ABILITÀ INFORMATICHE

Link moodle: https://moodle2.units.it/course/view.php?id=14606

Python: Exceptions

Exceptions are runtime errors occurring during the execution of a program.

Exceptions are objects that an instruction returns when its execution does not run successfully.
Either they are somehow handled by the developer, or they halt the script execution.

Python: Exception handling

Exceptions are runtime errors occurring during the execution of a program.

Exceptions are objects that an instruction returns when its execution does not run successfully.
Either they are somehow handled by the developer, or they halt the script execution.

However, exceptions in Python can also point to something else (e.g. see the case of *StopIteration*)

It’s useful to catch exceptions as soon as they arise and decide how to continue with following program instructions.

The blocks **try / except** allow you to handle exceptions.

Python: Exception handling

The blocks **try / except** allow you to handle exceptions.

try allows you to test a block of code for errors.

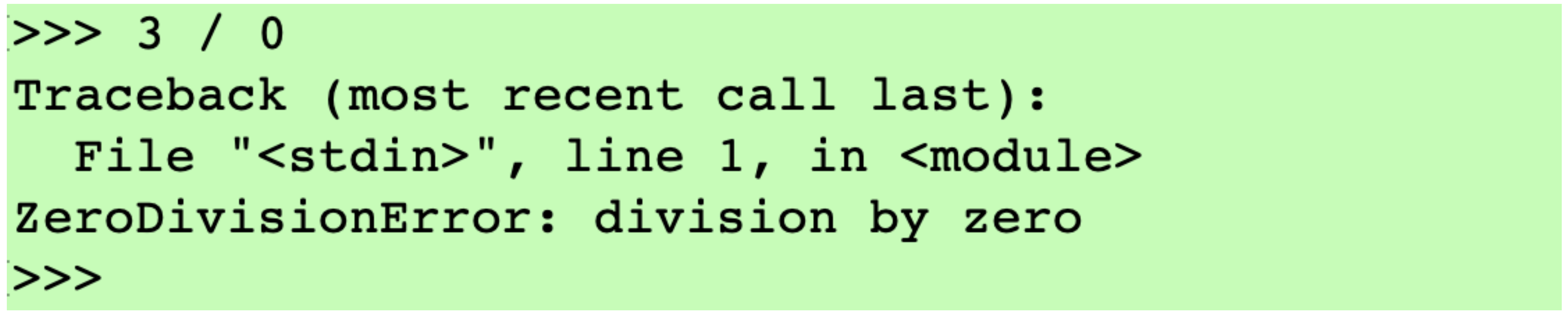
except lets you handle the error.

else lets you execute code when there is no error.

The **finally** block lets you execute code, regardless of the result of the try and except blocks.

Python: Exception handling

try allows you to test a block of code for errors

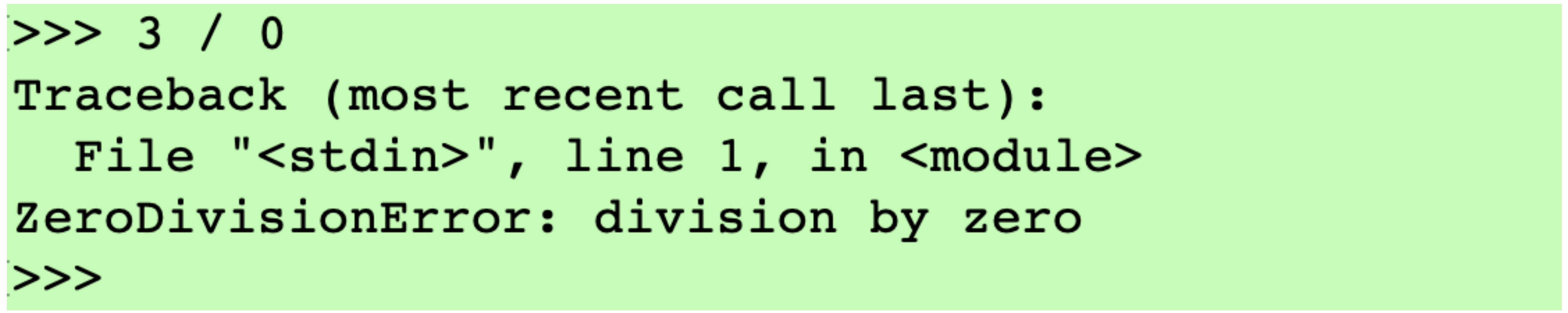


```
>>> 3 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

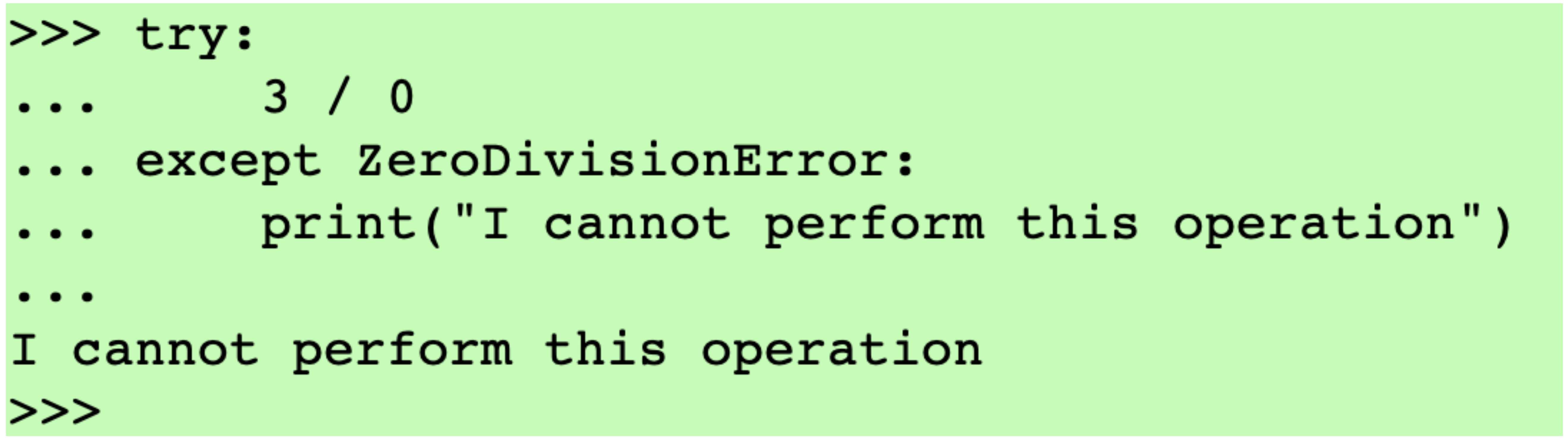
Python: Exception handling

try allows you to test a block of code for errors

except lets you handle the error



