

- Again, **f-strings** can be used in external files. See the Jupyter notebook for an example and elaborate further examples on your own.

Exercises

- Write a program receiving a string and returning a double list

```
[[  word 1  ,  frequency 1  ],
 [  word 2  ,  frequency 2  ],
      ⋮     ⋮       ⋮
 [  word n  ,  frequency n  ]]
```

  collating each character against the frequency with which it occurs in the string.

- Write a program `table.py` that takes an integer input `k`, creates an external file `k_rows.txt` (i.e. its name adapted to `k`) and writes two columns of numbers in it:

```
k { 1     1.000
    2    -2.000
    3     4.000
    4    -8.000
    ⋮     ⋮
```

- Write a program `modifytable.py` that takes an integer input `k` and creates or modifies an external file `k_rows.txt` by writing two columns of numbers at the end of it,

```
k { 1    -1.0000000000000000
    2     0.5000000000000000
    3    -0.2500000000000000
    4     0.1250000000000000
    ⋮     ⋮
```

  with the understanding that any (if at all) data preceding them will not be erased.

- Write a program `modifytable2.py` taking an integer input `k`, using the `"r+"` option instead of `"w"` or `"a"` (you can probe into this yourselves) and modifying an already-existing external file `k_rows.txt` as follows: the last `k` rows are made up of a concatenation of the columns of the previous rows, e.g. for `k=2` your output should leave `2_rows.txt` modified as follows:

```
1     1.000
2    -2.000
1    -1.0000000000000000
2    0.5000000000000000
1    1.000    1    -1.0000000000000000
2    -2.000    2    0.5000000000000000
```

- Write a program `modifytable3.py` that does the same as the above to an already-existing file but erases duplicate integer indices in the last `k` rows, i.e. the output for `2_rows.txt` after successively running `table.py`, `modifytable.py` and `modifytable3.py` with `k=2` should be

```
1     1.000
2    -2.000
1    -1.0000000000000000
2    0.5000000000000000
1    1.000    -1.0000000000000000
2    -2.000    0.5000000000000000
```

- Write a program `frequency.py` retrieving text from a file and performing a frequency analysis of its characters. Use a large text and see how close you get to the famous list (commonly referred to as ETAONRISH from Herbert Zim's treatise, although the exact order is controversial) used for deciphering messages in the English language.

8. Functions, program structure and module importation

- In other languages (e.g. `C` or `C++`) all commands are centralised in a **main function** (usually labelled `main()`) which is automatically called and executed by the program – and which calls all the remaining functions in that program. For instance, the following program in `C`

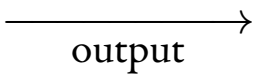
```
#include <stdio.h>
float sum (float, float);
int main ()
{
    printf("The sum of %f and %f is %f", 1., 2., sum(1.,2.));
    return 0;
}
float sum ( float a, float b ) {
    return (a+b);
}
```

  has two functions: one returning the sum of two numbers, and the main function calling it. *The program could not function without the latter*.
- In Python, however, there is *a priori* no hierarchy between functions nor a need for a main one, and the control flow carries out all commands – except of course for those in functions, which are only carried out if the functions are called.
- There is, however, a `main` function that can be *optionally* called in Python as well, and will be useful in a number of cases–and will justify our explanation of **module importation**.

- The following block of code in file `hello.py` will not write the contents of the main function:

```
def main():
    print("hello!")
print ("and goodbye")
```

contents of `hello.py`

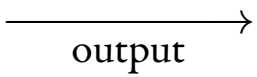



```
and goodbye
```

- Calling function `main`, of course, solves this:

```
def main():
    print("hello!")
main()
print ("and goodbye")
```

modified contents of `hello.py`

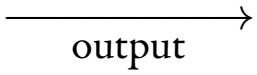



```
hello!
and goodbye
```

- A special variable `__name__` will categorise what place this `.py` file occupies within the project running it – if it is the main file being run, `__name__` will have the value `"__main__"`:

```
def main():
    print("hello!")
if __name__== "__main__":
    main()
print ("and goodbye")
```

modified contents of `hello.py`

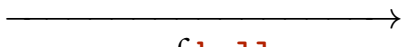



```
hello!
and goodbye
```

- Check the difference when we are compiling another program that **imports** file `hello.py`:

```
def main():
    print("hello!")
print ("and goodbye")

if __name__ == "__main__":
    main()

if __name__ == "hello":
    print("imported externally")
```

modified contents of `hello.py`



output of `hello.py`

```
hello!
and goodbye
```

```
import hello
```

contents of new file `greetings.py`

output of `greetings.py`

```
and goodbye
imported externally
```

- Referring to functions in the imported module can be done with a dot (`.`):

```
import hello

hello.main()
```

modified contents of file `greetings.py`

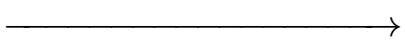
output of `greetings.py`

```
and goodbye
imported externally
hello!
```

- We can customise importations:

```
import hello as he

he.main()
```

modified contents of file `greetings.py`



output of `greetings.py`

```
and goodbye
imported externally
hello!
```

- Let us illustrate this with a function other than `main`:

```
def main():
    print("hello!")
def write_age( age ):
    print("You are",age,"years old")

if __name__ == "__main__":
    main()
if __name__ == "hello":
    print("imported externally")
```

modified contents of `hello.py`

output of `hello.py` →

```
hello!
```

```
import hello as he

he.write_age(20)
```

modified contents of file `greetings.py`

output of `greetings.py` →

```
imported externally
You are 20 years old
```

- We can obviously trim our outputs in this case:

```
def main():
    print("hello!")

def write_age( age ):
    print("You are",age,"years old")
```

modified contents of `hello.py`

No output

```
import hello as he

he.write_age(20)
```

modified contents of file `greetings.py`

output of `greetings.py` →

```
You are 20 years old
```

- Importation can be nested and reference to the main function can always be reproduced by `print`:

```
print("This is the first program")
print("__name__ is ", __name__)
def main():
    print("And main function in first program")
if __name__ == '__main__':
    main()
```

contents of `first_program.py`

→ output

```
This is the first program
__name__ is __main__
And main function in first program
```

```
import first_program as fp

print("Part of second program")
if __name__ == "second_program":
    print("Second program called by external")
```

contents of `second_program.py`

→ output

```
This is the first program
__name__ is first_program
Part of second program
```

```
import second_program as se

print("Third program")
se.fp.main()
print("__name__ is ", __name__)
```

contents of `third_program.py`

→ output

```
This is the first program
__name__ is first_program
Part of second program
Second program called by external
Third program
And main function in first program
__name__ is __main__
```

- As a rule of good practice in programming, and in order to have full control of the program(s) we are working with, **we recommend that you always define a `main` function**.

- **Packages** are directories where we store modules that are related to one another.
- They are useful as a means to organise and reuse modules.
- **Method to create a package**: create a folder containing a file named `__init__.py`.
- For example, name a new folder `my_folder` and fill it with the following files:

```
# Empty file!
```

contents of `__init__.py`

```
def add_list( list1, list2 ):
    list3=[]
    for i in range(len(list1)):
        list3.append(list1[i]+list2[i])
    return list3
```

contents of `vectors.py`

if we open another file *outside* of this folder,

```
from my_folder.vectors import add_list

print( add_list( [1,2],[3,4] ) )
```

contents of `use_module_from_outside.py`

$\longrightarrow$ output `[4, 6]`

and if we open another file *inside* the same directory `my_folder`,

```
from vectors import add_list

print( add_list( [1,2],[3,4] ) )
```

contents of `use_module_from_inside.py`

$\longrightarrow$ output `[4, 6]`

- Assume our folder `my_folder` and/or its modules have more "substance" to them:

`__init__.py`

```
def add_list( list1, list2 ):
    list3=[]
    for i in range(len(list1)):
        list3.append(list1[i]+list2[i])
    return list3
def subtract_list( list1, list2 ):
    list3=[]
    for i in range(len(list1)):
        list3.append(list1[i]-list2[i])
    return list3
```

contents of `vectors.py`

```
def scalar( number, list1 ):
    list2=[]
    for i in range(len(list1)):
        list2.append(number*list1[i])
    return list2
```

contents of `vector_and_number.py`

we now have *two* functions in `vectors.py`; we can import only what we need, or we can import *everything* with a `*`:

```
from my_folder.vectors import *
from my_folder.vector_and_number import scalar

print( add_list( [1,2],[3,4] ) )
print( subtract_list( [1,2],[3,4] ) )
print( scalar( 3.1,[3,4] ) )
print( scalar( -1,[2,4,5] ) )
```

contents of `use_module_from_outside.py`

output

```
[4, 6]
[-2, -2]
[9.3, 12.4]
[-2, -4, -5]
```

- **Subpackages** follow the same logic (just keep tabs of your extensions with `.`, e.g. `from my_folder.some_subfolder.some_module import *`).

Functions and variable inputs as arguments of functions

- Functions can call other functions as arguments. For instance,

```
def callingfunction( f, x ):
    return f(x+1)
def func1 ( x ):
    return x**2
def func2 ( x ):
    return x**3
print( callingfunction( func1, 3 ) )
print( callingfunction( func2, 3 ) )
```

output →

```
16
64
```

- You can also pass a variable amount of arguments, using the **unpacking operator** (*) which does what it name indicates to any list or tuple so that its elements can be passed as different parameters:

```
def sum (*args):
    s = 0
    for x in args:
        s+=x
    return s
print(sum(1, 2, 3.2))
print(sum(0.1, 0.2, -.9, 10))
```

output →

```
6.2
9.4
```

thus we can pass any amount of arguments to `sum` as long as they are numbers.

- On a side note, unpacking operators also work when we are not calling functions:

```
list1 = ["A", "B", "C", "D", 1, 2.1]
a, *b, c, d = list1
print(a)
print(b)
print(c)
print(d)
```

output →

```
A
['B', 'C', 'D']
1
2.1
```

- This turns our `Sqrt` function from Week 4 into one that can be called with one single argument:

```
def Sqrt ( a, *args ):
    nmax =1000
    tol=1.e-20
    if len(args)>0:
        nmax=args[0]
        if len(args)>1:
            tol=args[1]
    x0 = 1
    Abs = 1
    x=x0
    i=0
    while Abs >= tol and i<nmax:
        X = 0.5* (x+(a/x))
        Abs = abs(X-x)
        x=X
        i+=1
    if i>=nmax:
        print("max. number of iterations was reached before precision goal was achieved")
    return X
```

  - Calling `Sqrt(2)` is the same as calling `Sqrt(2,1000,1.e-20)` and both will yield `1.4142135623730949234`3.
  - Calling `Sqrt(2,3)` is the same as `Sqrt(2,3,1.e-20)` and only three iterations will be undertaken, yielding the very imprecise approximation `1.41421568627450966460`.
  - `Sqrt(2,5,1.e-5)` packs *two* elements to `args` and the output is `1.41421356237468986983`.

- `lambda` functions can also be useful: their syntax is:

```
lambda arguments : output
```

- They are easily amenable to redefinition, but the onus is on you to check the latest definition:

```
x = lambda x : x**2
print(x(2))
print(x(3))
x = lambda t : t**3
print(x(2))
print(x(3))
x = lambda t,u : t+u
print(x(3,4))
print(x(1,2))
```

output →

```
4
9
8
27
7
3
```

- And they can be called by other functions just like we did earlier:

```
def evaluate( func, a ):
    return func(a)

x = lambda t : t**2
print(evaluate(x,3))
print(evaluate(lambda z : (z+3)**4,3))
```

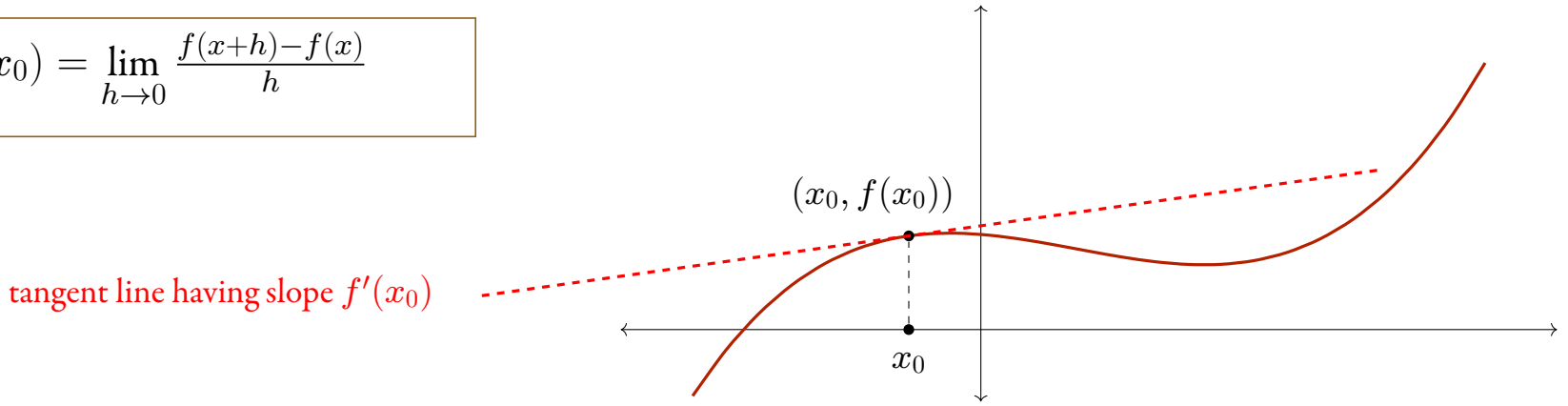
output →

```
9
1296
```

Exercises

- Write a function which, given function $f(x)$, returns an approximation of the **derivative** $f'(x_0)$:

$$f'(x_0) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$



You cannot compute the above limit in general, but you can *approximate* it:

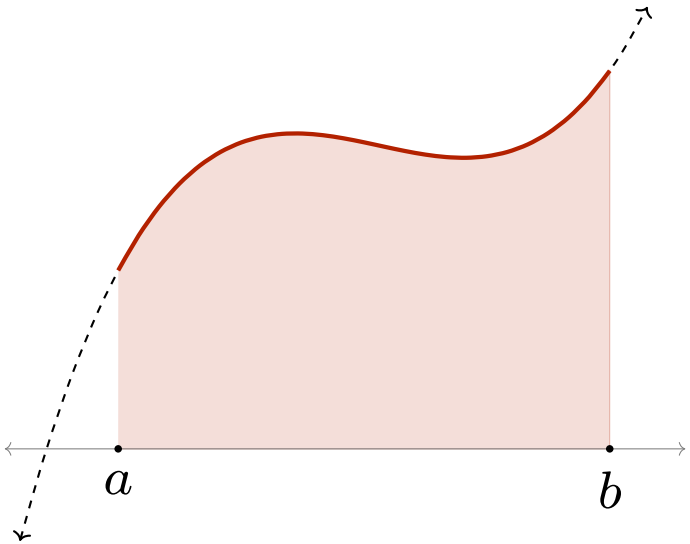
$$f'(x) \simeq \frac{f(x+h) - f(x)}{h}, \qquad |h| \text{ very small.}$$

Write all of your modules in a single folder called `derivation` containing at least the following:

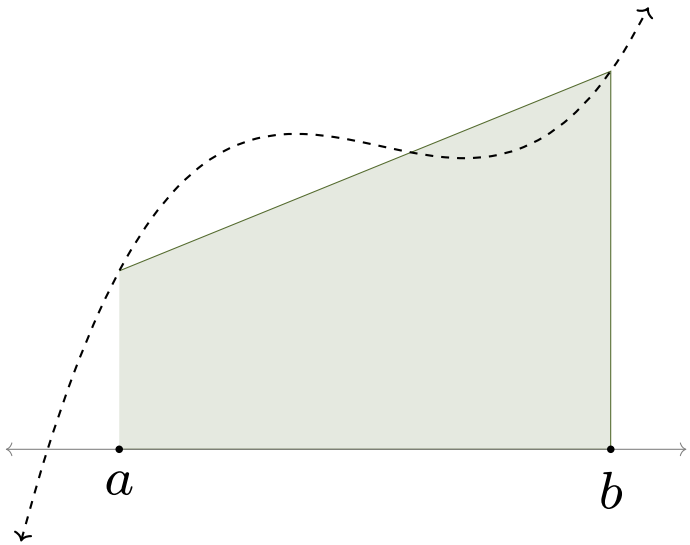
- anything turning this folder into a **package**;
- `derivation.py`, containing the main function and plenty of derivation examples;
- `functions.py` program, containing definitions of functions;

You can test your program on a number of functions whose actual derivatives you know. Think of how you would implement this in the most simple and comprehensive way.

- Write a function `trapezium` which given any function $f(x)$, defined on a given interval $[a, b]$, returns an approximation of the integral $\int_a^b f(x)dx$ (i.e., the area formed by the region bounded by the graph of f, the x-axis, and the vertical lines at $x = a, b$) by means of the **trapezium rule**:



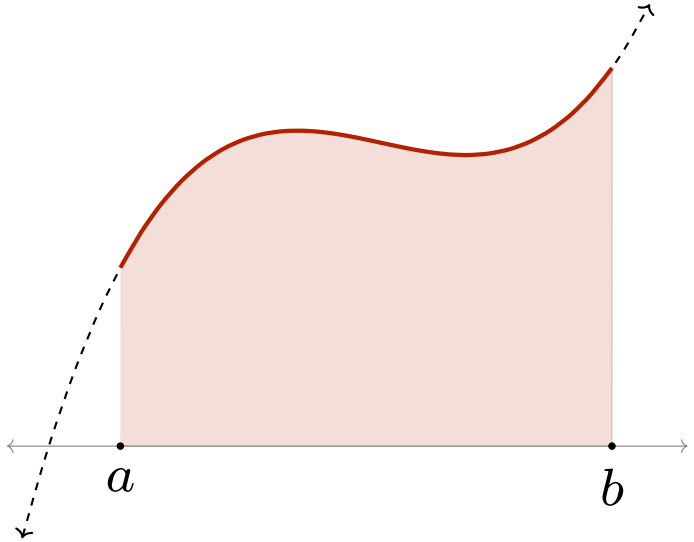
The shaded region is the actual area $\int_a^b f$



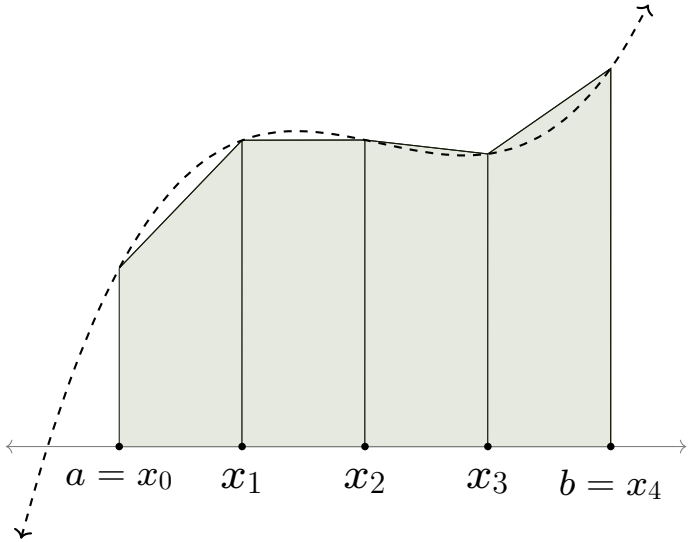
The shaded region is the approximation $T(f, a, b)$

The area of a **trapezium** ⏢ (i.e. a quadrilateral with at least two parallel sides) is the sum of the lengths of those parallel sides, times the distance between them, divided by two.

- Write another function also named `trapezium`, with one particularity: if more arguments are passed into it than those in the earlier `trapezium`, then the approximation of the integral will not be by means of the original *trapezium rule*, but the **composite trapezium rule**:



The shaded region is the actual area $\int_a^b f$



Shaded region is approximation $T_n(f, a, b)$ with $n = 4$

In other words, we divide $[a, b]$ in subintervals of equal lengths, construct smaller trapezia from them, then add the areas. Needless to say,

- $T_1(f, a, b)$ should return the original, *simple* trapezium rule (shown in the previous page)
- The larger the number n of sub-intervals, the better our approximation is... theoretically.
- However, as n grows, so does the number of operations and thus numerical error propagates.

- Remarks on the previous two exercises:
  - Your two versions of `trapezium` (the exclusively simple, and the potentially composite) should be able to comfortably change the function and the domain $[a, b]$.
  - Once you have secured a correct function for the first exercise (i.e. simple trapezium), feel free to use it as a tool for the second exercise (which consists on dividing the original interval into subintervals of equal length).
  - Avoid using the `math` package to compute functions. Write all of your modules in a single folder called `integration` containing at least the following:
    * anything turning this folder into a **package**;
    * `simple.py`, devoted to simple trapezium (including plenty of examples);
    * `composite.py` defining and implementing composite trapezium to many examples;
    * a `.py` program, containing definitions of functions *not* needing Taylor expansions;
    * a `.py` program, containing definitions of functions needing Taylor expansions;

    Your `trapezium` examples of application should include instances of `lambda`, as well as calls to functions defined by you elsewhere.

- You can test your program on a number of functions and intervals; below are some examples with their definite integrals, and approximations rounded to 20 exact decimal digits:
  - $\cos x$ in $[1, 3]$: area equals $\sin 3 - \sin 1 \simeq -0.70035097674802928455$
  - $\int_2^{10} \sqrt{x}\, dx = \frac{4}{3}\sqrt{2}\left(5\sqrt{5} - 1\right) \simeq 19.196232984625068815$
  - $\int_{-1}^{1} \sqrt{x^2 + 1}\, dx = \sqrt{2} + \operatorname{arcsinh}(1) \simeq 2.2955871493926380740$
  - $\int_0^1 \frac{1}{x^2-2} dx = -\frac{1}{\sqrt{2}} \operatorname{arctanh}\left(\frac{1}{\sqrt{2}}\right) \simeq -0.623225240140230513 39$

  In the above examples, you would only need a Taylor series expansion for $\cos x$. The rest can be defined with functions already seen by you.
- Examples of Taylor expansions (along $x = 0$):
  - $\cos x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}$
  - $\sin x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$
  - $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$
  - $\cosh x = \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!}$
  - $\sinh x = \sum_{k=0}^{\infty} \frac{x^{2k+1}}{(2k+1)!}$
  - $\ln(1 - x) = -\sum_{k=0}^{\infty} \frac{x^{k+1}}{k+1}$, if $-1 < x < 1$.

