Python Bootcamps 1, 2 and 3

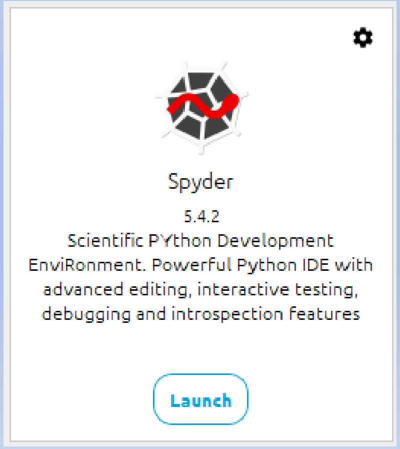
- 1: Getting up to speed with Python
- 2: Learning to use Python to solve typical problem scenarios
- 3: Detailed modeling of packed-bed and plug-flow reactors

Bootcamp 1 Outline

Installing the Integrated Development Environment (IDE)

There are several popular Python IDEs. These choices are beneficial, but they also present a problem. We have to choose one here. That will be the Spyder IDE because it is well suited to engineering and scientific computations, and it has an interface similar to MATLAB. Once we get into the details of Python, you can use any other IDE – you will just have to adapt to the Spyder illustrations used here.

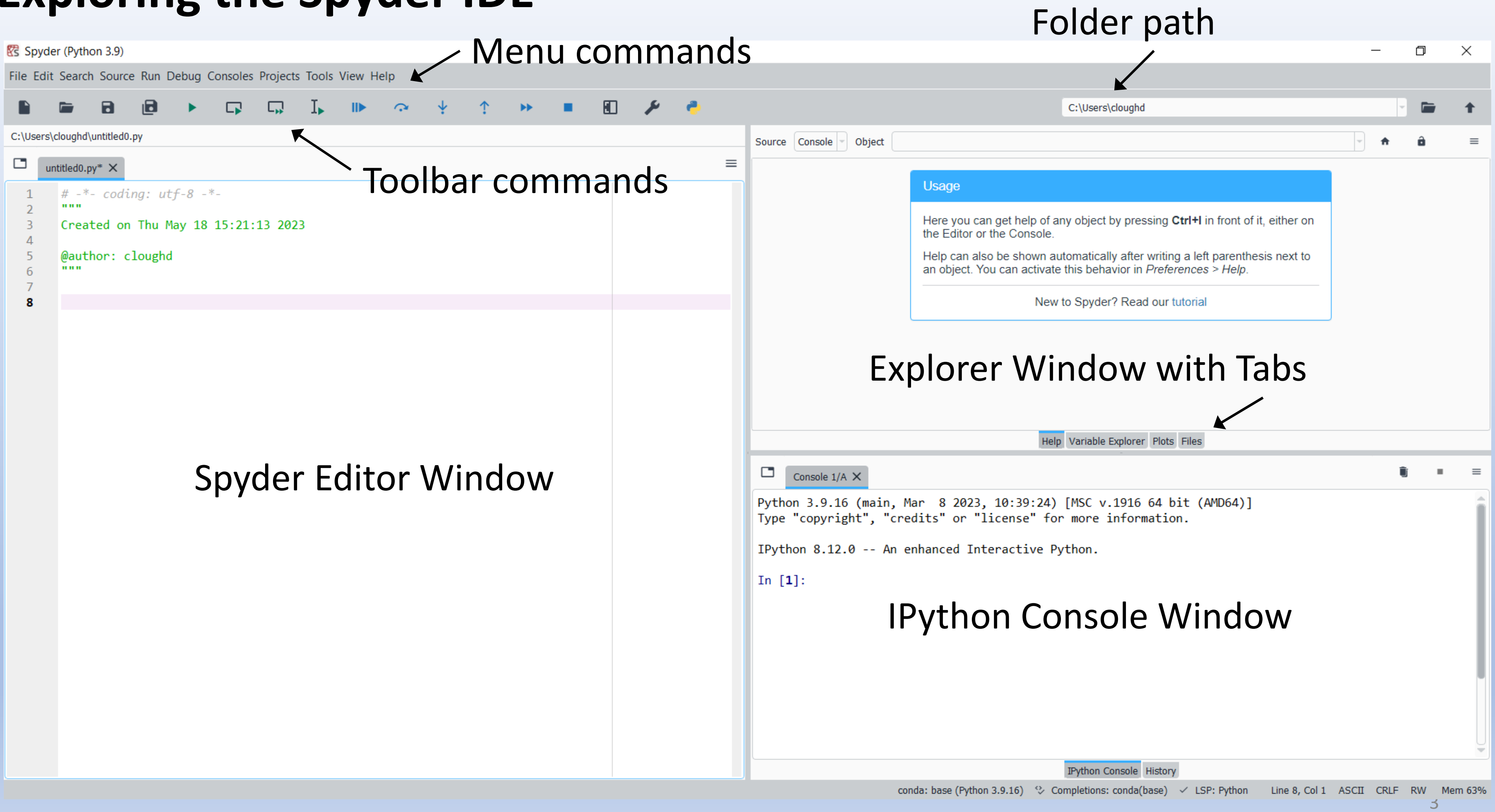
The Spyder IDE and corresponding Python programming language with common supporting modules are conveniently available at no cost by installing the Anaconda package. This is available via the URL: https//www.anaconda.com/distribution/ The Anaconda package includes numerous open-source software packages among them Spyder, which will appear with the icon


Spyder
5.4.2
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Once installed and launched, you can run Spyder without starting Anaconda and add its command icon to your display and/or taskbar.

Periodically, you can update the Spyder and Python versions via the Anaconda interface.

Exploring the Spyder IDE


Folder path
Menu commands
Toolbar commands
Explorer Window with Tabs
Spyder Editor Window
IPython Console Window
Spyder (Python 3.9)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\cloughd
C:\Users\cloughd\untitled0.py
untitled0.py*
Source Console Object
Help Variable Explorer Plots Files
Console 1/A
IPython Console History
conda: base (Python 3.9.16) Completions: conda(base) LSP: Python Line 8, Col 1 ASCII CRLF RW Mem 63%

```
# -*- coding: utf-8 -*-
"""
Created on Thu May 18 15:21:13 2023

@author: cloughd
"""

```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

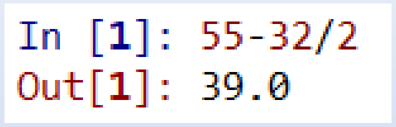
Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

New to Spyder? Read our tutorial

```
Python 3.9.16 (main, Mar  8 2023, 10:39:24) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 8.12.0 -- An enhanced Interactive Python.

In [1]:
```

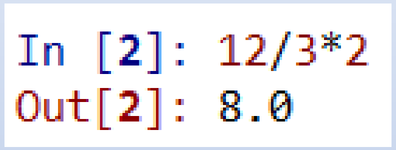
Using the IPython Console for Calculations

Examples



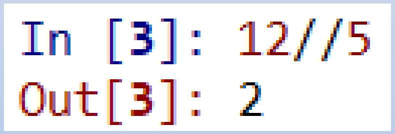
```
In [1]: 55-32/2
Out[1]: 39.0
```

/ takes place first



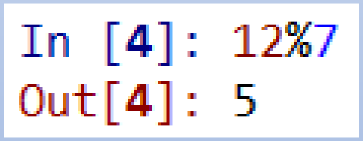
```
In [2]: 12/3*2
Out[2]: 8.0
```

left-to-right evaluation



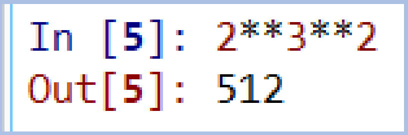
```
In [3]: 12//5
Out[3]: 2
```

integer division



```
In [4]: 12%7
Out[4]: 5
```

modulus (remainder)



```
In [5]: 2**3**2
Out[5]: 512
```

repeated ** right-to-left

↑ returns previous command

<u>Arithmetic Operators</u>

+	addition
–	subtraction and negation
*	multiplication
/	division (floating point)
//	division (integer)
%	modulus (remainder)
**	exponentiation

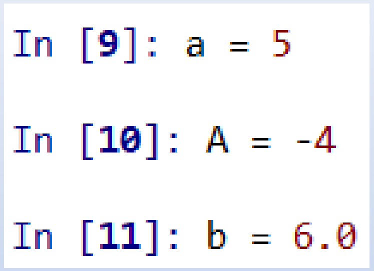
<u>Priority Order</u> (precedence)

**	highest
– (negation)	●
*, /, //, %	●
+, – (subtraction)	lowest

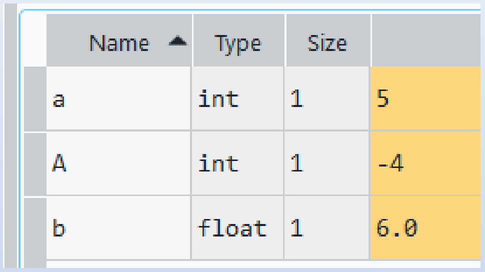
Evaluation left to right
except repeated exponentiation

Variables and Mathematical Functions

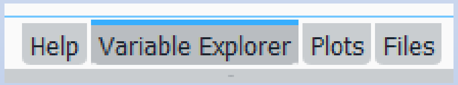
Assignment



```
In [9]: a = 5

In [10]: A = -4

In [11]: b = 6.0
```

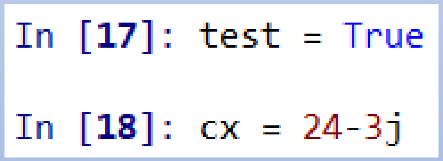


Name	Type	Size	
a	int	1	5
A	int	1	-4
b	float	1	6.0

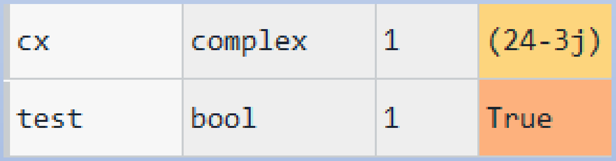


Notice a and A are different variables.
a and A are integer type (int)
b is floating point (float)

Other data types



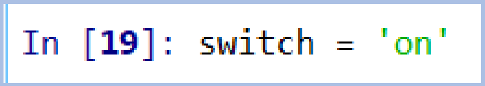
```
In [17]: test = True

In [18]: cx = 24-3j
```



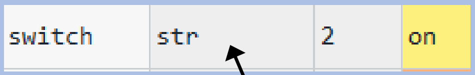
cx	complex	1	(24-3j)
test	bool	1	True

complex type

Boolean or T/F type



```
In [19]: switch = 'on'
```

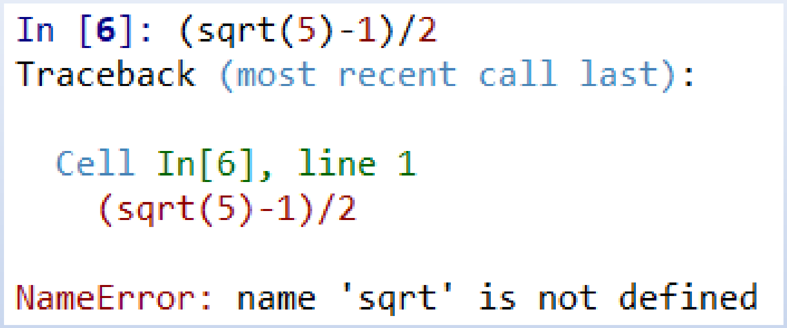


switch	str	2	on

character (or string) type

Variables and Mathematical Functions

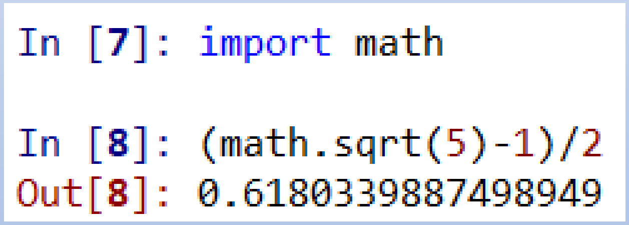
Built-in Functions



```
In [6]: (sqrt(5)-1)/2
Traceback (most recent call last):

  Cell In[6], line 1
    (sqrt(5)-1)/2

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

Uh, oh!
No sqrt function

Very few math functions directly available in Python: abs(•) and round(•)



```
In [7]: import math

In [8]: (math.sqrt(5)-1)/2
Out[8]: 0.6180339887498949
```

import the **math** module
for commonly used math functions
notice format: math.name(•)

The **math** module functions only operate on single quantities, not arrays. Later, we will use the **numpy** module that has similar functions that do operate on arrays.

math module includes

sin, cos, tan
asin, acos, atan
exp, log

sinh, cosh, tanh
asinh, acosh, atanh

<u>built-in constants</u>

pi   inf
e   nan

all referenced with math.name

Relational and Logical Operators

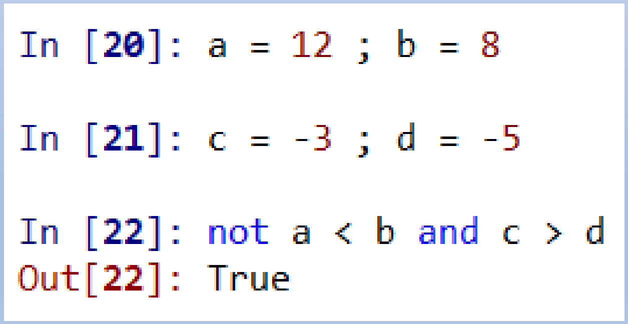
Relational Operators

==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Logical Operators
(from highest to lowest precedence)

not	logical negation
and	logical and
or	logical or

Example



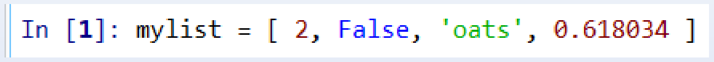
```
In [20]: a = 12 ; b = 8

In [21]: c = -3 ; d = -5

In [22]: not a < b and c > d
Out[22]: True
```

$\Rightarrow$ not False and True $\Rightarrow$ True and True

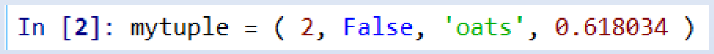
not applied before and

Collections of Data

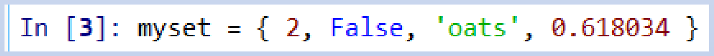
List	collection of various data types, [...]
Tuple	an immutable list (cannot be extended, shrunk, have elements removed or reassigned), (...)
Set	an unordered collection of unique objects, {...}
Dictionary	a collection of objects, each identified by a key, {value pairs}
Array	collection of a single data type indexed by integer subscripts, provided by the NumPy module



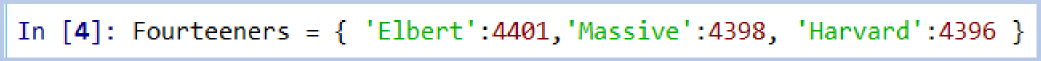
```
In [1]: mylist = [ 2, False, 'oats', 0.618034 ]
```



```
In [2]: mytuple = ( 2, False, 'oats', 0.618034 )
```



```
In [3]: myset = { 2, False, 'oats', 0.618034 }
```

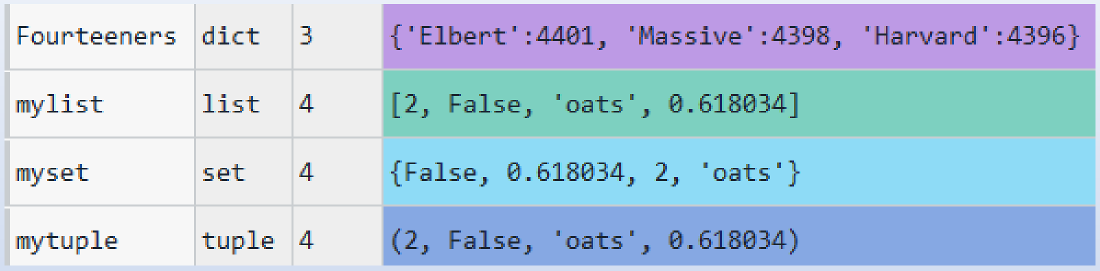


```
In [4]: Fourteeners = { 'Elbert':4401,'Massive':4398, 'Harvard':4396 }
```

See examples on following slides. Used extensively in numerical methods and applied statistics.

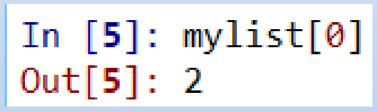
Collections of Data

Variable Explorer

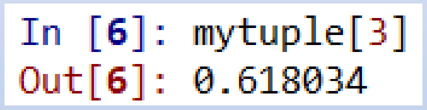


Fourteeners	dict	3	{'Elbert':4401, 'Massive':4398, 'Harvard':4396}
mylist	list	4	[2, False, 'oats', 0.618034]
myset	set	4	{False, 0.618034, 2, 'oats'}
mytuple	tuple	4	(2, False, 'oats', 0.618034)

Indexing
[...]



```
In [5]: mylist[0]
Out[5]: 2
```

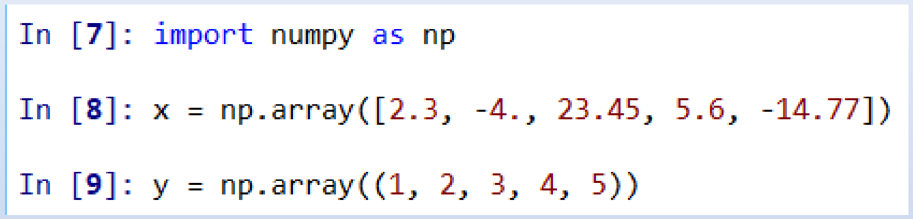


```
In [6]: mytuple[3]
Out[6]: 0.618034
```

Indices (or subscripts) are zero-based,
not one-based as in mathematical descriptions
and software packages such as MATLAB.
Excel/VBA is zero-based by default but can be
changed to one-based with the Option Base 1
declaration.

Collections of Data - Arrays

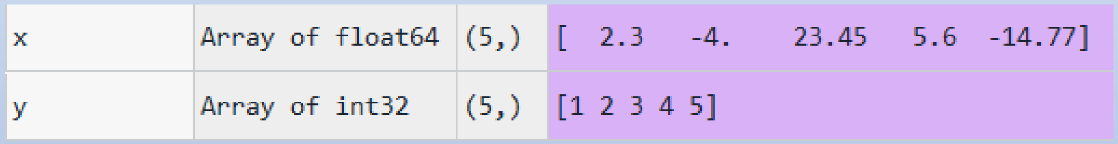
Arrays are created in a *class* called **ndarray** provided by the NumPy module.



```
In [7]: import numpy as np

In [8]: x = np.array([2.3, -4., 23.45, 5.6, -14.77])

In [9]: y = np.array((1, 2, 3, 4, 5))
```

We define **np** as an abbreviation of **numpy** because we use it so frequently.

The NumPy **array** function (or constructor) creates an array type from a list or a tuple.



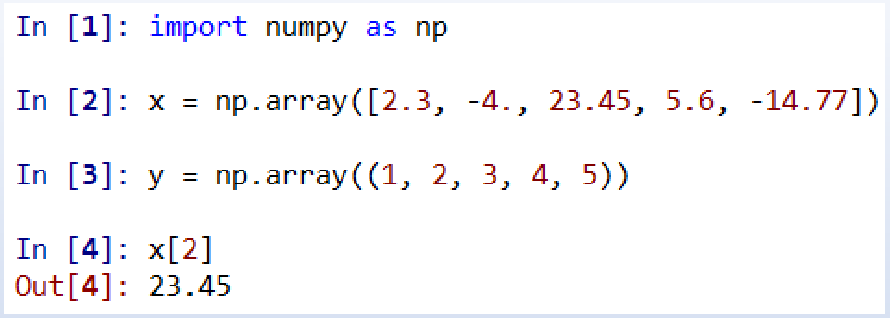
x	Array of float64	(5,)	[2.3 -4. 23.45 5.6 -14.77]
y	Array of int32	(5,)	[1 2 3 4 5]

- float64 indicates a numerical quantity is stored in 64 bits (8 bytes) according to the IEEE standard (https://ieeexplore.ieee.org/document/8766229).
- int32 shows that these are integer quantities and are stored in 32 bits (4 bytes).

Note that the size (5,) includes a comma. This allows for a second set of indices (a second dimension) to represent matrices.

Collections of Data - Arrays

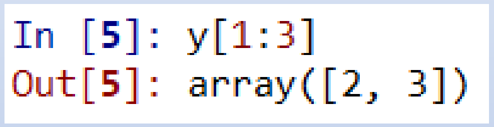
Referring to individual elements of an array with an index (or subscript):



```
In [1]: import numpy as np

In [2]: x = np.array([2.3, -4., 23.45, 5.6, -14.77])

In [3]: y = np.array((1, 2, 3, 4, 5))

In [4]: x[2]
Out[4]: 23.45
```

Note zero-based indexing.

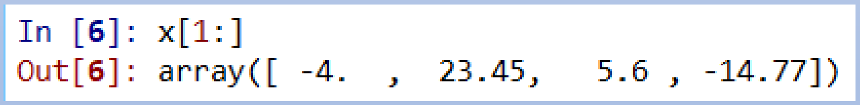
Use of the colon (:) in indexing.