```
>>> 3 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

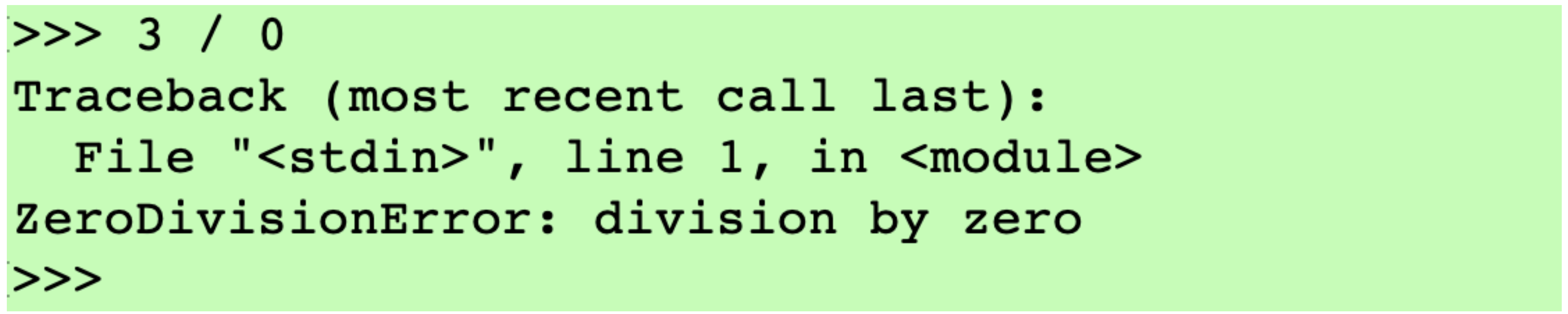


```
>>> try:
...     3 / 0
... except ZeroDivisionError:
...     print("I cannot perform this operation")
...
I cannot perform this operation
>>>
```