9. Predefined modules in Python

- We have seen that importation of programs written by ourselves is easy:

```
def Sqrt ( a, *args ):
    nmax =1000
    tol=1.e-20
    if len(args)>0:
        nmax=args[0]
        if len(args)>1:
            tol=args[1]
    x0 = 1
    Abs = 1
    x=x0
    i=0
    while Abs >= tol and i<nmax:
        X = 0.5* (x+(a/x))
        Abs = abs(X-x)
        x=X
        i+=1
    if i>=nmax:
        print("precision not reached")
    return X
```

Contents of `sqrt.py`

```
import sqrt as sq

x = float(input("Write x"))
print("Square root is {:.20f}".format(sq.Sqrt(x)))
```

Contents of `program_using_sqrt.py`

Running `program_using_sqrt.py` and inputting `3` yields `1.7320508075688719318`, etc.

- The option to customise module `sqrt.py` as `sq` is ours and we can call it anything we want – obviously modifying the way we invoke function `Sqrt` from this module, e.g.

  `import sqrt as whatever` entails changing line four to `....format(whatever.Sqrt(x)))`

- Modules can be collected into **libraries** according to some commonality (e.g. libraries devoted to maths modules, to statistics modules, to matrix modules, etc).
- However, if we have to program every single module and library from scratch like we did with `Sqrt`, the line between spending time and wasting it becomes blurred.
- With Python, we are lucky to have many modules that are predefined (e.g. programmed by someone else) but have been tested and checked enough times to be considered nearly *default* or *intrinsic* to the language itself.
- Most of these modules are part of the **Standard Library**; others are valuable extensions of it.
- You can find the Standard Library modules in `https://docs.python.org/3/library/`.
  Most commonly used Standard Library modules and external libraries:
  - the `math` module
  - the `sys` module
  - the `Matplotlib` library
  - the `NumPy` library
  - the `SciPy` library
  - the `SymPy` library
  - the `random` module
  - the `Astropy` library
  - the `time` module
- You can find manuals on these with `help`, e.g. by including `help(math)`.

The `math` package

- Contains most of the mathematical functions you will be using, e.g. `sqrt`. Assume we modify `program_using_sqrt.py` (while leaving our own, user-programmed `sqrt.py` intact):

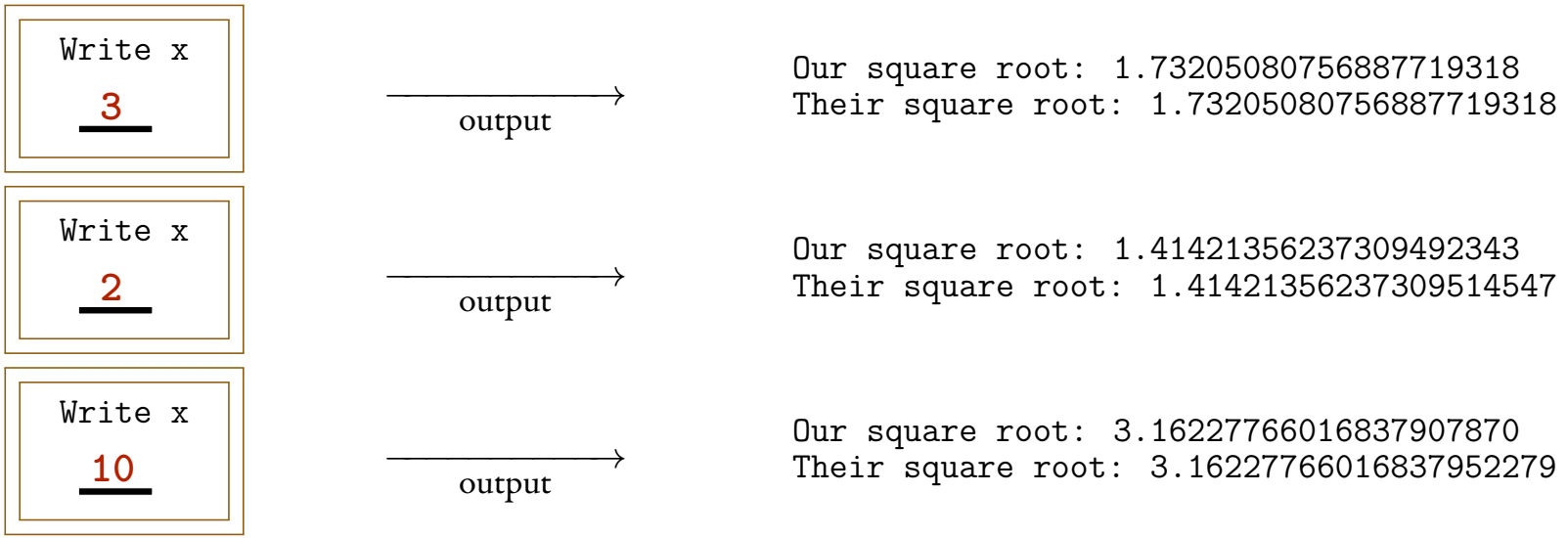
```
def Sqrt ( a, *args ):
    nmax =1000

    # ... Our program ...

    return X
```

Contents of `sqrt.py`

```
import math
import sqrt as sq

x = float(input("Write x"))
print("Our square root: :.20f".format(sq.Sqrt(x)))
print("Their square root: :.20f".format(math.sqrt(x)))
```

Contents of modified `program_using_sqrt.py`

Comparison seems to indicate at least 15 correct decimal digits after rounding up:



but most computations using square roots will not require all of our 20 digits.

- A few other functions in `math` (as usual, preceded by `math.` if you imported it as such):
  - `math.factorial(x)`: the already-known $x!$
  - `floor(x)`: $\lfloor x \rfloor = \max\{k \in \mathbb{Z} : k \leq x\}$
  - `fsum(L)`: sum of the elements in list L
  - `gcd(a,b)`: *greatest common divisor* $a, b \in \mathbb{Z}$
  - `exp(x)`: already-known *exponential* e^x
  - `log(x)`: *natural logarithm* $\ln x = \log_e x$
  - `log(x,b)`: *logarithm* in any base $\log_b x$
  - `pow(x,y)`: x^y regardless of whether $y \in \mathbb{Z}$
  - `cos(x)`: $\cos x$, *cosine* of x
  - `sin(x)`: $\sin x$, *sine* of x
  - `tan(x)`: $\tan x$, *tangent* of x
  - `cosh(x)`: $\cosh x$, *hyperbolic cosine* of x
  - `sinh(x)`: $\sinh x$, *hyperbolic sine* of x
  - `tanh(x)`: $\tanh x$, *hyperbolic tangent* of x
  - `acos(x)`: $\arccos x$, *arccosine* of x
  - `asin(x)`: $\arcsin x$, *arcsine* of x
  - `atan(x)`: $\arctan x$, *arctangent* of x
  - `acosh(x)`: $\text{arccosh}x$, *hyp. arccosine* of x
  - `asinh(x)`: $\text{arcsinh}x$, *hyp. arcsine* of x
  - `atanh(x)`: $\text{arctanh}x$, *hyp. arctangent* of x

Where $\mathbb{Z}$ stands for the set of all **integer numbers**.
x, y are assumed, by default to be general **real** numbers $\in \mathbb{R}$ (i.e. `float` types).

- Constants too:
  - `math.e`: $e \simeq 2.718281828459045$
  - `math.pi`: $\pi \simeq 3.141592653589793$
  - `math.inf`: ∞ or anything beyond a computable limit, e.g. `1.e1000`.
  - `math.nan`: NAN (*Not a number*, i.e. not amenable to arithmetic, e.g. `math.inf-math.inf`
- Sometimes we will need to "complete" already-existing functions anyway:

```
import math

def gcd(*nums):
    if len(nums)==1:
        return nums[0]
    if len(nums)>=2:
        return math.gcd(nums[0],gcd(*nums[1:]))
    return 1
print(gcd (578, 221, 255, 85))
```

$\xrightarrow[\text{output}]{}$ 17

which allows us to compute the GCD of *any* set of integers with variadic arguments.

The sys package

- Contains useful functions affecting our interaction with the environment where we call Python.
- For instance, `sys.argv` is a list which by default contains only one element – the file name:

```
import sys

def main():
    print(sys.argv)

if __name__ == '__main__':
    main()
```

`program.py` called from command line or Jupyter notebook

output → `['program.py']`

- If we now call the function from the command line or Jupyter notebook using more arguments,

```
In [3] : run "program.py" 1
In [4] : run "program.py" 0.1 hello
In [5] : _
```

JUPYTER NOTEBOOK

output →
```
['program.py', '1']
['program.py', '0.1', 'hello']
```

Thus if we ask for `sys.argv[2]` in the second case, we would obtain `'hello'`, etc. This can be useful to pass data to the program instead of explicitly asking for it via `input` from the user.

The `time` package

- As the name indicates, this module contains time-related functions.
- `time.time()` counts time in seconds since a given epoch (in Unix: January 1, 1970, 00:00:00):

```
import time
import sqrt as sq
import math

def main():
    start = time.time()
    squareroot1 = sq.Sqrt(18120915176)
    end = time.time()
    diff = end - start
    print("Our Sqrt: ",diff,"sec")

    start = time.time()
    squareroot2 = math.sqrt(18120915176)
    end = time.time()
    diff = end - start
    print("math.sqrt:",diff,"sec")
    cur = time.time()
    print("current time:",cur," sec" )

if __name__ == '__main__':
    main()
```

output

```
Our Sqrt: 2.002716064453125e-05 sec
math.sqrt: 3.0994415283203125e-06 sec
current time: 1572802240.834068 sec
```

Prologue to the NumPy library: Matrices

- **Matrices** are 2-dimensional arrays of numbers (*entries* or *coefficients*) arranged in *rows* and *columns*

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,j} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,j} & \dots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i,1} & a_{i,2} & \dots & a_{i,j} & \dots & a_{i,m} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,j} & \dots & a_{n,m} \end{pmatrix} \leftarrow \text{row } i$$

$$\uparrow$$

column j

Compact forms of notation are

$$A = \big(a_{i,j}\big) = \big(a_{i,j}\big)_{i,j} = \big(a_{i,j}\big)_{\substack{i=1,\dots,n \\ j=1,\dots,m}} = \big(a_{i,j}\big)_{\substack{1 \le i \le n \\ 1 \le j \le m}}$$

- $\mathrm{Mat}_{r,c}(\mathbb{R})$ = set of $r \times c$ matrices having real entries. If $r = c$ the matrices are called **square**

$$A = \begin{pmatrix} 1 & -2 & 0.1 \\ 0 & \pi & 3 \end{pmatrix} \in \mathrm{Mat}_{2,3}(\mathbb{R}), \qquad B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & -1 & 9 \\ 0 & 0 & 3 & 4 \\ 0 & 0 & 0 & 4 \end{pmatrix} \in \mathrm{Mat}_{4,4}(\mathbb{R}),$$

meaning B is square but A is not and $a_{1,1} = 1,\ a_{1,2} = -2,\ a_{1,3} = 0.1, b_{2,4} = 9, \dots$

- $1 \times n$ (one-row) or $m \times 1$ (one-column) matrices are called **row (resp. column) vectors**.
- Individual numbers (on their own or appearing as matrix entries) are called **scalars**.

- Two matrices can be **added** if they share the same numbers of rows and columns:

$$A + B = (a_{i,j} + b_{i,j})_{i,j} = \begin{pmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & \dots & a_{1,m} + b_{1,m} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & \dots & a_{2,m} + b_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} + b_{n,1} & a_{n,2} + b_{n,2} & \dots & a_{n,m} + b_{n,m} \end{pmatrix}$$

  For instance, the sum of two 2×3 matrices yields a 2×3 matrix:

$$\begin{pmatrix} 1.1 & -2 & 0.4 \\ 0 & 4.7 & 5.1 \end{pmatrix} + \begin{pmatrix} 2 & 9.1 & 0.7 \\ 4.8 & 6 & 0.4 \end{pmatrix} = \begin{pmatrix} 1.1+2 & -2+9.1 & 0.4+0.7 \\ 0+4.8 & 4.7+6 & 5.1+0.4 \end{pmatrix} = \begin{pmatrix} \mathbf{3.1} & \mathbf{7.1} & \mathbf{1.1} \\ \mathbf{4.8} & \mathbf{10.7} & \mathbf{5.5} \end{pmatrix}$$

- Same applies to inverse operation of $+$ (*subtraction*), replacing every occurrence of $+$ by $-$:

$$\begin{pmatrix} 1.1 & -2 & 0.4 \\ 0 & 4.7 & 5.1 \end{pmatrix} - \begin{pmatrix} 2 & 9.1 & 0.7 \\ 4.8 & 6 & 0.4 \end{pmatrix} = \begin{pmatrix} 1.1-2 & -2-9.1 & 0.4-0.7 \\ 0-4.8 & 4.7-6 & 5.1-0.4 \end{pmatrix} = \begin{pmatrix} \mathbf{-0.9} & \mathbf{-11.1} & \mathbf{-0.3} \\ \mathbf{-4.8} & \mathbf{-1.3} & \mathbf{4.7} \end{pmatrix}$$

- The particular case of row or column vectors becomes much simpler:

$$\begin{pmatrix} 0 & 1 & 3 \end{pmatrix} + \begin{pmatrix} 1 & 2 & 0.4 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 3.4 \end{pmatrix}$$

$$\begin{pmatrix} 3 \\ 4 \end{pmatrix} - \begin{pmatrix} \pi \\ -1000 \end{pmatrix} = \begin{pmatrix} 3-\pi \\ 1004 \end{pmatrix}$$

- Matrix addition *generalises* real number addition, i.e. given two numbers (i.e. two 1×1 matrices minus the brackets), adding them as usual numbers is equivalent to adding them as matrices.

- Matrices can be **multiplied** in a number of ways, but the one "natural" to Linear Algebra (i.e. representing compositions of linear maps) is the following, ***provided that the number of columns of the first matrix equals the number of rows of the second***:

$$\left.\begin{array}{l} A \in \mathrm{Mat}_{n,m} \\ B \in \mathrm{Mat}_{m,p} \end{array}\right\} \quad \boxed{A \cdot B = (c_{i,j})_{i,j} = \begin{pmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,p} \\ c_{2,1} & c_{2,2} & \dots & c_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \dots & c_{n,p} \end{pmatrix} \quad \text{where} \quad \boxed{c_{i,j} = \sum_{k=1}^{m} a_{i,k} b_{k,j}}}$$

- Hence, entry $_{i,j}$ in $A \cdot B$ is the **dot product** of two vectors: row i in A and column j in B:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{a_{i,1}} & \boldsymbol{a_{i,2}} & \boldsymbol{\dots} & \boldsymbol{a_{i,m}} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{1,m} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & \dots & \boldsymbol{b_{1,j}} & \dots & b_{1,p} \\ b_{2,1} & \dots & \boldsymbol{b_{2,j}} & \dots & b_{2,p} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{m,1} & \dots & \boldsymbol{b_{m,j}} & \dots & b_{m,p} \end{pmatrix} = \begin{pmatrix} \star & & \vdots & & \star \\ & & \vdots & & \\ \boldsymbol{\dots} & \boldsymbol{\dots} & \boxed{\boldsymbol{c_{i,j}}} & \boldsymbol{\dots} & \boldsymbol{\dots} \\ & & \vdots & & \\ \star & & \vdots & & \star \end{pmatrix}$$

  where $c_{i,j} = \langle \text{row } i \text{ in } A, \underline{\text{column } j \text{ in } A} \rangle = \boldsymbol{a_{i,1}\underline{b_{1,j}} + a_{i,2}\underline{b_{2,j}} + \dots + a_{m,1}\underline{b_{m,j}}}$.
- Again, $\cdot$ generalises the usual product in $\mathbb{R}$: for 1×1 matrices (a), (b), their product is the same as real numbers or according to the above definition.

- For example, given 3×4 and 4×2 matrices

$$A = \begin{pmatrix} 1 & 2 & -4 & 3 \\ 0 & 7 & 0.1 & 6 \\ 5 & 4.3 & -1 & 2.2 \end{pmatrix}, \qquad B = \begin{pmatrix} 1 & 0.2 \\ 2 & 1.3 \\ 4 & 0 \\ 11 & 1 \end{pmatrix}$$

- we cannot multiply $B \cdot A$ because the number of columns of $B \neq$ the number of rows in A.
- we can however multiply $A \cdot B$ because number of columns of $A =$ number of rows in A:

$$A \cdot B = \left(\begin{array}{c|c} \mathbf{1 \cdot \underline{1} + 2 \cdot \underline{2} + (-4) \cdot \underline{4} + 3 \cdot \underline{11}} & \mathbf{1 \cdot \underline{0.2} + 2 \cdot \underline{1.3} + (-4) \cdot \underline{0} + 3 \cdot \underline{1}} \\ \hline \mathbf{0 \cdot \underline{1} + 7 \cdot \underline{2} + 0.1 \cdot \underline{4} + 6 \cdot \underline{11}} & \mathbf{0 \cdot \underline{0.2} + 7 \cdot \underline{1.3} + 0.1 \cdot \underline{0} + 6 \cdot \underline{1}} \\ \hline \mathbf{5 \cdot \underline{1} + 4.3 \cdot \underline{2} + (-1) \cdot \underline{4} + 2.2 \cdot \underline{11}} & \mathbf{5 \cdot \underline{0.2} + 4.3 \cdot \underline{1.3} + (-1) \cdot \underline{0} + 2.2 \cdot \underline{1}} \end{array}\right)$$

$$= \begin{pmatrix} \mathbf{22.} & \mathbf{5.8} \\ \mathbf{80.4} & \mathbf{15.1} \\ \mathbf{33.8} & \mathbf{8.79} \end{pmatrix}$$

which, unsurprisingly, has

  - the same number of rows (3) as the first matrix A,
  - and the same number of columns (2) as the second matrix B.

The *third* number, i.e. the dimension shared by both matrices ($4 =$ col. of $A =$ rows of B) plays no further role after the multiplication is completed.

The NumPy library

- Contains useful tools for several disciplines but particularly for Linear Algebra, i.e. manipulation of arrays of one or more dimensions (vectors and matrices).
- External to the Standard Library, hence must be imported; see info in `https://numpy.org/`
- The centerpiece is type `numpy.array` which, *at a first glance*, works exactly like a list:

```
import numpy as np

a = np.array([1, 2, 3, 4])
print(type(a))
print(a[0], a[3])
a[0] = 5
print(a)
```

output

```
<class 'numpy.ndarray'>
1 4
[5 2 3 4]
```

- It can also be converted to a list and there is a `range` analogue in `np.array` types::

```
print(a)
ll=list(a)
print(ll)
```

output

```
[5 2 3 4]
[5, 2, 3, 4]
```

```
x = np.arange(3)
print(x)
print(type(x))
```

output

```
[0 1 2]
<class 'numpy.ndarray'>
```

- Most importantly, `numpy` contains functions like those in `math`, but handles those functions, e.g. np.cos, `np.sin`, `np.exp` much faster than their `math` counterparts. Hence **we strongly recommend using `numpy` instead of `math` for special functions and constants.**

- For `float` types, we can decide whether we want **half**, **single** or **double floating-point precision**:

```
import numpy as np
x = np.array([1.1,2], dtype=np.float16)
y = np.array([1.1,2], dtype=np.float32)
z = np.array([1.1,2], dtype=np.float64)
print("%.30f" % x[0])
print("%.30f" % y[0])
print("%.30f" % z[0])
```

output

```
1.099609375000000000000000000000
1.100000023841857910156250000000
1.100000000000000088817841970013
```

- We can also alter data types for integers when it comes to ranges:
  - `int8` -128 to 127
  - `int16` -32768 to 32767
  - `int32` -2147483648 to 2147483647
  - `int64` -9223372036854775808 to 9223372036854775807

With the understanding that attention will have to be paid if we do not wish to overflow variables:

```
x = np.array([1.1,2], dtype=np.int16)
y = np.array([1.1,2], dtype=np.int32)
y[0]=x[0]=500000
print("%d" % x[0])
print("%d" % y[0])
```

output

```
-24288
500000
```

If you are not sure what precision your computer is working with by default,

```
print( np.dtype(int) )
```

→ `int64`

→ `int32`

- Other ways of converting and presetting precision (for matrices, vectors and numbers) are:

```
import numpy as np

x = np.float32(2.4)
print("x = %.20f" % x)

X = np.float_(2.4)
print("X = %.20f" % X)

XX = np.float16(2.4)
print("XX = %.20f" % XX)

y = np.int_([10.11, 20.56, -1.2])
print("y=",y)

print("unsigned int y=", end="")
y = np.uint8([10.11, 20.56, -1.2])
print(y)

z = np.arange(4, dtype=np.int8)
print("z=",z)
zz=z.astype(np.float16)
print("float z=",zz)

t = np.array([1, 2], dtype='f')
print(t)

u = np.array([1, 2], dtype=np.float16)
print(u)
```

output

```
x = 2.40000009536743164062

X = 2.39999999999999991118

XX = 2.40039062500000000000

y= [10 20 -1]

unsigned int y=[ 10 20 255]

z= [0 1 2 3]

float z= [0. 1. 2. 3.]

[1. 2.]

[1. 2.]
```

- Special matrices, e.g.
  - matrices (including vectors) all of whose entries equal the same element,

$$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \begin{pmatrix} 3 & 3 \\ 3 & 3 \\ 3 & 3 \end{pmatrix}, \quad \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 2 & 2 & 2 & 2 & 2 \end{pmatrix}$$