```
In [5]: y[1:3]
Out[5]: array([2, 3])
```

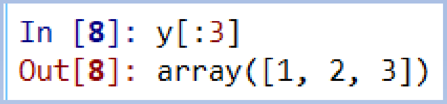
Using zero-based indexing, we might expect [1:3] to return the 2nd through the 4th element. Not so (sorry). The [i:j] subscript extracts the zero-based (i-1)th element up to, but not including, the (j-1)th element.

[1:] selects from the 2nd element to the end



```
In [6]: x[1:]
Out[6]: array([ -4.   ,  23.45,   5.6 , -14.77])
```

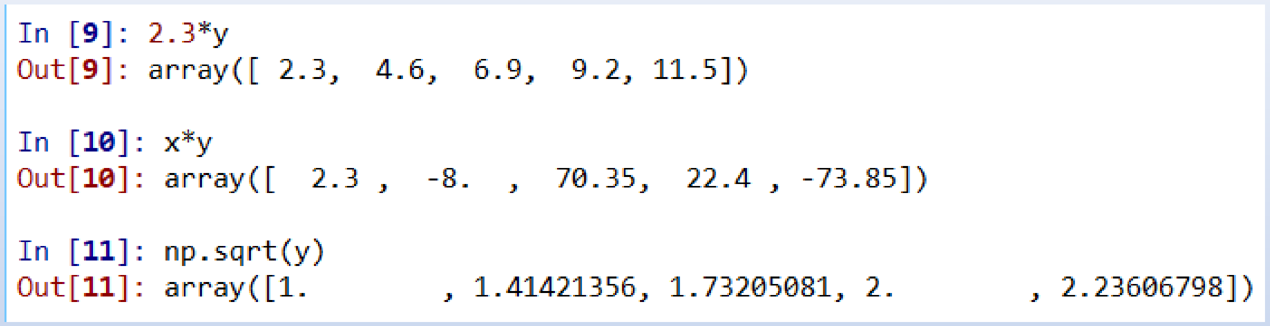
[:3] selects from the first element to the
index (3-1) or 3rd element



```
In [8]: y[:3]
Out[8]: array([1, 2, 3])
```

Collections of Data - Arrays

Array operations



```
In [9]: 2.3*y
Out[9]: array([ 2.3,  4.6,  6.9,  9.2, 11.5])

In [10]: x*y
Out[10]: array([  2.3 ,  -8.  ,  70.35,  22.4 , -73.85])

In [11]: np.sqrt(y)
Out[11]: array([1.        , 1.41421356, 1.73205081, 2.        , 2.23606798])
```

Array operations are carried out item-by-item

NumPy’s built-in functions (**sqrt** here) work with arrays, item-by-item.
The Math module’s functions do not.

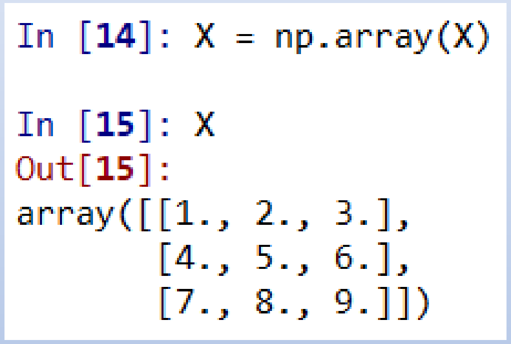
You cannot carry out array operations with lists or tuples. We use NumPy functions more frequently than Math functions.

Collections of Data - Arrays

Two-dimensional arrays

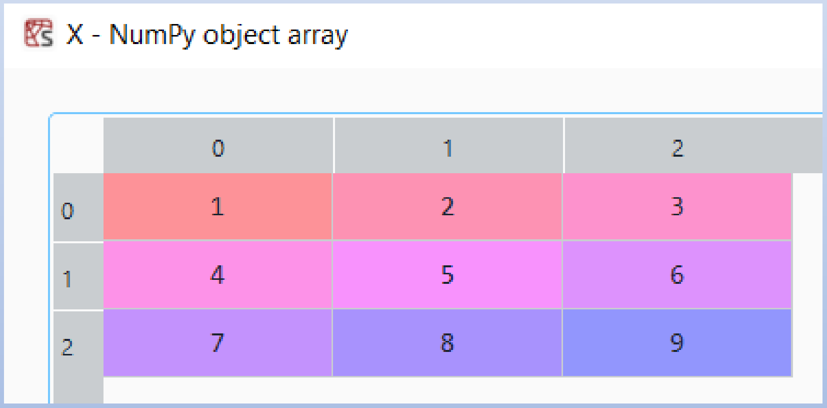
Creating a two-dimensional array from a two-dimensional list:

```
X = [ [1.,2.,3.] , [4.,5.,6.] , [7.,8.,9.] ]
```



```
In [14]: X = np.array(X)

In [15]: X
Out[15]:
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

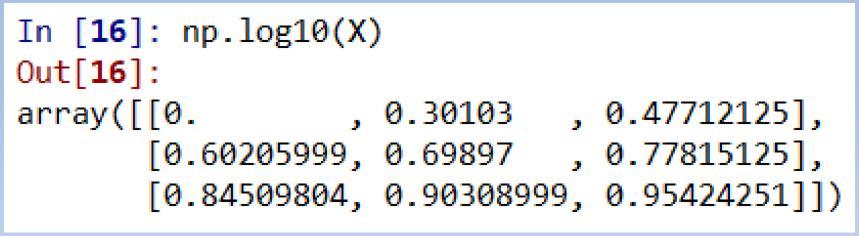
Double-click X in the Variable Explorer



X - NumPy object array

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

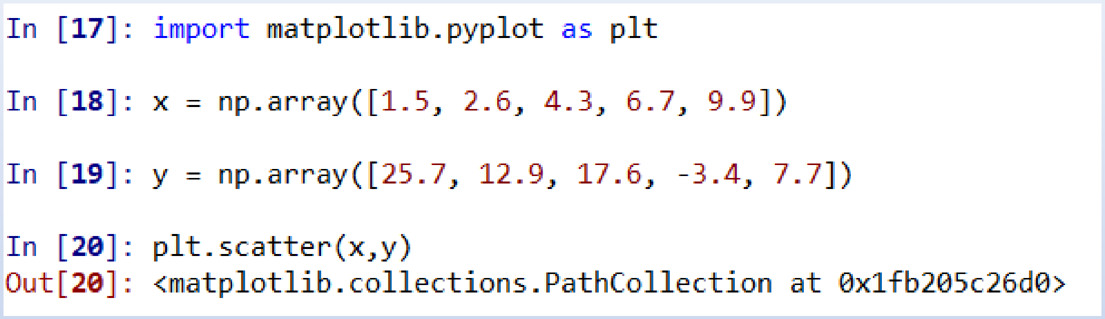
Array operations are valid with a two-dimensional array



```
In [16]: np.log10(X)
Out[16]:
array([[0.        , 0.30103   , 0.47712125],
       [0.60205999, 0.69897   , 0.77815125],
       [0.84509804, 0.90308999, 0.95424251]])
```

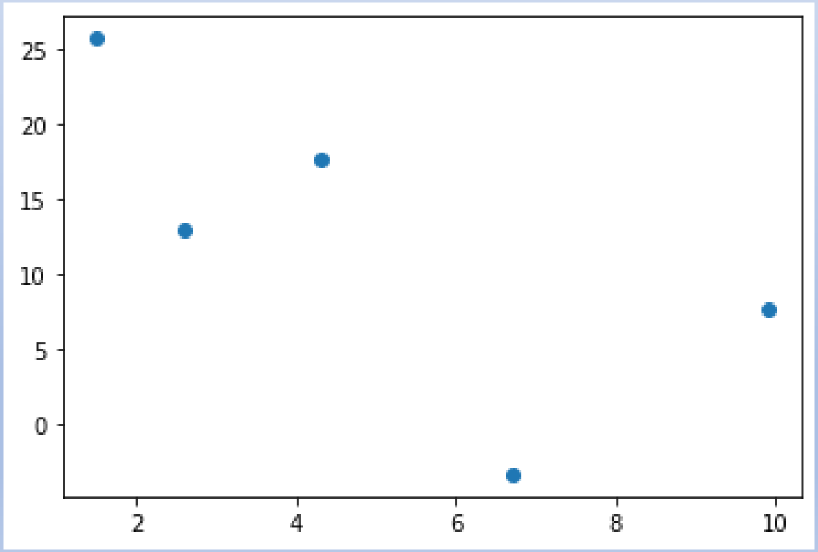
Creating Simple Plots

Matplotlib module and its **pyplot** submodule



```
In [17]: import matplotlib.pyplot as plt

In [18]: x = np.array([1.5, 2.6, 4.3, 6.7, 9.9])

In [19]: y = np.array([25.7, 12.9, 17.6, -3.4, 7.7])

In [20]: plt.scatter(x,y)
Out[20]: <matplotlib.collections.PathCollection at 0x1fb205c26d0>
```

Create **x** and **y** arrays and a scatter plot of **y** versus **x**

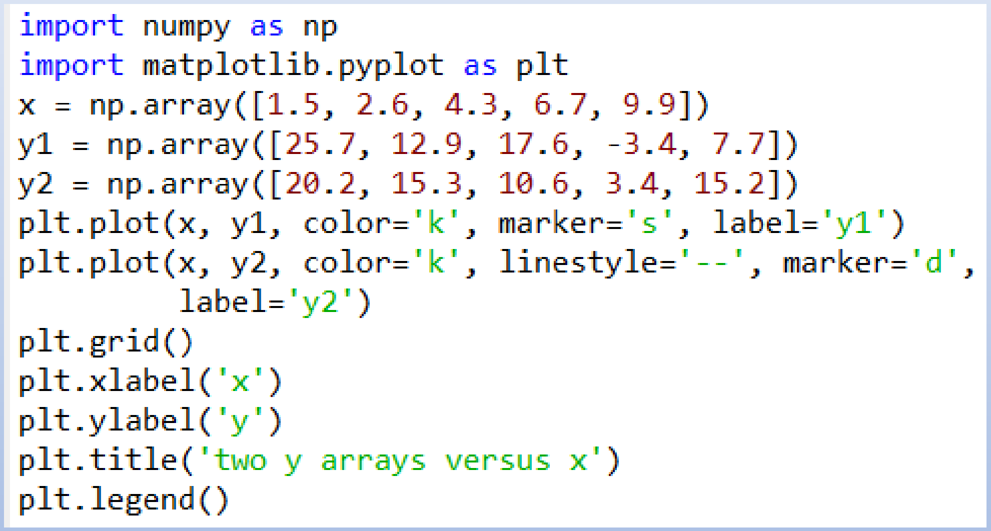


In order to customize the plot, we cannot enter additional commands into the Console, rather we must build a script in the Editor window.

Using the Spyder Editor and Getting Help

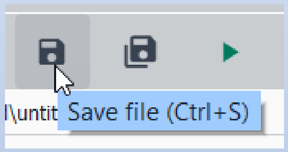
For most Python tasks involving multiple commands, we prefer to enter those as a script in the Editor window.

Here is an example that creates a plot of two **y** arrays versus an **x** array.



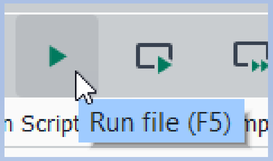
```
import numpy as np
import matplotlib.pyplot as plt
x = np.array([1.5, 2.6, 4.3, 6.7, 9.9])
y1 = np.array([25.7, 12.9, 17.6, -3.4, 7.7])
y2 = np.array([20.2, 15.3, 10.6, 3.4, 15.2])
plt.plot(x, y1, color='k', marker='s', label='y1')
plt.plot(x, y2, color='k', linestyle='--', marker='d',
         label='y2')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.title('two y arrays versus x')
plt.legend()
```

Enter this into the Editor window.
Click the Save file button (or Ctrl-S).



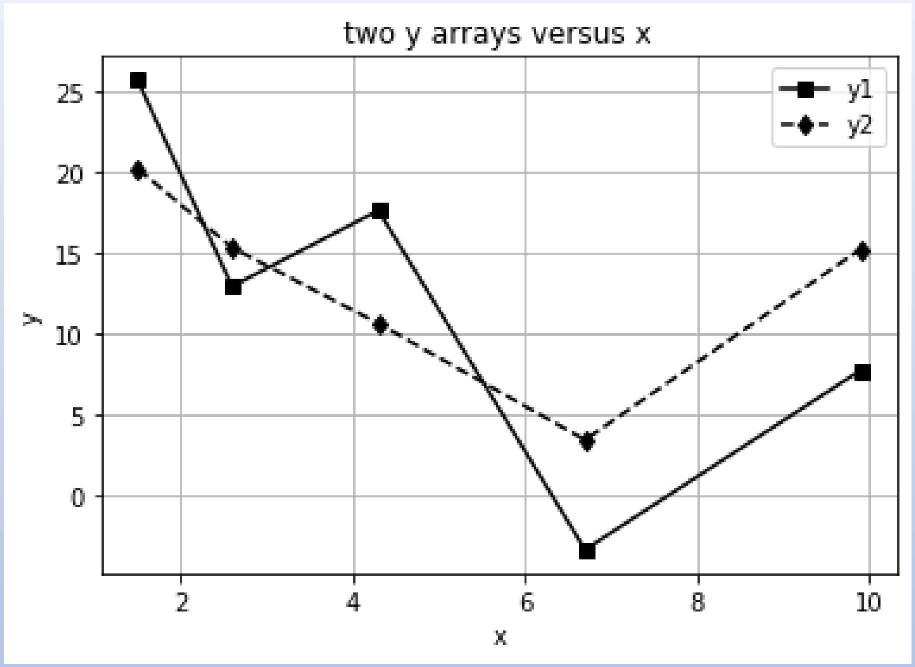
Browse to a folder of your choice and save the script as **firstplot.py**
(the **py** will be added by default)

Run the script with the Run button (or F5).



See the resulting plot on the next slide.

Using the Spyder Editor and Getting Help



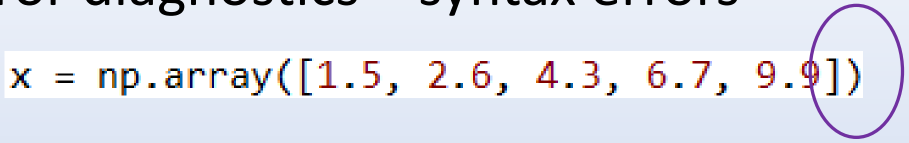
This plot should appear in the Explorer window, Plots tab.

We will get to more details about plotting a bit later.

For now, we concentrate on the Editor.

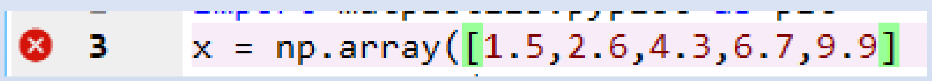
Using the Spyder Editor and Getting Help

Error diagnostics – syntax errors



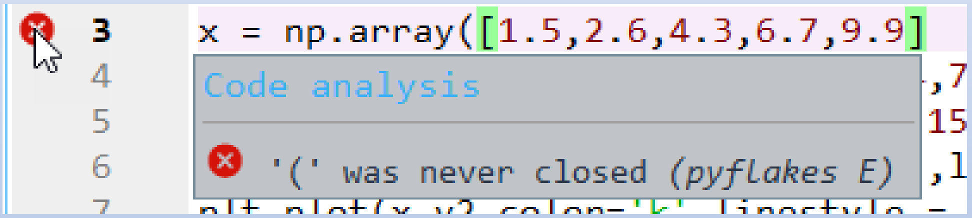
```
x = np.array([1.5, 2.6, 4.3, 6.7, 9.9])
```

remove the right parenthesis, **)**



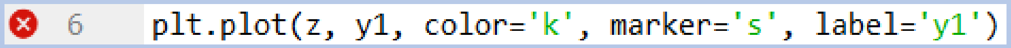
```
x = np.array([1.5,2.6,4.3,6.7,9.9]
```

a red X appears on the line



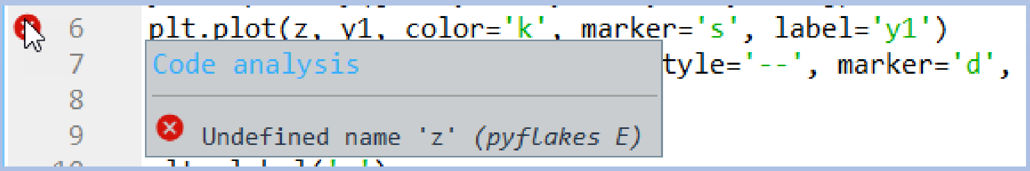
```
x = np.array([1.5,2.6,4.3,6.7,9.9]
Code analysis
'(' was never closed (pyflakes E)
```

if the mouse pointed is hovered over the X, an informative error message is displayed



```
plt.plot(z, y1, color='k', marker='s', label='y1')
```

make a typographical error by entering **z** instead of **x**

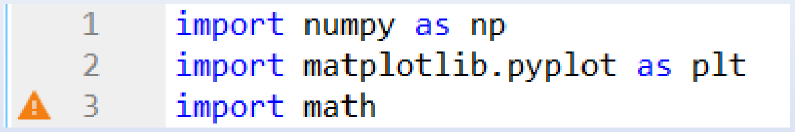


```
plt.plot(z, y1, color='k', marker='s', label='y1')
Code analysis
Undefined name 'z' (pyflakes E)
```

error message is on point

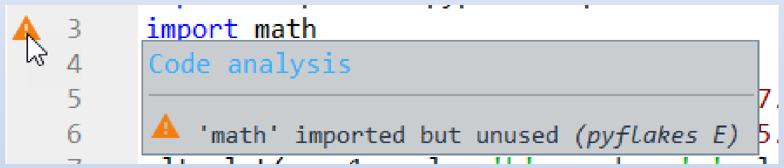
Using the Spyder Editor and Getting Help

Error diagnostics – warnings



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

orange triangle



```
import math
Code analysis

'math' imported but unused (pyflakes E)
```

indicates the **math** module has been imported but its routines were not used (yet)

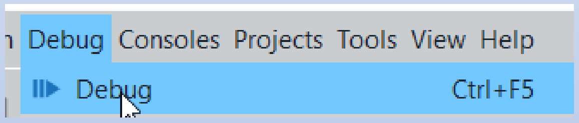
Using the Spyder Editor and Getting Help

Debugging code for execution errors

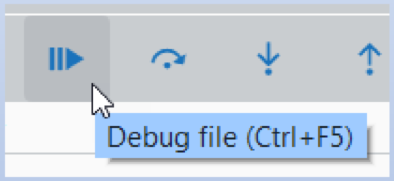
These errors show up when the script is run.

A typical technique is to single-step the code and observe variable values in the Explorer window.

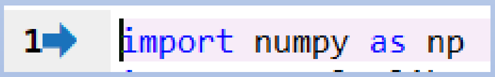
To start Debug mode for single-stepping,



or



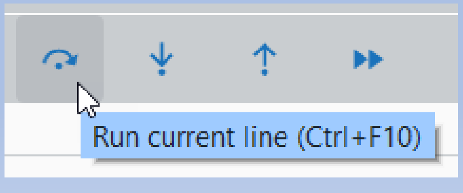
Note Ctrl-F5 shortcut



```
import numpy as np
```

arrow appears next to first statement
execution is stopped there

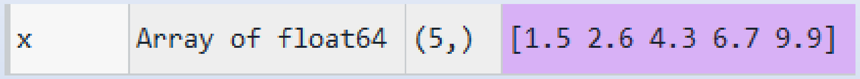
Advance execution with Ctrl-F10 or



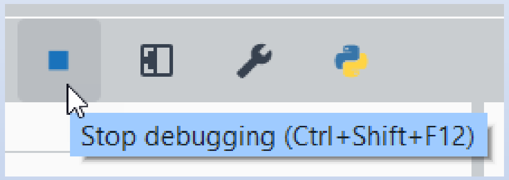
Using the Spyder Editor and Getting Help

Debugging code for execution errors

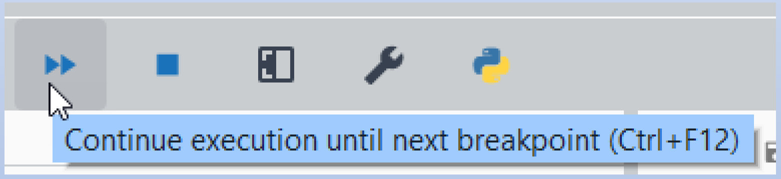
Single-step past the **x = ...** statement and Variable Explorer shows


```
x    Array of float64    (5,)    [1.5 2.6 4.3 6.7 9.9]
```

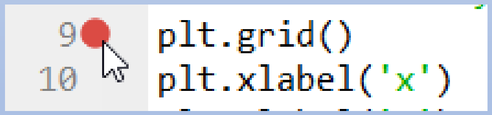
Stop debugging mode with


Stop debugging (Ctrl+Shift+F12)

Continue execution (no single-stepping) with

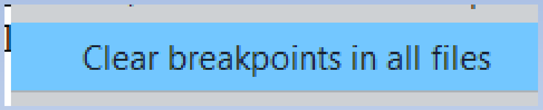

Continue execution until next breakpoint (Ctrl+F12)

Place a breakpoint.


```
9   plt.grid()
10  plt.xlabel('x')
```

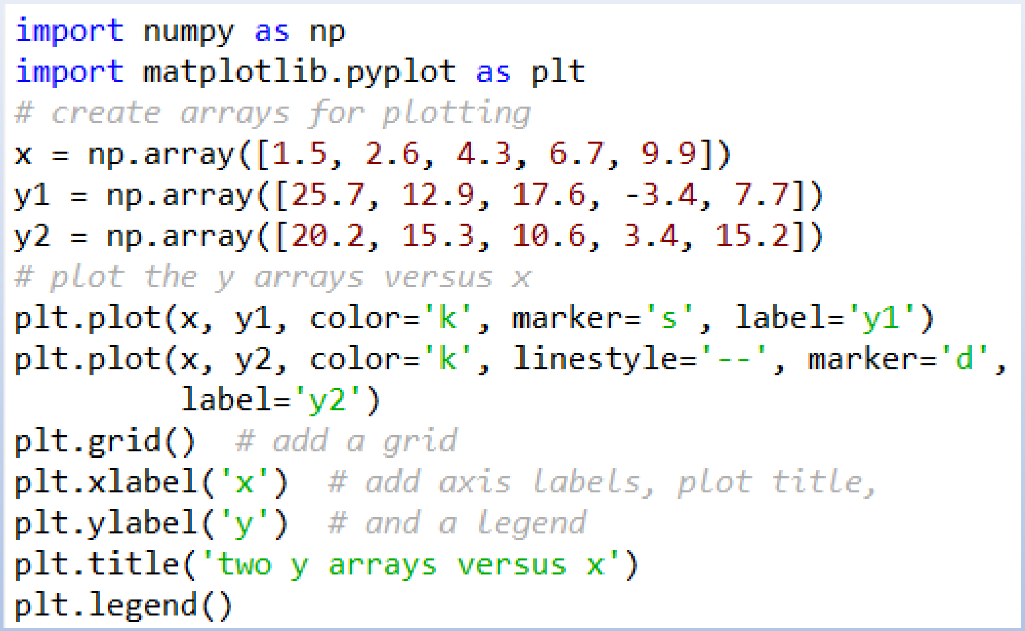
Execution will stop there.

Clear a breakpoint by clicking on the red dot,
also clear all breakpoints with
on the Debug menu


Clear breakpoints in all files

Using the Spyder Editor and Getting Help

Adding comments to scripts



```python
import numpy as np
import matplotlib.pyplot as plt
# create arrays for plotting
x = np.array([1.5, 2.6, 4.3, 6.7, 9.9])
y1 = np.array([25.7, 12.9, 17.6, -3.4, 7.7])
y2 = np.array([20.2, 15.3, 10.6, 3.4, 15.2])
# plot the y arrays versus x
plt.plot(x, y1, color='k', marker='s', label='y1')
plt.plot(x, y2, color='k', linestyle='--', marker='d',
         label='y2')
plt.grid()  # add a grid
plt.xlabel('x')  # add axis labels, plot title,
plt.ylabel('y')  # and a legend
plt.title('two y arrays versus x')
plt.legend()
```

firstplot_withcomments.py

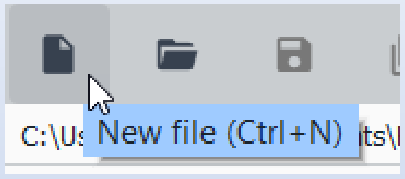
Start with **#**

Either on their own line or appended to a line of code

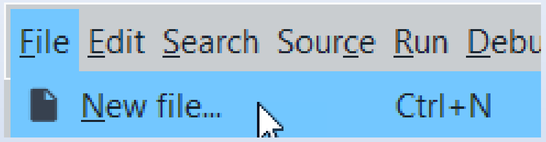
Comments are useful and recommended, especially for scripts that will be seen and used by others. They also serve as a good reminder for the author after not seeing the script for some time.

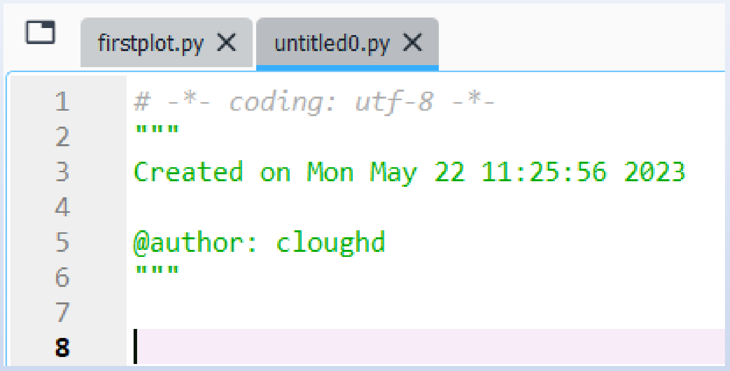
Using the Spyder Editor and Getting Help

Creating a new file in the Editor



or

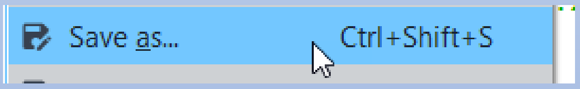


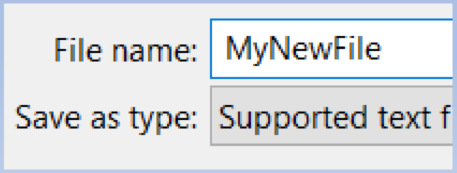


```
# -*- coding: utf-8 -*-
"""
Created on Mon May 22 11:25:56 2023

@author: cloughd
"""

```

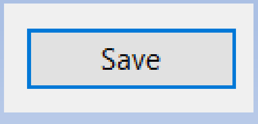
By default, initial script window shows a **comment** set off by **#** and a template for a **docustring** between " " "
These can be deleted or edited.