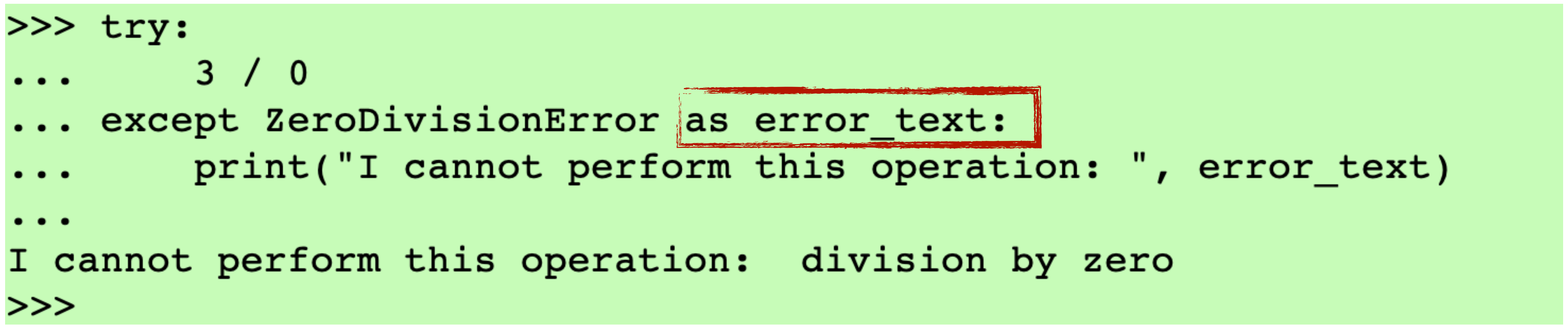
Python: Exception handling

try allows you to test a block of code for errors

except lets you handle the error



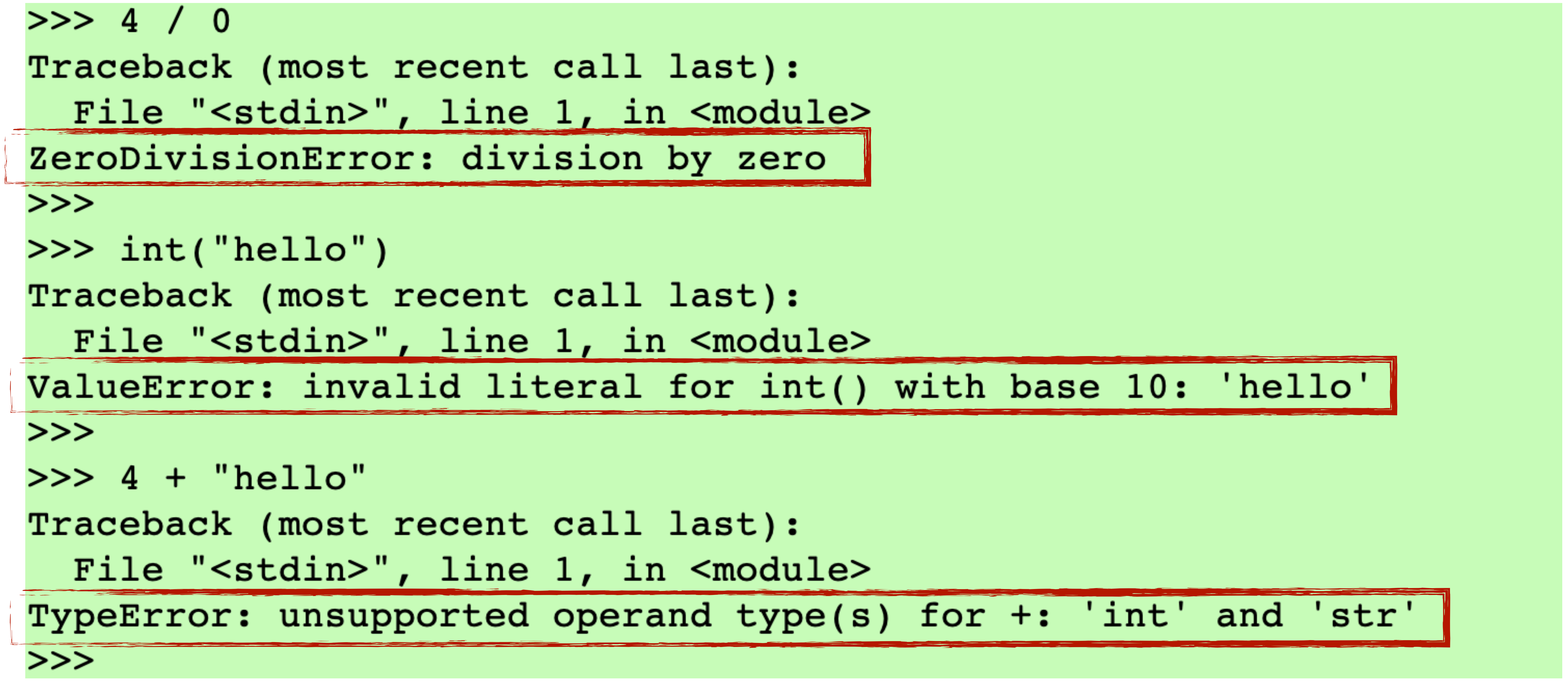
```
>>> 3 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```



```
>>> try:
...     3 / 0
... except ZeroDivisionError as error_text:
...     print("I cannot perform this operation: ", error_text)
...
I cannot perform this operation:  division by zero
>>>
```

Python: Exception handling

A single **try - except** block can deal with multiple operations and different errors



```
>>> 4 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
>>> int("hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hello'
>>>
>>> 4 + "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Python: Exception handling

A single **try - except** block can deal with multiple operations and different errors

```
>>> try:
...     4 / 0
...     int("hello")
...     4 + "hello"
... except ZeroDivisionError:
...     print("I cannot perform this operation")
... except ValueError:
...     print("I cannot cast a string into an int")
... except TypeError:
...     print("I cannot sum a string to an int")
...
I cannot perform this operation
>>>
```

Python: Exception handling

A single **try - except** block can deal with multiple operations and different errors

```
>>> try:
...     4 / 0
...     int("hello")
...     4 + "hello"
... except ZeroDivisionError:
...     print("I cannot perform this operation")
... except ValueError:
...     print("I cannot cast a string into an int")
... except TypeError:
...     print("I cannot sum a string to an int")
...
I cannot perform this operation
>>>
```

```
>>> try:
...     int("hello")
...     4 / 0
...     4 + "hello"
... except ZeroDivisionError:
...     print("I cannot perform this operation")
... except ValueError:
...     print("I cannot cast a string into an int")
... except TypeError:
...     print("I cannot sum a string to an int")
...
I cannot cast a string into an int
>>>
```

Python: Exception handling

Exceptions can be grouped in different (sub-)types.

The **except** block catches all the exceptions of a given type.

Main type is **Exception**. All the others stem from it.

```
>>> try:
...     6 / 0
... except Exception as error_text:
...     print("I cannot perform this operation: ", error_text)
...
I cannot perform this operation:  division by zero
>>>
```

ZeroDivisionError is a sub-type of **Exception**.

```
>>> try:
...     2 + "bye"
... except Exception as error_text:
...     print("I cannot perform division: ", error_text)
...
I cannot perform division:  unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Python: Exception handling

Exceptions can be grouped in different (sub-)types.

The **except** block catches all the exceptions. Main type is **Exception**. All the other types stem from it.

Exceptions should be rather captured from the least to the most specific.

```
>>> try:
...     6 / 0
... except ZeroDivisionError as error_text:
...     print("I cannot perform this operation: ", error_text)
... except Exception as error_text:
...     print("Issues due to: ", error_text)
...
I cannot perform this operation:  division by zero
>>>
>>> try:
...     2 + "bye"
... except ZeroDivisionError as error_text:
...     print("I cannot perform this operation: ", error_text)
... except Exception as error_text:
...     print("Issues due to: ", error_text)
...
Issues due to:  unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Python: Exception handling

Exceptions can be grouped in different (sub-)types.