  - the **identity matrix** playing the role of a multiplicative neutral element in matrices:

$$\mathrm{Id}_n = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \dots & 1 \end{pmatrix} \quad \text{eg.} \quad \mathrm{Id}_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \ \mathrm{Id}_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \dots \quad \text{and} \quad \begin{cases} A \cdot \mathrm{Id} = A \\ \mathrm{Id} \cdot B = B \\ \text{for any } A, B \end{cases}$$

  - or matrices (including vectors) whose entries are random `float` numbers,

are easy to input as arrays here, e.g.

```
A = np.ones((2,3))
print(A)
B = np.zeros((2,2))
print(B)
C = np.full((3,3), 4)
print(C)
D = np.eye(2)
print(D)
E = np.eye(2,dtype=np.int16)
print(E)
F = np.random.random((2,4))
print(F)
```

output

```
[[1. 1. 1.]
 [1. 1. 1.]]
[[0. 0.]
 [0. 0.]]
[[4 4 4]
 [4 4 4]
 [4 4 4]]
[[1. 0.]
 [0. 1.]]
[[1 0]
 [0 1]]
[[0.19851442 0.19231117 0.15831314 0.09988122]
 [0.63140522 0.5235444 0.2604196 0.95926241]]
```

- **Matrix arithmetic**:
  - $\pm$ works by overriding operators `+`, `-` or by using the `add` and `subtract` functions in `numpy`:

```
x = np.array([[1,2,3],[4,5,6]], dtype=np.float64)
y = np.array([[-3,1,4],[3,2,8]], dtype=np.float64)
print(x + y)
print(np.add(x, y))
print(x - y)
print(np.subtract(x, y))
```

output

```
[[-2.  3.  7.]
 [ 7.  7. 14.]]
[[-2.  3.  7.]
 [ 7.  7. 14.]]
[[ 4.  1. -1.]
 [ 1.  3. -2.]]
[[ 4.  1. -1.]
 [ 1.  3. -2.]]
```

  - `*` or `multiply` do not correspond to matrix multiplication as defined earlier, but to **entrywise multiplication** (the **Hadamard product**):

```
print(x * y)
print(np.multiply(x, y))
```

output

```
[[-3.  2. 12.]
 [12. 10. 48.]]
[[-3.  2. 12.]
 [12. 10. 48.]]
```

  - `/` or `divide`, accordingly, produce **Hadamard division**:

```
print(x / y)
print(np.divide(x, y))
```

output

```
[[-0.33333333 2.   0.75 ]
 [ 1.33333333 2.5  0.75 ]]
[[-0.33333333 2.   0.75 ]
 [ 1.33333333 2.5  0.75 ]]
```

- **Matrix arithmetic**:
  - Dot products $\langle \star, \star \rangle$ and matrix multiplication as seen pages ago, need `numpy` function `dot`:

```
import numpy as np

mat1 = np.array([[3,-2],[1,10]])
mat2 = np.array([[1,4],[3,6]])
vec1 = np.array([1,2],dtype='f')
vec2 = np.array([11, 12])

# Dot product of vectors: two ways
print(vec1.dot(vec2))
print(np.dot(vec1, vec2))

# Matrix-vector product: two ways
print(mat1.dot(vec1))
print(np.dot(mat1, vec1))
print(mat1.dot(vec2))
print(np.dot(mat1, vec2))

# Matrix-matrix product: two ways
print(mat1.dot(mat2))
print(np.dot(mat1, mat2))
```

output →

```
35.0

35.0

[-1. 21.]

[-1. 21.]

[ 9 131]

[ 9 131]

[[-3 0]
[31 64]]

[[-3 0]
[31 64]]
```

  - Needless to say, if dimensions are not compatible (i.e. # columns of the first matrix $\neq$ # of rows of second) multiplication is impossible:

```
mat1 = np.array([[3,-2,1],[1,10,4]])
mat2 = np.array([[1,4],[3,6]])
print(mat1.dot(mat2))
```

output →

```
ValueError:
shapes (2,3) and (2,2)
not aligned
```

Exercises

- Imagine you live in a world without `numpy`.
  - Define a function `input_vector` inputting a vector as a **list** from keyboard.
  - Define a function `input_matrix` inputting a matrix as a **list (of lists)** from keyboard.
  - Your already-known recursive and non-recursive functions `array_sum` can add vectors. Now do the same for an `array_subtract`.
  - Define functions `add_matrices`, `subtract_matrices` compatible with your `input_vector`, `input_matrix` above.
  - Define a function `multiply_matrix_vector` compatible with your `input_vector`, `input_matrix` above for the usual matrix-vector product. Make sure your function checks compatibility of the two matrices.
  - Define a function `multiply_matrices` compatible with your `input_matrix` above for the usual matrix product. Make sure your function checks compatibility of the two matrices.
  - Define a function `Hadamard_product` compatible with your `input_matrix` above for the usual matrix product. Make sure your function checks row and column number identities.
  - Write functions such as the above, but in this case taking matrices and vectors as inputs from *external* files.

- Imagine you live in a world with `numpy` but you don't know all of its features. Every **square** matrix has a **determinant**, which can be computed as follows:
  - If the matrix is 2×2,

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \Rightarrow \quad \det A = \|A\| = ad - bc$$

  - Assume A has $n \geq 3$ rows and columns. Choose a row or column, for instance in 4×4

$$\begin{pmatrix} 1 & 3 & 0 & 4 \\ \mathbf{2} & \mathbf{3} & \mathbf{0} & \mathbf{10} \\ 0 & 1 & 9 & 7 \\ 1 & 1 & 1 & 2 \end{pmatrix}$$

  - The **minor** of a matrix entry is the determinant of the smaller matrix obtained by deleting the row and column containing that entry, for instance,

$$\text{the minor of } \boxed{\mathbf{2}} \text{ in } \begin{pmatrix} 1 & 3 & 0 & 4 \\ \boxed{\mathbf{2}} & \mathbf{3} & \mathbf{0} & \mathbf{10} \\ 0 & 1 & 9 & 7 \\ 1 & 1 & 1 & 2 \end{pmatrix} \quad \text{would be} \quad \det \begin{pmatrix} 3 & 0 & 4 \\ 1 & 9 & 7 \\ 1 & 1 & 2 \end{pmatrix} = \begin{vmatrix} 3 & 0 & 4 \\ 1 & 9 & 7 \\ 1 & 1 & 2 \end{vmatrix}$$

  - The determinant of A equals the sum of the products of each element $_{i,j}$ in the chosen row/column times its **cofactor** – i.e. its **minor**, multiplied by $(-1)^{i+j}$

- For instance, in the above matrix we draw a chessboard-style scheme to determine cofactor sign:

$$\begin{pmatrix} 1 & 3 & 0 & 4 \\ \mathbf{2} & \mathbf{3} & \mathbf{0} & \mathbf{10} \\ 0 & 1 & 9 & 7 \\ 1 & 1 & 1 & 2 \end{pmatrix} \qquad \begin{matrix} + & - & + & - \\ - & + & - & + \\ + & - & + & - \\ - & + & - & + \end{matrix} \quad \Rightarrow \quad (-\mathbf{1}) \cdot \mathbf{2} \quad (+\mathbf{1}) \cdot \mathbf{3} \quad (-\mathbf{1}) \cdot \mathbf{0} \quad (+\mathbf{1}) \cdot \mathbf{10}$$

- and compute its determinant using smaller determinants, each computed the same way:

$$\begin{aligned} \begin{vmatrix} 1 & 3 & 0 & 4 \\ 2 & 3 & 0 & 10 \\ 0 & 1 & 9 & 7 \\ 1 & 1 & 1 & 2 \end{vmatrix} &= (-\mathbf{1}) \cdot \mathbf{2} \det \begin{bmatrix} 3 & 0 & 4 \\ 1 & 9 & 7 \\ 1 & 1 & 2 \end{bmatrix} + (+\mathbf{1}) \cdot \mathbf{3} \det \begin{bmatrix} 1 & 0 & 4 \\ 0 & 9 & 7 \\ 1 & 1 & 2 \end{bmatrix} \\ &+ (-\mathbf{1}) \cdot \mathbf{0} \det \begin{bmatrix} 1 & 3 & 4 \\ 0 & 1 & 7 \\ 1 & 1 & 2 \end{bmatrix} + (+\mathbf{1}) \cdot \mathbf{10} \det \begin{bmatrix} 1 & 3 & 0 \\ 0 & 1 & 9 \\ 1 & 1 & 1 \end{bmatrix} \\ &= \cdots = 113. \end{aligned}$$

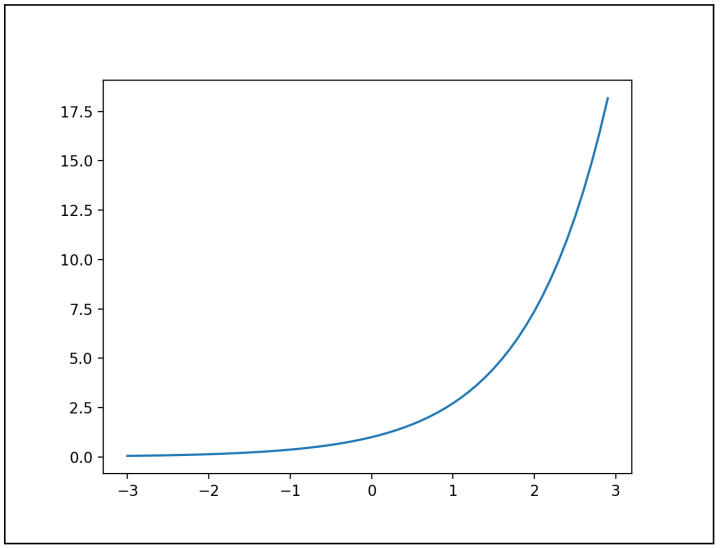
- Compute a program that takes a square matrix, no matter how large, from a previously filled-out external file and computes its determinant with a recursive function. Make sure your program also prints out the time taken to perform this task.

10 The `matplotlib` library

- Useful to plot data in all its forms (function graphs, sets of points...)
- External to the Standard Library; see documentation in `https://matplotlib.org/index.html`
- We will focus on module `matplotlib.pyplot` whose inner workings are similar to MatLab (which you will see in semester 2).
- First example:

```
import numpy as np
import matplotlib.pyplot as plt

# array of equidistant numbers
x = np.arange(-3, 3, 0.1)
# array of their exponentials
y = np.exp(x)

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

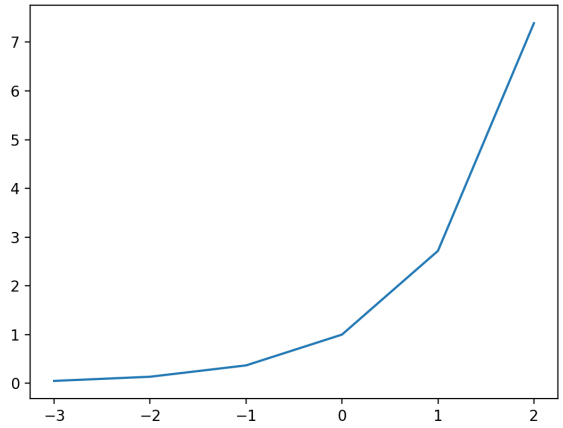
output



- Making the mesh of this partition larger (from `0.1` to `1.`), it becomes apparent that by default the dots in our plot are *joined* instead of drawn separately:

```
x = np.arange(-3, 3, 1.)
y = np.exp(x)
plt.plot(x, y)
plt.show()
```

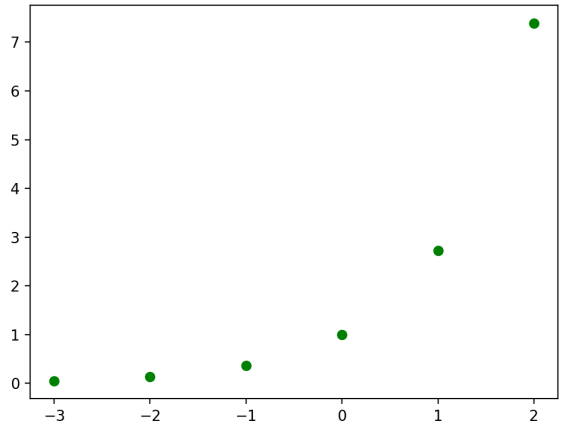
output



- To draw these points separately (regardless of shape or colour) we need to write a bit more:

```
x = np.arange(-3, 3, 1.)
y = np.exp(x)
plt.plot(x, y,'go')
plt.show()
```

output



- Here is a list of the most-often used markers:

Marker	Shape	Marker	Shape
`'.'`	Point	`'|'`	Vertical Line
`'+'`	Plus	`'_'`	Horizontal Line
`'o'`	Circle	`'^'`	Triangle upwards
`','`	Pixel	`'*'`	Star
`'s'`	Square	`(n,s,a)`	Regular `n`-polygon rotated by an angle `a`, with style `s`: `s`=0 : *convex* regular polygon `s`=1 : *concave* (star-shaped) regular polygon `s`=2 : an *asterisk* (limit case of `s`=1) `s`=3 : a *circle* and `n`, `a` are ignored

- Line styles (default is `-`):

Style	Description	Style	Description
`'-'`	Solid line	`'--'`	Dashed line
`'-.'`	Dash-dot line	`':'`	Dotted line

- Colours (default is `b`):

Letter	Colour	Letter	Colour	Letter	Colour	Letter	Colour
`'r'`	Red	`'k'`	Black	`'w'`	White	`'y'`	Yellow
`'g'`	Green	`'m'`	Magenta	`'c'`	Cyan	`'b'`	Blue

- Let us create two different files:
  - `largefile.txt` with many points forming a smooth curve,
  - `smallfile.txt` with just a few, their ordinates (y-coordinates) created *randomly*

```
import numpy as np
import matplotlib.pyplot as plt

large = open ("largefile.txt","w")
small = open ("smallfile.txt","w")
# a few random numbers with integer x-coordinates
for xk in range(10):
    yk=np.random.random()
    small.write("%.20f %.20f" % (xk,yk))
# a lot of numbers following a smooth curve (cosine)
for i in range(1000):
    xk = i/200.
    yk=np.cos(xk)
    large.write("%.20f %.20f" % (xk,yk))
large.close()
small.close()
```

contents of `store.py`

as you can see, `numpy` is not only useful for operations with arrays and matrices.

Exercises

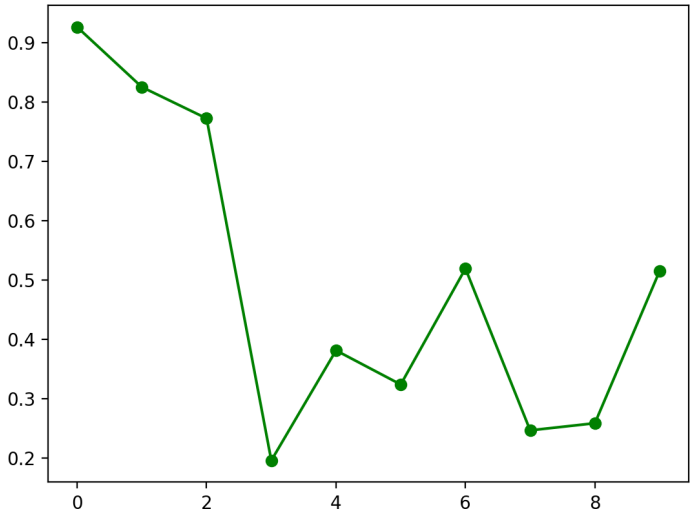
- This file is incomplete and needs you to fill out the contents of function `read_from_file`:

```
import numpy as np
import matplotlib.pyplot as plt

# Function to read data from a file:
def read_from_file( filename ):
    """ this function should return two lists:
    - x containing the numbers of the first column of filename
    - y containing the numbers in the second column of filename
    """
    return x,y

smallx, smally = read_from_file( "smallfile.txt" )
largex, largey = read_from_file( "largefile.txt" )

# any other preliminary options
plt.plot(..., ...,"...")
plt.show()
```

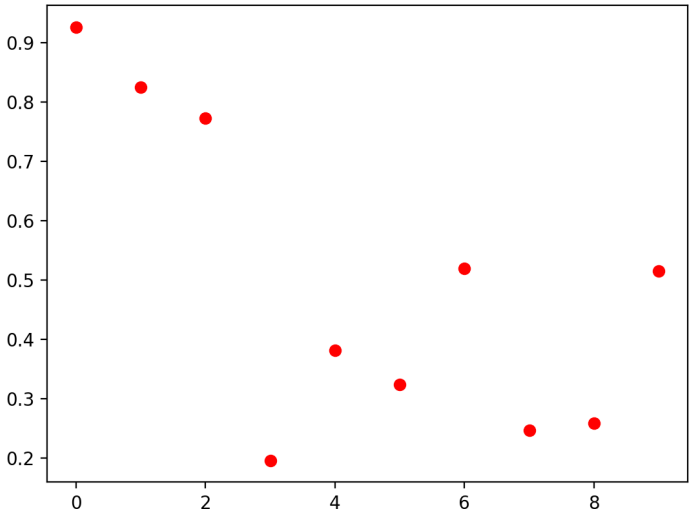
contents of `plot.py`

The following pages show outputs of running `plot.py`, *depending on what we write in the last* ***three*** *lines*.

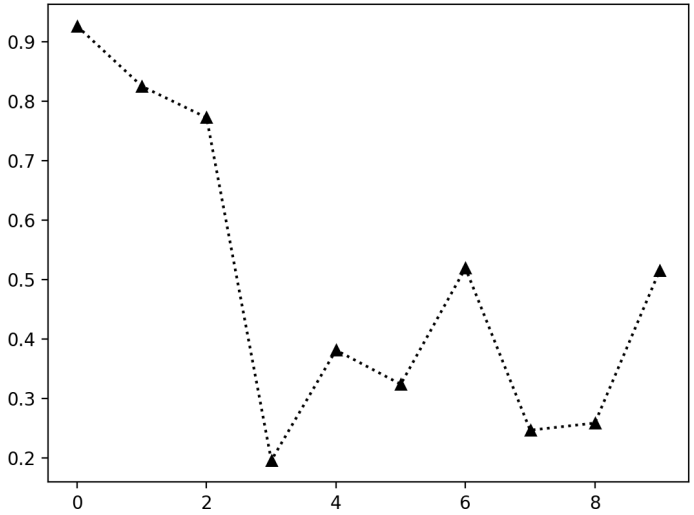
For the smaller file. It is advised that you practice all possible options.



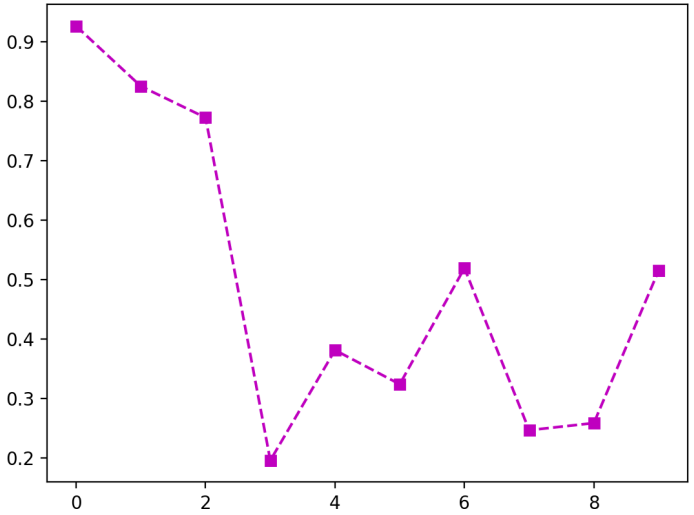

```
plt.plot(smallx, smally,"go-")
```




```
plt.plot(smallx, smally,"ro")
```

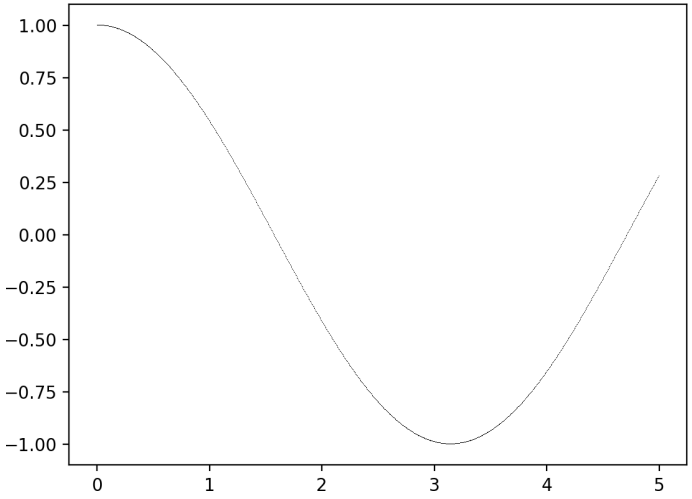



```
plt.plot(smallx, smally,"^k:")
```

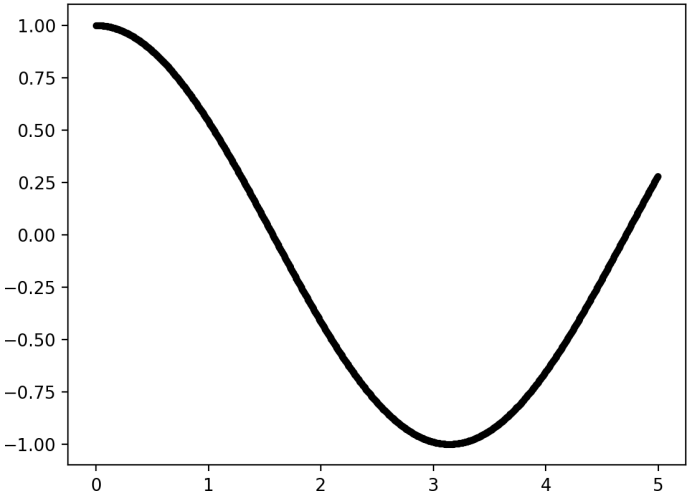



```
plt.plot(smallx, smally,"ms--")
```

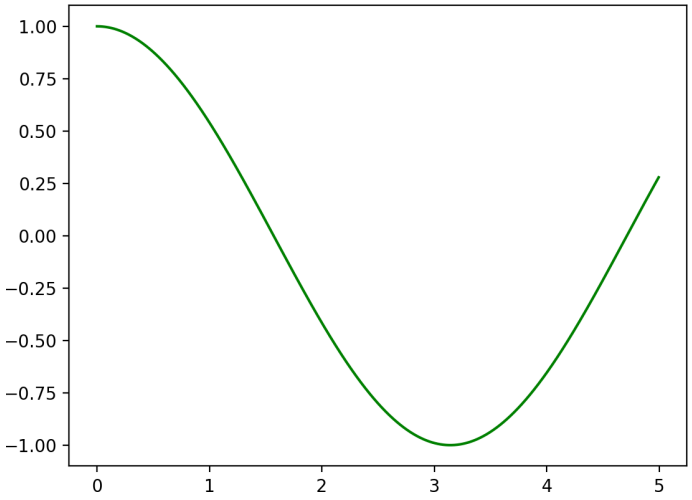
For the larger file. It is advised that you practice all possible options.



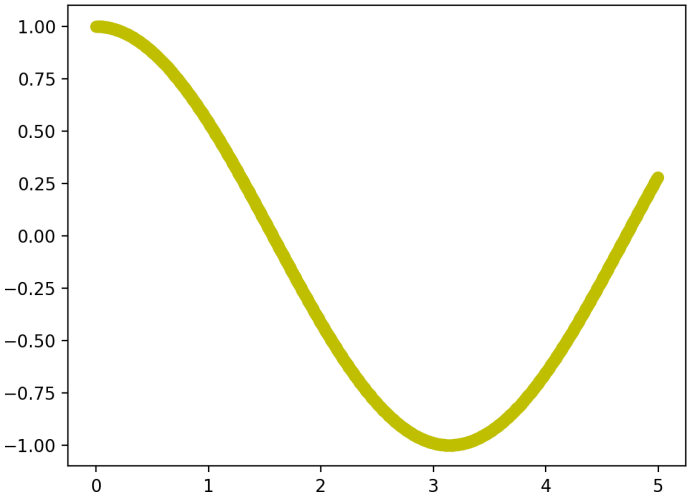