Save the new file with an appropriate filename via Save As.



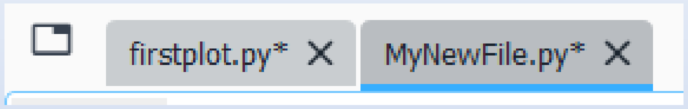


File name: MyNewFile
Save as type: Supported text f

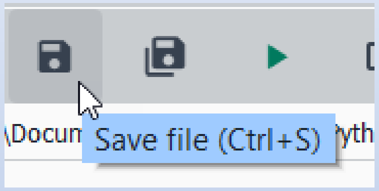


Using the Spyder Editor and Getting Help

Saving and closing files

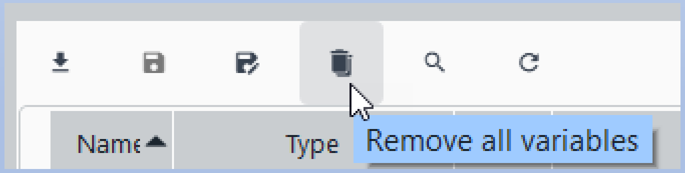


If changes have been made to a file, the name shows with an asterisk (*). That is a reminder that the file should be saved with



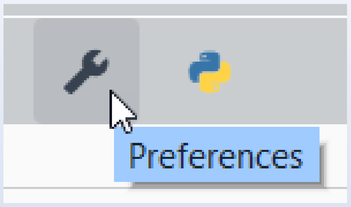
Close a file by clicking the X on its tab. If it needs to be saved, a reminder will appear.

Variables will accumulate in the Explorer window. You can clear the window with

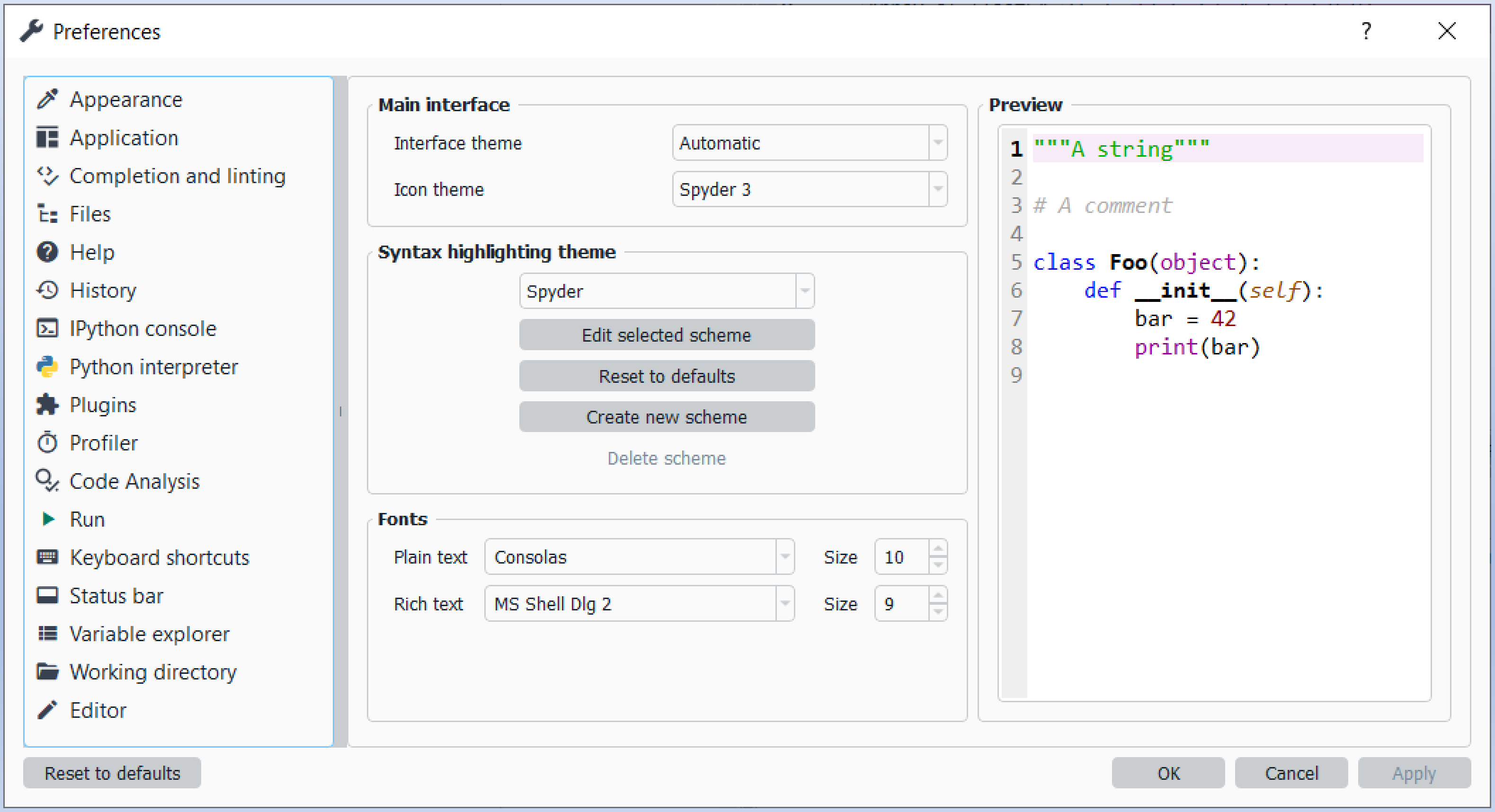


Using the Spyder Editor and Getting Help

Changing Spyder settings



Many settings are available for customization.



Main interface

Interface theme: Automatic

Icon theme: Spyder 3

Syntax highlighting theme

Spyder

Fonts

Plain text: Consolas Size: 10

Rich text: MS Shell Dlg 2 Size: 9

Preview

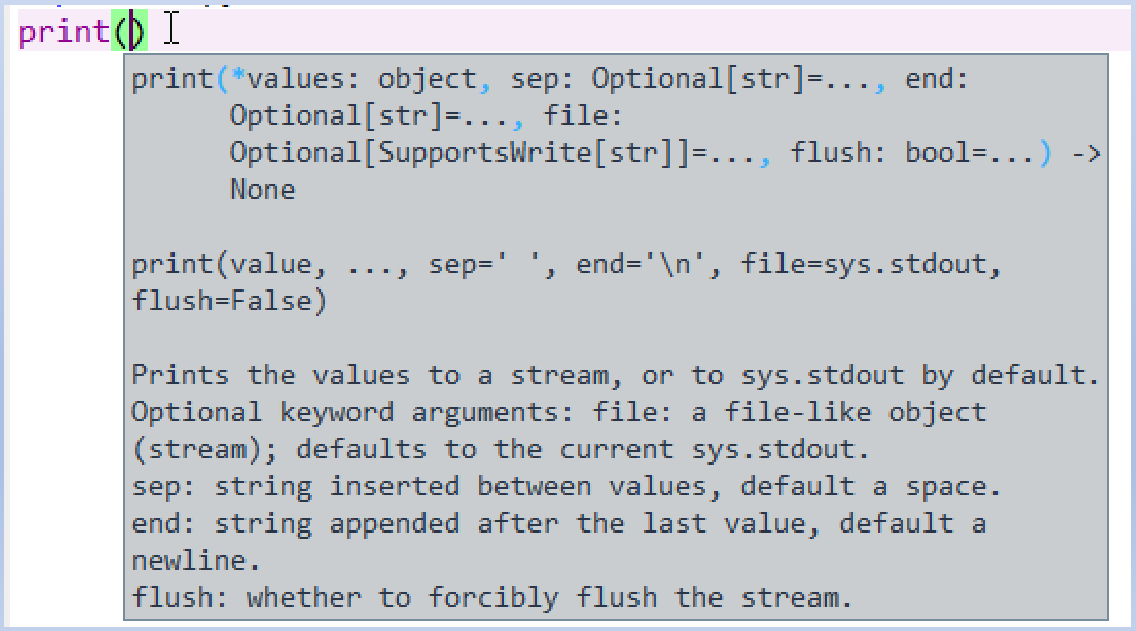
```python
"""A string"""

# A comment

class Foo(object):
    def __init__(self):
        bar = 42
        print(bar)
```

Using the Spyder Editor and Getting Help

"Flavors" of help

- Recalling arguments to built-in functions
- Finding information on a function or feature
- Getting the answer to "How do I . . . in Python?"

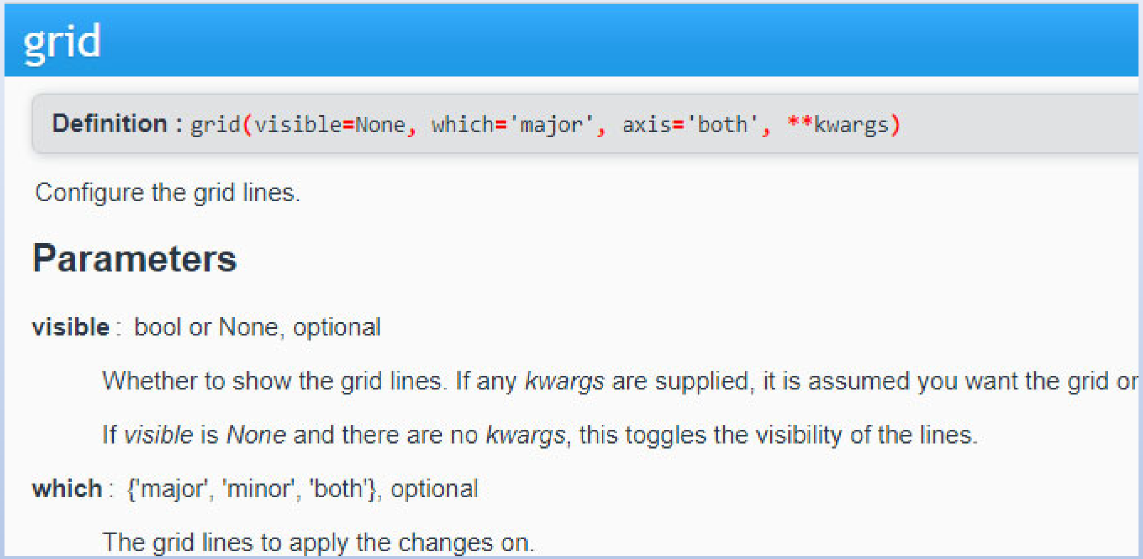
Tooltips that appear while typing in a function:



```
print()

print(*values: object, sep: Optional[str]=..., end:
      Optional[str]=..., file:
      Optional[SupportsWrite[str]]=..., flush: bool=...) ->
      None

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

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

Using the Spyder Editor and Getting Help

Hover the mouse pointer on a function or keyword and press Ctrl-I



grid

Definition : `grid(visible=None, which='major', axis='both', **kwargs)`

Configure the grid lines.

Parameters

visible : bool or None, optional

Whether to show the grid lines. If any *kwargs* are supplied, it is assumed you want the grid or

If *visible* is *None* and there are no *kwargs*, this toggles the visibility of the lines.

which : {'major', 'minor', 'both'}, optional

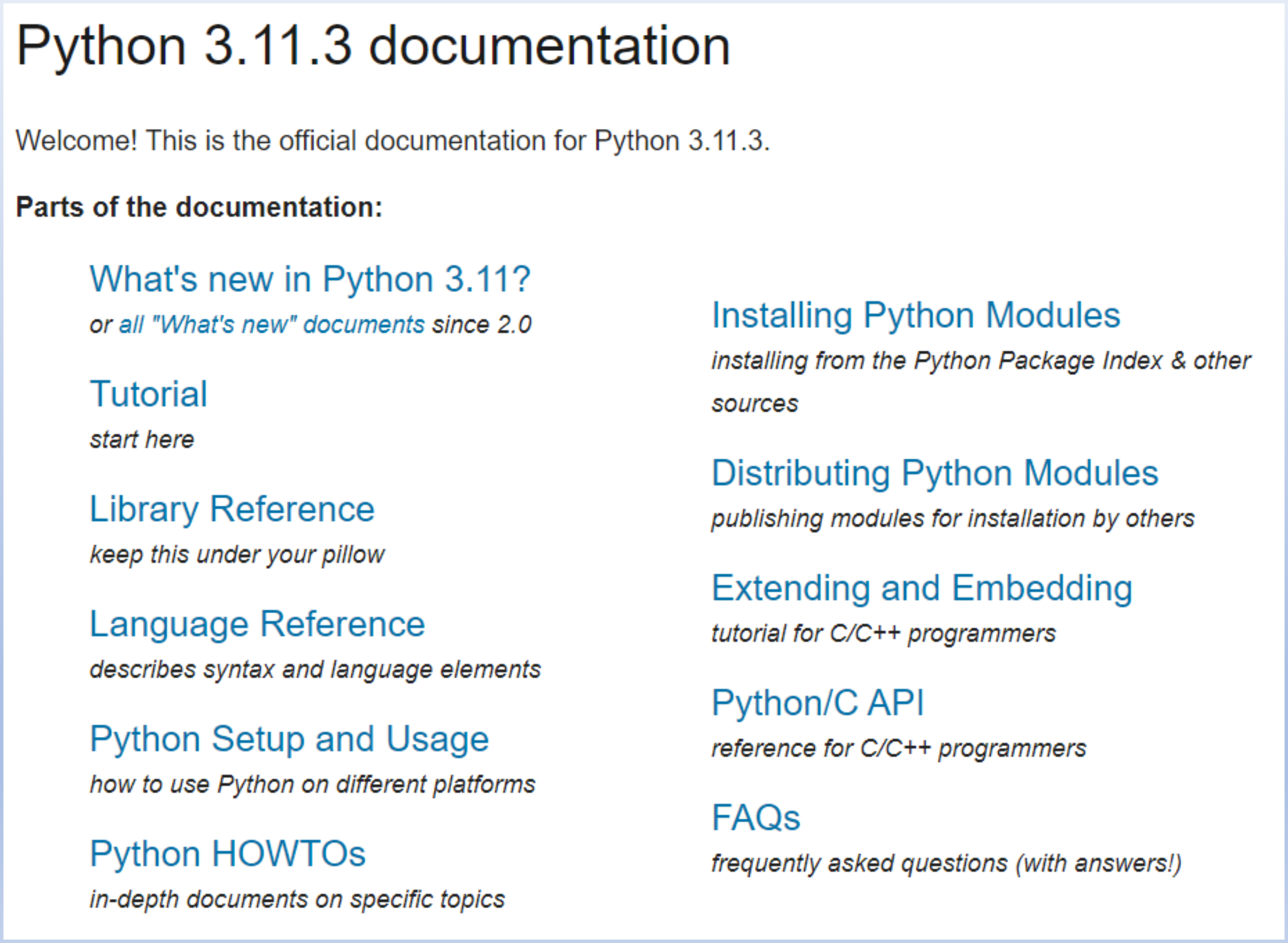
The grid lines to apply the changes on.

Help appears in the Explorer window

Note: The Spyder Help menu is not that helpful.

Using the Spyder Editor and Getting Help

Internet help on Python [3.11.3 Documentation (python.org)]



Python 3.11.3 documentation

Welcome! This is the official documentation for Python 3.11.3.

Parts of the documentation:

What's new in Python 3.11?
or all "What's new" documents since 2.0

Tutorial
start here

Library Reference
keep this under your pillow

Language Reference
describes syntax and language elements

Python Setup and Usage
how to use Python on different platforms

Python HOWTOs
in-depth documents on specific topics

Installing Python Modules
installing from the Python Package Index & other sources

Distributing Python Modules
publishing modules for installation by others

Extending and Embedding
tutorial for C/C++ programmers

Python/C API
reference for C/C++ programmers

FAQs
frequently asked questions (with answers!)

Using the Spyder Editor and Getting Help

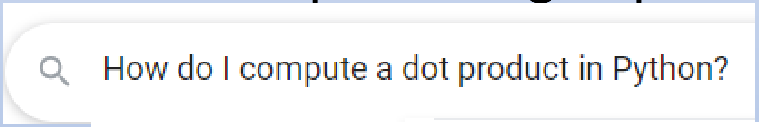
Internet help on NumPy, SciPy, and Matplotlib

NumPy user guide — NumPy v1.24 Manual

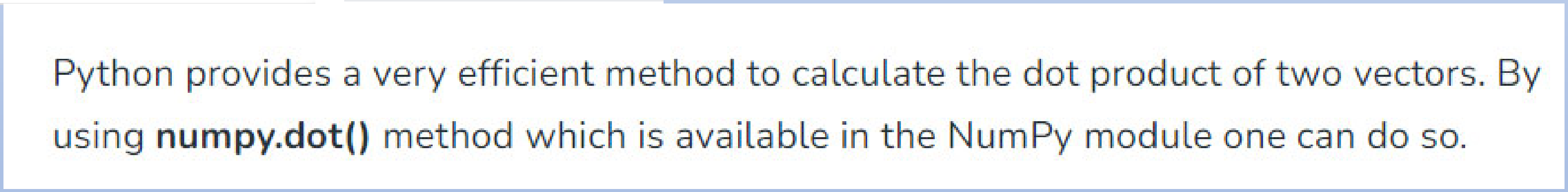
SciPy User Guide — SciPy v1.10.1 Manual

Users guide — Matplotlib 3.7.1 documentation

Internet help – asking a question



How do I compute a dot product in Python?



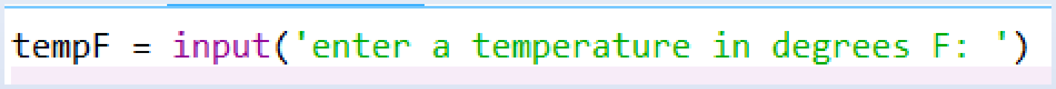
Python provides a very efficient method to calculate the dot product of two vectors. By using **numpy.dot()** method which is available in the NumPy module one can do so.

Help on using Spyder

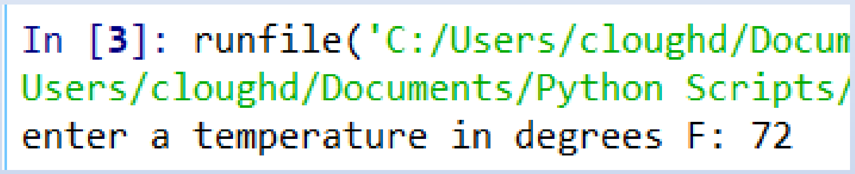
Welcome to Spyder's Documentation — Spyder 5 documentation (spyder-ide.org)

Input/Output

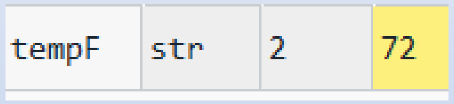
Get user response from the Console with the **input** function



```
tempF = input('enter a temperature in degrees F: ')
```



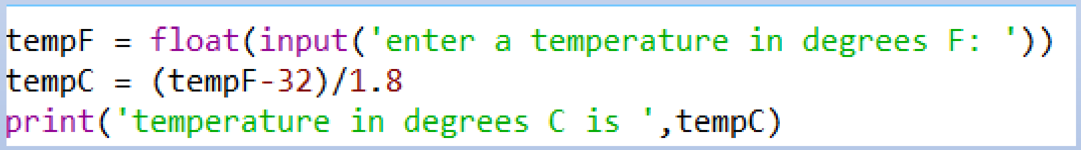
```
In [3]: runfile('C:/Users/cloughd/Docum
Users/cloughd/Documents/Python Scripts/
enter a temperature in degrees F: 72
```



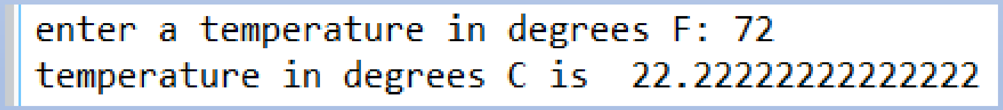
tempF	str	2	72

Result is a string (text)

Displaying a result in the Console with the **print** function



```
tempF = float(input('enter a temperature in degrees F: '))
tempC = (tempF-32)/1.8
print('temperature in degrees C is ',tempC)
```



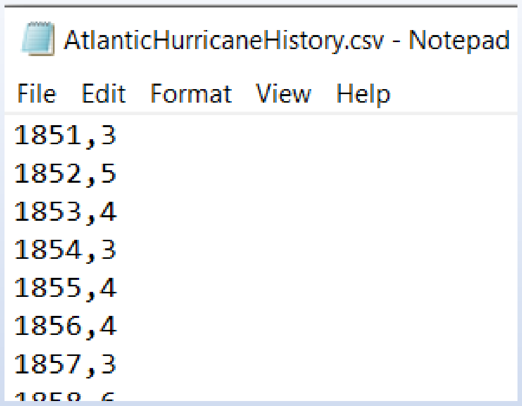
```
enter a temperature in degrees F: 72
temperature in degrees C is  22.2222222222222
```

Result displayed in full precision

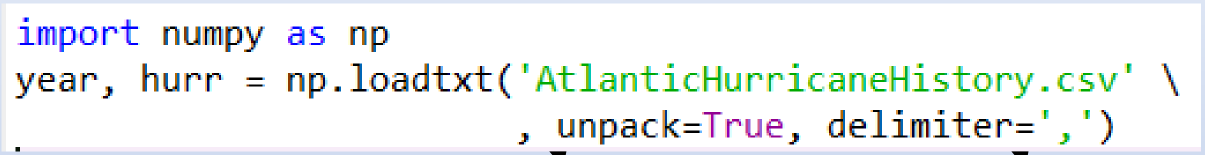
Input/Output

Input from text files (.txt or .csv)

Example: AtlanticHurricaneHistory.csv


AtlanticHurricaneHistory.csv - Notepad
File Edit Format View Help

```
1851,3
1852,5
1853,4
1854,3
1855,4
1856,4
1857,3
```

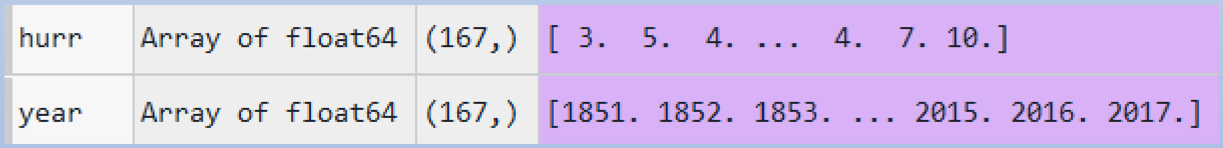