The **except** block catches all the exceptions. Main type is **Exception**. All the other types stem from it.

```
>>> try:
...      int("hello")
...      4 / 0
...      4 + "hello"
... except (ZeroDivisionError, ValueError, TypeError) as error_text:
...     print("Issue: ", error_text)
...
Issue:  invalid literal for int() with base 10: 'hello'
>>>
>>> try:
...      4 + "hello"
... except (ZeroDivisionError, ValueError, TypeError) as error_text:
...     print("Issue: ", error_text)
...
Issue:  unsupported operand type(s) for +: 'int' and 'str'
>>>
>>> try:
...      4 / 0
... except (ZeroDivisionError, ValueError, TypeError) as error_text:
...     print("Issue: ", error_text)
...
Issue:  division by zero
```

A single **except** instruction can handle different types of exceptions: they have to be listed within a tuple.

Python: Exception handling

Besides **except**, the **try** block can be followed by **finally**.

Finally contains all the instructions which will be executed independently of the presence of exceptions.

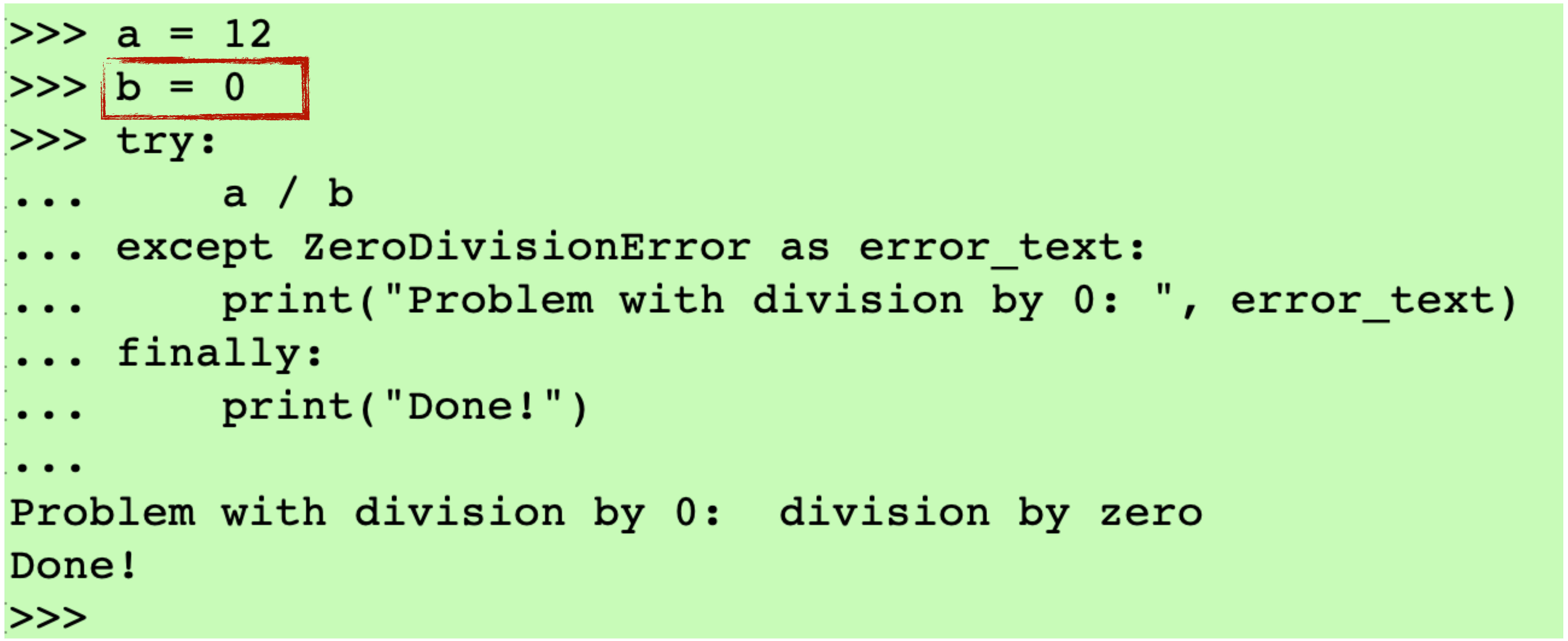
```
>>> a = 12
>>> b = 3
>>> try:
...     a / b
... except ZeroDivisionError as error_text:
...     print("Problem with division by 0: ", error_text)
... finally:
...     print("Done!")
...
4.0
Done!
```

Python: Exception handling

Besides **except**, the **try** block can be followed by **finally**.

Finally contains all the instructions which will be executed independently of the presence of exceptions.