```
plt.plot(largex, largey,"k,")
```




```
plt.plot(largex, largey,"k.")
```



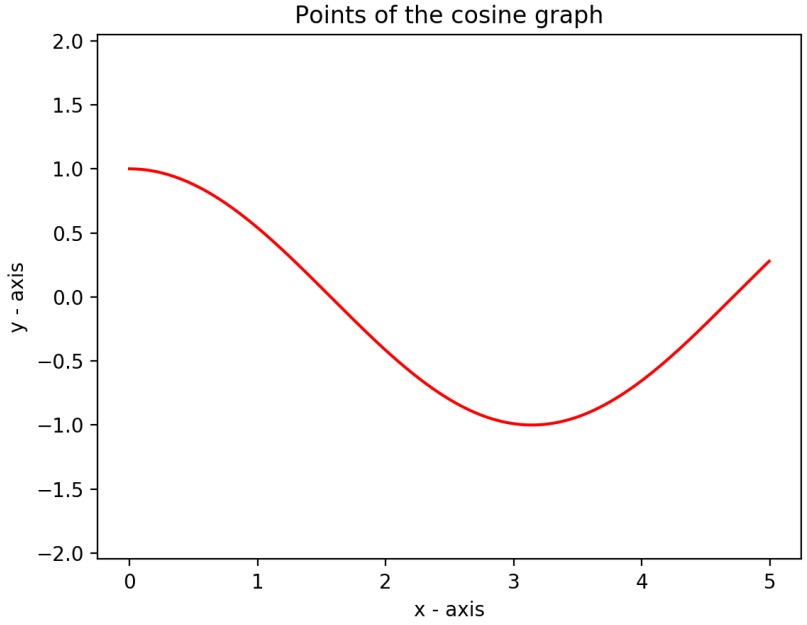

```
plt.plot(largex, largey,'g-')
```



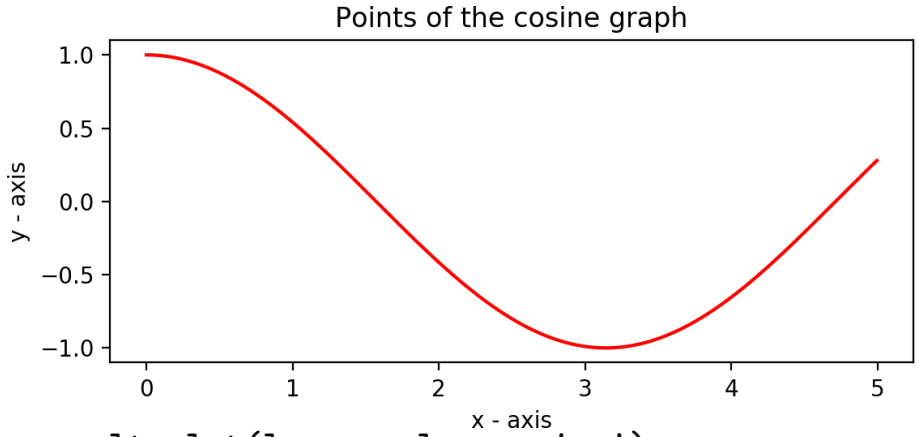

```
plt.plot(largex, largey,'yo')
```

We can add other options, e.g.

- modify the aspect ratio and/or trim our image
- place text and labels

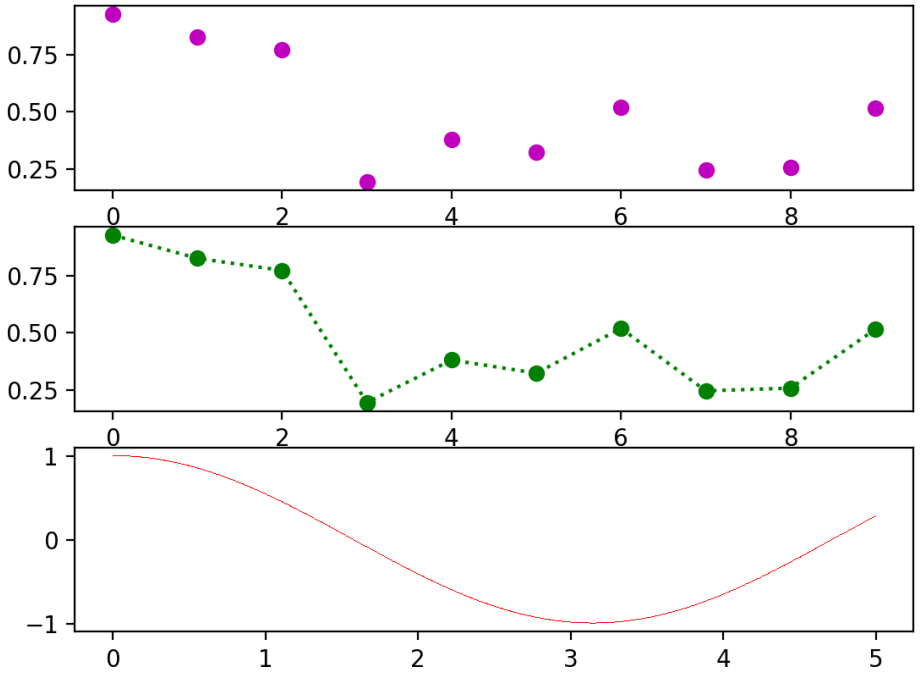



```
plt.plot(largex, largey,'r-')
plt.axes().set_aspect('equal', 'datalim')
plt.xlabel('x - axis')
plt.ylabel('y - axis')
plt.title('Points of the cosine graph')
plt.show()
```

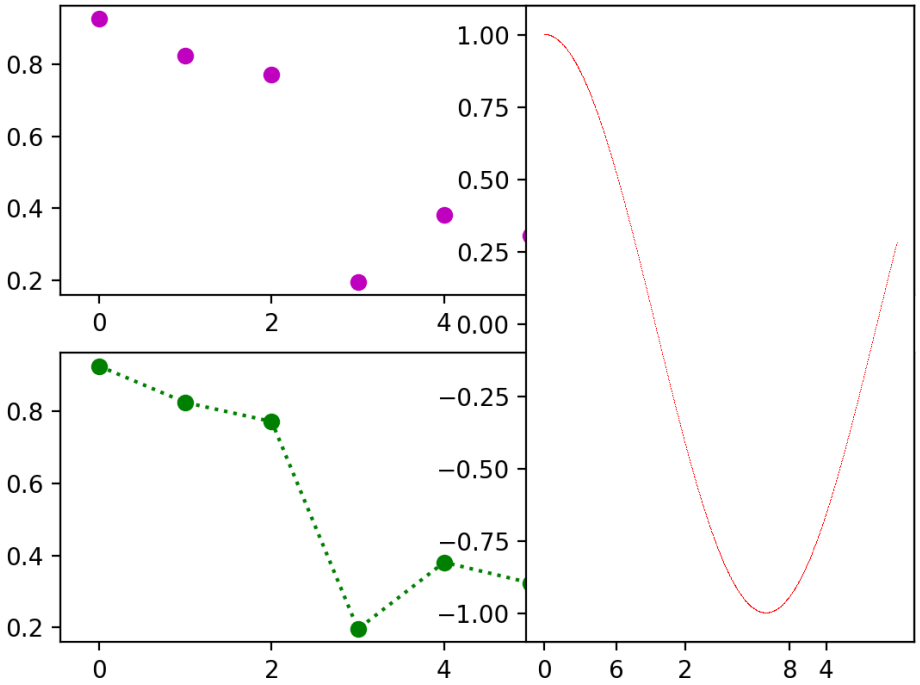



```
plt.plot(largex, largey,'r-')
plt.axes().set_aspect('equal', 'box')
plt.xlabel('x - axis')
plt.ylabel('y - axis')
plt.title('Points of the cosine graph')
plt.show()
```

We can also typify plots as variables using `figure` and use functions involving them, e.g.

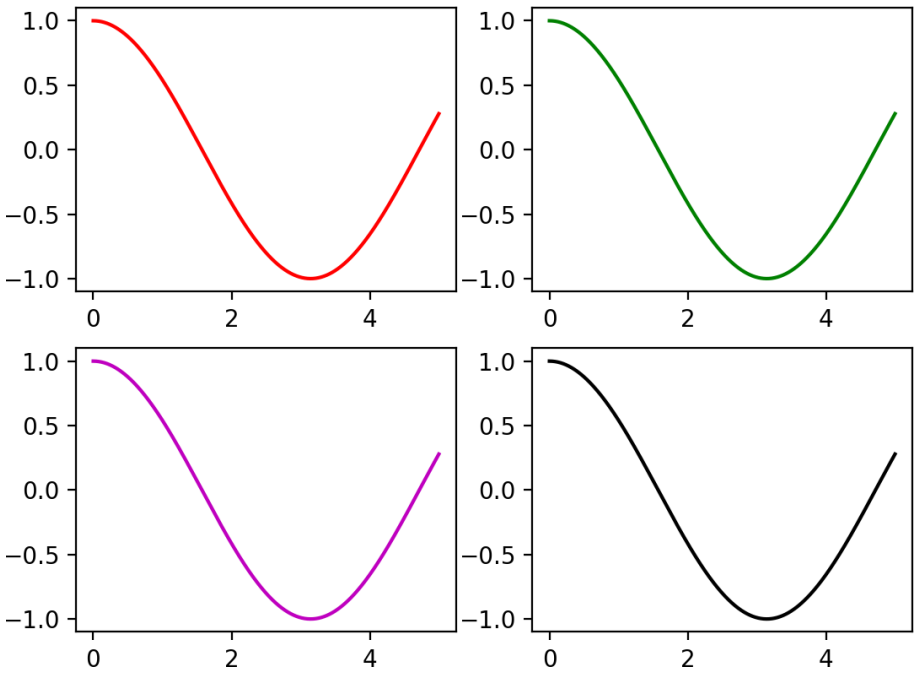



```
fig1 = plt.figure() # Make a new figure window
plt.clf() # clear current figure
fig1.add_subplot(3, 1,1) # 3 rows 1 column, plot 1
plt.plot(smallx, smally,'mo')
fig1.add_subplot(3, 1,2) # 3 rows 1 column, plot 2
plt.plot(smallx, smally,'go:')
fig1.add_subplot(3, 1,3) # 3 rows 1 column, plot 3
plt.plot(largex, largey,'r,')
plt.show()
```

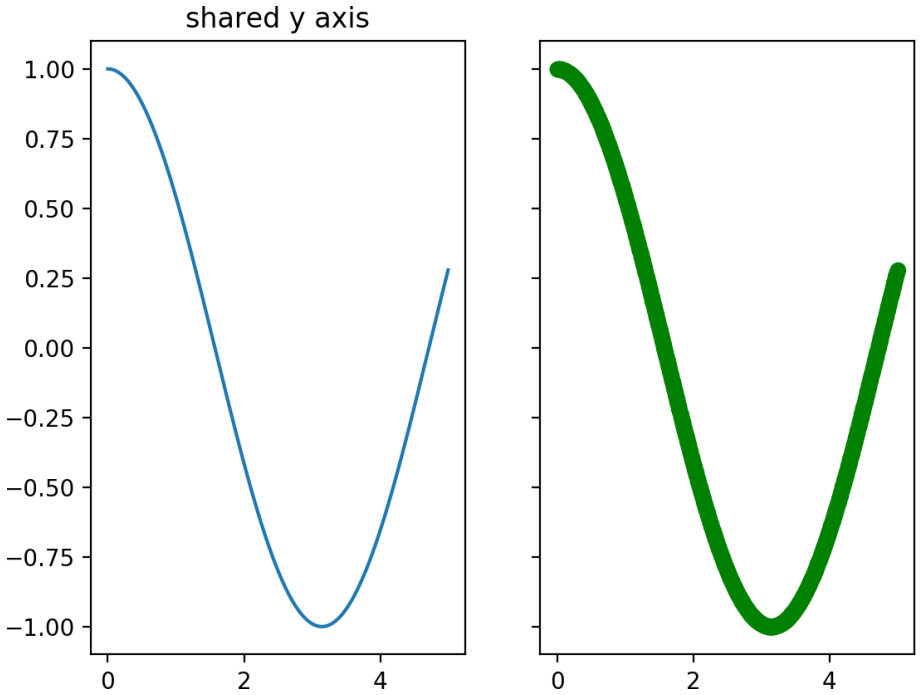



```
fig1 = plt.figure()
plt.clf() # clear current figure
fig1.add_subplot(2,1,1)
plt.plot(smallx, smally,'mo')
fig1.add_subplot(2,1,2)
plt.plot(smallx, smally,'go:')
fig1.add_subplot(1,2,2)
plt.plot(largex, largey,'r,')
plt.show()
```

More grids of subplots with other methods:

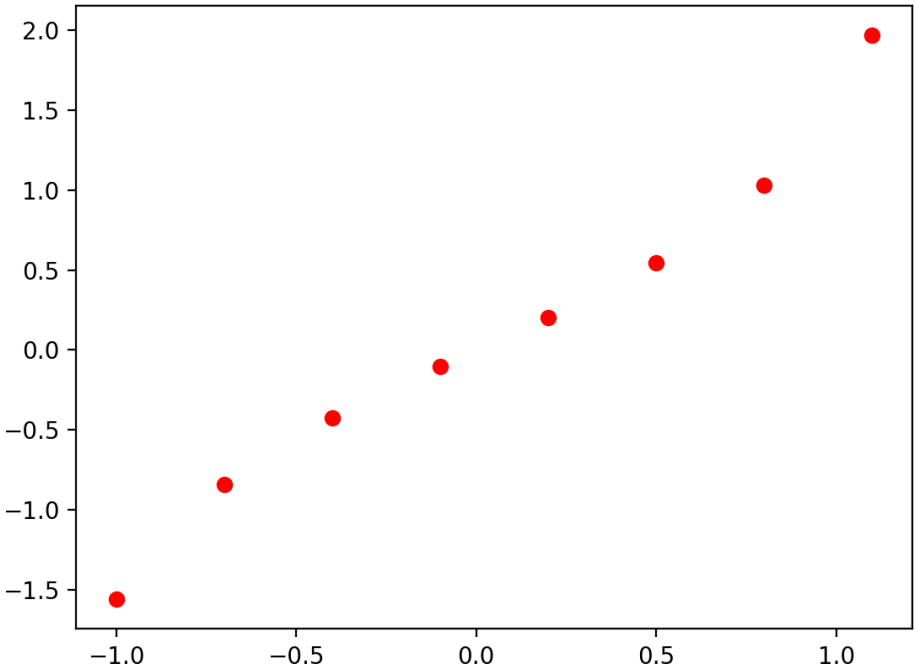



```
fig, ax = plt.subplots(2, 2)
ax[0, 0].plot(largex, largey, 'r')
ax[1, 0].plot(largex, largey, 'm')
ax[0, 1].plot(largex, largey, 'g')
ax[1, 1].plot(largex, largey, 'k')
plt.show()
```

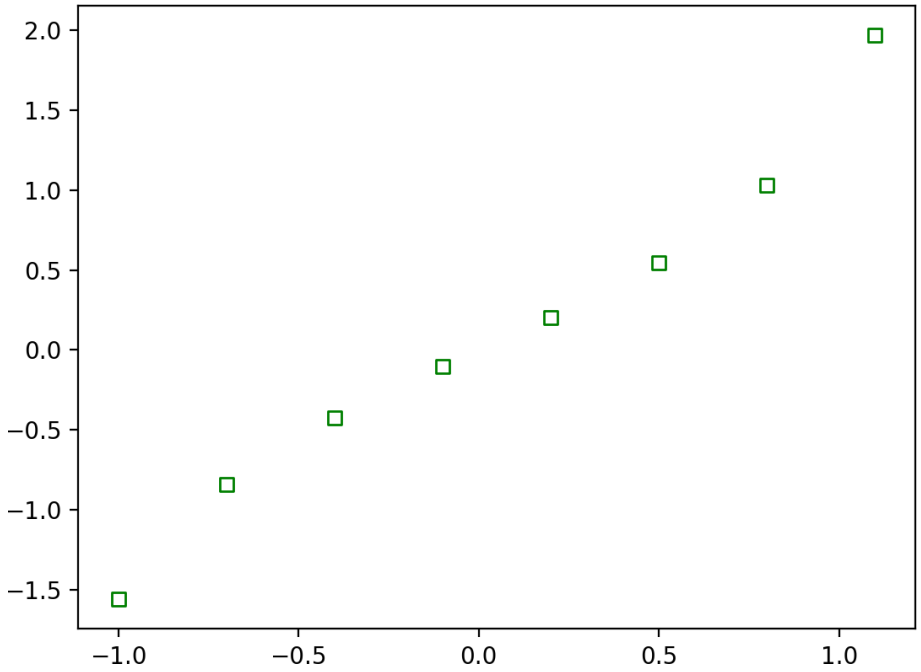



```
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(largex, largey)
ax1.set_title('shared y axis')
ax2.plot(largex, largey,'go')
plt.show()
```

`scatter` works like `plot ... 'o'`, i.e. without joining the dots:

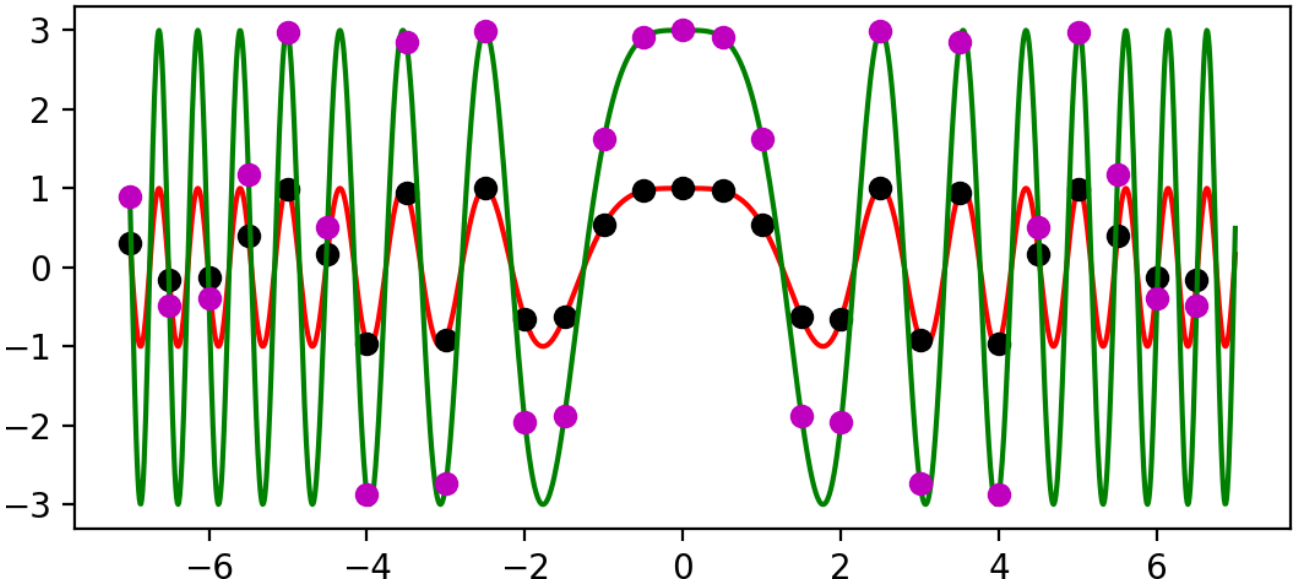



```
x = np.arange(-1, 1.1, 0.3)
y = np.tan(x)
plt.scatter(x, y,c='r')
plt.show()
```




```
x = np.arange(-1, 1.1, 0.3)
y = np.tan(x)
plt.scatter(x, y,c='w',marker='s',edgecolors='g')
plt.show()
```

Important to note that ***joint plots*** are also immediate to program:




```
x = np.arange(-7, 7, 0.5)
y = np.cos(x**2)
X = np.arange(-7, 7, 0.01)
Y = np.cos(X**2)
xx = np.arange(-7, 7, 0.5)
yy = 3*np.cos(xx**2)
XX = np.arange(-7, 7, 0.01)
YY = 3*np.cos(X**2)
plt.axes().set_aspect('equal', 'box')
plt.plot(X, Y,'r',x, y,'ko',XX,YY,'g-',xx, yy,'mo')
plt.show()
```

- We saw two weeks ago a method to approximate integrals: the **composite trapezium rule**, i.e. divide interval $[a, b]$ into n subintervals of equal length, then apply simple trapezium to each.
- The larger n, the smaller the length h of each subinterval, thus (theoretically) the closer $T_n =$ `trapezium(f,a,b,n)` is to the actual area.

  (i) Write a program that plots $h = \frac{b-a}{n}$ against $\left|\int_a^b f - T_n\right|$.

  (ii) We apply **Richardson extrapolation** to the well-known formula for the numerical error:

$$T_n = \int_a^b f + Ah^2 + \text{higher terms of } h, \quad \text{for some } A, \quad \text{where } h = \frac{b-a}{n}.$$

  if we compute T_n and T_{2n}, we can combine them to obtain better approximations:

$$\left.\begin{array}{l} T_n = \int_a^b f + Ah^2 + \ldots, \\ T_{2n} = \int_a^b f + A(\frac{h}{2})^2 + \ldots, \end{array}\right\} \Rightarrow \boxed{R_2 = \frac{2^2 T_{2n} - T_n}{2^2 - 1}} = \int_a^b f + \text{lowest term } h^4 \ldots$$

  This is called **Romberg's method** and can be brought to higher orders (think how).

  (iii) Plot Romberg output errors along with the plot in (i), check that indeed they provide better approximations. Think of ways of plotting this that entail some clear visibility of all information returned by your program.

11 In-class exercise

- A **system of linear equations**,

$$\left.\begin{array}{rcl} b_1 & = & a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,m}x_m \\ b_2 & = & a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,m}x_m \\ \vdots & \vdots & \vdots \\ b_n & = & a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,m}x_m \end{array}\right\} \qquad \text{(LS)}$$

can be written in matrix form:

$$\boxed{A\mathbf{x} = \mathbf{b}} \quad \text{where} \quad A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

For instance,

$$\left.\begin{array}{r} 2x - y + t = 1 \\ 2x + 9t + 3y = -5 \\ -4t + 8x + y + z = 7 \end{array}\right\} \Rightarrow A = \begin{pmatrix} 2 & -1 & 0 & 1 \\ 2 & 3 & 0 & 9 \\ 8 & 1 & 1 & -4 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1 \\ -5 \\ 7 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix}$$

A and $\mathbf{b}$ given, to **solve** (LS) is to find $\mathbf{x}$ satisfying all equations in the system.

- Possible cases if the matrix A is **square** $(n = m)$:
  - If $\det A \neq 0$, (LS) has a **unique** solution **x**.
  - If $\det A = 0$, (LS) can either have:
    - **infinitely many** solutions **x**.
    - or **no solution** at all.
- Examples:
  - A system with only one solution:

$$\left.\begin{aligned} x - y + 3z &= 1 \\ 2x + 4y + 2z &= 1 \\ 7x + z &= 3 \end{aligned}\right\} \Rightarrow \boxed{\begin{aligned} x &= \tfrac{9}{23} \\ y &= \tfrac{4}{23} \\ z &= \tfrac{6}{23} \end{aligned}} \quad \text{unsurprisingly because } \det \begin{pmatrix} 1 & -1 & 3 \\ 2 & 4 & 2 \\ 7 & 0 & 1 \end{pmatrix} = -92 \neq 0$$

  - A system with infinitely many solutions (in this case with two *free* variables, e.g. x and y):

$$\overbrace{\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 4 & 8 & 12 \end{pmatrix}}^{\det=0} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \frac{1}{3} \end{pmatrix} + x \begin{pmatrix} 1 \\ 0 \\ -\frac{1}{3} \end{pmatrix} + y \begin{pmatrix} 0 \\ 1 \\ -\frac{2}{3} \end{pmatrix}$$

  - A system with no solutions:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 4 & 8 & 12 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}.$$

- **Cramer's rule** provides a formula for the solution of $A\mathbf{x} = \mathbf{b}$ if $n = m$ and $\det A \neq 0$:

$$x_1 = \frac{|A_1|}{|A|}, \quad x_2 = \frac{|A_2|}{|A|}, \quad \dots \quad x_n = \frac{|A_n|}{|A|},$$

  where
  - $|A|$ is the determinant of the whole original matrix
  - $\left|A_j\right|$ is the determinant of the matrix obtained by replacing column j in A by vector $\mathbf{b}$.
- For instance, for

$$\begin{pmatrix} 3 & 1 & 2 \\ 0 & 4 & 2 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix}$$

  the matrix has determinant 6 and Cramer's rule implies:

$$x_1 = \frac{\begin{vmatrix} \mathbf{1} & 1 & 2 \\ \mathbf{2} & 4 & 2 \\ \mathbf{4} & 0 & 1 \end{vmatrix}}{6} = -\frac{11}{3}, \qquad x_2 = \frac{\begin{vmatrix} 3 & \mathbf{1} & 2 \\ 0 & \mathbf{2} & 2 \\ 1 & \mathbf{4} & 1 \end{vmatrix}}{6} = -\frac{10}{3}, \qquad x_3 = \frac{\begin{vmatrix} 3 & 1 & \mathbf{1} \\ 0 & 4 & \mathbf{2} \\ 1 & 0 & \mathbf{4} \end{vmatrix}}{6} = \frac{23}{3}.$$

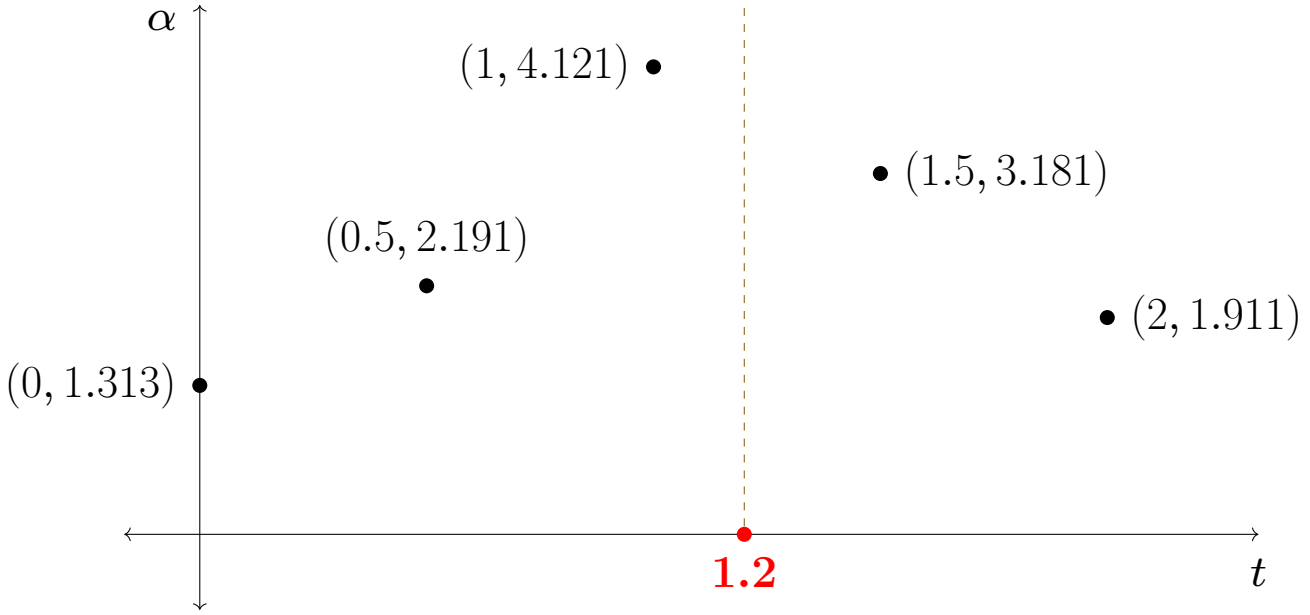
- Write a program that takes any system as input from an external file, does the adequate checks and then solves it using Cramer. Make sure it keeps control of the time spent in the process.

12 Comments about the coursework[this refers to 2019/2020]

- **Interpolation** is the "completion" of known (and finite) data by assuming it fits a simple pattern.
- Some points are given in a table, for instance the relation between time t and the angle formed by a satellite with the Ecliptic plane:

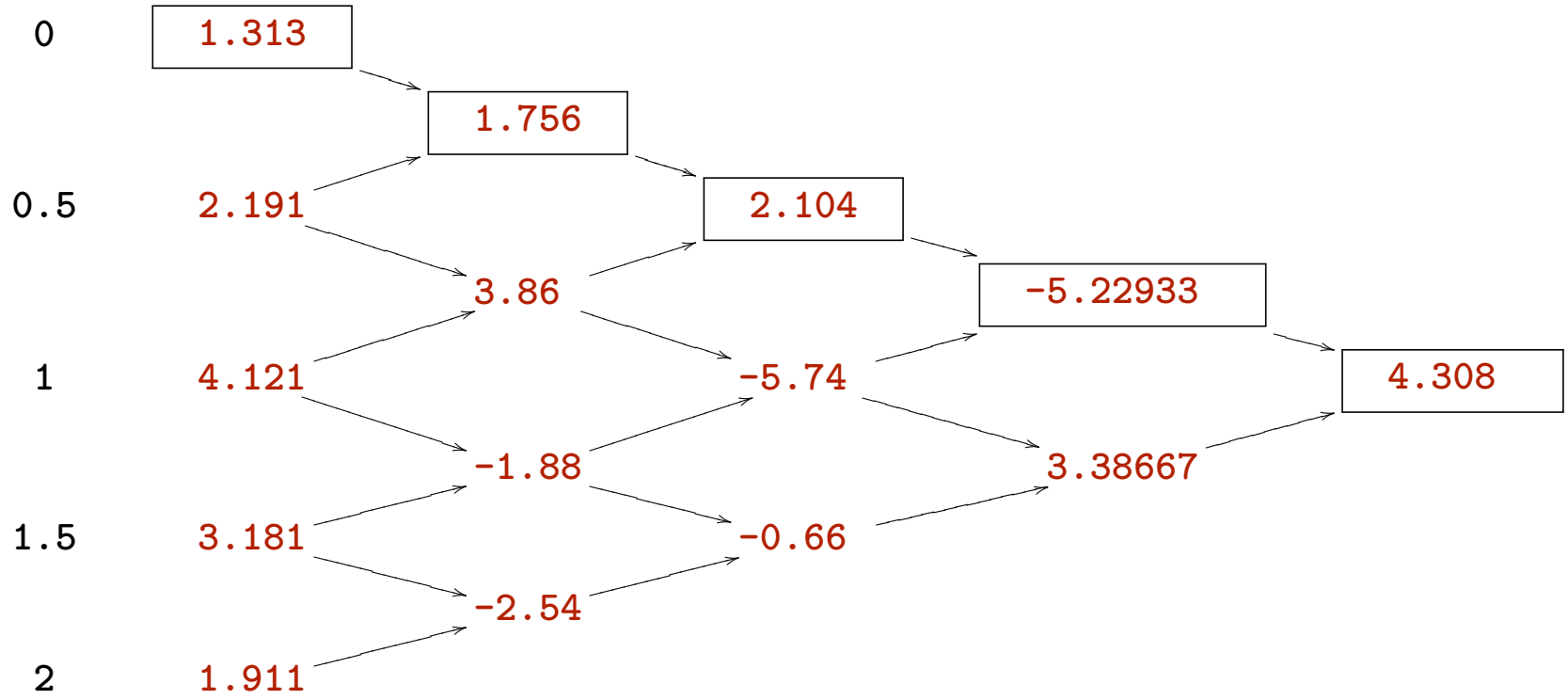
t	0.	0.5	1.	1.5	2
α	1.313	2.191	4.121	3.181	1.911

- We would like to know the angle α if $t = 1.2$, which does not appear in our experimental table:



- We do not know whether α is indeed a function of t in real life; interpolation ignores this question and proceeds to *approximate* α by a simple function $p(t) = a_0 + a_1 t + \dots$

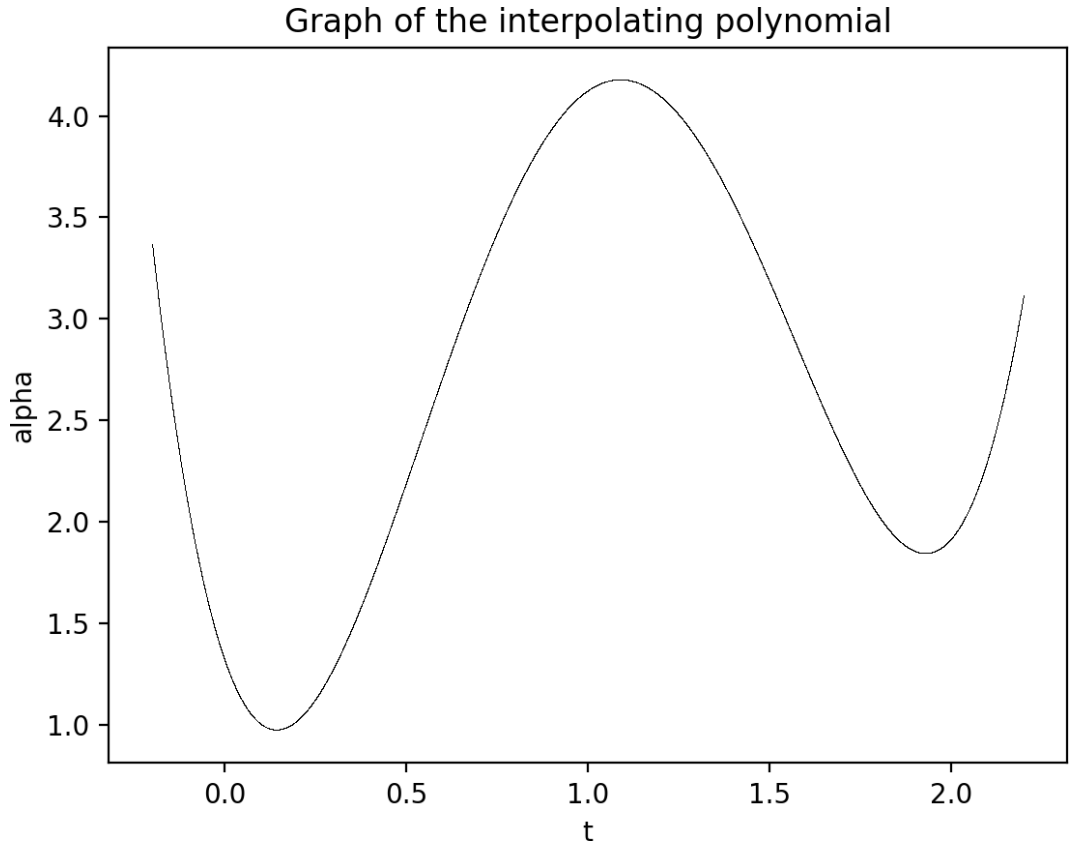
- This *interpolating polynomial* $p(t)$ has degree at most one minus the number of points. Hence in this case we expect a polynomial of degree **four**: $p(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4$.
- The method proposed is using **Newton's divided differences**:



then the polynomial uses the marked elements in the upper row:

$$\begin{aligned} p(t) &= \boxed{1.313} + (t-0)\left\{\boxed{1.756} + (t-0.5)\left[\boxed{2.104} + (t-1)\left(\boxed{-5.22933} + (t-1.5)\boxed{4.308}\right)\right]\right\} \\ &= 4.308t^4 - 18.1533t^3 + 21.795t^2 - 5.14167t + 1.313. \end{aligned}$$

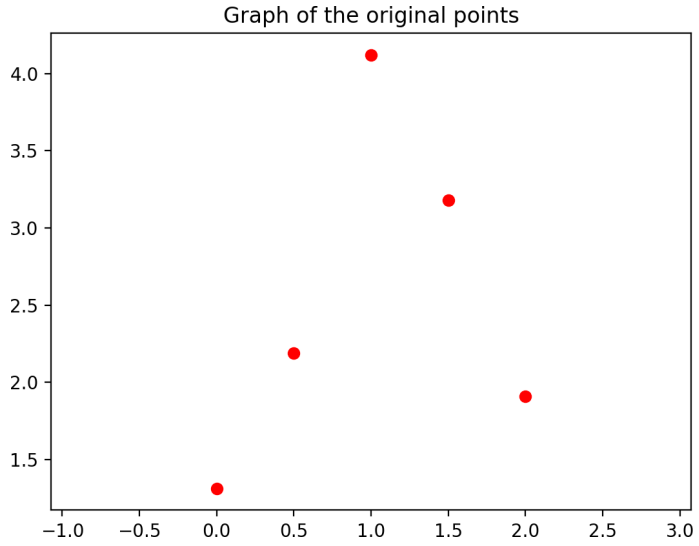
- Expanded expression $4.308t^4 - 18.1533t^3 + 21.795t^2 - 5.14167t + 1.313$ is easy to complete and not practical to us: it is the *other* expression $\boxed{\texttt{1.313}} + (t - \texttt{0})\,\{\ldots\}$ that minimises operations.
- The fact it minimises operations means you can evaluate $p(t)$ numertically in thousands of values of t in less than one second (and then plot the resulting graph of points):

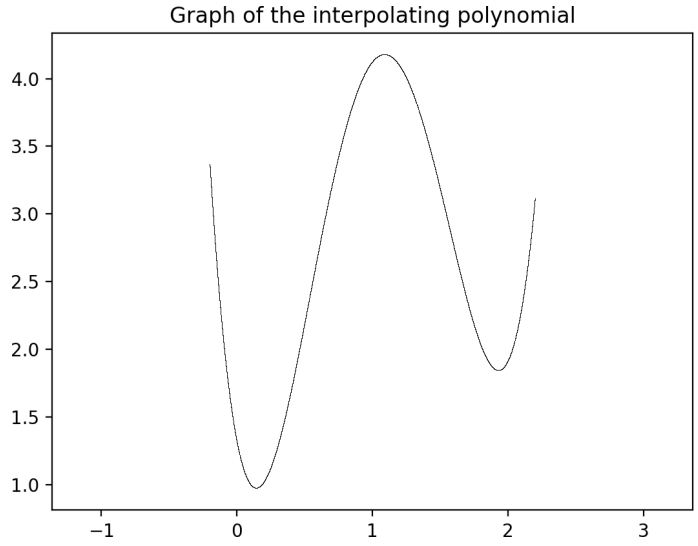


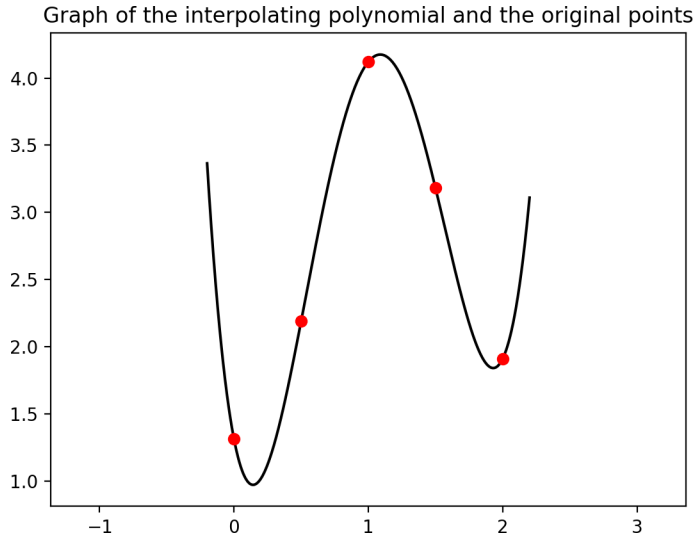
and our desired approximation for $\alpha(1.2)$ (our initial question) is $p(1.2) = 4.09191$:

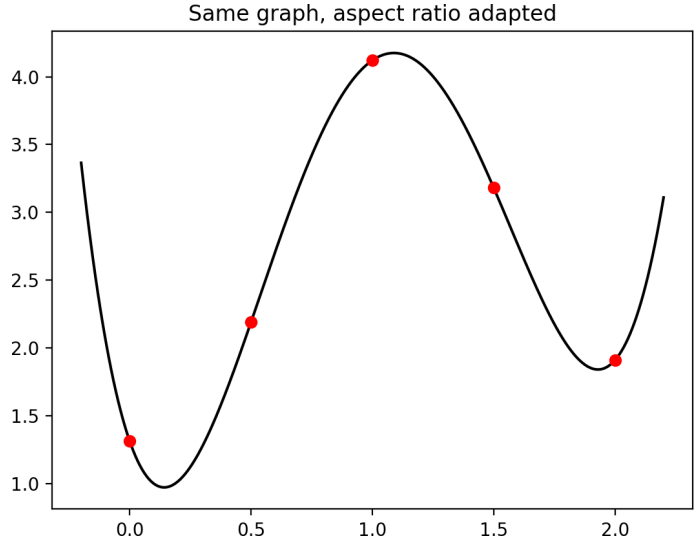
$$p(1.2) = 1.313 + (1.2 - 0)\,\{1.756 + (1.2 - 0.5)\,[2.104 + (1.2 - 1)\,(-5.22933 + (1.2 - 1.5)4.308)]\} = \boxed{4.09191}$$

- Your program should be able to return (among other things) something like this:









- Check the sample Jupyter notebook provided in the assessment folder for sample things your program could be providing.

- Your Jupyter notebook should **not** look remotely similar to the one provided. Write it in your own style, with your own checks and peculiarities (any additions to it will be welcome as well).
- as a way to **check** your calculations, you can also solve the linear system attached to the interpolating problem, using the method explained last week, e.g. for the given problem in the first slide, we're looking for a polynomial of degree one less than the number of points, $p(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4$, and fitting the polynomial to the table entails

$$\begin{cases} p(0) = 1.313 \\ p(0.5) = 2.191 \\ p(1) = 4.121 \\ p(1.5) = 3.181 \\ p(2) = 1.911 \end{cases} \Rightarrow \begin{cases} a_0 = 1.313 \\ a_0 + 0.5a_1 + 0.25a_2 + 0.125a_3 + 0.0625a_4 = 2.191 \\ a_0 + a_1 + a_2 + a_3 + a_4 = 4.121 \\ a_0 + 1.5a_1 + 2.25a_2 + 3.375a_3 + 5.0625a_4 = 3.181 \\ a_0 + 2a_1 + 4a_2 + 8a_3 + 16a_4 = 1.911 \end{cases}$$

which you can solve for $a_0, \ldots, a_4$ using Cramer's rule and obtain $a_0 = 1.313, a_1 = -5.14167, a_2 = 21.795, a_3 = -18.1533, a_4 = 4.308$, unsurprisingly the same result obtained earlier by expanding the polynomial.

We need to insist: **solving the linear system is only a <u>check</u> (and a computationally ineffective one) and will constitute <u>0</u> marks in itself. *Your coursework must use divided differences*.**

- You are free to compute the set of divided differences in any way or form you deem fit (matrices, recursive functions...). However,
  - if you use an entire matrix to compute the differences, you are allocating memory for it and that can be costly if the matrix is large;
  - and if you compute divided differences recursively, you are computing most differences twice (think why).

  Hence if you avoid both matrices and recursiveness, you will get bonus marks.

Sergi Simon, December 6, 2019.

13 Classes and objects

- To be continued...

INTRODUCTION TO COMPUTATIONAL PHYSICS

First Coursework

U24200 –Academic Session 2019–2020

Instructions

a) This is worth **50%** of your total mark for this unit.

b) **You must undertake this assignment individually.**

c) Submission method: **by Moodle through the available dropbox**.

d) Submission deadline: **January 13, 2019**

1 Introduction

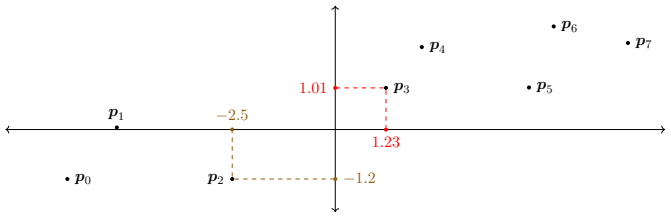
Interpolation consists in building a simple function f (here, a *polynomial*, $f(x) = a_0 + a_1 x + \cdots + a_k x^k$) whose graph curve passes through a given set of points

$$\boldsymbol{p}_0 = (x_0, y_0)\,, \quad \boldsymbol{p}_1 = (x_1, y_1)\,, \quad \ldots\,, \quad \boldsymbol{p}_m = (x_m, y_m)\,,$$

i.e. such that $f(x_0) = y_0$, $f(x_1) = y_1$, $\ldots$, $f(x_m) = y_m$. This can be seen, for instance in the case in which

$$\begin{array}{llll} \boldsymbol{p}_0 = (-6.5, -1.2)\,, & \boldsymbol{p}_1 = (-5.3, 0.05)\,, & \boldsymbol{p}_2 = (-2.5, -1.2)\,, & \boldsymbol{p}_3 = (1.23, 1.01)\,, \\ \boldsymbol{p}_4 = (2.1, 2)\,, & \boldsymbol{p}_5 = (4.7, 1.02)\,, & \boldsymbol{p}_6 = (5.3, 2.5)\,, & \boldsymbol{p}_7 = (7.1, 2.1)\,. \end{array}$$

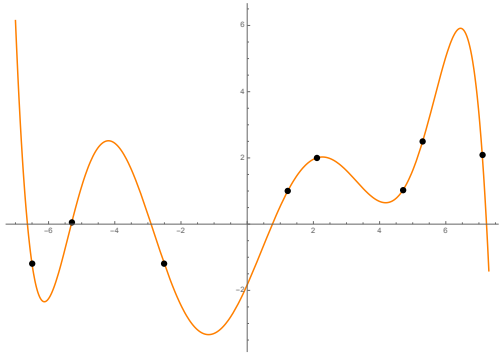
Let us first represent these points graphically:



We wish to find a polynomial whose graph traverses each one of these points. After you have finished your program, you will find that such a function can be approximated as

$$f(x) = -0.000175066x^7 + 0.000289133x^6 + 0.014541x^5 - 0.0211649x^4 - 0.34218x^3 + 0.477095x^2 + 2.24967x - 1.83489,$$

and has the shape shown in the next page. We draw it along with the original points $\boldsymbol{p}_i$.