```python
import numpy as np
year, hurr = np.loadtxt('AtlanticHurricaneHistory.csv' \
                        , unpack=True, delimiter=',')
```

Separate items on each file record into distinct array variables.

A comma (**,**) separates the items on each file record, as indicated by .csv

Note the use of the backslash (****) as a line continuation character.



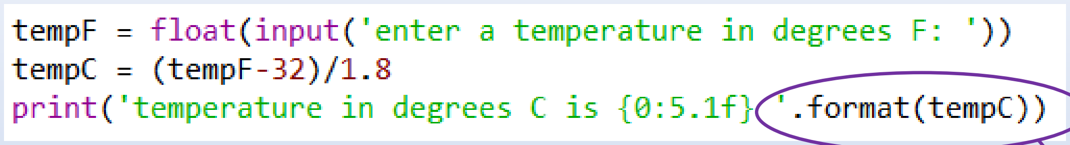
hurr	Array of float64	(167,)	[3. 5. 4. ... 4. 7. 10.]
year	Array of float64	(167,)	[1851. 1852. 1853. ... 2015. 2016. 2017.]

There is a NumPy **savetxt** function to write data to a text file.
There are **load** and **save** functions to read and write binary files.

http://tropical.atmos.colostate.edu/Realtime/index.php?arch&loc=northatlantic

Input/Output

Formatting output



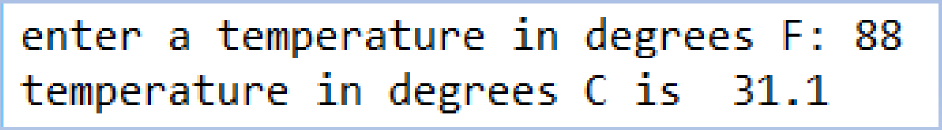
```
tempF = float(input('enter a temperature in degrees F: '))
tempC = (tempF-32)/1.8
print('temperature in degrees C is {0:5.1f}'.format(tempC))
```

f	floating point
d	decimal (integer)
e	exponent (scientific)
g	general (**f** or **e**)

`{0:5.1f}` **f** : floating point display

- `0`: First value to be formatted
- `5`: Field width for the display
- `.1`: No. of decimal places

format is a "method" that applies to the preceding string "object"
tempC is an argument to the format method

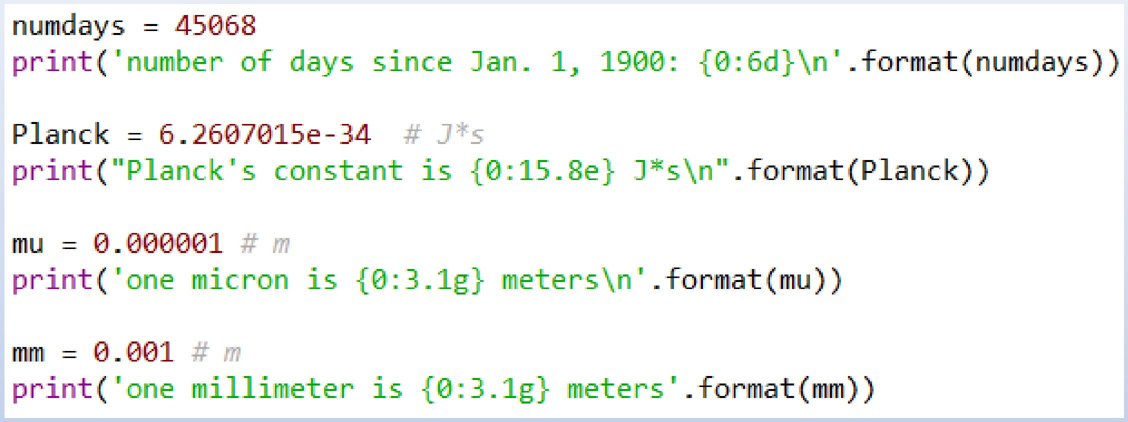


```
enter a temperature in degrees F: 88
temperature in degrees C is  31.1
```

Input/Output

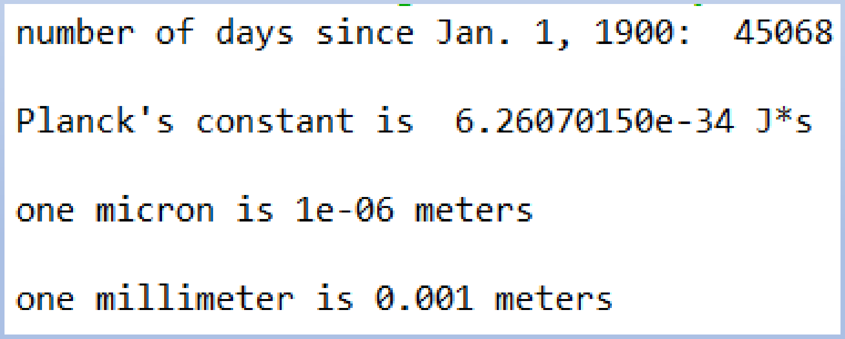
Formatting output - examples

FormattingExamples.py



```python
numdays = 45068
print('number of days since Jan. 1, 1900: {0:6d}\n'.format(numdays))

Planck = 6.2607015e-34  # J*s
print("Planck's constant is {0:15.8e} J*s\n".format(Planck))

mu = 0.000001 # m
print('one micron is {0:3.1g} meters\n'.format(mu))

mm = 0.001 # m
print('one millimeter is {0:3.1g} meters'.format(mm))
```

Notice use of alternate string delimiter **“. . .”** to preserve use of **‘** in **Planck’s**.



```
number of days since Jan. 1, 1900:  45068

Planck's constant is  6.26070150e-34 J*s

one micron is 1e-06 meters

one millimeter is 0.001 meters
```

g format chooses
e for micron and
f for millimeter

Plotting with Matplotlib's pyplot

Plot of an analytical function

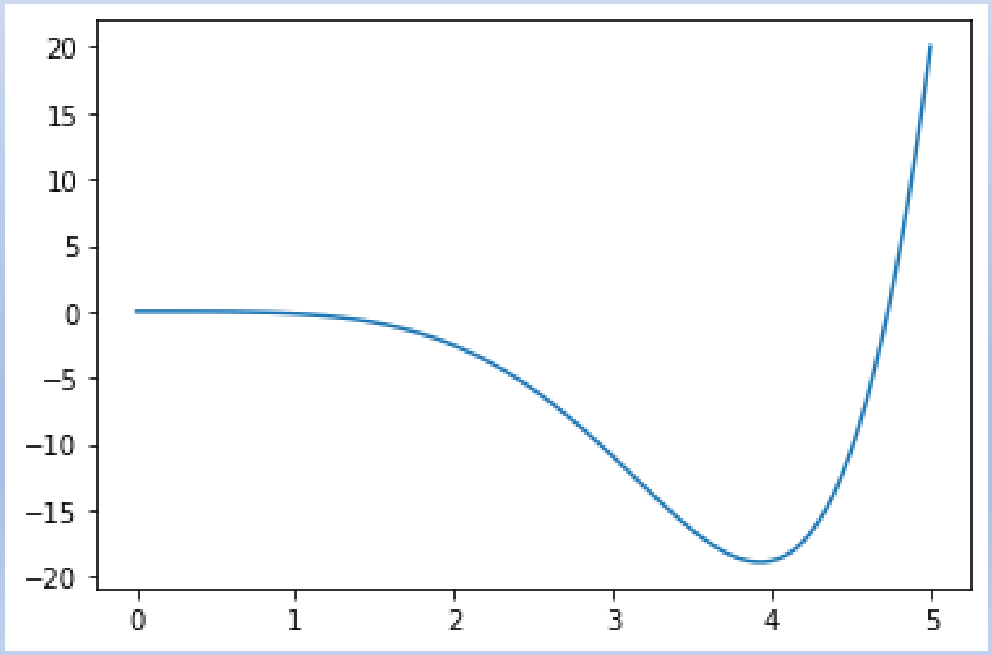
plot_analytical_start.py

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0,5,100)
y = np.cos(x)*np.cosh(x)-1
plt.plot(x,y)
plt.show()
```

linspace function creates an array of 100 equally spaced points between 0 and 5 assigned to **x**

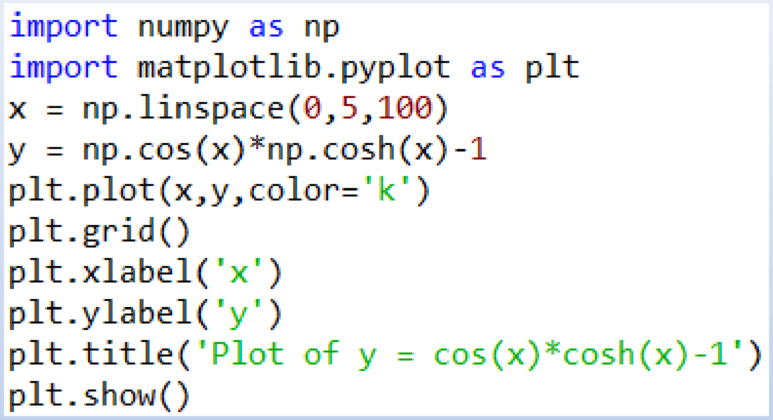
vectorized expression to compute the **y** array from the **x** array

not required in the Spyder IDE but may be required in other Python implementations

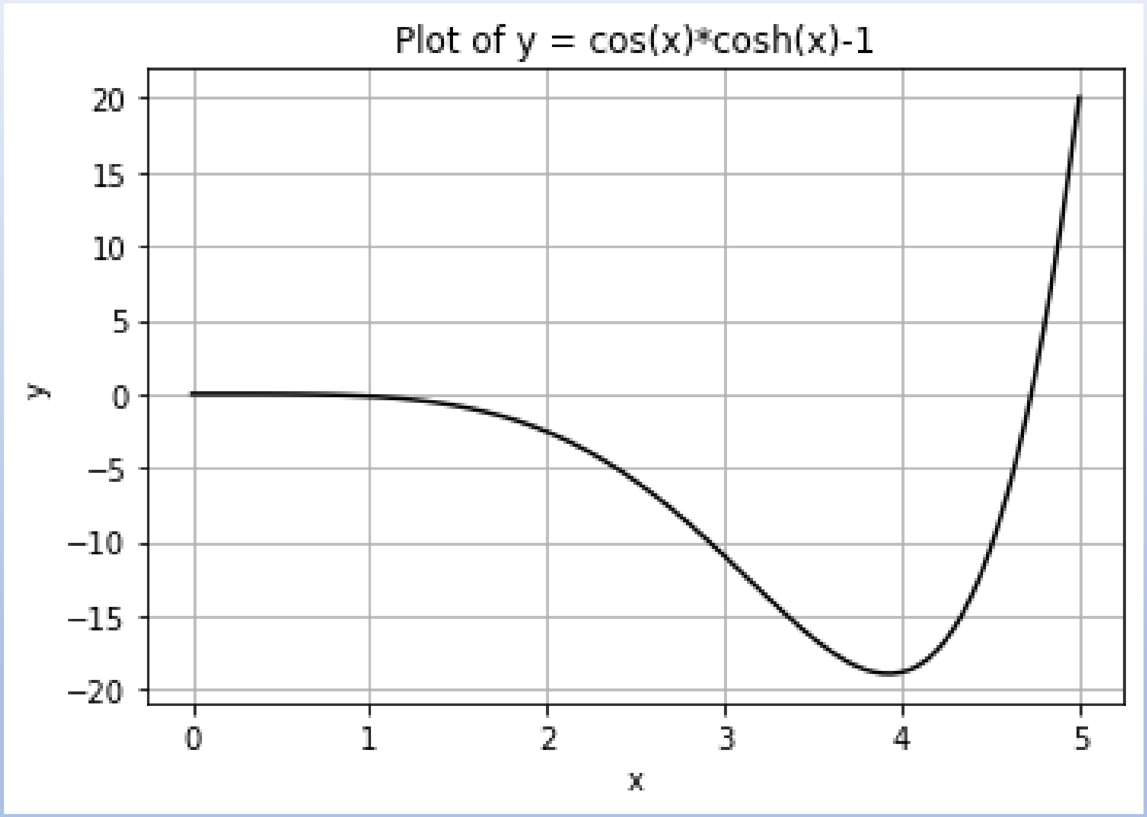


Plotting with Matplotlib’s pyplot

Customizing the plot – add grid and labels, change line color to black



```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0,5,100)
y = np.cos(x)*np.cosh(x)-1
plt.plot(x,y,color='k')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Plot of y = cos(x)*cosh(x)-1')
plt.show()
```



<u>Color codes</u>

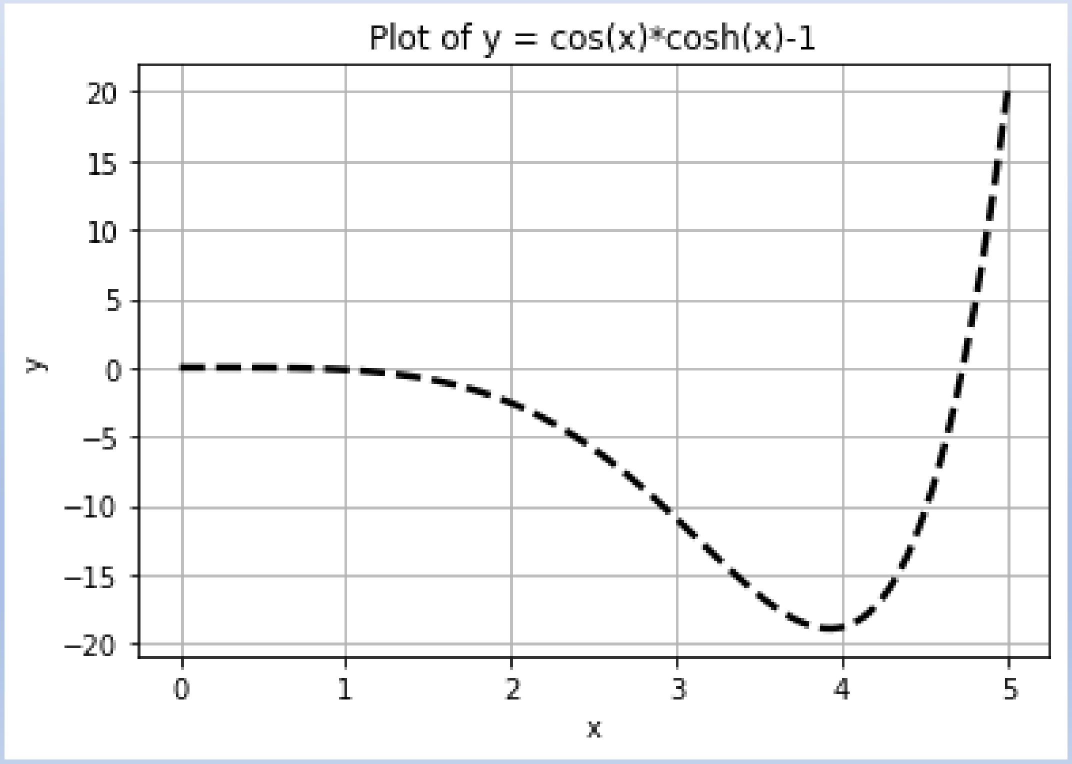
k	black
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
w	white

Plotting with Matplotlib's pyplot

plot_analytical1.py

Customizing the plot – change linestyle and line width

```
plt.plot(x,y,c='k',ls='--',lw=2.5)
```



<u>Abbreviations</u>

c	color
ls	linestyle
lw	line width

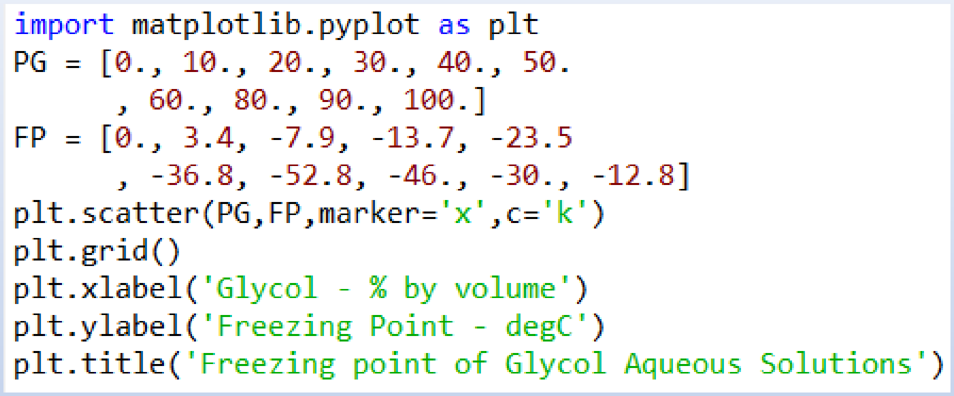
<u>Linestyle Codes</u>

-	solid
--	dashed
:	dotted
-.	dash-dot

Plotting with Matplotlib's pyplot

Plots of data with the **scatter** function

plot_glycol_data_starter.py

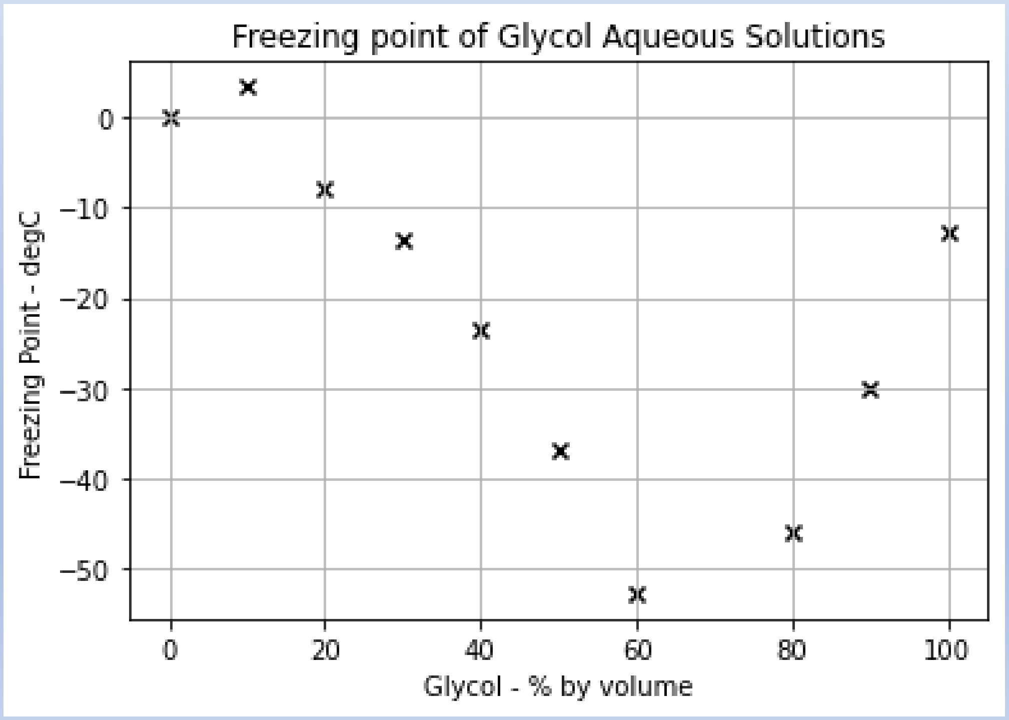


```
import matplotlib.pyplot as plt
PG = [0., 10., 20., 30., 40., 50.
      , 60., 80., 90., 100.]
FP = [0., 3.4, -7.9, -13.7, -23.5
      , -36.8, -52.8, -46., -30., -12.8]
plt.scatter(PG,FP,marker='x',c='k')
plt.grid()
plt.xlabel('Glycol - % by volume')
plt.ylabel('Freezing Point - degC')
plt.title('Freezing point of Glycol Aqueous Solutions')
```

scatter will plot numerical lists
(not **NumPy** arrays here – could be)

<u>Marker Codes</u>

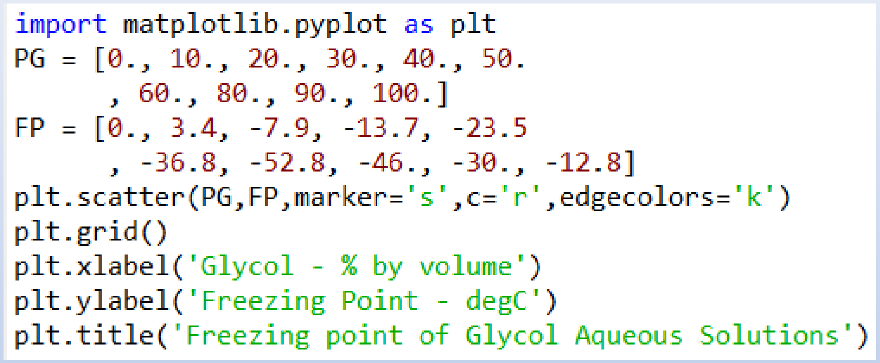
.	point
o	circle
+	plus sign
x	x
D	diamond
V	del
^	triangle
s	square



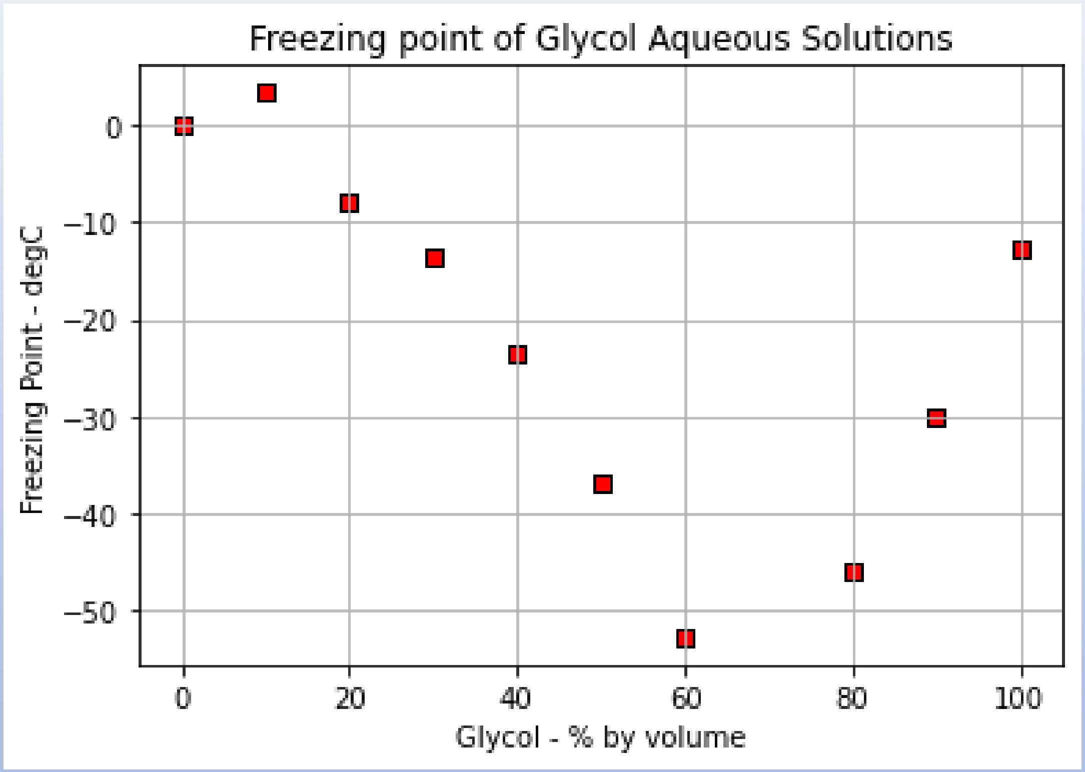
Plotting with Matplotlib's pyplot

Plots of data – customizing markers on scatter plots

plot_glycol_data.py



```
import matplotlib.pyplot as plt
PG = [0., 10., 20., 30., 40., 50.
      , 60., 80., 90., 100.]
FP = [0., 3.4, -7.9, -13.7, -23.5
      , -36.8, -52.8, -46., -30., -12.8]
plt.scatter(PG,FP,marker='s',c='r',edgecolors='k')
plt.grid()
plt.xlabel('Glycol - % by volume')
plt.ylabel('Freezing Point - degC')
plt.title('Freezing point of Glycol Aqueous Solutions')
```