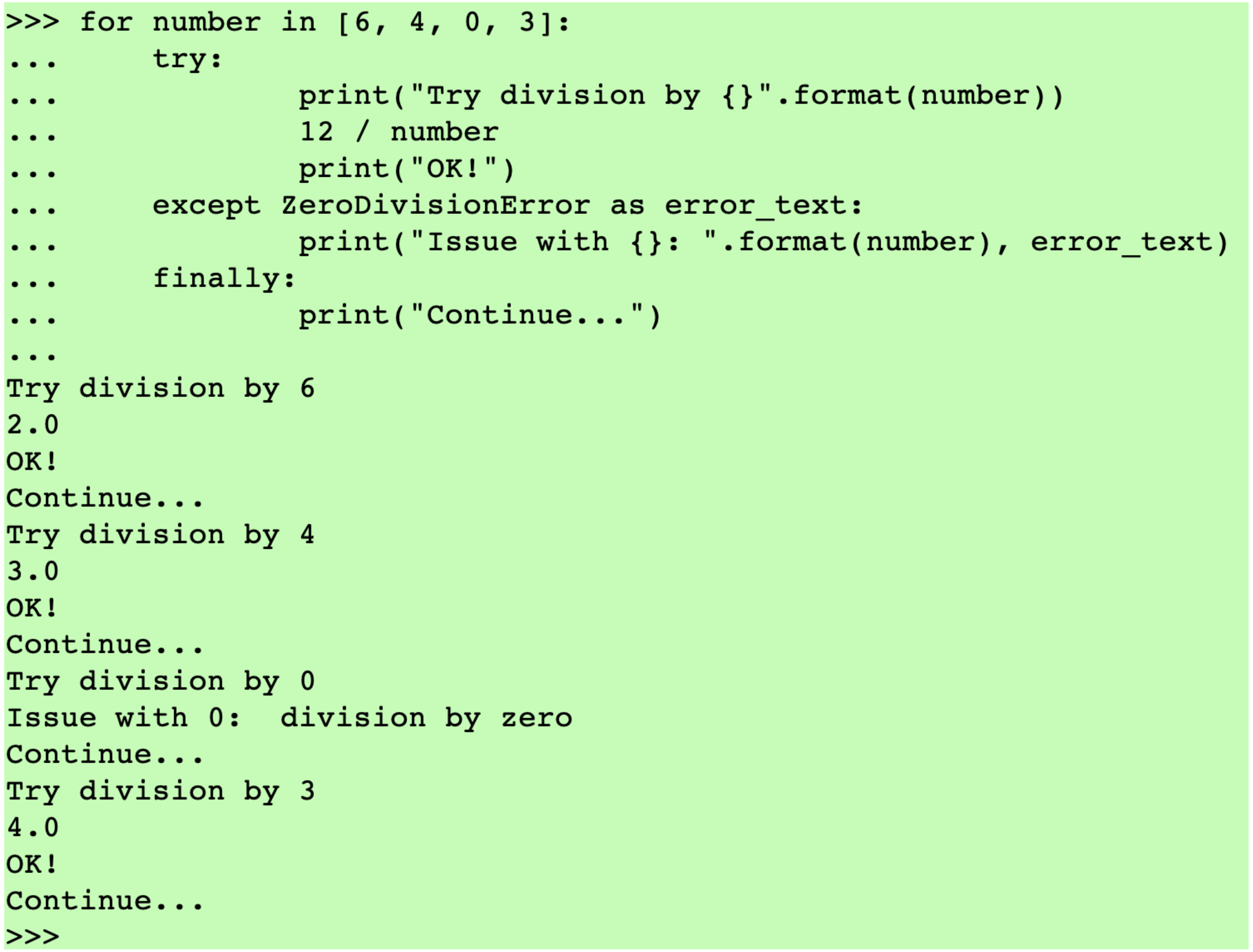
Finally: extremely useful to perform instructions even if errors/exceptions occur.



```
>>> a = 12
>>> b = 0
>>> try:
...     a / b
... except ZeroDivisionError as error_text:
...     print("Problem with division by 0: ", error_text)
... finally:
...     print("Done!")
...
Problem with division by 0:  division by zero
Done!
>>>
```

Python: Exception handling

Last block that **try** can execute before **finally** is **else**.



```
>>> for number in [6, 4, 0, 3]:
...     try:
...             print("Try division by {}".format(number))
...             12 / number
...             print("OK!")
...     except ZeroDivisionError as error_text:
...             print("Issue with {}: ".format(number), error_text)
...     finally:
...             print("Continue...")
...
Try division by 6
2.0
OK!
Continue...
Try division by 4
3.0
OK!
Continue...
Try division by 0
Issue with 0:  division by zero
Continue...
Try division by 3
4.0
OK!
Continue...
>>>
```

Python: Exception handling

Last block that **try** can execute before **finally** is **else**.

Else contains instructions which are computed only if exceptions did not occur.

```
>>> for number in [6, 0, 3]:
...     try:
...             print("Try division by {}".format(number))
...             12 / number
...             print("OK!")
...     except ZeroDivisionError as error_text:
...             print("Issue with {}: ".format(number), error_text)
...     else:
...             print("No issue")
...     finally:
...             print("Continue...")
...
Try division by 6
2.0
OK!
No issue
Continue...
Try division by 0
Issue with 0:  division by zero
Continue...
Try division by 3
4.0
OK!
No issue
Continue...
>>>
```

Python: exercise

https://github.com/MilenaValentini/TRMD_2024/blob/main/file1_Group_Pos_Data_100Mpc-N256-DM.txt

1. Plottare prima colonna VS ciascuna delle altre tre, e poi seconda VS terza e quarta, terza VS seconda e quarta, quarta VS seconda e terza.
2. Considerare la struttura più massiccia. Calcolare le distanze di ciascuna delle altre strutture dalla più massiccia, e plottare nel piano y=massa VS x=distanza.
3. Graficare l'istogramma che mostra la distribuzione delle masse degli aloni.
4. Ripetere 1., organizzando i plot in modo che contengano due panel (ad esempio tramite https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplot.html), così da vedere le proiezioni delle posizioni (ad esempio top panel mostra coordinata y VS x, mentre bottom panel mostra z VS x; o anche left panel che mostra y VS x e right panel che mostra y VS z). Gestire gli assi che i due panel di volta in volta condividono.
5. Ripetere 3., prevedendo una scala logaritmica per le masse delle strutture, e provando poi ad usare bin logaritmici (non equispaziati linearmente).