The following result is fundamental to our purpose:

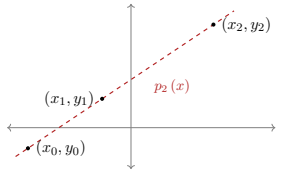
Theorem (Existence and uniqueness of the interpolating polynomial). *Let $(x_0, y_0), \ldots, (x_n, y_n)$ be $n+1$ points in the plane such that x_i are pairwise different ($x_i \neq x_j$ if $i \neq j$). Then there exists a unique polynomial of degree* at most *n, $p_n(x) = a_0 + a_1 x + \cdots + a_n x^n$, interpolating these points, i.e. such that*

$$p(x_0) = y_1, \quad p(x_1) = y_2, \quad \ldots, \quad p(x_n) = y_n. \tag{1}$$

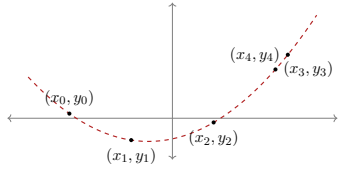
Remarks.

1. A well-known fact in Geometry, namely that any two points are traversed simultaneously by a unique line, is a particular case of the above: a line is the graph of a degree-one polynomial $y = ax + b$, and using the above two notation we would have $n = 1$ for two points $(x_0, y_0), (x_1, y_1)$.

2. n is the *maximum* value (hence an upper bound) of the degree of the interpolating polynomial but the actual degree could be less than n depending on the disposition of the points. For instance,



Three points hence $n = 2$ but they are *aligned*, thus interpolating polynomial is that line: $p(x) = a + bx + \boxed{0}\,x^2$



Five points ($n = 4$) but they are *all in the same parabola*, thus interpolating polynomial is that parabola: $p_4(x) = A + Bx + Cx^2 + D\boxed{0}\,x^3 + E\boxed{0}\,x^4$

3. Interpolation (and a similar concept called *extrapolation*) will be useful whenever you are given a table of experimental data and need to guess theoretical outputs for values not belonging to that table.

There are many ways of computing the interpolating polynomial p_n of $n+1$ points (but remember: the polynomial, due to its uniqueness, *is still the same* and depends only on the points chosen). Most notably:

- solving the linear system defined by the interpolating conditions
- method of Lagrange polynomials;
- Newton's method of divided differences;
- Aitken's method, Neville's method, etc.

We will see the third method. The first two are mostly of theoretical utility but we will give you an example of application of the second one to further illustrate the uniqueness of the polynomial for each table.

2 Lagrange polynomials

Let $x_0, \ldots, x_n$ be $n+1$ pairwise different abscissae. For every $k = 0, 1, \ldots, n$, the k^{th} **Lagrange polynomial** linked to the abscissae $x_0, \ldots, x_n$ is

$$L_k = \prod_{i=0,\, i\neq k}^{n} \frac{x - x_i}{x_k - x_i} = \frac{(x-x_0)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)}$$

Then, the interpolating polynomial for the table $\{(x_0, y_0), \ldots, (x_n, y_n)\}$ is

$$p_n(x) = y_0 L_0(x) + y_1 L_1(x) + \cdots + y_n L_n(x).$$

Example. Assume we want to find the interpolating polynomial for the following table:

x	1	2	4	8	15
$f(x)$	-0.5	0.4	0.9	1.5	1.9

(2)

The Lagrange polynomials are:

$$\begin{aligned}
L_0(x) &= \frac{(x-2)(x-4)(x-8)(x-15)}{(1-2)(1-4)(1-8)(1-15)} = \frac{(x-2)(x-4)(x-8)(x-15)}{294},\\
L_1(x) &= \frac{(x-1)(x-4)(x-8)(x-15)}{(2-1)(2-4)(2-8)(2-15)} = -\frac{(x-1)(x-4)(x-8)(x-15)}{156},\\
L_2(x) &= \frac{(x-1)(x-2)(x-8)(x-15)}{(4-1)(4-2)(4-8)(4-15)} = \frac{(x-1)(x-2)(x-8)(x-15)}{264},\\
L_3(x) &= \frac{(x-1)(x-2)(x-4)(x-15)}{(8-1)(8-2)(8-4)(8-15)} = -\frac{(x-1)(x-2)(x-4)(x-15)}{1176},\\
L_4(x) &= \frac{(x-1)(x-2)(x-4)(x-8)}{(15-1)(15-2)(15-4)(15-8)} = \frac{(x-1)(x-2)(x-4)(x-8)}{14014}
\end{aligned}$$

and the interpolating polynomial is

$$\begin{aligned}
p_4(x) &= (-0.5)\frac{(x-2)(x-4)(x-8)(x-15)}{294} - (0.4)\frac{(x-1)(x-4)(x-8)(x-15)}{156} + (\ldots)\\
&- (1.5)\frac{(x-1)(x-2)(x-4)(x-15)}{1176} + (1.9)\frac{(x-1)(x-2)(x-4)(x-8)}{14014}\\
&= -2.18962 + 2.18947x - 0.55636x^2 + 0.0585058x^3 - 0.00199562x^4,
\end{aligned} \tag{3}$$

or $p_4(x) = -\frac{153427}{70070} + \frac{306833}{140140}x - \frac{6683}{12012}x^2 + \frac{8199}{140140}x^3 - \frac{839}{420420}x^4$ if you had been working with exact rational amounts (i.e. replacing -0.5 by $-\frac{1}{2}$, 0.4 by $\frac{2}{5}$, etc in the original table). In general, you will not have the easy option to convert to rational form and you will need to work with `float` as done in (3).

3 Newton's divided differences

Assume p_n interpolates $\{(x_k, y_k) : 0 \le k \le n\}$ and we wish to find $p_n(x)$ for different values of x (for instance, $p_n(3), p_n(3.4), p_n(6.7)$). The method described in §2 would make this difficult unless we perform the final expansion (3); otherwise we would need to compute new Lagrange polynomials for each value of x. If, however, we could fix some coefficients $A_0, A_1, \ldots, A_n$ *depending only on the fixed abscissae* $x_0, \ldots, x_n$ *and not on the mobile point* x and could fit them in an expression

$$\begin{aligned}
p_n(x) &= A_0 + A_1(x-x_0) + A_2(x-x_0)(x-x_1) + A_3(x-x_0)(x-x_1)(x-x_2) + \cdots +\\
&+ A_n(x-x_0)(x-x_1)\cdots(x-x_{n-1})
\end{aligned} \tag{4}$$

this would make it easier for us to work out $p_n(x)$ in *any* x using a tool that reduces the number of operations: **generalised Horner's method**: start with $U_n = A_n$ and perform the following:

$$U_k = U_k(x - x_{k-1}) \quad \text{and} \quad U_{k-1} = U_k + A_{k-1}, \qquad k = n, \ldots, 1. \tag{5}$$

For instance, for $n = 3$ we would have

$$p_3(x) = \overbrace{A_0 + (x-x_0)\left\{\overbrace{A_1 + (x-x_1)\left[\overbrace{A_2 + \overbrace{(x-x_2)A_3}^{U_3}}^{U_2}\right]}^{U_1}\right\}}^{U_0}.$$

Newton's method of divided differences provides an expression of the form (4). Given a table $\{(x_k, y_k) : k = 0, \ldots, n\}$, $x_0, \ldots, x_m$ pairwise different, we recursively define the **divided differences** as:

$$\begin{aligned}
\boxed{y[x_k]} &= \boxed{y_k} = y_k,\\
\boxed{y[x_k, x_{k+1}]} &= \boxed{y_{k,k+1}} = \frac{y_{k+1} - y_k}{x_{k+1} - x_k},\\
\boxed{y[x_k, x_{k+1}, x_{k+2}]} &= \boxed{y_{k,k+1,k+2}} = \frac{y_{k+1,k+2} - y_{k,k+1}}{x_{k+2} - x_k},\\
&\vdots\\
\boxed{y[x_k, \ldots, x_{k+m}]} &= \boxed{y_{k,k+1,\ldots,k+m}} = \frac{y_{k+1,\ldots,k+m} - y_{k,\ldots,k+m-1}}{x_{k+m} - x_k}.
\end{aligned}$$

Both notations (boxed or double-boxed) are equally correct. These provide the $A_0, \ldots, A_n$ in (4) for p_n:

Theorem. *The interpolating polynomial for a table* $\{(x_k, y_k) : 0 \le k \le n\}$ *is*

$$\begin{aligned}
p_n(x) &= y[x_0] + y[x_0, x_1](x-x_0) + y[x_0, x_1, x_2](x-x_0)(x-x_1) +\\
&+ y[x_0, x_1, x_2, x_3](x-x_0)(x-x_1)(x-x_2) + \cdots +\\
&+ y[x_0, x_1, \ldots, x_n](x-x_0)\cdots(x-x_{n-1})
\end{aligned} \tag{6}$$

Example. We return to the table in (2). The divided difference scheme if we use pen and paper is

k	x_k	y_k	$y_{k-1,k}$	$y_{k-2,k-1,k}$	$y_{k-3,k-2,k-1,k}$	$y_{k-4,\ldots,k}$
0	1	$-\frac{1}{2}$	$\frac{\frac{4}{10}-\left(-\frac{1}{2}\right)}{2-1}=\frac{9}{10}$			
1	2	$\frac{4}{10}$	$\frac{\frac{9}{10}-\frac{4}{10}}{4-2}=\frac{1}{4}$	$\frac{\frac{1}{4}-\frac{9}{10}}{4-1}=-\frac{13}{60}$		
2	4	$\frac{9}{10}$	$\frac{\frac{3}{2}-\frac{9}{10}}{8-4}=\frac{3}{20}$	$\frac{\frac{3}{20}-\frac{1}{4}}{8-2}=-\frac{1}{60}$	$\frac{1}{35}$	$-\frac{839}{420420}$
3	8	$\frac{3}{2}$	$\frac{\frac{19}{10}-\frac{3}{2}}{15-8}=\frac{2}{35}$	$\frac{\frac{2}{35}-\frac{3}{20}}{15-4}=-\frac{13}{1540}$	$\frac{19}{30030}$	
4	15	$\frac{19}{10}$				

(7)

and we keep the elements in the upper diagonal row. The polynomial is

$$\begin{aligned} p_4(x) &= y_0 + y_{0,1}(x-x_0) + y_{0,1,2}(x-x_0)(x-x_1) + y_{0,1,2,3}(x-x_0)(x-x_1)(x-x_2) \\ &\quad + y_{0,1,2,3,4}(x-x_0)(x-x_1)(x-x_2)(x-x_3) \\ &= -\frac{1}{2} + \frac{9}{10}(x-1) - \frac{13}{60}(x-1)(x-2) + \frac{1}{35}(x-1)(x-2)(x-4) - \frac{839}{420420}(x-1)(x-2)(x-4)(x-8) \\ &= -\frac{153427}{70070} + \frac{306833}{140140}x - \frac{6683}{12012}x^2 + \frac{8199}{140140}x^3 - \frac{839}{420420}x^4 \end{aligned} \tag{8}$$

Unsurprisingly, the same polynomial we found for this given set of points using Lagrange's method. If we want to compute $p_4(x)$ for any x, if we use (8) we need to carry out 14 $+/-$, 10 $*$. However, using Horner,

$$p_4(x) = -\frac{1}{2} + (x-1)\left\{\frac{9}{10} + (x-2)\left[-\frac{13}{60} + (x-4)\left(\frac{1}{35} - \frac{839}{420420}(x-8)\right)\right]\right\}$$

which only requires eight sums or subtractions and four products.

Again, in general (**and more specifically in this coursework**) you will not have the luxury of working with pen and paper, thus you will need to keep numbers in `float` decimal point form:

x_k	y_k	$y_{k-1,k}$	$y_{k-2,k-1,k}$	$y_{k-3,k-2,k-1,k}$	$y_{k-4,\ldots,k}$
1	-0.5	$\frac{0.4-(-0.5)}{2-1}=$ 0.9			
2	0.4	$\frac{0.9-0.4}{4-2}=$ 0.25	$\frac{0.25-0.9}{4-1}\simeq$ -0.216667	0.0285714	
4	0.9	$\frac{1.5-0.9}{8-4}=$ 0.15	$\frac{0.15-0.25}{8-2}\simeq$ -0.0166667	0.000632701	-0.00199562
8	1.5	$\frac{1.9-1.5}{15-8}\simeq$ 0.0571429	$\frac{0.0571429-0.15}{15-4}\simeq$ -0.00844155		
15	1.9				

(9)

The boxed terms in (9) are the ones we keep for the polynomial. Note that regardless of whether you use finite precision (9) or the exact rational values (7), you need to compute *the entire table* to retrieve that upper row of boxed items. Now for every x, the value of $p_4(x)$ at x is

$$p_4(x) = -0.5 + (x-1)\cdot(0.9 + (x-2)\cdot(-0.216667 + (x-4)\cdot(0.0285714 + (x-8)\cdot(-0.00199562))))$$

which, if expanded, has the same form $p_4 = -2.1896249 + 2.1894750x + \ldots$ as in (3). Instead of expanding it, however, if we wanted to compute it for different values of x we would use Horner (5):

$$\begin{aligned} U_4 &= -0.00199562, \quad U_4 = U_4(x-8), \quad U_3 = U_4 + 0.0285714, \quad U_3 = U_3(x-4), \\ U_2 &= U_3 - 0.216667, \quad U_2 = U_2(x-2), \quad U_1 = U_2 + 0.9, \quad U_1 = U_1(x-1), \quad U_0 = U_1 - 0.5 \end{aligned}$$

and the value of $p_4(x)$ for that particular x would be U_0.

4 Coursework exercises

1. These will be the functions used to compute p_n:

(i) Write up a function `Newton_differences` with the following arguments:

- an array of abscissae $x_0, \ldots, x_n$ that have been previously checked to be pairwise different;
- an array of ordinates $y_0, \ldots, y_n$ comprising the other half of the table we wish to interpolate.

and returning the array of divided differences $(y_0, y_{0,1} \ldots, y_{0,\ldots,n})$.

(ii) Write up a function `Horner` with the following arguments:

- an array of abscissae $x_0, \ldots, x_n$ that have been previously checked to be pairwise different;
- an array of terms $A_0, \ldots, A_n$,
- and a variable floating-point number x,

and returning the output $A_0 + (x-x_0)(A_1 + (x-x_1)(A_2 + \ldots))$ in the manner described in (5).

(iii) Combine Exercises (i) and (ii) above to interpolate a given set of points and compute $p_n(x)$ for any given x. Think of possible ways to minimise the number of operations used.

2. And these constitute an example of how to apply Exercise 1:

(i) Create a text file `table.txt` and fill it with pairs of numbers distributed in two columns:

```
x0  y0
x1  y1
 ⋮   ⋮
```

make sure `x0, x1,...` are all different (10)

(ii) Write a function `read_from_file` with two arguments: a file `name` (in string form) and a tolerance `tol`. The function will read two-column data such as (10) externally from file `name` and will check that there are no abscissae `x0, x1,...` (in any order) differing from each other by less than `tol` in absolute value.

(iii) Write a function `write_to_files` one of whose arguments must be the number of points of the interpolating polynomial you wish to plot. It will write these points, again in the same disposition as (10), in a new file called `function_to_plot.txt`. It will also store Newton's divided differences in another file named `divided_differences.txt`.

All that is needed for you to upload to Moodle is:

- One Python file `UPXXXXXX.py` containing your student number.
- One Jupyter library showcasing applications of (and any necessary comments about) your Python file.
- One file `table.txt` containing a sample table to interpolate.
- One file `function_to_plot.txt` with a larger (> 1000) set of points interpolating those in `table.txt`.
- One file `divided_differences.txt` containing the divided differences for the data in `table.txt`.
- A plot of the data in `function_to_plot.txt` (you can skip this if your Jupyter file contains plots).

INTRODUCTION TO COMPUTATIONAL PHYSICS

Second Assessment for Teaching Block 1

U24200 –Academic Session 2019–2020

Instructions

a) The proportion of marks for this coursework

b) **You must undertake this assignment individually.**

c) Submission method: **by Moodle through the available dropbox**.

d) Submission deadline: **August 7, 2020**

1 Reminder about interpolation (already seen in Coursework 1)

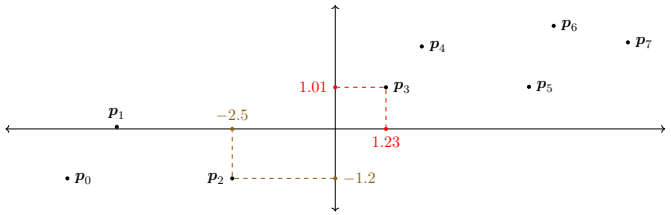
Interpolation consists in building a simple function f (here, a *polynomial*, $f(x) = a_0 + a_1 x + \cdots + a_k x^k$) whose graph curve passes through a given set of points

$$\boldsymbol{p}_0 = (x_0, y_0)\,, \quad \boldsymbol{p}_1 = (x_1, y_1)\,, \quad \ldots\,, \quad \boldsymbol{p}_m = (x_m, y_m)\,,$$

i.e. such that $f(x_0) = y_0$, $f(x_1) = y_1$, $\ldots$, $f(x_m) = y_m$. This can be seen, for instance in the case in which

$$\begin{array}{llll} \boldsymbol{p}_0 = (-6.5, -1.2)\,, & \boldsymbol{p}_1 = (-5.3, 0.05)\,, & \boldsymbol{p}_2 = (-2.5, -1.2)\,, & \boldsymbol{p}_3 = (1.23, 1.01)\,, \\ \boldsymbol{p}_4 = (2.1, 2)\,, & \boldsymbol{p}_5 = (4.7, 1.02)\,, & \boldsymbol{p}_6 = (5.3, 2.5)\,, & \boldsymbol{p}_7 = (7.1, 2.1)\,. \end{array}$$

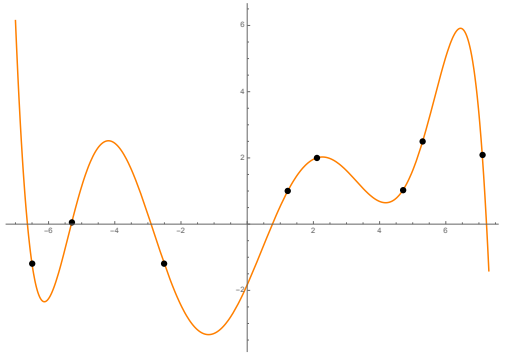
Let us first represent these points graphically:



We wish to find a polynomial whose graph traverses each one of these points. After you have finished your program, you will find that such a function can be approximated as

$$f(x) = -0.000175066x^7 + 0.000289133x^6 + 0.014541x^5 - 0.0211649x^4 - 0.34218x^3 + 0.477095x^2 + 2.24967x - 1.83489,$$

and has the shape shown in the next page. We draw it along with the original points $\boldsymbol{p}_i$.



The following result is fundamental to our purpose:

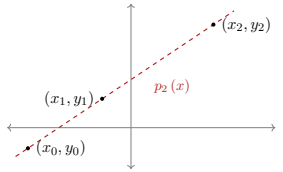
Theorem (Existence and uniqueness of the interpolating polynomial). *Let $(x_0, y_0), \ldots, (x_n, y_n)$ be $n+1$ points in the plane such that x_i are pairwise different ($x_i \neq x_j$ if $i \neq j$). Then there exists a unique polynomial of degree* at most *n, $p_n(x) = a_0 + a_1 x + \cdots + a_n x^n$, interpolating these points, i.e. such that*

$$p(x_0) = y_1, \quad p(x_1) = y_2, \quad \ldots, \quad p(x_n) = y_n. \tag{1}$$

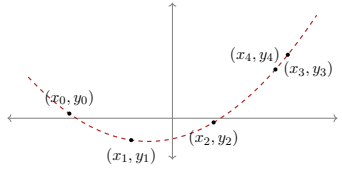
Remarks.

1. A well-known fact in Geometry, namely that any two points are traversed simultaneously by a unique line, is a particular case of the above: a line is the graph of a degree-one polynomial $y = ax + b$, and using the above two notation we would have $n = 1$ for two points $(x_0, y_0), (x_1, y_1)$.

2. n is the *maximum* value (hence an upper bound) of the degree of the interpolating polynomial but the actual degree could be less than n depending on the disposition of the points. For instance,



Three points hence $n = 2$ but they are *aligned*, thus interpolating polynomial is that line: $p(x) = a + bx + \boxed{0}\,x^2$



Five points ($n = 4$) but they are *all in the same parabola*, thus interpolating polynomial is that parabola: $p_4(x) = A + Bx + Cx^2 + D\boxed{0}\,x^3 + E\boxed{0}\,x^4$

3. Interpolation (and a similar concept called *extrapolation*) will be useful whenever you are given a table of experimental data and need to guess theoretical outputs for values not belonging to that table.

There are many ways of computing the interpolating polynomial p_n of $n+1$ points (but remember: the polynomial, due to its uniqueness, *is still the same* and depends only on the points chosen). Most notably:

- solving the linear system defined by the interpolating conditions
- method of Lagrange polynomials;
- Newton's method of divided differences;
- Aitken's method, Neville's method, etc.

In Coursework 1 we saw the third method. We will now focus on the first method for this Coursework.

2 Solving a linear system

The polynomial $p_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ must verify (1), i.e.

$$\begin{aligned} a_n x_0^n + a_{n-1}x_0^{n-1} + \cdots + a_1 x_0 + a_0 &= y_0, && (2)\\ a_n x_1^n + a_{n-1}x_1^{n-1} + \cdots + a_1 x_1 + a_0 &= y_1, && (3)\\ &\vdots && (4)\\ a_n x_n^n + a_{n-1}x_n^{n-1} + \cdots + a_1 x_n + a_0 &= y_n, && (5) \end{aligned}$$

these are $n+1$ linear equations with $n+1$ unknowns $a_0, \ldots, a_n$. It can be proved that the matrix of this linear system, called a **Vandermonde matrix**,

$$V = \begin{pmatrix} x_0^n & x_0^{n-1} & \ldots & x_0 & 1 \\ x_1^n & x_1^{n-1} & \ldots & x_1 & 1 \\ x_2^n & x_2^{n-1} & \ldots & x_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_n^n & x_n^{n-1} & \ldots & x_n & 1 \end{pmatrix} \qquad (6)$$

has a determinant $\neq 0$, thus the linear system defined by (2), (3), ..., (5) has a unique solution $(a_n, a_{n-1}, \ldots, a_0)$. These are the coefficients of the polynomial we are looking for.

Sometimes you will also see the matrix written the other way,

$$\begin{pmatrix} 1 & x_0 & \ldots & x_0^{n-1} & x_0^n \\ 1 & x_1 & \ldots & x_1^{n-1} & x_1^n \\ 1 & x_2 & \ldots & x_2^{n-1} & x_2^n \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_n & \ldots & x_n^{n-1} & x_n^n \end{pmatrix} \qquad (7)$$

in which case the solution will be written in the opposite order: $(a_0, a_1, \ldots, a_{n-1}, a_n)$. This is just as correct as if you used (6).

3 Coursework exercises

1. These will be the functions used to compute p_n:

(i) Implement a function `Cramer` to solve linear systems with an invertible matrix. If you check through the Moodle material for TB1 you will find this, but make sure you write it in your own style (don't just copy it) and test it on separate systems before adapting it to your coursework.

(ii) Write up a function `VandermondeSolution` with the following arguments:

- an array of abscissae $\boldsymbol{x} = (x_0, \ldots, x_n)$ that have been previously checked to be pairwise different;
- an array of ordinates $\boldsymbol{y} = (y_0, \ldots, y_n)$ comprising the other half of the table we wish to interpolate.

and returning matrix V and the solution $\boldsymbol{a}$ of the system $V\boldsymbol{a} = \boldsymbol{y}$, where

- V is the Vandermonde matrix for $x_0, \ldots, x_n$. Feel free to choose either version: (6) or (7).
- $\boldsymbol{a} = (a_n, \ldots, a_0)$ or $\boldsymbol{a} = (a_0, \ldots, a_n)$ is the array of coefficients of the desired $a_n x^n + \cdots + a_0$.

(iii) Write up a function `Horner` with the following arguments:

- an array of terms $a_0, \ldots, a_n$,
- and a variable floating-point number x,

and returning the output $a_0 + x(a_1 + x(a_2 + \ldots))$ where you minimise the number of operations used. You can adapt the generalised Horner's method seen in Coursework 1, only that in this simpler case you will multiply each new block by x instead of by the successive $x - x_{n-1}$, $x - x_{n-2}$, etc.

(i), (ii), (iii) above are enough to interpolate a given set of points and compute $p_n(x)$ for any given x.

2. And these constitute an example of how to apply Exercise 1:

(i) Create a text file `table.txt` and fill it with pairs of numbers distributed in two columns:

$$\left.\begin{array}{cc} \texttt{x0} & \texttt{y0} \\ \texttt{x1} & \texttt{y1} \\ \vdots & \vdots \end{array}\right\} \quad \text{make sure } \texttt{x0, x1,...} \text{ are all different} \qquad (8)$$

(ii) Write a function `read_from_file` with two arguments: a file `name` (in string form) and a tolerance `tol`. The function will read two-column data such as (8) externally from file `name` and will check that there are no abscissae `x0`, `x1`, ... (in any order) differing from each other by less than `tol` in absolute value.

(iii) Write a function `write_to_files` one of whose arguments must be the number of points of the interpolating polynomial you wish to plot. It will write these points, again in the same disposition as (8), in a new file called `function_to_plot.txt`. It will also store the Vandermonde matrix, ordinates $y_0, \ldots, y_n$ and coefficients of your polynomial in another file named `matrix_and_vectors.txt`.

All that is needed for you to upload to Moodle is:

- One Python file `UPXXXXXX.py` containing your student number.
- One Jupyter library showcasing applications of (and any necessary comments about) your Python file.
- One file `table.txt` containing a sample table $\boxed{x_i \mid y_i}$ to interpolate.
- One file `matrix_and_vectors.txt` containing the Vandermonde matrix V, the ordinates $\boldsymbol{y}$ and the solution $\boldsymbol{a} = (a_n, \ldots, a_0)$ to the system $V\boldsymbol{a} = \boldsymbol{y}$, i.e. the coefficients of p_n.
- One file `function_to_plot.txt` with a larger (> 1000) set of points interpolating those in `table.txt`.
- A plot of the data in `function_to_plot.txt` (you can skip this if your Jupyter file contains plots).

Mark scheme: if your program

- does none of the following: run without errors, produce a plot or a numerical output: **0-50 marks**
- fails to do *at least* one of the things described above: **30-60 marks**
- runs without errors and produces *one* plot, but lacks what is said in the items below: **50-70 marks**
- same as item above and the interpolating process is correct for arbitrary degrees: **60-90 marks**
- same as item above and a good Jupyter notebook is present: **80-100 marks**

Mark ranges overlap because the global layout of your program and your ability to detect glitches will also count.

INTRODUCTION TO COMPUTATIONAL PHYSICS

First Coursework

U24200 –Academic Session 2020–2021

Instructions

a) This is worth **50%** of your total mark for this unit.

b) **You must undertake this assignment individually.**

c) Submission method: **by Moodle through the available dropbox**.

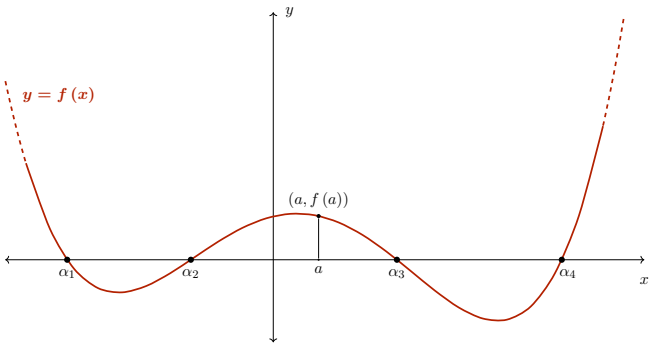
d) Submission deadline: **January 22, 2020, 4PM**

1 Introduction

Let $f(x)$ be a continuous function of a single variable. It can be:

- a simple function, e.g. a polynomial $f(x) = a_n x^n + \cdots + a_1 x + a_0$ (for instance $f(x) = x - 1$, $f(x) = 3x^2 + 4x - 5$, etc)
- or a more complicated function such as $f(x) = \cos x$, $f(x) = e^x$, ...

Having one real input x and one real output $y = f(x)$ for every input, this function can be represented by a two-dimensional graph plotting *abscissae* x against *ordinates* y:

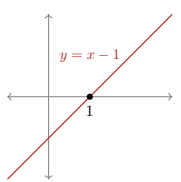


We have highlighted four values $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ in the above example. They are values of x such that $f(x) = 0$, i.e. the graph of f intersects the x-axis. We usually call such values **zeros** (or **roots**) of the function f, or **roots** (or simply *solutions*) of the equation $f(x) = 0$.

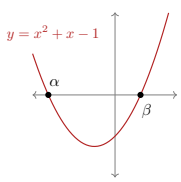
Two situations can arise, depending on the function $f(x)$ you are working with:

- sometimes finding or describing zeros of a function *symbolically* will be an easy task, e.g.
  - $f(x) = x - 1$ has only one root $\alpha = 1$,
  - $f(x) = x^2 + x - 1$ has two roots $\alpha = \frac{-\sqrt{5}-1}{2}$, $\beta = \frac{\sqrt{5}-1}{2}$ that are easy to find exactly using the *quadratic formula*,
  - $f(x) = \cos(x)$ has infinitely many roots but we know them: $\frac{(2k+1)\pi}{2}$ (k any integer),
  - and $f(x) = e^x$ has no zeros at all, because it is strictly positive on all values of x;

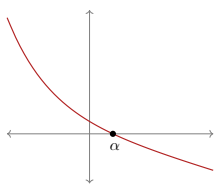
  by *symbolically* we mean roots we can represent exactly because we do not require decimal approximations to write them down (even if we may need approximations to *operate* with them in real-life scenarios, using $\sqrt{5} \simeq 2.23607\ldots$ and $\pi \simeq 3.14159\ldots$);
- most oftentimes, however, we may determine the *amount* of roots – but finding one, let alone all, symbolically will be either difficult or impossible. For instance, a thorough study of the roots of $e^{-x} = x$ (i.e. the zeros of function $f(x) = e^{-x} - x$) reveals that there is only one such zero α and it lies in the interval $(0.5, 0.6)$. But we cannot describe α in closed form, unlike the above examples; we cannot write it as an integer, a multiple of a known constant (e.g. π), a combination of square roots or simple functions of known values, etc. The only thing we can do is approximate α with a fixed amount of correct significant digits, e.g. $\alpha \simeq 0.5671432904097838$ with 16 correct decimal digits and $\alpha \simeq 0.56714329040978387299996866221035$ with 32 correct decimal digits.



$f(x) = 0$ equates to $x = 1$



Formula $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$ applied to $ax^2 + bx + c$, $a = 1$, $b = 1$, $c = -1$



Function $f(x) = e^{-x} - x$ has only one zero α but we cannot find it exactly

Numerical solution of equations consists in approximating one or more roots of a given equation $f(x) = 0$ using certain *root-finding algorithms*. Some remarks are in order here:

- these algorithms will work regardless of whether or not we can find the roots symbolically, e.g. for $f(x) = x - 1$ we know the exact solution, $x = 1$, but we can also apply a root-finding algorithm if we want to; if implemented correctly, it will approximate

  $x \simeq 1.$ large amount of 0's other digits or $x \simeq 0.$ large amount of 9's other digits

  and the better the approximation, the more 0's or 9's after the decimal point.
- As said above, most functions will not afford us the luxury of a symbolic solution, and numerical root-finding algorithms to approximate it will be the only tools available.
- Before applying the root-finding algorithm, we need to know the *precision*, i.e. the number of correct digits that we desire, e.g. 16 in the first approximation of α for $e^{-x} = x$ above.

Most root-finding algorithms are **iterative** methods: they entail the creation of a sequence of numbers $\{x_k\}_{k\geq 0}$ such, that every new iteration (i.e. element of the sequence) x_k can be obtained as a fixed function of a fixed number of previous iterations:

$$\boxed{x_k = G\left(x_{k-1}, x_{k-2}, \ldots, x_{k-j}\right)} \qquad G \text{ and } j \text{ being fixed and depending on the method.} \quad (1)$$

This requires, as you will realise yourselves, a set of j initial conditions $x_0, \ldots, x_{j-1}$.

Assume you want to fix a precision; more specifically, assume you wish to approximate a root α with p correct decimal digits. You can do so by fixing a certain tolerance $\varepsilon = \frac{1}{2}10^{-p}$, which is what you will use as a criterion to stop your iteration process. Let us represent a few steps of this method:

$$\begin{aligned}
\boldsymbol{x_j} &= G\left(x_{j-1}, x_{j-2}, \ldots, x_0\right) \quad (1^{\text{st}} \text{ actual step of the method, using only initial conditions})\\
\boldsymbol{x_{j+1}} &= G\left(\boldsymbol{x_j}, x_{j-1}, \ldots, x_1\right) \quad (2^{\text{nd}} \text{ step, using previous one } x_j \text{ and some initial conditions})\\
\boldsymbol{x_{j+2}} &= G\left(\boldsymbol{x_{j+1}}, \boldsymbol{x_j}, x_{j-1}, \ldots, x_2\right) \quad (3^{\text{rd}} \text{ step})\\
&\vdots \quad \vdots \quad \vdots\\
\boldsymbol{x_m} &= G\left(\boldsymbol{x_{m-1}}, \boldsymbol{x_m}, \ldots, \boldsymbol{x_{m-j}}\right) \quad (m^{\text{th}} \text{ step})\\
&\vdots \quad \vdots \quad \vdots
\end{aligned}$$

The process will stop at a step n where two consecutive iterations differ by less than ε:

$$|\boldsymbol{x_n} - \boldsymbol{x_{n-1}}| < \frac{1}{2}10^{-p}, \qquad \text{i.e. } \boldsymbol{x_n} \text{ and } \boldsymbol{x_{n-1}} \text{ share their first } p \text{ decimal digits.} \quad (2)$$

Obviously, the challenge here is finding a function G, depending on the original function f, such that the sequence generated above converges to the root we are looking for: $\lim_{n\to\infty} \boldsymbol{x_n} = \alpha$. Once G is established (and we will not explain the mathematical reasoning behind this now), **condition (2) ensures, in most cases, that the last iteration $\boldsymbol{x_n}$ in our method will share p correct decimal digits with α as well**.

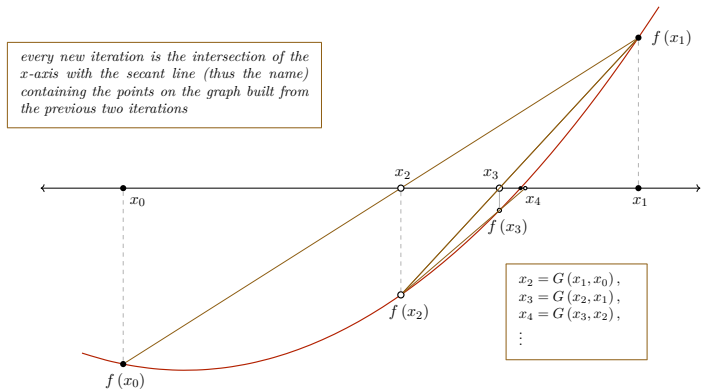
There are many examples of iterative methods, i.e. choices of G:

- the **secant method**, which may or may not converge depending on initial conditions;
- **Newton's method**, which may or may not converge but if it does, is usually faster;
- the **bisection method**, which converges more often but does so extremely slowly;
- other methods, such as Steffensen's method, fixed-point iteration, Halley's method, etc.

We will see the first three methods.

2 The secant method

This is a two-point iterative method, i.e. $j = 2$ meaning every iteration is a function of the previous two. Assume we have chosen two initial conditions x_0, x_1, chosen close to the root α that we are after:



x_4 is already fairly close to (but still distinguishable from) the root. It is reasonable to assume that $x_5, x_6, \ldots$ will be even closer, and proceeding in this manner we will obtain an x_k sharing the desired number of decimal digits with α. Simple geometry shows that the intersection of line $\overline{(x_{n-1}, f(x_{n-1}))\ (x_{n-2}, f(x_{n-2}))}$ with the x-axis is $G(x_{n-1}, x_{n-2}) = x_{n-1} - f(x_{n-1})\frac{x_{n-1}-x_{n-2}}{f(x_{n-1})-f(x_{n-2})}$, meaning the secant method has the following form:

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}, \quad n \geq 2, \quad \text{having chosen initial conditions } x_0, x_1 \qquad (3)$$

Example. Assume we want to find the largest root of $f(x) = x^2 - 1$. We know how to find it symbolically (it is $= 1$), hence we do not need a numerical method – but that does not mean we cannot apply one. First let us express G in terms of f:

$$G(x_{n-1}, x_{n-2}) = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} = x_{n-1} - \frac{\left(x_{n-1}^2 - 1\right)(x_{n-1} - x_{n-2})}{x_{n-1}^2 - x_{n-2}^2} = \frac{x_{n-2}x_{n-1} + 1}{x_{n-2} + x_{n-1}}$$

Thus our iteration method will be the following. We are using pen and calculator instead of Python here, so we can afford simplifying G:

$$x_n = \frac{x_{n-2}x_{n-1} + 1}{x_{n-2} + x_{n-1}}, \qquad n \geq 2.$$

It being a two-point method, we need two initial conditions. This is up to the method user. **Bear in mind that convergence (as well as convergence speed) of the method may depend on an adequate choice of these conditions**. Choosing unwisely can result in convergence to the wrong root (-1), slower convergence to 1, or no convergence at all.

For example, choose $x_0 = 0, x_1 = 2$. Let us fix precision $\varepsilon = \frac{1}{2}10^{-16}$ and start our process:

$$
\begin{aligned}
x_0 &= 0 \quad \text{(initial condition chosen by us)}\\
x_1 &= 2 \quad \text{(initial condition chosen by us)}\\
x_2 &= G(x_1,x_0) = \frac{x_0x_1+1}{x_0+x_1} = \frac{0\cdot 2+1}{0+2} = \frac{1}{2}\\
x_3 &= G(x_2,x_1) = \frac{x_1x_2+1}{x_1+x_2} = \frac{2\cdot\frac{1}{2}+1}{2+\frac{1}{2}} = \frac{4}{5} = 0.8\\
x_4 &= G(x_3,x_2) = \frac{x_2x_3+1}{x_2+x_3} = \frac{\frac{1}{2}\cdot\frac{4}{5}+1}{\frac{1}{2}+\frac{4}{5}} = \frac{14}{13} \simeq 1.0769230769230769231\\
x_5 &= G(x_4,x_3) = \frac{x_3x_4+1}{x_3+x_4} = \frac{\frac{4}{5}\cdot\frac{14}{13}+1}{\frac{4}{5}+\frac{14}{13}} = \frac{121}{122} \simeq 0.99180327868852459016\\