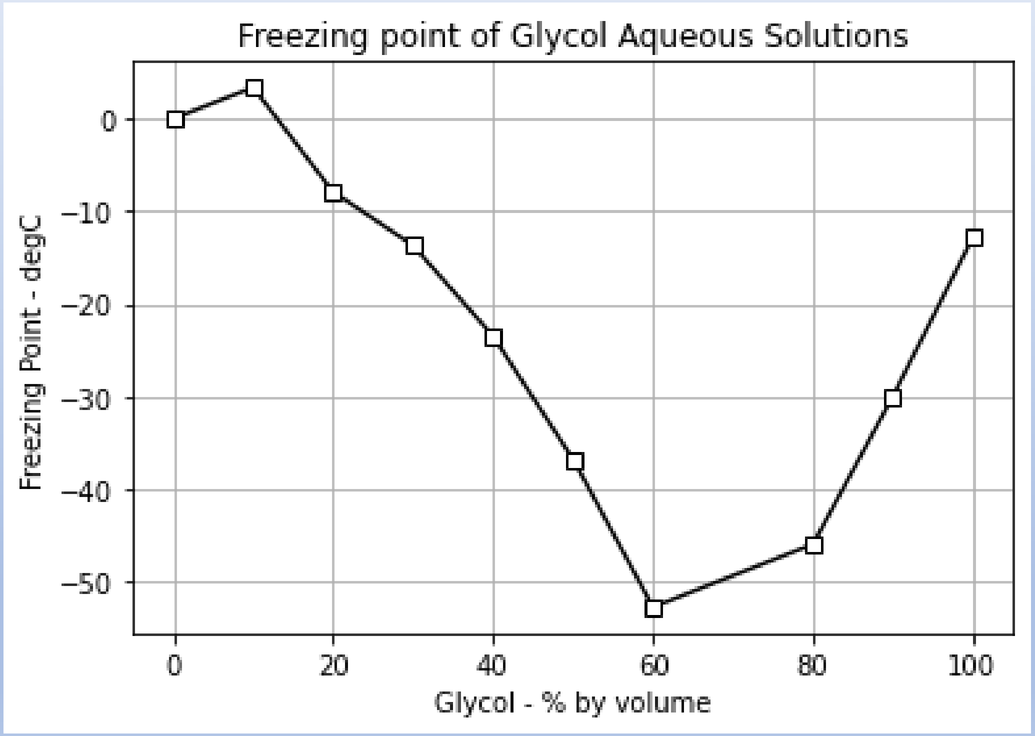
c	interior color
edgecolors	edges

Plotting with Matplotlib's pyplot

Plots of data – markers with lines using the **plot** function

```
plt.plot(PG,FP,c = 'k',marker='s',markeredgecolor='k' \
         , markerfacecolor='w')
```

plot_glycol_datalines.py

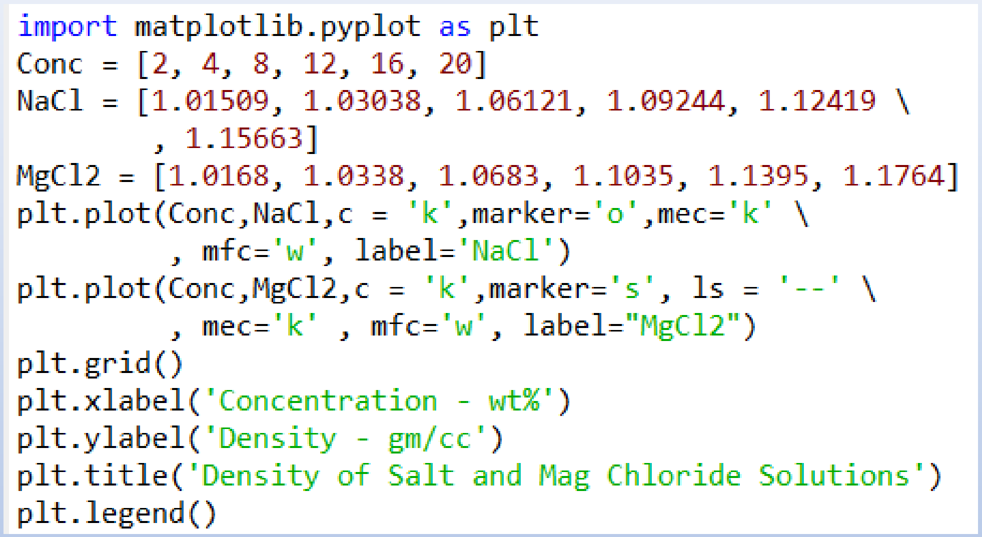


<u>Abbreviations</u>

mec	markeredgecolor
mfc	markerfacecolor

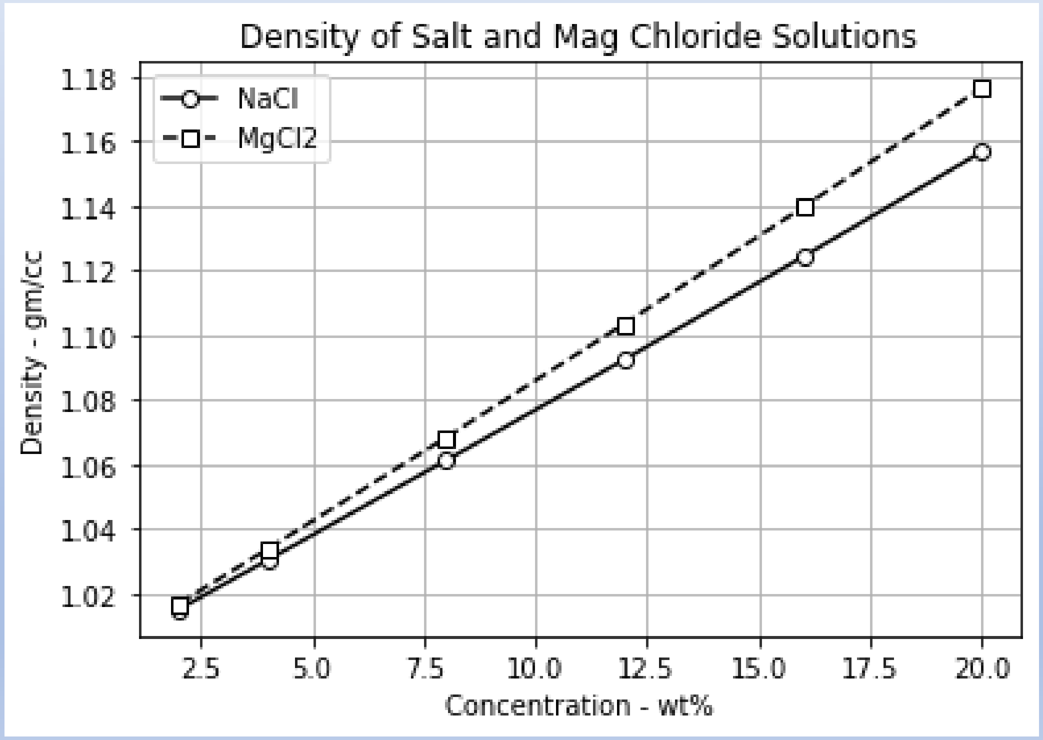
Plotting with Matplotlib’s pyplot

Plots of data – plotting more than one series with a legend



```
import matplotlib.pyplot as plt
Conc = [2, 4, 8, 12, 16, 20]
NaCl = [1.01509, 1.03038, 1.06121, 1.09244, 1.12419 \
       , 1.15663]
MgCl2 = [1.0168, 1.0338, 1.0683, 1.1035, 1.1395, 1.1764]
plt.plot(Conc,NaCl,c = 'k',marker='o',mec='k' \
         , mfc='w', label='NaCl')
plt.plot(Conc,MgCl2,c = 'k',marker='s', ls = '--' \
         , mec='k' , mfc='w', label="MgCl2")
plt.grid()
plt.xlabel('Concentration - wt%')
plt.ylabel('Density - gm/cc')
plt.title('Density of Salt and Mag Chloride Solutions')
plt.legend()
```

SaltandMagClDensities.py



label field in plot command provides text for **legend** function

Plotting with Matplotlib's pyplot

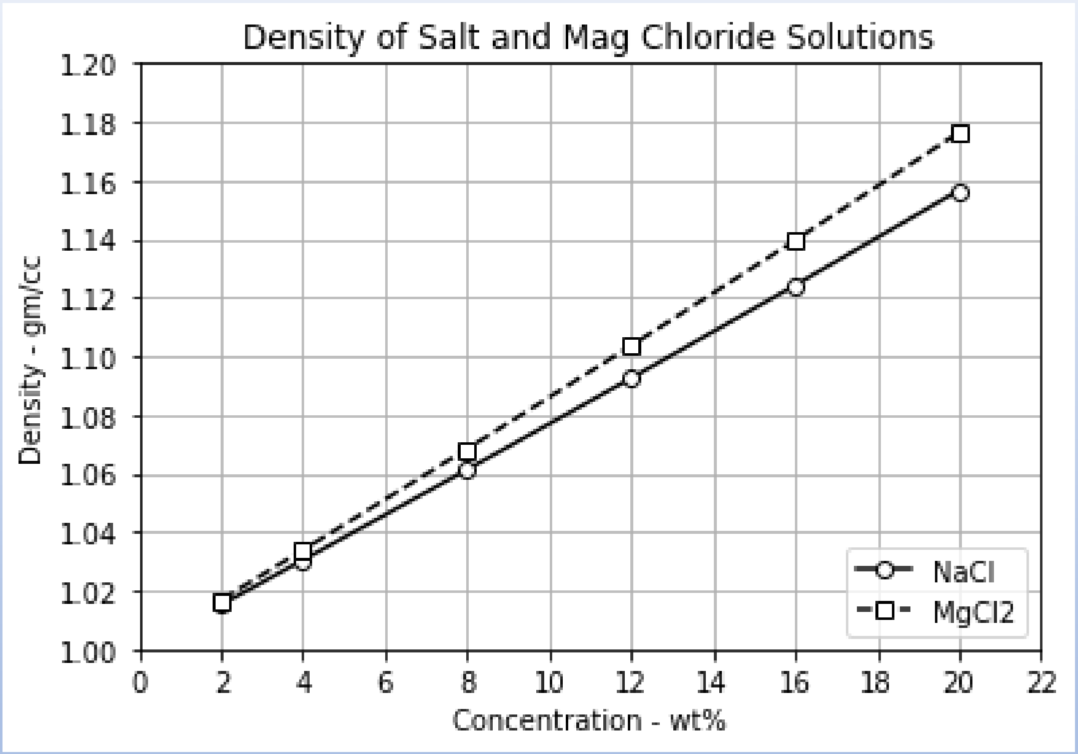
SaltandMagClDensities_AxesTicks.py

Plots of data – plotting more than one series

Adjusting legend position

Changing axis limits and tick intervals

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

Conc = [2, 4, 8, 12, 16, 20]
NaCl = [1.01509, 1.03038, 1.06121, 1.09244, 1.12419 \
        , 1.15663]
MgCl2 = [1.0168, 1.0338, 1.0683, 1.1035, 1.1395, 1.1764]

plt.plot(Conc,NaCl,c = 'k',marker='o',mec='k' \
         , mfc='w', label='NaCl')
plt.plot(Conc,MgCl2,c = 'k',marker='s', ls = '--' \
         , mec='k' , mfc='w', label="MgCl2")
plt.grid()
plt.xlim(0.,22.)
plt.ylim(1.,1.2)
plt.xticks(np.arange(0,24,2))
plt.yticks(np.arange(1,1.22,0.02))
plt.xlabel('Concentration - wt%')
plt.ylabel('Density - gm/cc')
plt.title('Density of Salt and Mag Chloride Solutions')
plt.legend(loc='lower right')
```



NumPy arrange function

arrange(start_value,end-value,interval)

last value is <u>not</u> included

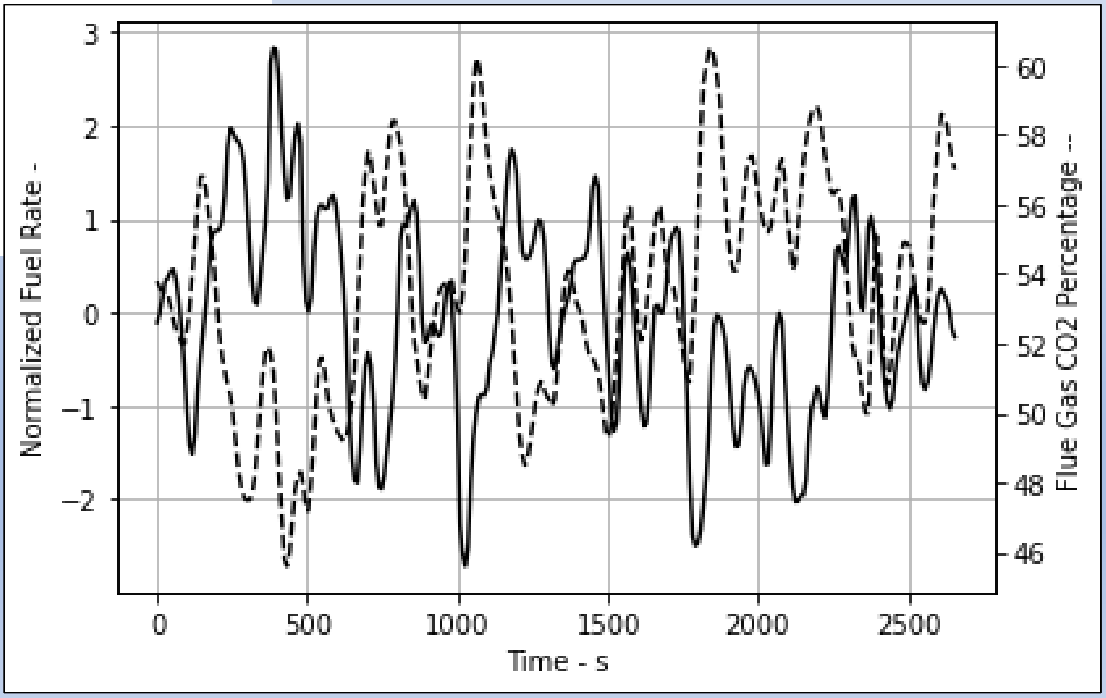
Plotting with Matplotlib's pyplot

Plots of data – plotting more than one series right and left axes

Twinaxes.py

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

t,fuel,co2 = np.loadtxt('furnacedata.txt',delimiter='\t',unpack=True)

plt.plot(t,fuel,c='k')
plt.grid()
plt.xlabel('Time - s')
plt.ylabel('Normalized Fuel Rate -')

plt.twinx()
plt.plot(t,co2,c='k',ls='--')
plt.ylabel('Flue Gas CO2 Percentage --')
```

A disadvantage here is the lack of a legend.



Uses the **twinx** command to shift to the right axis.

Plotting with Matplotlib's pyplot

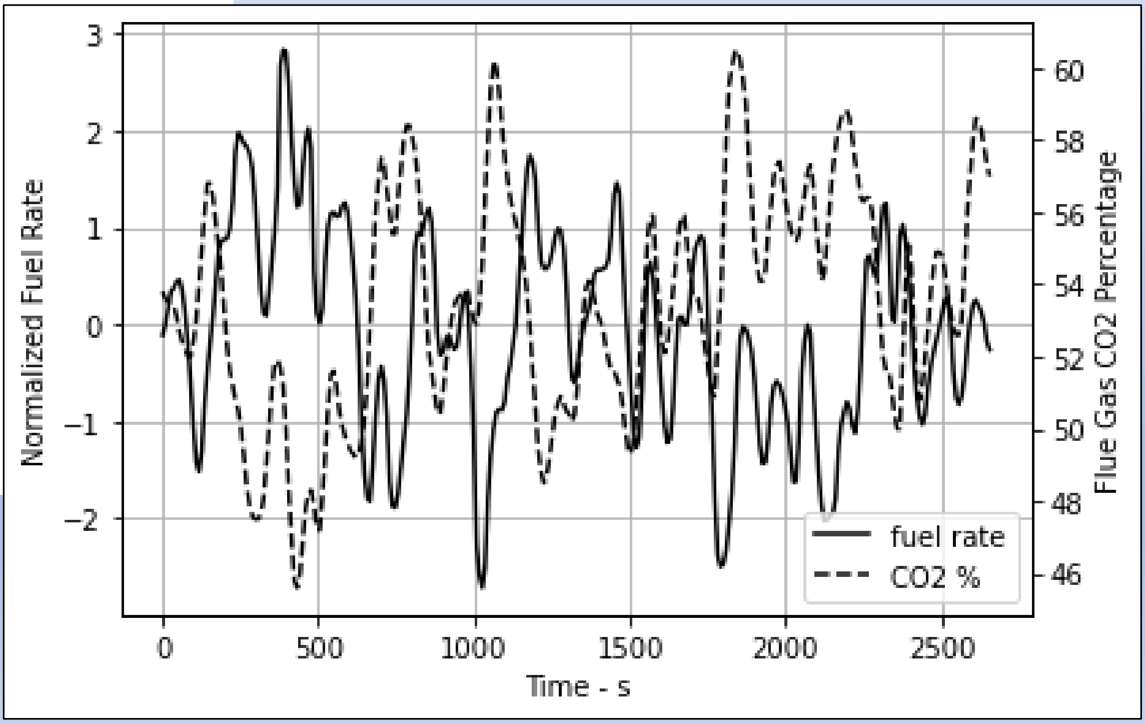
Plots of data – plotting more than one series

right and left axes

Adding a legend is possible, but it is complicated. Sorry about that.

Twinaxeswithlegend.py

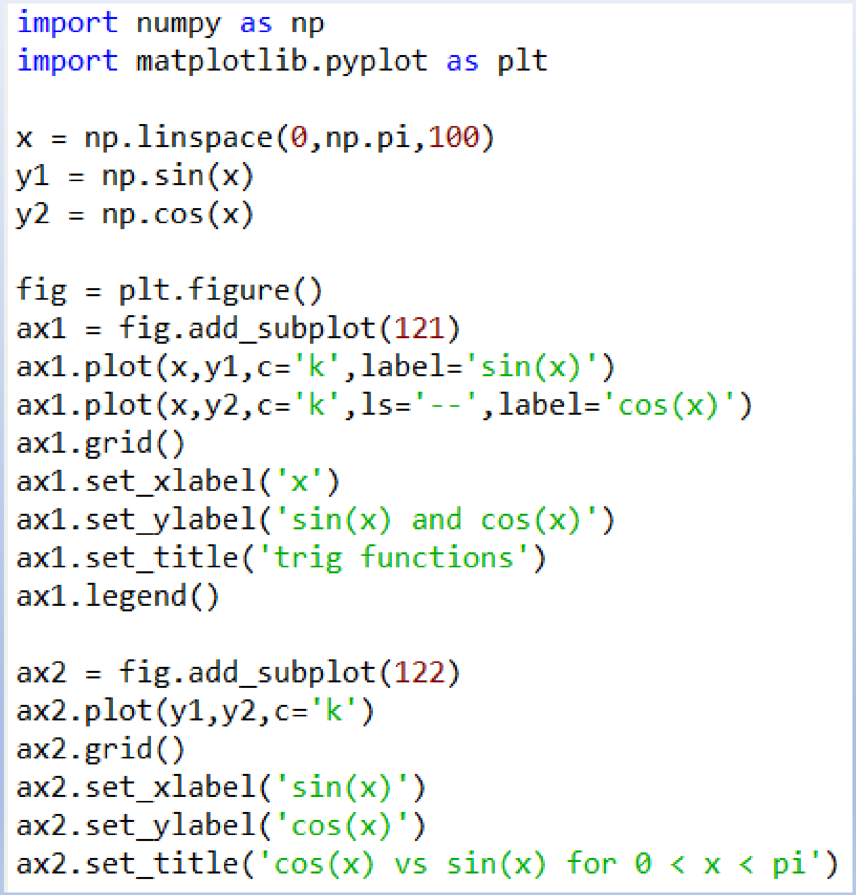
```python
import numpy as np
import matplotlib.pyplot as plt

t,fuel,co2 = np.loadtxt('furnacedata.txt',delimiter='\t',unpack=True)

curve1 = plt.plot(t,fuel,c='k',label='fuel rate')
plt.grid()
plt.xlabel('Time - s')
plt.ylabel('Normalized Fuel Rate')

plt.twinx()
curve2 = plt.plot(t,co2,c='k',ls='--',label='CO2 %')
plt.ylabel('Flue Gas CO2 Percentage')

curves = curve1 + curve2
labels = []
for curve in curves:
    labels.append(curve.get_label())
plt.legend(curves,labels,loc='lower right')
```



Plotting with Matplotlib's pyplot

Using figure window objects

figure_windows_example_starter.py



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

x = np.linspace(0,np.pi,100)
y1 = np.sin(x)
y2 = np.cos(x)

fig = plt.figure()
ax1 = fig.add_subplot(121)
ax1.plot(x,y1,c='k',label='sin(x)')
ax1.plot(x,y2,c='k',ls='--',label='cos(x)')
ax1.grid()
ax1.set_xlabel('x')
ax1.set_ylabel('sin(x) and cos(x)')
ax1.set_title('trig functions')
ax1.legend()

ax2 = fig.add_subplot(122)
ax2.plot(y1,y2,c='k')
ax2.grid()
ax2.set_xlabel('sin(x)')
ax2.set_ylabel('cos(x)')
ax2.set_title('cos(x) vs sin(x) for 0 < x < pi')
```

`plt.figure()` creates a figure window object

`fig.add_subplot(121)` specifies 1 row and 2 columns of subplots and IDs the first with the 3rd argument and creates an axes object, ax1

the following commands have the syntax:
object.method
e.g. `ax1.plot . . .`

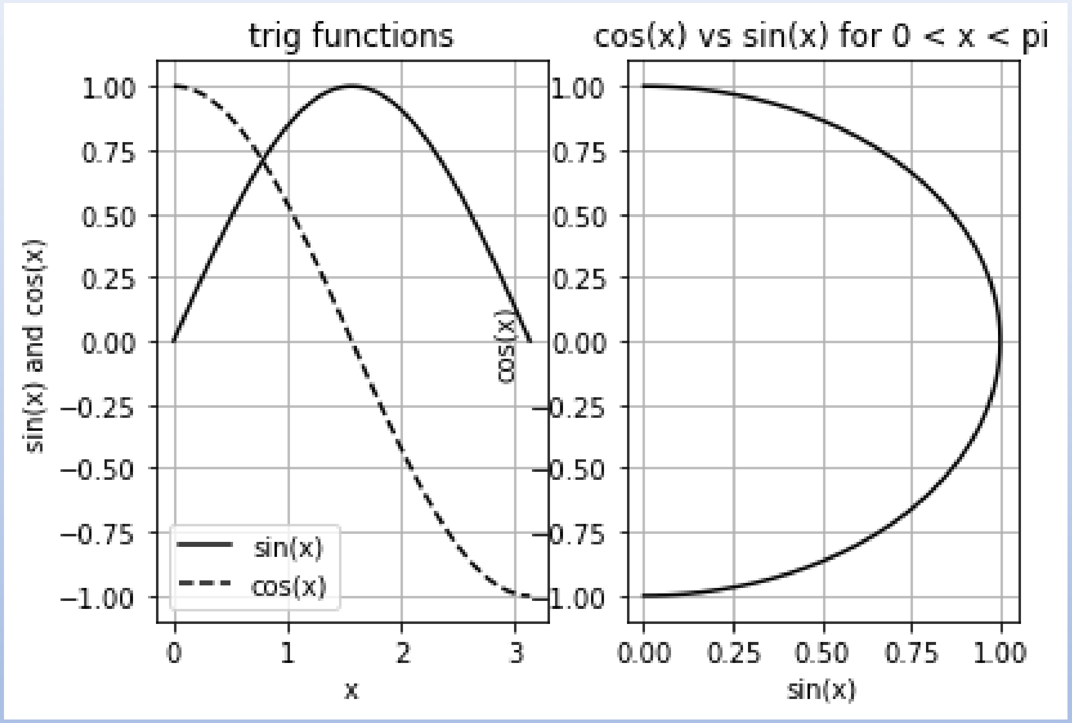
`fig.add_subplot(122)` IDs the second subplot in the 1-by-2 arrangement
and creaes an axes object, ax2

Notice the `set_` part of the label and title methods.

Plotting with Matplotlib's pyplot

Using figure window objects

Resulting plot --
subplots crowd each other

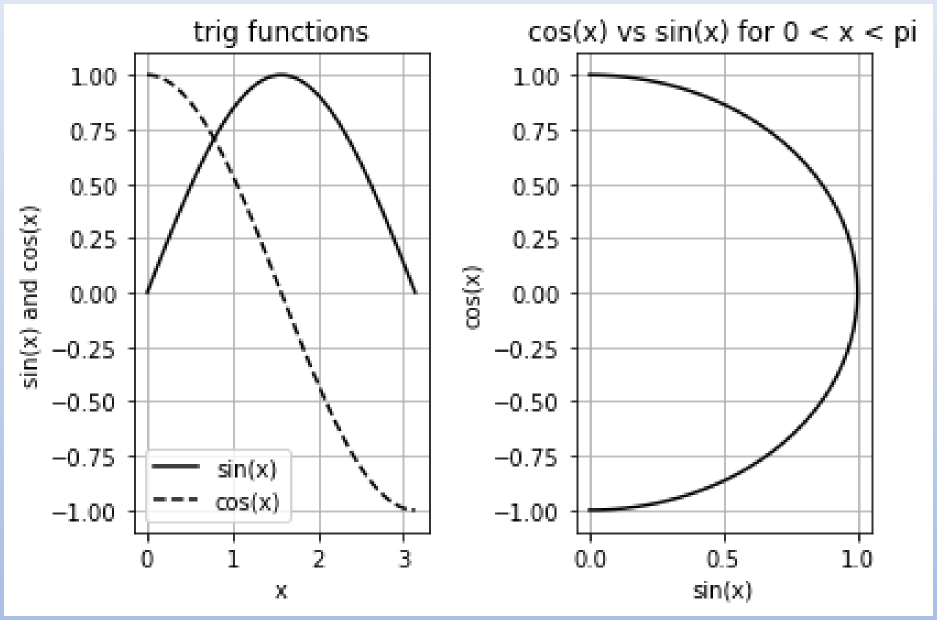


Plotting with Matplotlib's pyplot

Making adjustments to subplots

adding space between the subplots