Python: exercise

Exercise.

a) — Create two arrays (for instance with numpy.arange).
The first one has 60 even numbers.
The second one has 60 odd numbers.
Plot them (e.g. scatter; x-array is array_even, y-array is array_odd) and use a label for them.

b) — On the same plot, overplot (with a different colour) a selection of the aforementioned arrays, where you only pick every other array entry (i.e., one entry yes, the following one no, and so on).

c) — In addition, use a function to compute the sum of the x-array in a) and its log_10, and square the result (i.e., **2), and do the same for y-array.
Plot the new vectors, always choosing different colours and label.
Use log scale if needed.

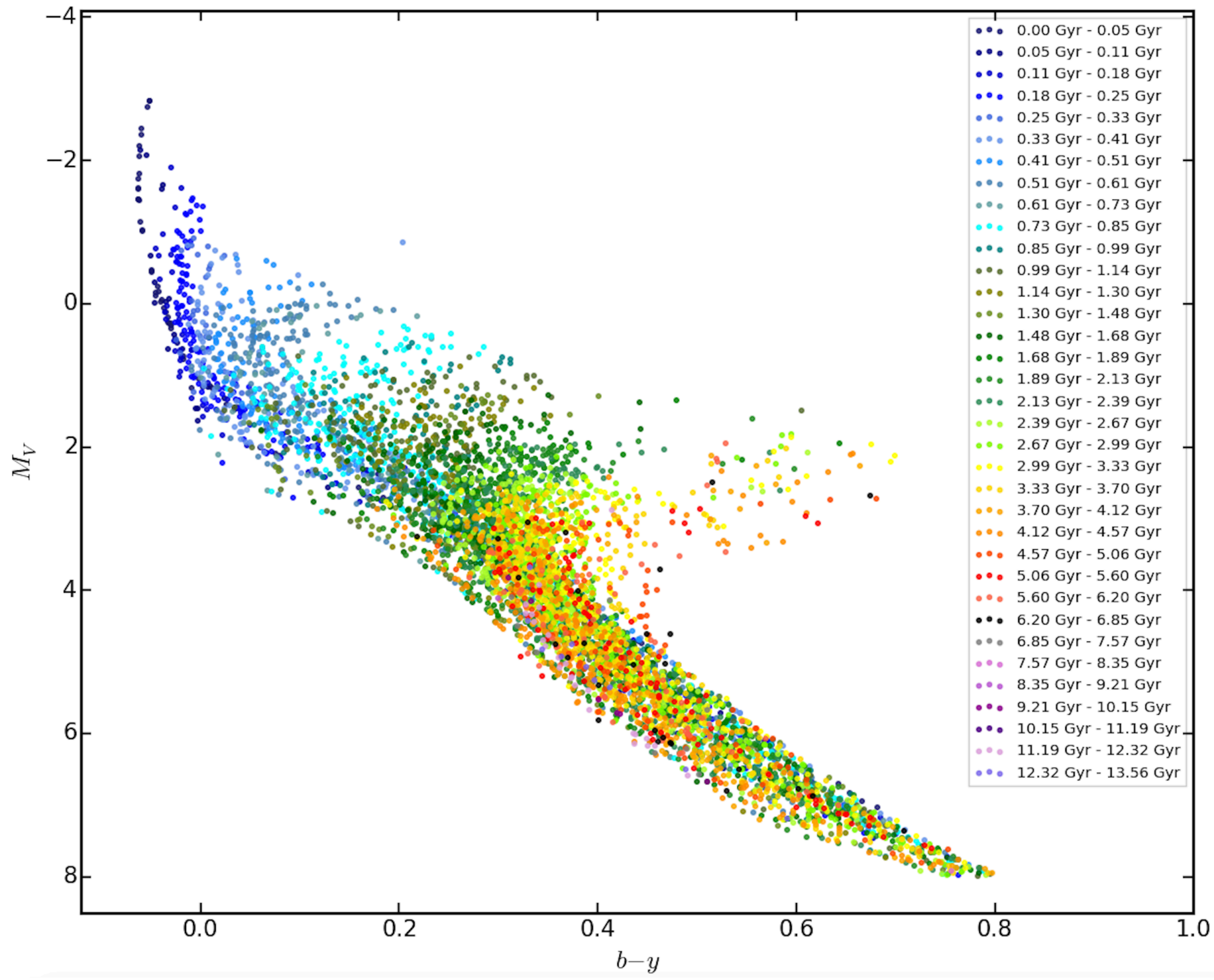
d) — Write all the plotted vectors in a file.