x_6 &= G(x_5,x_4) = \frac{3280}{3281} \simeq 0.99969521487351417251\\
x_7 &= G(x_6,x_5) = \frac{797162}{797161} \simeq 1.0000012544517355967\\
x_8 &= G(x_7,x_6) = \frac{5230176601}{5230176602} \simeq 0.99999999980880186730\\
x_9 &= G(x_8,x_7) = \frac{8338590849833284}{8338590849833285} \simeq 0.9999999999999998801\\
x_{10} &= G(x_9,x_8) = \frac{872246055045600895355852 54}{872246055045600895355852 53} \simeq 1.000000000000000000000000001146.\\
x_{11} &= G(x_{10},x_9) = \frac{145466059468128540431523291324612122334 0241}{145466059468128540431523291324612122334 0242} \simeq 0.\boxed{42\text{ digits} = 9}3125544
\end{aligned}
$$

We have $|x_{11} - x_{10}| < \frac{1}{2}10^{-16}$, thus x_{11} is our last iteration and the process can stop. Our approximation for the root is $x_{11} = 0.999\cdots$, and if we round it up to 16 digits we get 1.

Needless to say, choosing other initial conditions will change the numbers (and potentially, the number of iterations needed) in the above process. Exercise: try this yourselves, e.g. $x_0 = 1/2$ and $x_1 = 3/2$.

Example. Let us try $f(x) = e^{-x} - x$ shown in page 2. Unlike the previous example, we now do not have a simple representation of root α. We know (after looking at its graph) that it lies between 0.5 and 0.6, thus these two can be the initial conditions. Our iteration function will be

$$
\begin{aligned}
G(x_{n-1},x_{n-2}) &= x_{n-1} - f(x_{n-1})\frac{x_{n-1}-x_{n-2}}{f(x_{n-1})-f(x_{n-2})} = x_{n-1} - \frac{(e^{-x_{n-1}}-x_{n-1})(x_{n-1}-x_{n-2})}{x_{n-2}-e^{-x_{n-2}}+e^{-x_{n-1}}-x_{n-1}}\\
&= \frac{e^{x_{n-2}}x_{n-2} - e^{x_{n-1}}x_{n-1}}{e^{x_{n-1}}\left(e^{x_{n-2}}(x_{n-2}-x_{n-1})-1\right)+e^{x_{n-2}}}
\end{aligned}
$$

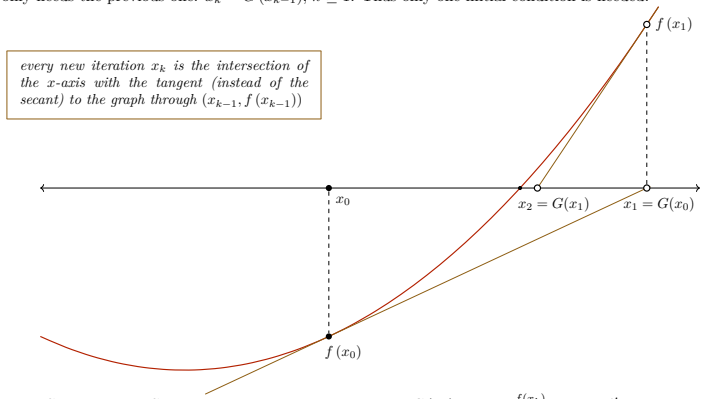
thus our process will be:

$$
\begin{aligned}
x_2 &= G(x_1,x_0) = G(0.6,0.5) = \frac{e^{0.5}0.5 - e^{0.6}0.6}{e^{0.6}\left(e^{0.5}(0.5-0.6)-1\right)+e^{0.5}} \simeq 0.5675445848373013947\\
x_3 &= G(x_2,x_1) = \frac{e^{x_1}x_1 - e^{x_2}x_2}{e^{x_2}\left(e^{x_1}(x_1-x_2)-1\right)+e^{x_1}} \simeq 0.5671409166735748153,\\
x_4 &= G(x_3,x_2) = \frac{e^{x_2}x_2 - e^{x_3}x_3}{e^{x_3}\left(e^{x_2}(x_2-x_3)-1\right)+e^{x_2}} \simeq 0.5671432905821386311\\
&\vdots\ \ \vdots\ \ \vdots
\end{aligned}
$$

until $x_7 \simeq 0.567143290409783873$ which satisfies $|x_7 - x_6| < \frac{1}{2}10^{-16}$, thus we have our approximation 0.567143290409783873 of the root with 16 correct decimal digits.

3 Newton's Method

Also called **Newton-Raphson method**, it is a one-point algorithm, i.e. $j = 1$: every iteration only needs the previous one: $x_k = G(x_{k-1})$, $k \geq 1$. Thus only one initial condition is needed:



Basic Geometry and Calculus yield an explicit expression: $G(x_k) = x_k - \frac{f(x_k)}{f'(x_k)}$, where f' is the **derivative** of f. Thus our method will be

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad n \geq 1, \quad \text{having chosen initial condition } x_0 \tag{4}$$

This method usually requires less iterations than secant to approximate the root with the same precision. Same as with secant, convergence to the desired root is not guaranteed here. Try to think of situations leading to lack of convergence.

Example. Return to $f(x) = e^{-x} - x$. $f'(x) = -e^{-x} - 1$, thus our iteration function will be

$$G(x) = x - \frac{f(x)}{f'(x)} = x - \frac{e^{-x}-x}{-e^{-x}-1} = \frac{x+1}{e^x+1},$$

Choose initial condition $x_0 = 1$ and the first iterations of (4) become:

$$
\begin{aligned}
x_1 &= G(x_0) = x_0 - \frac{f(x_0)}{f'(x_0)} = \frac{2}{1+e} \simeq 0.53788284273999024150\\
x_2 &= G(x_1) = x_1 - \frac{f(x_1)}{f'(x_1)} \simeq 0.5669869914054132388\\
x_3 &= G(x_2) = x_2 - \frac{f(x_2)}{f'(x_2)} \simeq 0.5671432859891229440\\
x_4 &= G(x_3) = x_3 - \frac{f(x_3)}{f'(x_3)} \simeq 0.56714329040978 3869\\
x_5 &= G(x_4) = x_4 - \frac{f(x_4)}{f'(x_4)} \simeq 0.56714329040978 3873;
\end{aligned}
$$

iteration x_5 shares at least 16 digits with x_4 ($|x_5 - x_4| < \frac{1}{2}10^{-16}$), thus the mathematical discussion we have omitted here implies that it also shares those digits with the actual root. Thus we can stop here and 0.5671432904097838 is our approximation. Exercise: try an initial condition x_0 closer to the root, say 0.55, and observe the algorithm converge even faster.

4 The bisection method

This is the most intuitive method and has the advantage of converging regardless of our choice of initial conditions *if the zero entails a change in sign*, as well as not requiring derivatives or complicated manipulation of our function $f(x)$ – but is also the slowest method available.

The idea behind bisection is simple: if a root α lies between a and b (i.e. inside the interval $[a,b]$) and we divide, or *bisect*, this interval into two subintervals of equal length, then α belongs to either the left half $\left[a, \frac{a+b}{2}\right]$, or the right half $\left[\frac{a+b}{2}, b\right]$. Once we know which of the two subintervals it belongs to, we apply the exact same argument to that subinterval: dividing it into two, and discerning which half contains α. Each subinterval will be $1/2$ the size of the previous one, thus we will be progressively cornering α in smaller intervals until we have an interval whose length is smaller than the precision tolerance $\frac{1}{2}10^{-p}$.

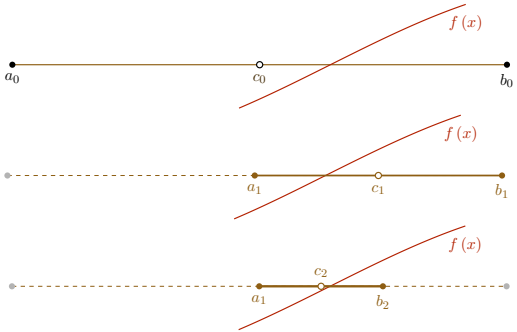
Let us describe the iteration. We need two initial conditions a_0, b_0 (extremes of the initial interval containing α) and then proceed as follows for $n = 0, 1, \ldots$, until $|a_n - b_n| < \frac{1}{2}10^{-p}$.

What is done in iteration n:

- At this stage we have $[a_n, b_n]$ containing α. Compute the midpoint $c_n = \frac{a_n+b_n}{2}$.

  Option 1: If $f(a_n)\, f(c_n) < 0$ that means $\alpha \in [a_n, c_n]$. Start iteration $n+1$, replacing $[a_n, b_n]$ by the smaller interval $[a_n, c_n]$, and repeat the process.

  Option 2: If $f(b_n)\, f(c_n) < 0$ the right half-interval $[c_n, b_n]$ contains α. Start iteration $n+1$ by defining $[a_{n+1}, b_{n+1}]$ to be this interval $[c_n, b_n]$, and repeat the process.



Needless to say, iteration $n = 3$ in the above example would arise from taking $a_3 = c_2$ and $b_3 = b_2$.

5 The Actual Coursework

Write a Python program to approximate the roots of any function. It must contain, at least, the following functions.

1. A routine `initial_plot` plotting the graph of any $f(x)$, then giving the user the option to change the range interval and create new plots, or stop plotting altogether. You will use this function to zoom in or out until you have spotted a root of f, thus giving you good candidates for initial condition(s) close to it before you implement the methods in 2 below.

2. Functions `Secant`, `Newton`, `Bisection` for approximating roots numerically. Each of these functions should have at least the following parameters passed to them as inputs:

- the function $f(x)$ itself and, in one of the three methods, its derivative $f'(x)$;
- the initial condition(s) `x0`,... required by the given method;
- the tolerance `tol` determining the precision with which you wish to approximate the root.

Your functions must write initial conditions and all iterations in an external file. Think of ways in which to fill out this external file with at least two columns, so that:

- one of the columns corresponds to the index $k = 0, \ldots$ determining the iteration;
- and another one for the iteration x_k itself.

Make sure you write this in a way that makes it readable from another Python routine.

3. Function `read_from_file` reading an external file and returning the column corresponding to the iterations of whatever method (`Secant`, `Newton` or `Bisection`) filled the file originally. If the file is not arrayed in proper columns, `read_from_file` should detect this and notify the user.

4. A routine `final_plot` doing the exact same thing as `final_plot`, and in addition having slightly larger, properly labelled dots for some of the iterations, along with captions containing all the relevant information.

Let `XXXXXX` be your student number. All that is needed for you to upload to Moodle is:

- One Python file `UPXXXXXX_MAIN.py` containing the main body of your program.
- One Python file `UPXXXXXX_METHODS.py` containing your methods and, perhaps, other routines.
- Optional `UPXXXXXX_FUNCTIONS.py` if you wish to define your different $f(x)\,, f'(x)$ in a separate file.
- One Jupyter notebook showcasing applications of, and comments about, your Python project.
- A few sample `.txt` files containing iterations of all three methods applied to functions.
- A few initial plots, showing how you located and zoomed into roots of at least one function (you can skip this if your Jupyter file contains plots)
- A final plot obtained from `final_plot`. Again, you can skip this if your Jupyter file contains plots.

Mark ranges: if your program

- does none of these: run without errors, produce a plot or a numerical output: **0-45 marks**
- fails to do *at least* one of the things described above: **30-60 marks**
- runs without errors and produces *one* plot, but lacks what is said below: **45-70 marks**
- same as item above and the process is correct for arbitrary $f(x)$: **60-90 marks**
- same as item above and a good Jupyter notebook is present: **80-100 marks**

Mark ranges overlap because the global layout of your program and your ability to detect glitches will also count.

INTRODUCTION TO COMPUTATIONAL PHYSICS

Referral/Deferral – Python Section

U24200 –Academic Session 2020–2021

Instructions

a) **You must undertake this assignment individually.**

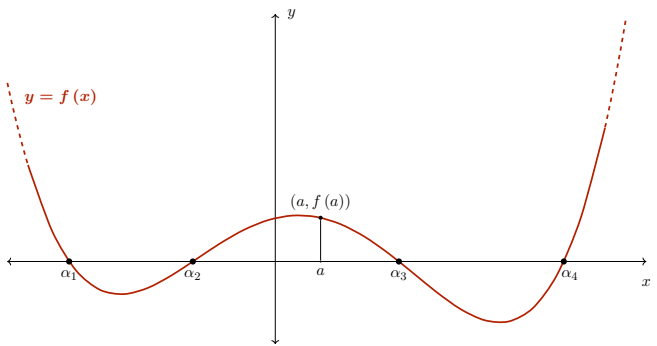
b) Submission method: **by Moodle through the available dropbox**.

1 Introduction

Let $f(x)$ be a continuous function of a single variable. It can be:

- a simple function, e.g. a polynomial $f(x) = a_n x^n + \cdots + a_1 x + a_0$ (for instance $f(x) = x - 1$, $f(x) = 3x^2 + 4x - 5$, etc)
- or a more complicated function such as $f(x) = \cos x$, $f(x) = e^x$, ...

Having one real input x and one real output $y = f(x)$ for every input, this function can be represented by a two-dimensional graph plotting *abscissae* x against *ordinates* y:

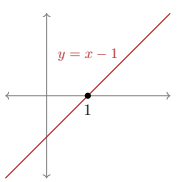


We have highlighted four values $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ in the above example. They are values of x such that $f(x) = 0$, i.e. the graph of f intersects the x-axis. We usually call such values **zeros** (or **roots**) of the function f, or **roots** (or simply *solutions*) of the equation $f(x) = 0$.

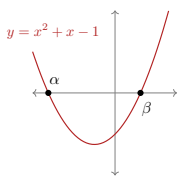
Two situations can arise, depending on the function $f(x)$ you are working with:

- sometimes finding or describing zeros of a function *symbolically* will be an easy task, e.g.
  - $f(x) = x - 1$ has only one root $\alpha = 1$,
  - $f(x) = x^2 + x - 1$ has two roots $\alpha = \frac{-\sqrt{5}-1}{2}$, $\beta = \frac{\sqrt{5}-1}{2}$ that are easy to find exactly using the *quadratic formula*,
  - $f(x) = \cos(x)$ has infinitely many roots but we know them: $\frac{(2k+1)\pi}{2}$ (k any integer),
  - and $f(x) = e^x$ has no zeros at all, because it is strictly positive on all values of x;

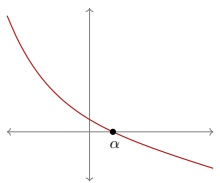
  by *symbolically* we mean roots we can represent exactly because we do not require decimal approximations to write them down (even if we may need approximations to *operate* with them in real-life scenarios, using $\sqrt{5} \simeq 2.23607\ldots$ and $\pi \simeq 3.14159\ldots$);
- most oftentimes, however, we may determine the *amount* of roots – but finding one, let alone all, symbolically will be either difficult or impossible. For instance, a thorough study of the roots of $e^{-x} = x$ (i.e. the zeros of function $f(x) = e^{-x} - x$) reveals that there is only one such zero α and it lies in the interval $(0.5, 0.6)$. But we cannot describe α in closed form, unlike the above examples; we cannot write it as an integer, a multiple of a known constant (e.g. π), a combination of square roots or simple functions of known values, etc. The only thing we can do is approximate α with a fixed amount of correct significant digits, e.g. $\alpha \simeq 0.5671432904097838$ with 16 correct decimal digits and $\alpha \simeq 0.567143290409783872999968662210 35$ with 32 correct decimal digits.



$f(x) = 0$ equates to $x = 1$



Formula $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$ applied to $ax^2 + bx + c$, $a = 1$, $b = 1$, $c = -1$



Function $f(x) = e^{-x} - x$ has only one zero α but we cannot find it exactly

Numerical solution of equations consists in approximating one or more roots of a given equation $f(x) = 0$ using certain *root-finding algorithms*. Some remarks are in order here:

- these algorithms will work regardless of whether or not we can find the roots symbolically, e.g. for $f(x) = x - 1$ we know the exact solution, $x = 1$, but we can also apply a root-finding algorithm if we want to; if implemented correctly, it will approximate

  $x \simeq 1.$ large amount of 0's other digits or $x \simeq 0.$ large amount of 9's other digits

  and the better the approximation, the more 0's or 9's after the decimal point.
- As said above, most functions will not afford us the luxury of a symbolic solution, and numerical root-finding algorithms to approximate it will be the only tools available.
- Before applying the root-finding algorithm, we need to know the *precision*, i.e. the number of correct digits that we desire, e.g. 16 in the first approximation of α for $e^{-x} = x$ above.

Most root-finding algorithms are **iterative** methods: they entail the creation of a sequence of numbers $\{x_k\}_{k\geq 0}$ such, that every new iteration (i.e. element of the sequence) x_k can be obtained as a fixed function of a fixed number of previous iterations:

$$\boxed{x_k = G\left(x_{k-1}, x_{k-2}, \ldots, x_{k-j}\right)} \qquad G \text{ and } j \text{ being fixed and depending on the method.} \quad (1)$$

This requires, as you will realise yourselves, a set of j initial conditions $x_0, \ldots, x_{j-1}$.

Assume you want to fix a precision; more specifically, assume you wish to approximate a root α with k correct decimal digits. You can do so by fixing a certain tolerance $\varepsilon = \frac{1}{2}10^{-k}$, which is what you will use as a criterion to stop your iteration process. The process (1) will stop at a step n where two consecutive iterations differ by less than ε:

$$|\boldsymbol{x_n} - \boldsymbol{x_{n-1}}| < \frac{1}{2}10^{-k}, \qquad \text{i.e. } \boldsymbol{x_n} \text{ and } \boldsymbol{x_{n-1}} \text{ share their first } k \text{ decimal digits.} \quad (2)$$

Obviously, the challenge here is finding a function G, depending on the original function f, such that the sequence generated above converges to the root we are looking for: $\lim_{n\to\infty} \boldsymbol{x_n} = \alpha$. Once G is established (and we will not explain the mathematical reasoning behind this now), **condition (2) ensures, in most cases, that the last iteration $\boldsymbol{x_n}$ in our method will share p correct decimal digits with α as well**.

There are many examples of iterative methods, i.e. choices of G:

- **secant**, **Newton** and **bisection**, all of which you saw during the first semester;
- other methods, such as Steffensen's method, fixed-point iteration, Halley's method, etc.

We will see a family of methods generalizing Newton.

2 Householder's methods

Denote $^{(p)}$ to be the p^{th} order derivative, e.g. $f^{(0)} = f, f^{(1)} = f', f^{(2)} = f'' = \frac{d^2 f}{dx^2}$, etc.

Householder methods are one-point algorithms, i.e. $j = 1$: every iteration only needs the previous one: $x_k = G\left(x_{k-1}\right)$, $k \geq 1$. Thus only one initial condition is needed. The general form of the p^{th} Householder method is:

$$x_{n+1} = x_n + p\frac{F^{(p-1)}\left(x_n\right)}{F^{(p)}\left(x_n\right)}, \qquad \text{where } F\left(x\right) := \frac{1}{f\left(x\right)}. \quad (3)$$

For example, if $p = 1$ Householder becomes **Newton's method** which you already know:

$$F^{(p-1)}\left(x\right) = F^{(0)}\left(x\right) = F\left(x\right) = \frac{1}{f\left(x\right)}, \qquad F^{(p)}\left(x\right) = F'\left(x\right) = \left(1/f\right)'\left(x\right) = -\frac{f'(x)}{f(x)^2},$$

using the quotient or the chain rule for derivation. Thus (3) becomes in this case

$$x_{n+1} = x_n + 1 \cdot \frac{\frac{1}{f(x_n)}}{-\frac{f'(x_n)}{f(x_n)^2}} = x_n - \frac{f\left(x_n\right)}{f'\left(x_n\right)},$$

which should be familiar to you from TB1 when you were programming Newton's method.

The **order of convergence** of the method, if it does converge, is $p+1$ (which is why Newton was generally considered to be quadratic, i.e. of order 2, hence the number of correct decimal places roughly doubled after every iteration).

Thus, for higher values of p, you obtain faster methods. The only drawback, of course, is that you need to make the additional effort to compute higher derivatives; you will realize this yourselves when you find the expression for Householder for values $p > 1$.

3 The Actual Coursework

Write a Python program to approximate the roots of any function. It must contain, at least, the following functions.

1. A routine `initial_plot` plotting the graph of any $f\left(x\right)$, then giving the user the option to zoom and create new plots, or stop plotting altogether. This is identical to the TB1 coursework.

2. A function `Householder` for approximating roots numerically using Householder methods. This function should have at least the following parameters passed to them as inputs:

- the value of p described above; **your program should be operational for at least one value $\boxed{>1}$ of p; you already know how to program Newton ($p = 1$) so that would entail 0 marks here**.
- the function $f\left(x\right)$ itself and its derivatives $f'\left(x\right), f''\left(x\right), ...$, upper bound depending on which values of p you choose; for instance, if you only choose $p = 3$ then your program should be able to use derivatives $f'\left(x\right), f''\left(x\right), f'''\left(x\right)$.
- the initial condition `x0` required by the given method;
- the tolerance `tol` determining the precision with which you wish to approximate the root.

Your function must write initial conditions and all iterations in external files (for different functions and different values of p), arrayed in two columns, so that:

- one of the columns corresponds to the index $k = 0, \ldots$ determining the iteration;
- and another one for the iteration x_k itself.

Make sure you write this in a way that makes it readable from another Python routine.

3. Function `read_from_file` reading an external file and returning the column corresponding to the iterations with which the method (`Householder`) filled the file originally. Identical to the same function described in the TB1 coursework.

4. A routine `final_plot` doing the exact same thing as `final_plot`, and in addition having slightly larger, properly labelled dots for some of the iterations, along with captions containing all the relevant information. Identical to the TB1 coursework.

Let `XXXXXX` be your student number. All that is needed for you to upload to Moodle is:

- One Python file `UPXXXXXX_MAIN.py` containing the main body of your program.
- One Python file `UPXXXXXX_METHODS.py` containing your methods and, perhaps, other routines.
- Optional `UPXXXXXX_FUNCTIONS.py` if you wish to define functions and derivatives in a separate file.
- A few sample `.txt` files containing iterations of some methods (i.e. values of p) applied to functions.
- A few initial plots, showing how you located and zoomed into roots of at least one function.
- A final plot obtained from `final_plot`.
- Optional: a Jupyter notebook if it makes any of the above easier.

Mark ranges: if your program

- does none of these: run without errors, produce a plot or a numerical output: **0-45 marks**
- fails to do *at least* one of the things described above: **30-60 marks**
- runs without errors and produces *one* plot, but lacks what is said below: **45-70 marks**
- same as item above and the process is correct for arbitrary $f(x)$: **60-100 marks**
- *two* Householder methods instead of one (e.g. $p = 2, 3$, $p = 2, 4$, etc): **10 bonus marks**

Mark ranges overlap because the global layout of your program and your ability to detect glitches will also count.