```
fig.subplots_adjust(wspace=0.5)
```



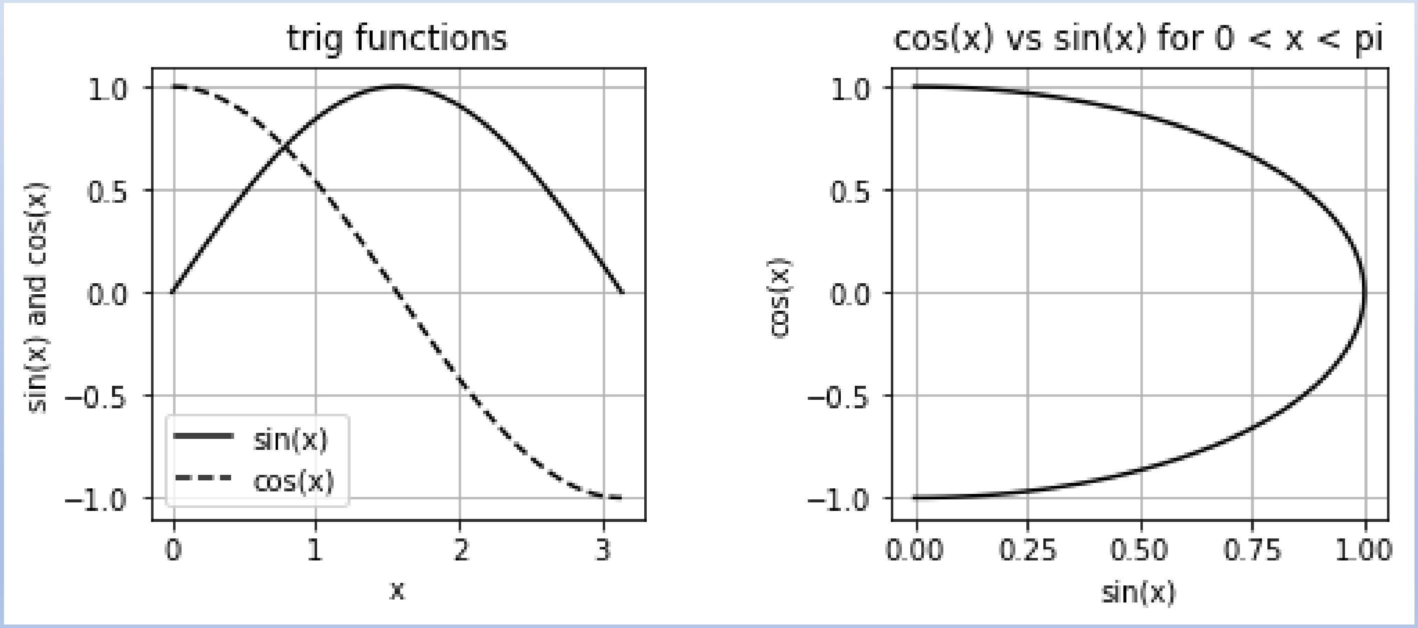
Plotting with Matplotlib’s pyplot

Making adjustments to subplots

figure_windows_example.py

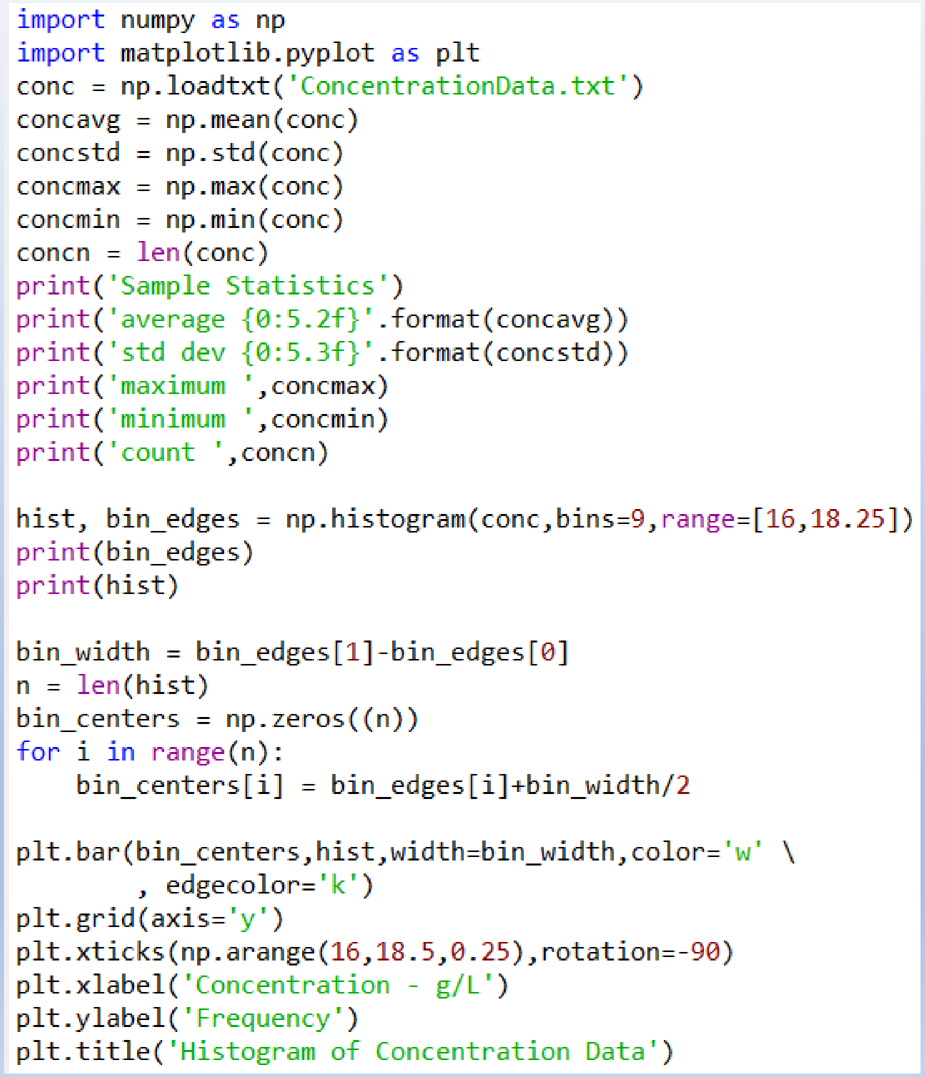
changing the aspect ratio of the figure window

```
fig = plt.figure(figsize=(8,3))
```



Plotting with Matplotlib's pyplot

Bar charts - histograms



```
import numpy as np
import matplotlib.pyplot as plt
conc = np.loadtxt('ConcentrationData.txt')
concavg = np.mean(conc)
concstd = np.std(conc)
concmax = np.max(conc)
concmin = np.min(conc)
concn = len(conc)
print('Sample Statistics')
print('average {0:5.2f}'.format(concavg))
print('std dev {0:5.3f}'.format(concstd))
print('maximum ',concmax)
print('minimum ',concmin)
print('count ',concn)

hist, bin_edges = np.histogram(conc,bins=9,range=[16,18.25])
print(bin_edges)
print(hist)

bin_width = bin_edges[1]-bin_edges[0]
n = len(hist)
bin_centers = np.zeros((n))
for i in range(n):
    bin_centers[i] = bin_edges[i]+bin_width/2

plt.bar(bin_centers,hist,width=bin_width,color='w' \
        , edgecolor='k')
plt.grid(axis='y')
plt.xticks(np.arange(16,18.5,0.25),rotation=-90)
plt.xlabel('Concentration - g/L')
plt.ylabel('Frequency')
plt.title('Histogram of Concentration Data')
```

Reads data in from a text file.
Calculates and displays sample statistics using NumPy functions and the Python **len** function.

Uses the NumPy **histogram** function.
Selects 9 bins at intervals of 0.25.
Function returns bin counts (frequencies) and bin edges.

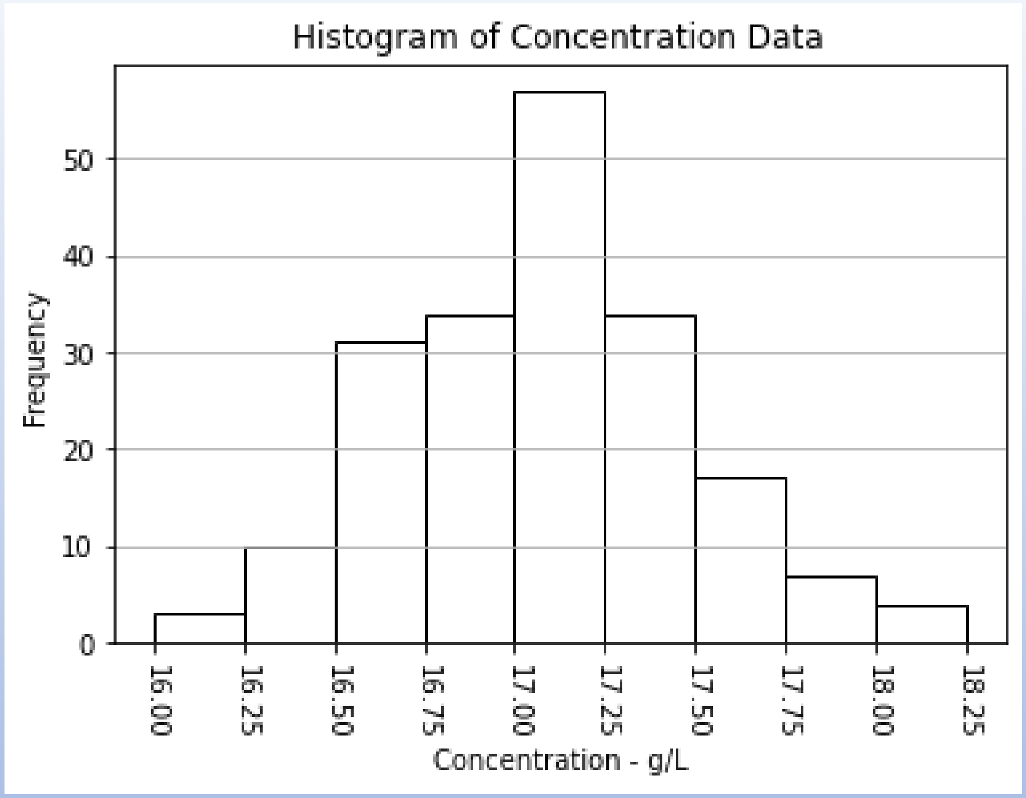
Creates an array of bin centers using a **for** loop (more on that later).
Uses the NumPy **bar** function to create the chart.
Adds grid lines in the y direction only.
Sets tick values for the x asis, rotated.

ConcentrationHistogram.py

Plotting with Matplotlib’s pyplot

Bar charts - histograms

```
Sample Statistics
average 17.06
std dev 0.398
maximum  18.2
minimum  16.1
count  197
[16.    16.25 16.5  16.75 17.    17.25 17.5  17.75 18.    18.25]
[ 3 10 31 34 57 34 17  7  4]
```



Note: With 197 data, appropriate number of bins should be in the range

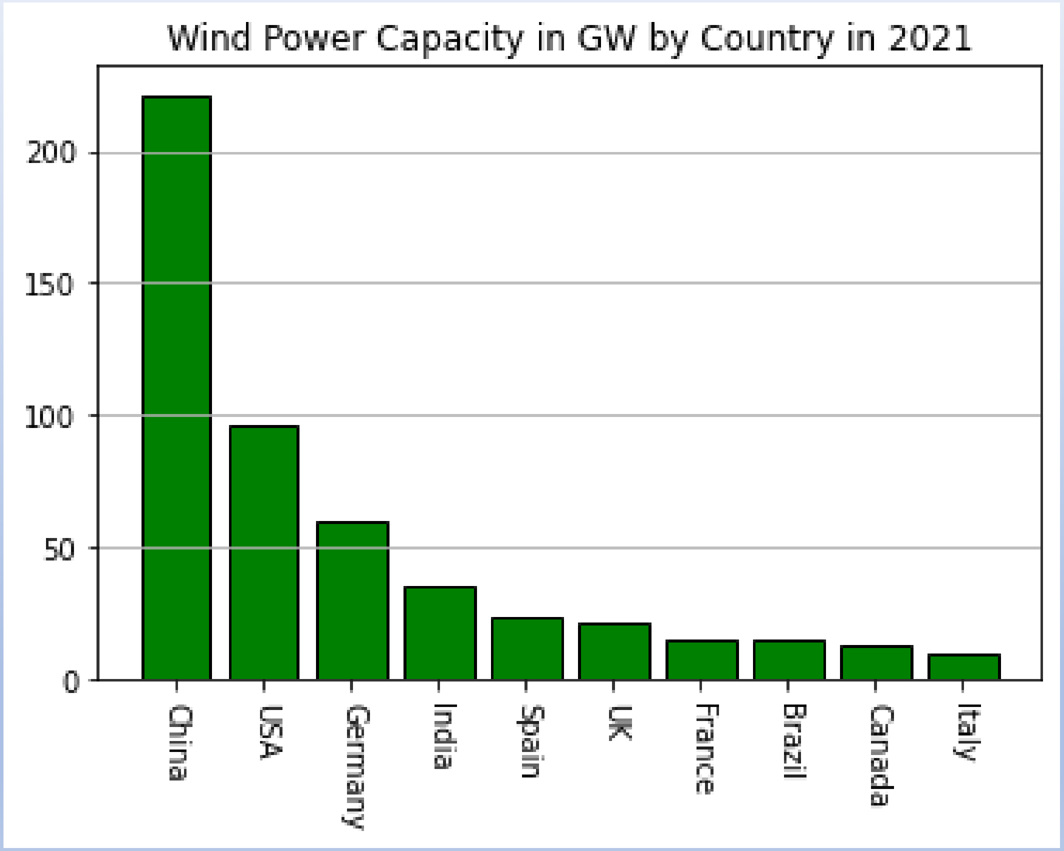
$$\text{int}\left(\log_2\left(197\right)\right)+1=8$$

$$\text{int}\left(\sqrt{197}\right)=14$$

Plotting with Matplotlib's pyplot

Bar charts – a Pareto chart based on categories

```python
import matplotlib.pyplot as plt
country = ['China','USA','Germany','India','Spain' \
          ,'UK','France','Brazil','Canada','Italy']
GW = [221.0, 96.4, 59.3, 35.0, 23.0, 21.7, 15.3 \
      , 14.5, 12.8, 10.0]
plt.bar(country,GW,edgecolor='k',facecolor='g')
plt.xticks(rotation = -90)
plt.grid(axis='y')
plt.title('Wind Power Capacity in GW by Country in 2021')
```



windpower_ParetoChart.py

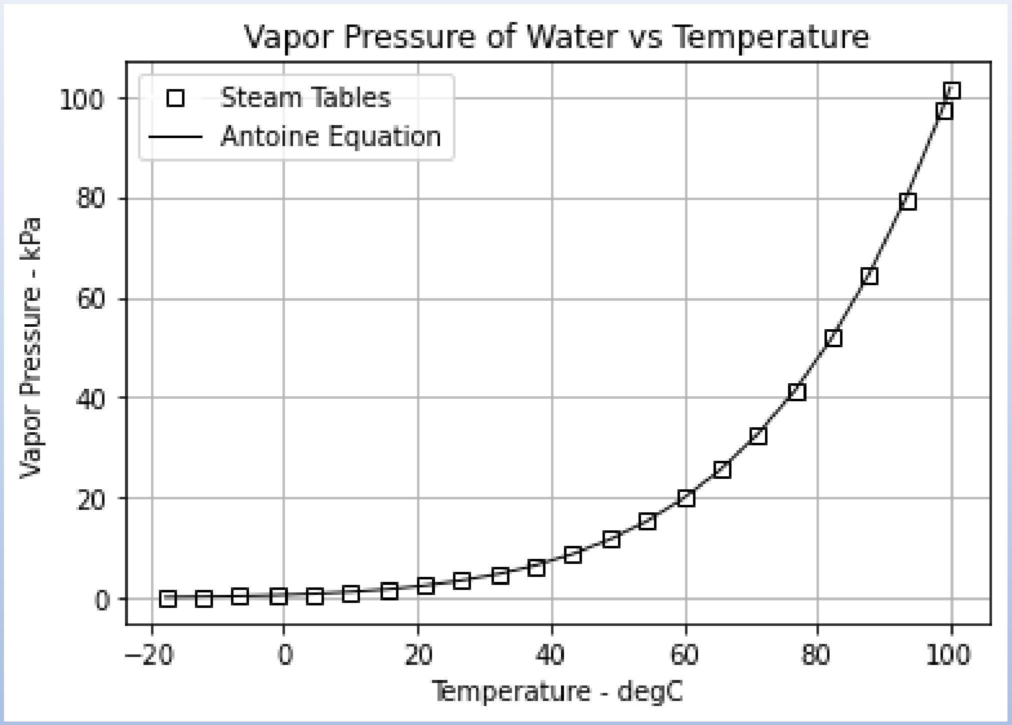
Plotting with Matplotlib’s pyplot

Plots with logarithmic scales

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

TF = np.arange(0.,220.,10.)
TF = np.append(TF,212.)
TC = (TF-32)/1.8

PV = np.array([0.1275, 0.2128, 0.3479, 0.5078, 0.8386
               , 1.227, 1.7659, 2.5023, 3.4946, 4.8136
               , 6.5016, 8.7911, 11.6712, 15.3276, 19.921
               , 25.6384, 32.692, 41.319, 51.7883, 64.3993
               , 79.4718, 97.3781, 101.3289])

A = 7.2325 ; B = 1750.3 ; C = 235.
PA = 10**(A-B/(TC+C))

plt.plot(TC,PV,c='w',marker='s',mec='k',mfc='w', \
        label = 'Steam Tables')
plt.plot(TC,PA,c='k',lw = 1.0, label = 'Antoine Equation')
plt.grid()
plt.xlabel('Temperature - degC')
plt.ylabel('Vapor Pressure - kPa')
plt.title('Vapor Pressure of Water vs Temperature')
plt.legend()
```

VaporPressureH2O_starter.py

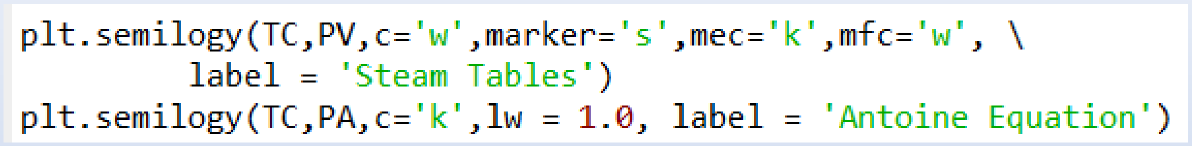
Plot with linear scales



The largest to smallest vapor pressure are a ratio of 800 to 1. Suggests a logarithmic vertical scale.

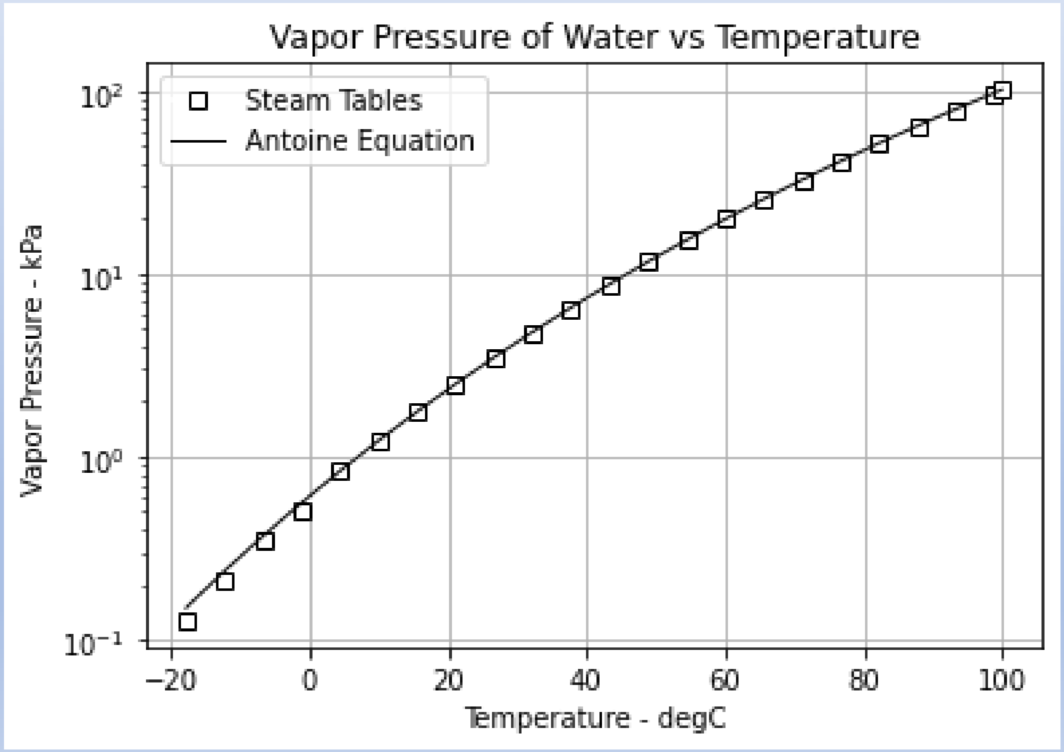
Plotting with Matplotlib's pyplot

Plots with logarithmic scales



```
plt.semilogy(TC,PV,c='w',marker='s',mec='k',mfc='w', \
         label = 'Steam Tables')
plt.semilogy(TC,PA,c='k',lw = 1.0, label = 'Antoine Equation')
```

Use the **semilogy** function instead of **plot**.