Python: exercises

https://github.com/MilenaValentini/TRM_Dati/blob/main/Nemo_6670.dat

Columns of interest in the file:

M_ass
b-y
age_parent [Gyr]



Python: exercises

https://github.com/MilenaValentini/TRM_Dati/blob/main/Nemo_6670.dat

Split the data in age bins (use np.where),
and use a for loop to (over)plot data in different age bins.

matplotlib.pyplot.figure(“my_figure”, figsize=(14,11))

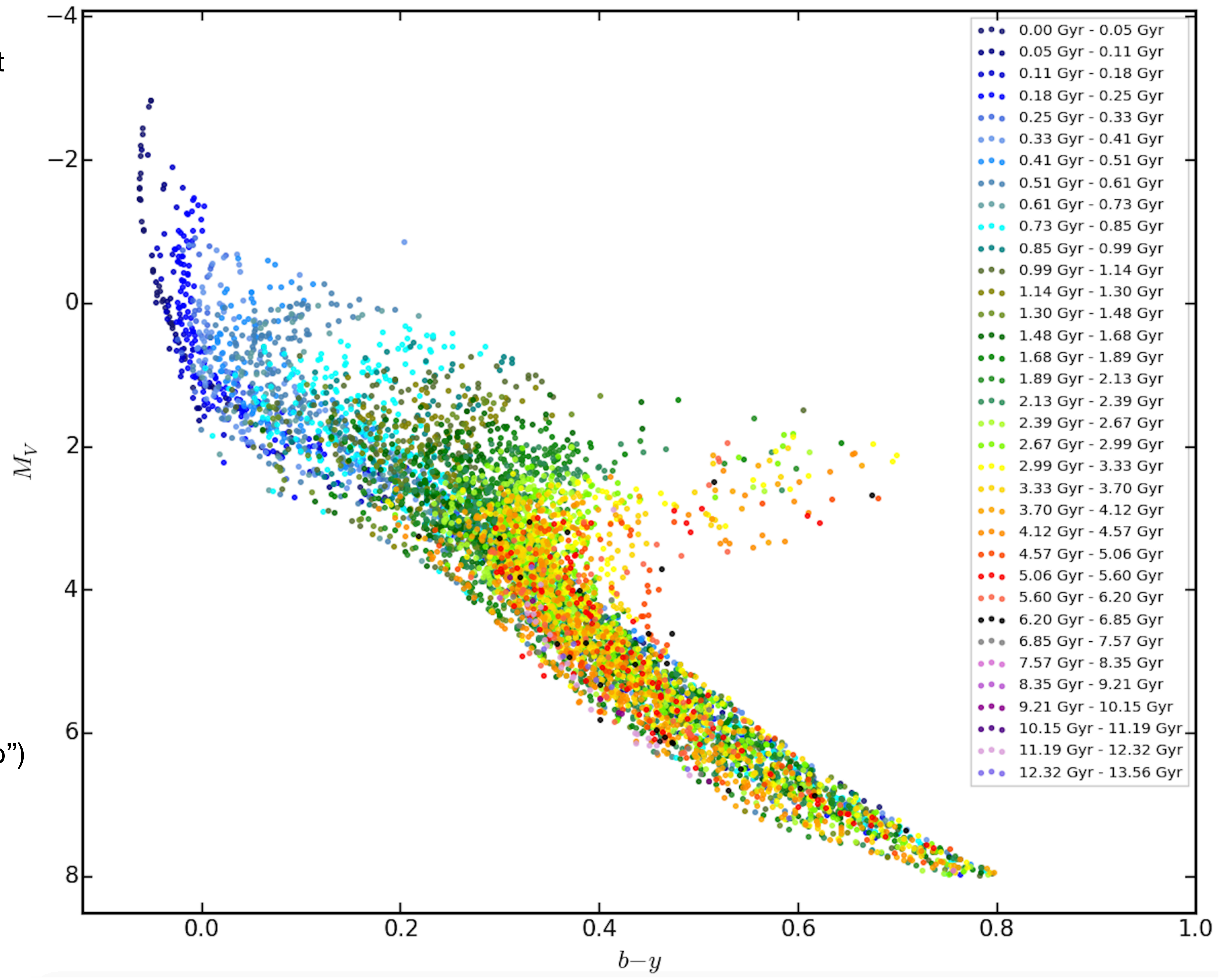
https://matplotlib.org/stable/gallery/color/named_colors.html

Useful for the scatterplot:

a = matplotlib.pyplot.scatter(array_x, array_y, c=array_z, cmap=“colormap”)

https://matplotlib.org/stable/gallery/color/colormap_reference.html

matplotlib.pyplot.colorbar(a)



Python: exercises

https://github.com/MilenaValentini/TRM_Dati/blob/main/Nemo_6670.dat

leggere le seguenti colonne:

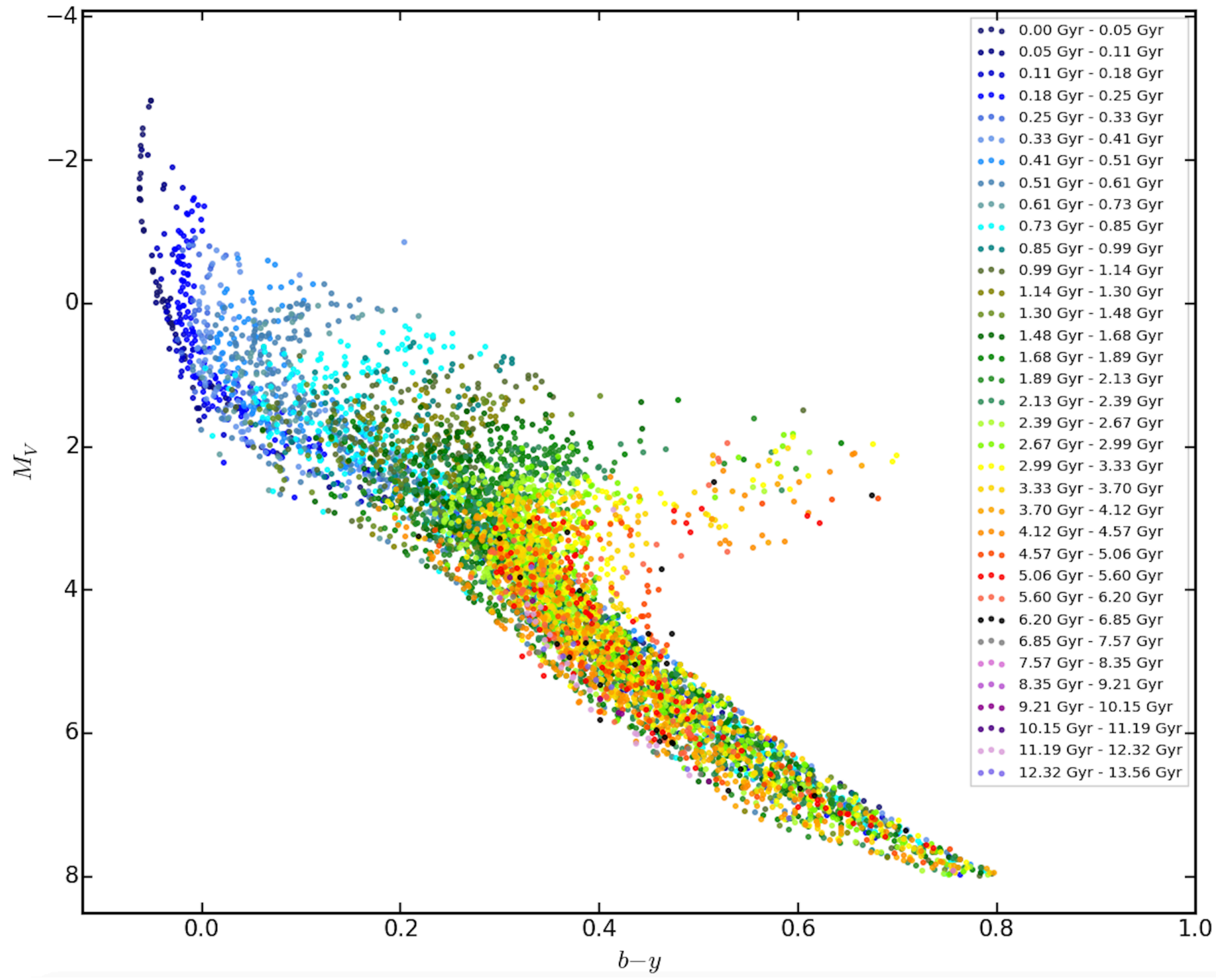
M_ass
b-y
age_parent [Gyr]

e produrre un grafico simile a quello riportato a destra.

In questo grafico si riporta M_ass in funzione di b-y (diagramma colore-magnitudine) per le 6670 stelle riportate nel file.
Il colore codifica l'età (age_parent) di stelle in un determinato bin di età e deve essere spiegato nella legenda.

Leggere poi anche la prima colonna del file (MsuH, cioè la metallicità delle stelle).
Dividere il campione di 6670 stelle in tre sotto-popolazioni a seconda della loro età (ad esempio per age_parent < 1 Gyr, compresa tra 1 e 5 Gyr, e > 5 Gyr) e studiare la distribuzione delle metallicità stellari per le stelle nei tre sotto-campioni (tramite tre istogrammi sullo stesso plot).
Identificare gli istogrammi con diversi colori e riportare una legenda sul grafico.
Calcolare anche media e mediana delle tre distribuzioni e riportarne con delle linee verticali i valori sul plot.

Leggere poi la seconda colonna (m_ini, cioè la massa iniziale) del file.
Graficare per le tre popolazioni individuate al punto precedente la metallicità in funzione della massa.
Riportare le tre distribuzioni sullo stesso grafico.
Opzionale: capire come ottimizzare la visualizzazione del contenuto del plot (ad esempio valutando diverse trasparenze dei simboli; o provando ad utilizzare contorni di densità per una o più distribuzioni ad esempio con l'istruzione matplotlib.pyplot.contour o pylab.contour; oppure provando a produrre un istogramma 2D ad esempio con numpy.histogram2d).


M_V
$b-y$
0.00 Gyr - 0.05 Gyr
0.05 Gyr - 0.11 Gyr
0.11 Gyr - 0.18 Gyr
0.18 Gyr - 0.25 Gyr
0.25 Gyr - 0.33 Gyr
0.33 Gyr - 0.41 Gyr
0.41 Gyr - 0.51 Gyr
0.51 Gyr - 0.61 Gyr
0.61 Gyr - 0.73 Gyr
0.73 Gyr - 0.85 Gyr
0.85 Gyr - 0.99 Gyr
0.99 Gyr - 1.14 Gyr
1.14 Gyr - 1.30 Gyr
1.30 Gyr - 1.48 Gyr
1.48 Gyr - 1.68 Gyr
1.68 Gyr - 1.89 Gyr
1.89 Gyr - 2.13 Gyr
2.13 Gyr - 2.39 Gyr
2.39 Gyr - 2.67 Gyr
2.67 Gyr - 2.99 Gyr
2.99 Gyr - 3.33 Gyr
3.33 Gyr - 3.70 Gyr
3.70 Gyr - 4.12 Gyr
4.12 Gyr - 4.57 Gyr
4.57 Gyr - 5.06 Gyr
5.06 Gyr - 5.60 Gyr
5.60 Gyr - 6.20 Gyr
6.20 Gyr - 6.85 Gyr
6.85 Gyr - 7.57 Gyr
7.57 Gyr - 8.35 Gyr
8.35 Gyr - 9.21 Gyr
9.21 Gyr - 10.15 Gyr
10.15 Gyr - 11.19 Gyr
11.19 Gyr - 12.32 Gyr
12.32 Gyr - 13.56 Gyr