Alternate functions are
semilogx
and
loglog

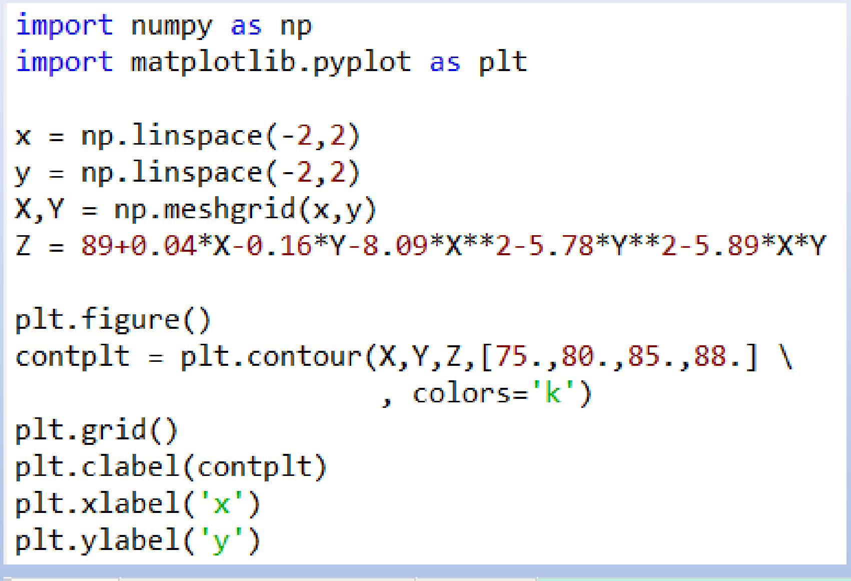
VaporPressureH2O.py

Plotting with Matplotlib’s pyplot

Contour and surface plots

ContourAnalytical.py

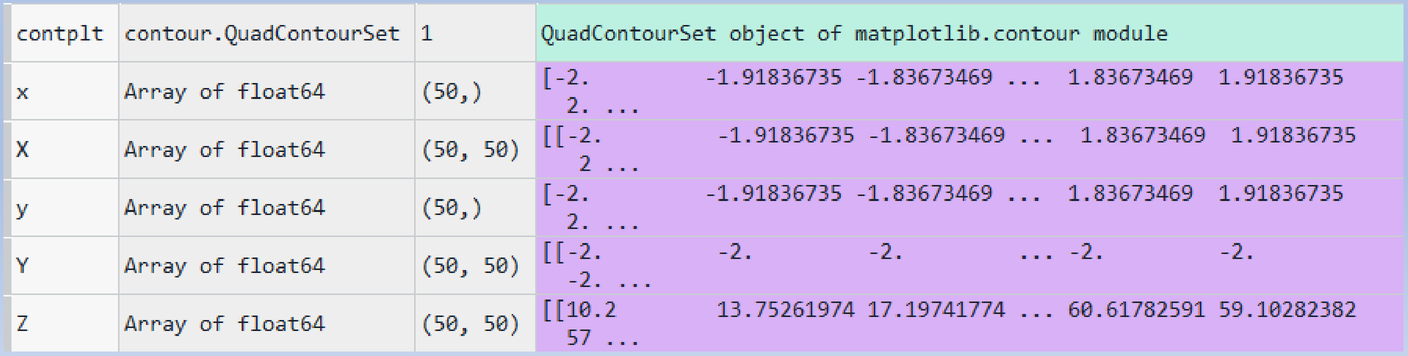
contour plot of an analytical function



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

x = np.linspace(-2,2)
y = np.linspace(-2,2)
X,Y = np.meshgrid(x,y)
Z = 89+0.04*X-0.16*Y-8.09*X**2-5.78*Y**2-5.89*X*Y

plt.figure()
contplt = plt.contour(X,Y,Z,[75.,80.,85.,88.] \
                        , colors='k')
plt.grid()
plt.clabel(contplt)
plt.xlabel('x')
plt.ylabel('y')
```

Create **x** and **y** arrays with 50 elements each.
Create a 50x50 meshgrid in **X** and **Y** arrays.
Evaluate function at each grid point.

Create contour plot with contours at 75.,...
Label the contours with their **z** values.

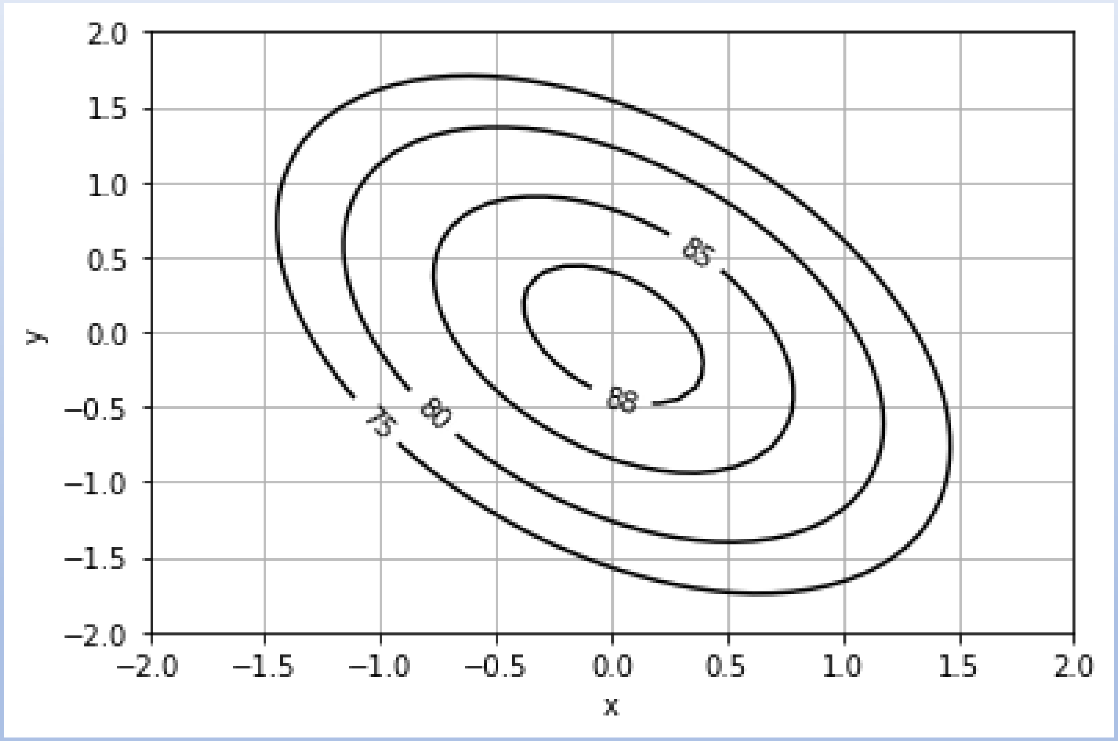


contplt	contour.QuadContourSet	1	QuadContourSet object of matplotlib.contour module
x	Array of float64	(50,)	[-2. -1.91836735 -1.83673469 ... 1.83673469 1.91836735 2. ...
X	Array of float64	(50, 50)	[[-2. -1.91836735 -1.83673469 ... 1.83673469 1.91836735 2 ...
y	Array of float64	(50,)	[-2. -1.91836735 -1.83673469 ... 1.83673469 1.91836735 2. ...
Y	Array of float64	(50, 50)	[[-2. -2. -2. ... -2. -2. -2. ...
Z	Array of float64	(50, 50)	[[10.2 13.75261974 17.19741774 ... 60.61782591 59.10282382 57 ...

Plotting with Matplotlib’s pyplot

Contour and surface plots

contour plot of an analytical function



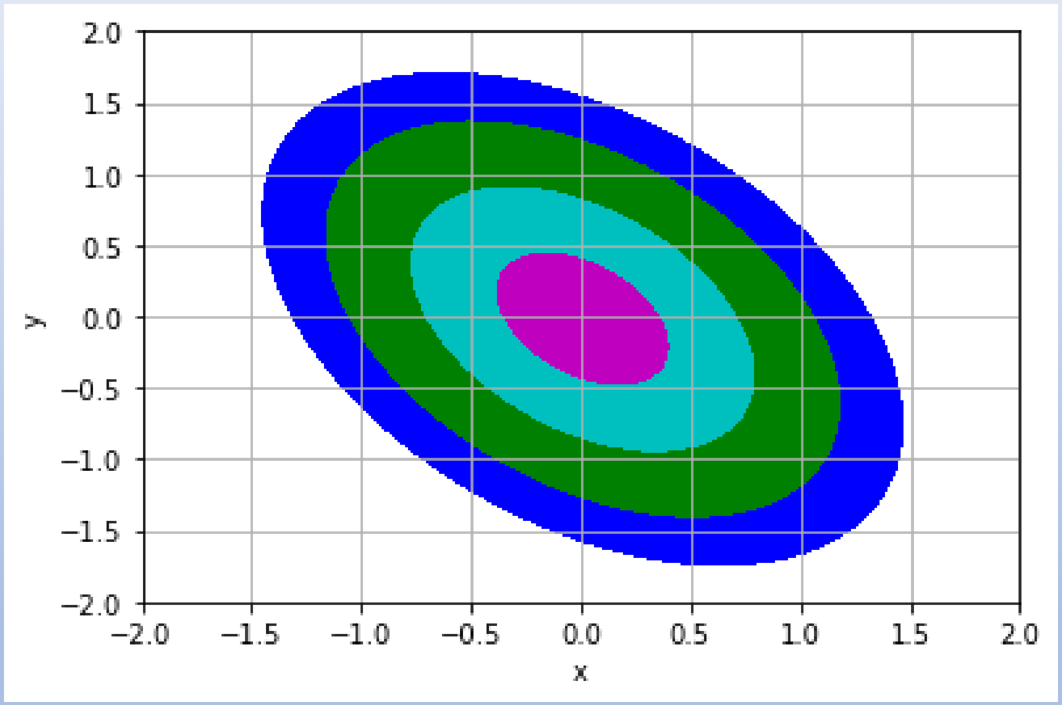
Plotting with Matplotlib’s pyplot

Contour and surface plots

filled contour plot of an analytical function

ContourFilledAnalytical1.py

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

x = np.linspace(-2,2)
y = np.linspace(-2,2)
X,Y = np.meshgrid(x,y)
Z = 89+0.04*X-0.16*Y-8.09*X**2-5.78*Y**2-5.89*X*Y

plt.figure()
contplt = plt.contourf(X,Y,Z,[75.,80.,85.,88.,92.] \
                      , colors=['b','g','c','m'])
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
```



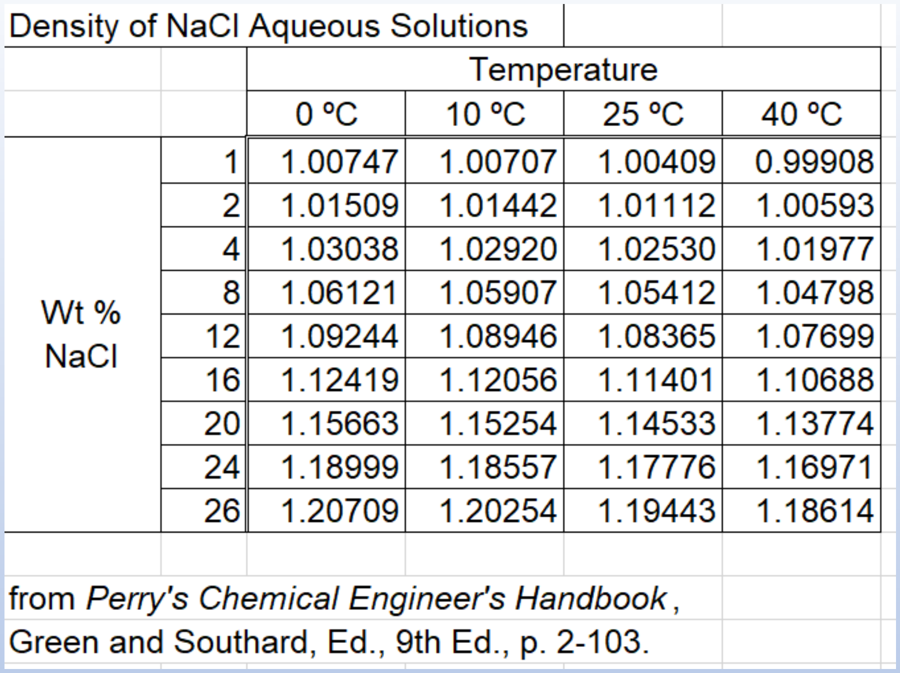
Plotting with Matplotlib’s pyplot

Contour and surface plots

contour plot based on data

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

wtpct = np.array([1.,2.,4.,8.,12.,16.,20.,24.,26.])
temp = np.array([0.,10.,25.,40.,60.,80.,100.])
W, T = np.meshgrid(wtpct,temp)
D = np.loadtxt('SaltDensity.csv',delimiter=',',unpack=True)

plt.figure()
contplt = plt.contour(W,T,D,colors='k')
plt.grid()
plt.clabel(contplt)
plt.xlabel('Concentration - wt%')
plt.ylabel('Temperature - degC')
plt.title('Density of NaCl Solutions in gm/cc')
```

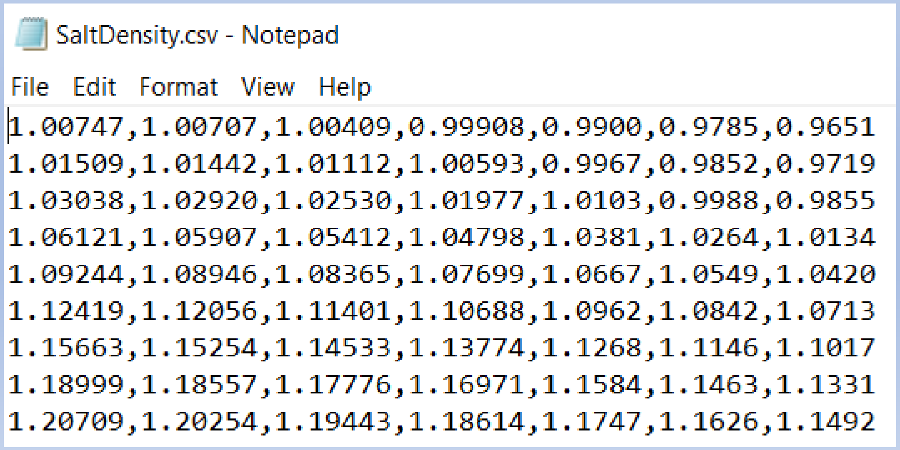
NaClDensityContourPlot.py



Density of NaCl Aqueous Solutions

		Temperature			
		0 °C	10 °C	25 °C	40 °C
Wt % NaCl	1	1.00747	1.00707	1.00409	0.99908
	2	1.01509	1.01442	1.01112	1.00593
	4	1.03038	1.02920	1.02530	1.01977
	8	1.06121	1.05907	1.05412	1.04798
	12	1.09244	1.08946	1.08365	1.07699
	16	1.12419	1.12056	1.11401	1.10688
	20	1.15663	1.15254	1.14533	1.13774
	24	1.18999	1.18557	1.17776	1.16971
	26	1.20709	1.20254	1.19443	1.18614

from *Perry's Chemical Engineer's Handbook*, Green and Southard, Ed., 9th Ed., p. 2-103.



SaltDensity.csv - Notepad

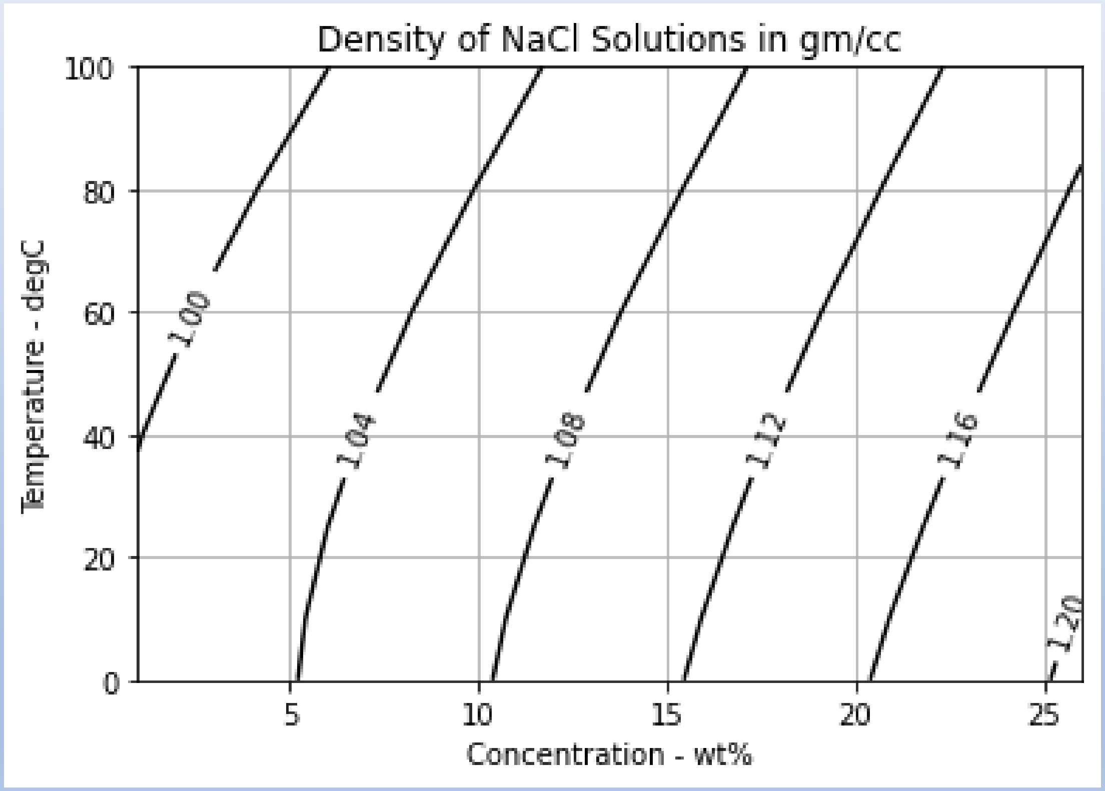
File Edit Format View Help

```
1.00747,1.00707,1.00409,0.99908,0.9900,0.9785,0.9651
1.01509,1.01442,1.01112,1.00593,0.9967,0.9852,0.9719
1.03038,1.02920,1.02530,1.01977,1.0103,0.9988,0.9855
1.06121,1.05907,1.05412,1.04798,1.0381,1.0264,1.0134
1.09244,1.08946,1.08365,1.07699,1.0667,1.0549,1.0420
1.12419,1.12056,1.11401,1.10688,1.0962,1.0842,1.0713
1.15663,1.15254,1.14533,1.13774,1.1268,1.1146,1.1017
1.18999,1.18557,1.17776,1.16971,1.1584,1.1463,1.1331
1.20709,1.20254,1.19443,1.18614,1.1747,1.1626,1.1492
```

Plotting with Matplotlib's pyplot

Contour and surface plots

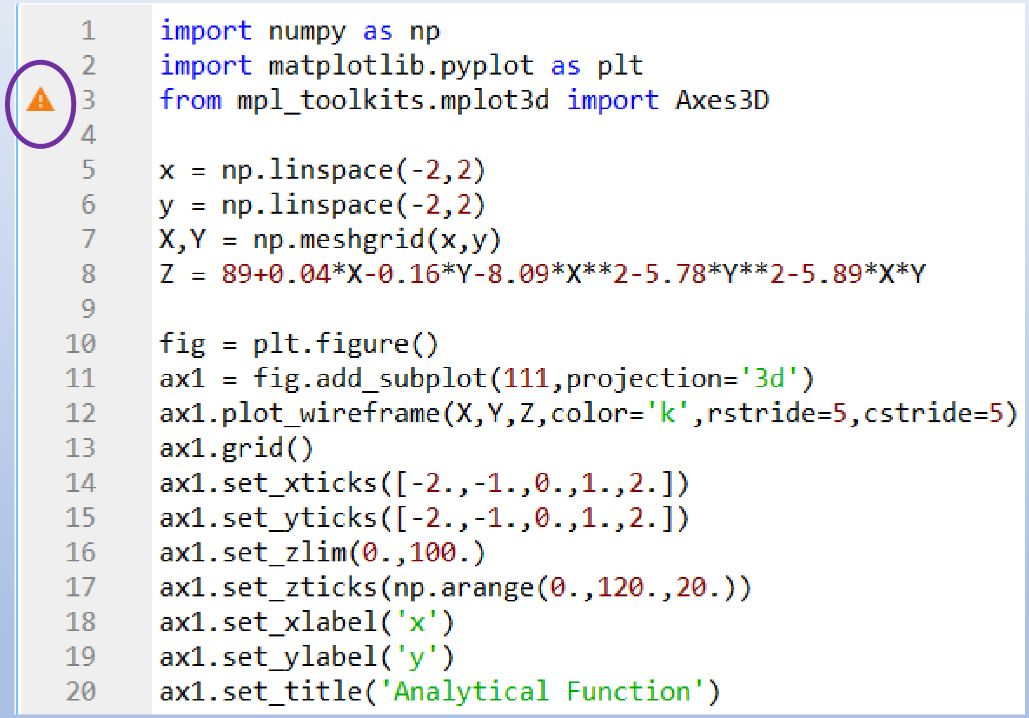
contour plot based on data



Plotting with Matplotlib’s pyplot

Contour and surface plots

wireframe plot based on an analytical function



```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

x = np.linspace(-2,2)
y = np.linspace(-2,2)
X,Y = np.meshgrid(x,y)
Z = 89+0.04*X-0.16*Y-8.09*X**2-5.78*Y**2-5.89*X*Y

fig = plt.figure()
ax1 = fig.add_subplot(111,projection='3d')
ax1.plot_wireframe(X,Y,Z,color='k',rstride=5,cstride=5)
ax1.grid()
ax1.set_xticks([-2.,-1.,0.,1.,2.])
ax1.set_yticks([-2.,-1.,0.,1.,2.])
ax1.set_zlim(0.,100.)
ax1.set_zticks(np.arange(0.,120.,20.))
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('Analytical Function')
```

Warning here is false.
This module is required.
Sorry.

stride arguments indicate data count intervals for mesh lines

SurfaceMeshAnalytical.py

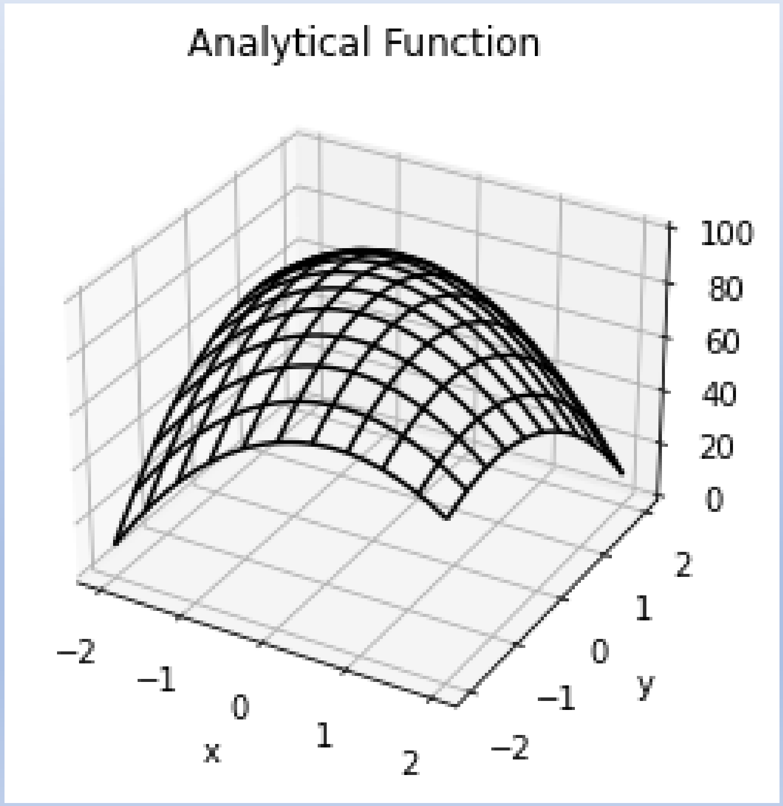
Plotting with Matplotlib’s pyplot

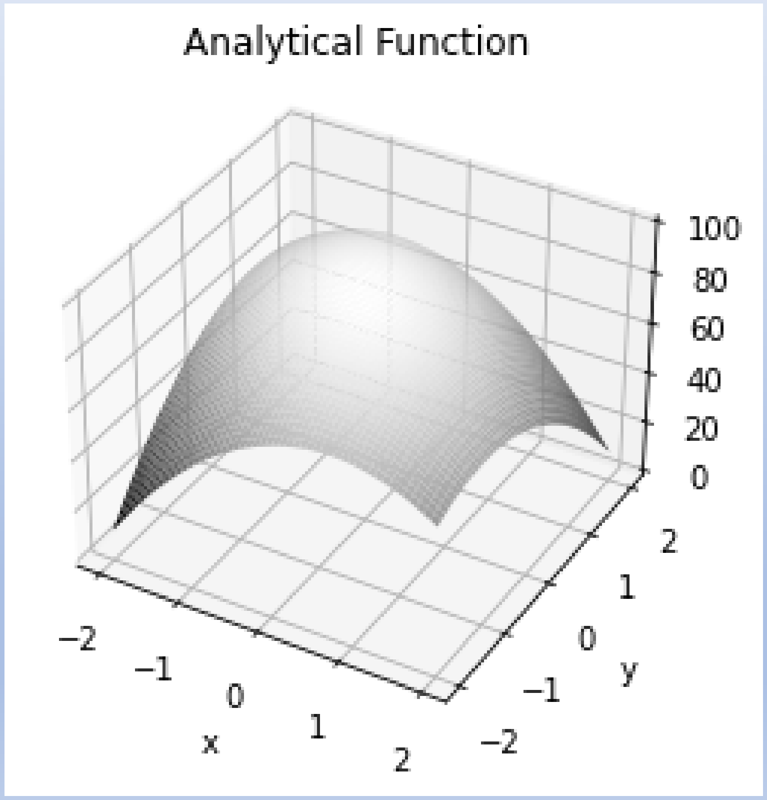
Contour and surface plots

wireframe plot based on
an analytical function

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

```
ax1.plot_surface(X,Y,Z,cmap=cm.gray)
```

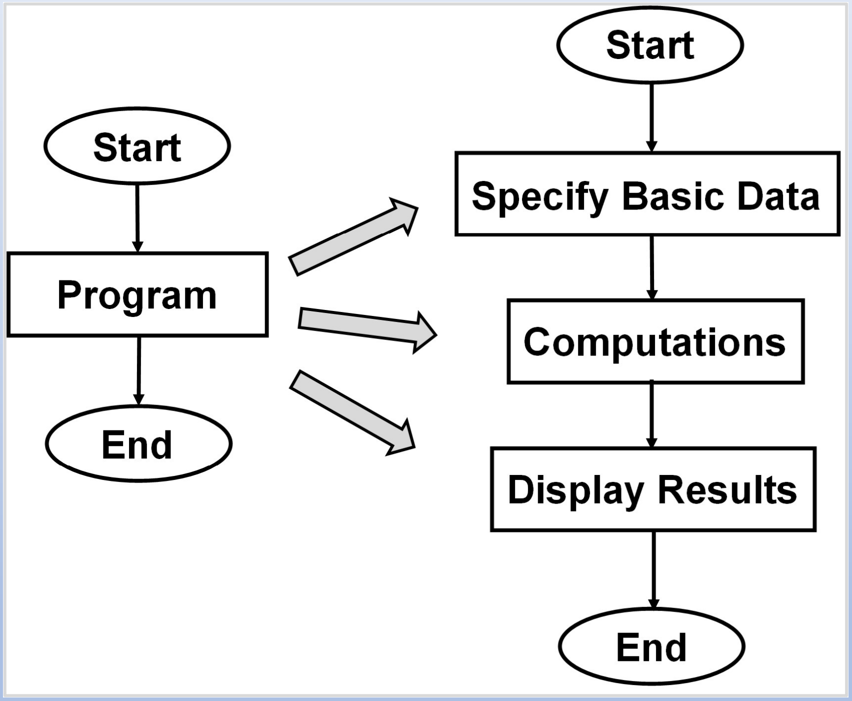




SurfaceAnalytical.py

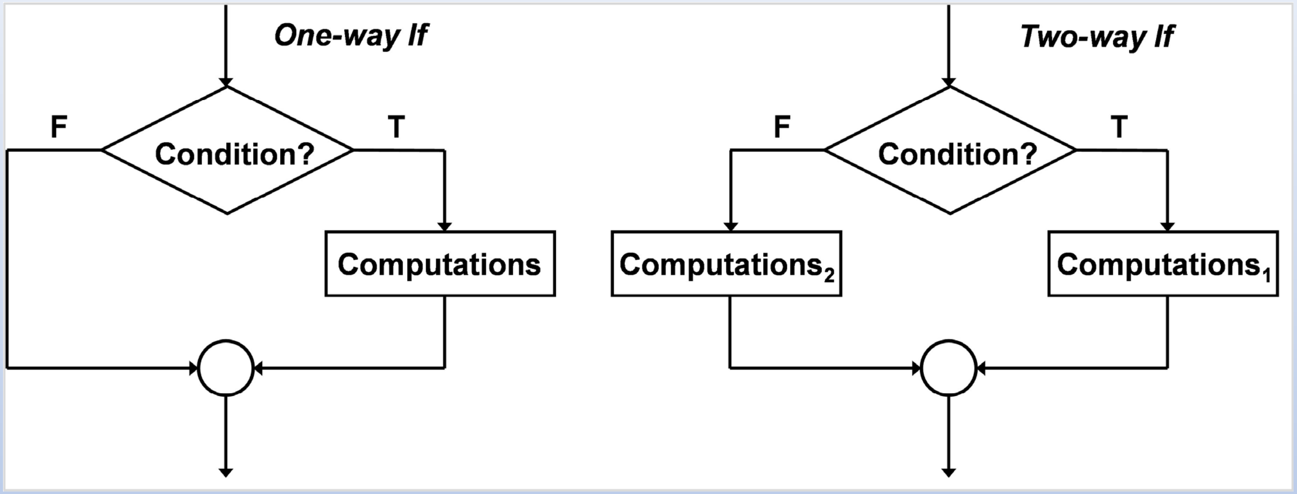
Program Structure and User-defined Functions

Overall program structure – sequential flow



Program Structure and User-defined Functions

Selection structures: one-way and two-way if



```
if condition:
    statements1
else:
    statements2
```

```
if x < 0:
    Sgnx2 = - x**2
else:
    Sgnx2 = x**2
```

```
if condition:
    statements
```

```
Sgnx2 = x**2
if x < 0:
    Sgnx2 = -Sgnx2
```

indent must be 4 spaces

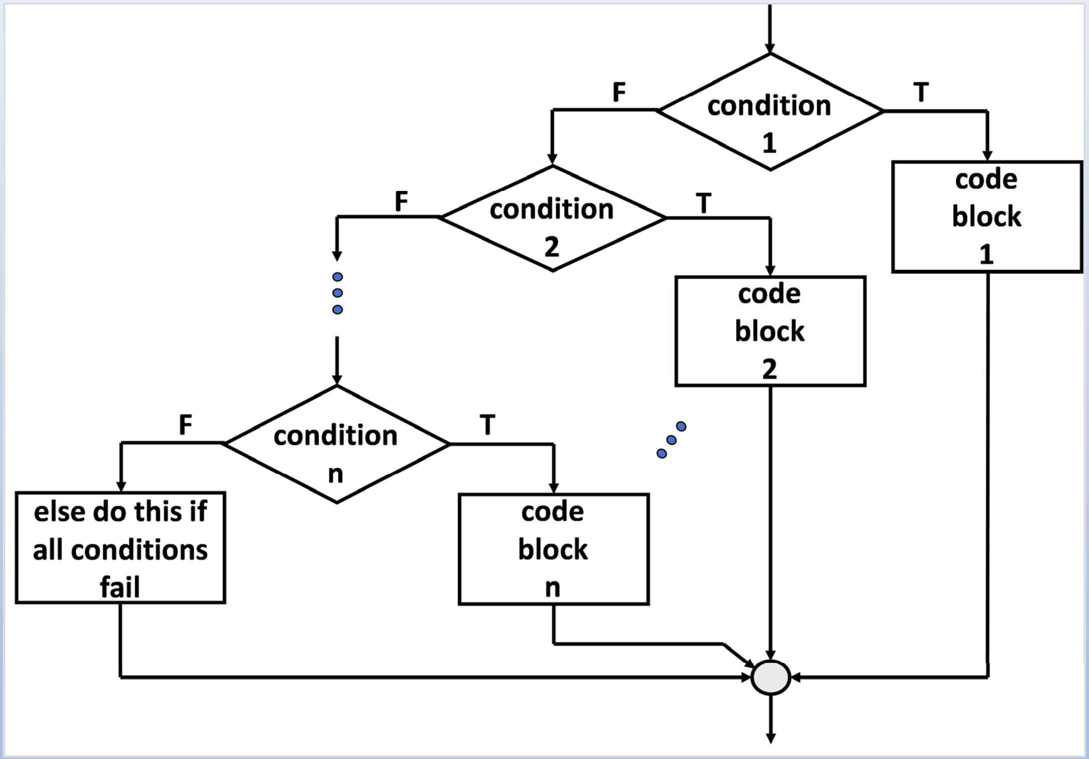
or one-line version

```
Sgnx2 = x**2
if x < 0: Sgnx2 = -Sgnx2
```

Note: There is no end statement. The structure is terminated when the indentation is removed.

Program Structure and User-defined Functions

Selection structures: multialternative if



```
if condition1:
    statements1
elif condition2:
    statements2
•
•
elif condition_n:
    statements_n
else:
    else statements
```

The **else** clause is not required.

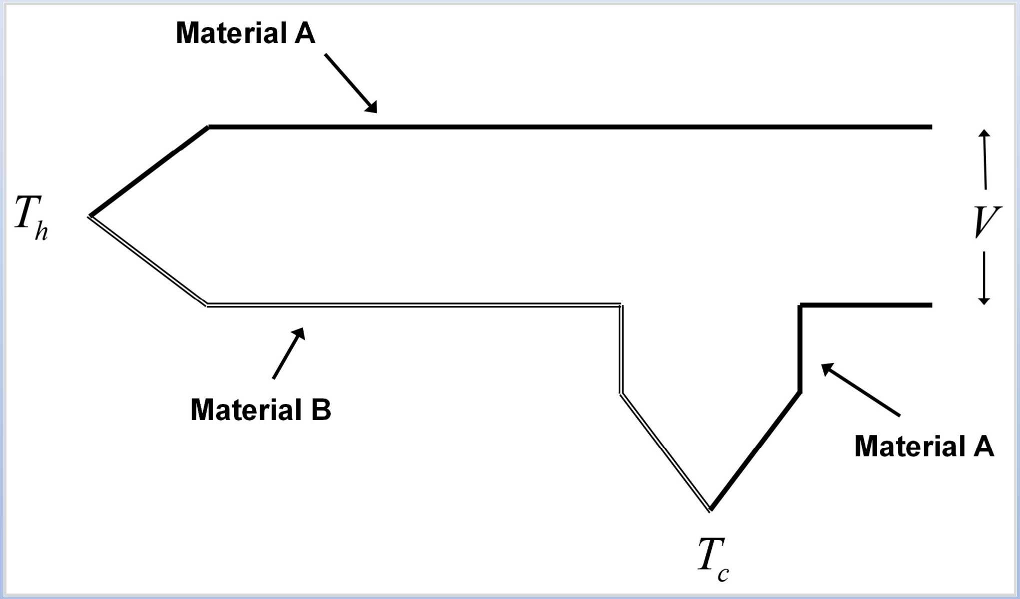
Program Structure and User-defined Functions

TypeJ_TC.py

Selection structures: multialternative if

Example: Type J thermocouple, temperature vs voltage

$V \propto (T_h - T_c) = dT$ T_c : reference junction temperature, known or controlled



For $-8.10 \le V \le 0.0\ mV$:

$$dT = 19.53V - 1.229V^2 - 1.075V^3 - 0.5909V^4 - 0.1726V^5 - 0.02813V^6 - 2.396\times10^{-3}V^7 - 8.382\times10^{-5}V^8$$

For $0.0 \le V \le 42.92\ mV$:

$$dT = 19.78V - 0.2001V^2 + 0.01037V^3 - 2.550\times10^{-4}V^4 + 3.585\times10^{-6}V^5 - 5.344\times10^{-8}V^6 + 5.100\times10^{-10}V^7$$

For $42.92 \le V \le 69.55\ mV$:

$$dT = -3114.0 + 300.5V - 9.948V^2 + 0.1703V^3 - 1.430\times10^{-3}V^4 + 4.739\times10^{-6}V^5$$

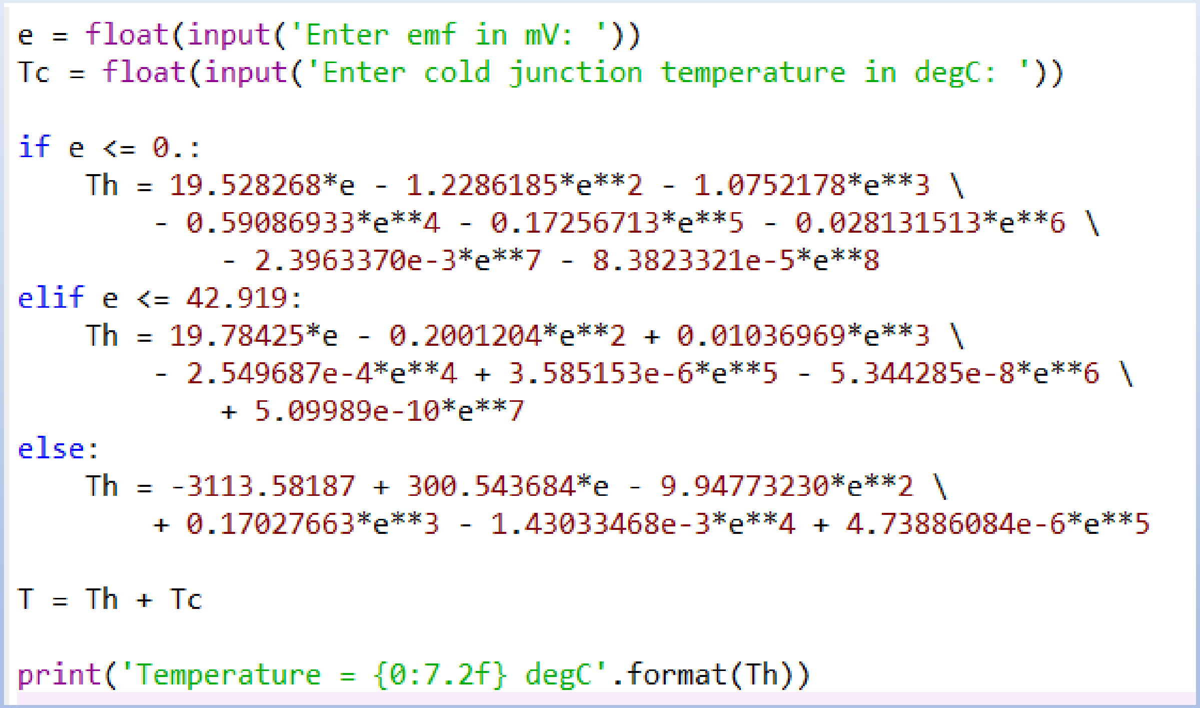
Thermocouples are the most common devices for industrial temperature measurement.

Type J is the most common thermocouple. Type J materials are iron and constantan. Constantan is an alloy, 55% Cu and 45% Ni.

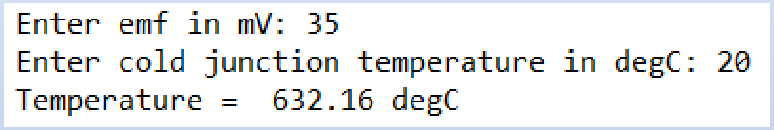
Program Structure and User-defined Functions

Selection structures: multialternative if

Example: Type J thermocouple, temperature vs voltage



```
e = float(input('Enter emf in mV: '))
Tc = float(input('Enter cold junction temperature in degC: '))

if e <= 0.:
    Th = 19.528268*e - 1.2286185*e**2 - 1.0752178*e**3 \
        - 0.59086933*e**4 - 0.17256713*e**5 - 0.028131513*e**6 \
            - 2.3963370e-3*e**7 - 8.3823321e-5*e**8
elif e <= 42.919:
    Th = 19.78425*e - 0.2001204*e**2 + 0.01036969*e**3 \
        - 2.549687e-4*e**4 + 3.585153e-6*e**5 - 5.344285e-8*e**6 \
            + 5.09989e-10*e**7
else:
    Th = -3113.58187 + 300.543684*e - 9.94773230*e**2 \
        + 0.17027663*e**3 - 1.43033468e-3*e**4 + 4.73886084e-6*e**5

T = Th + Tc

print('Temperature = {0:7.2f} degC'.format(Th))
```

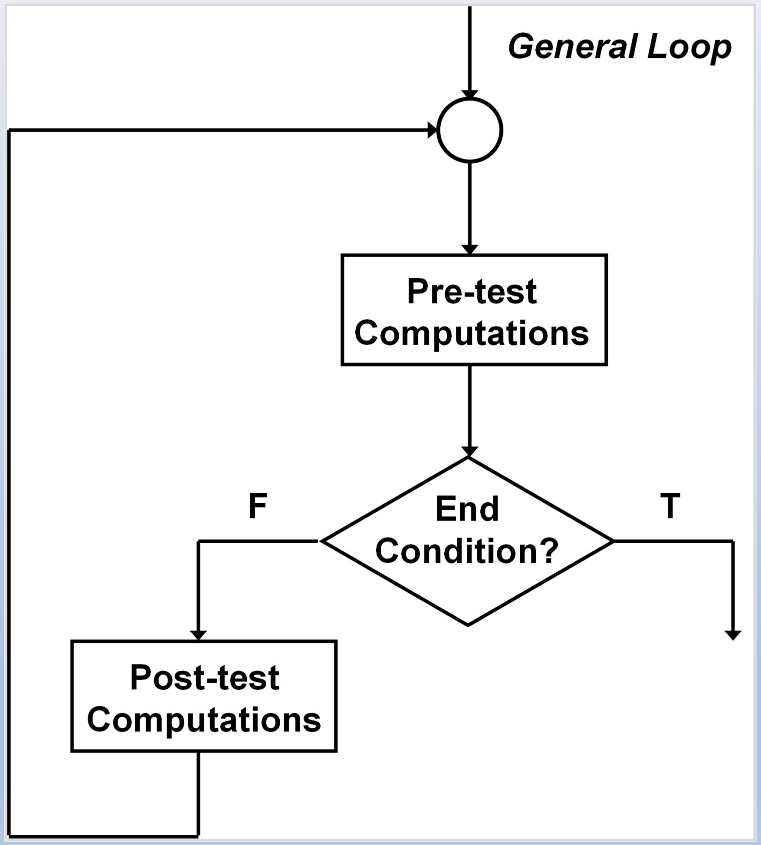


```
Enter emf in mV: 35
Enter cold junction temperature in degC: 20
Temperature =  632.16 degC
```

Note: This script doesn’t protect adequately for erroneous user input. We will consider that later.

Program Structure and User-defined Functions

Repetition structures: the general loop



Python implementation – **while** loop

```
while condition:
    loop statements
```

Loop repeats as long as condition is True.
Adapt this structure for the general loop:

```
while True:
    pre-test statements
    if condition: break
    post-test statements
```

Pre-test or post-test statements may be absent (not both!).
Similar adaptation as with MATLAB.

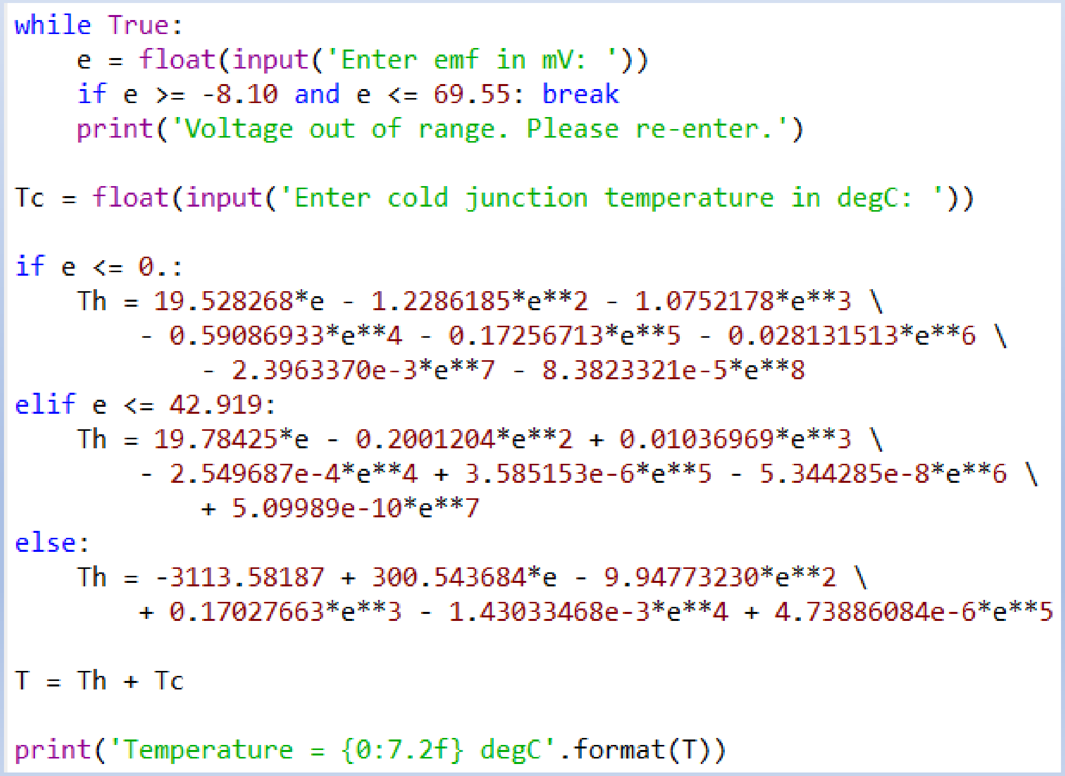
Program Structure and User-defined Functions

Repetition structures: the general loop

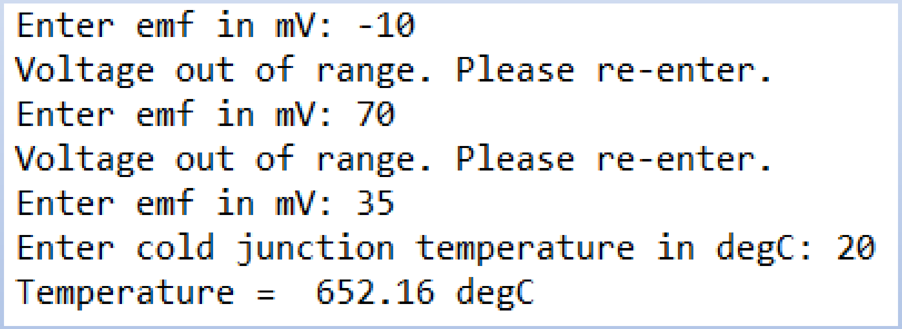
Typical application: input validation

```
while True:
    acquire input value
    if input is valid: break
    print error/corrective message
```

Input validation for
the thermocouple calculation



```python
while True:
    e = float(input('Enter emf in mV: '))
    if e >= -8.10 and e <= 69.55: break
    print('Voltage out of range. Please re-enter.')

Tc = float(input('Enter cold junction temperature in degC: '))

if e <= 0.:
    Th = 19.528268*e - 1.2286185*e**2 - 1.0752178*e**3 \
        - 0.59086933*e**4 - 0.17256713*e**5 - 0.028131513*e**6 \
            - 2.3963370e-3*e**7 - 8.3823321e-5*e**8
elif e <= 42.919:
    Th = 19.78425*e - 0.2001204*e**2 + 0.01036969*e**3 \
        - 2.549687e-4*e**4 + 3.585153e-6*e**5 - 5.344285e-8*e**6 \
            + 5.09989e-10*e**7
else:
    Th = -3113.58187 + 300.543684*e - 9.94773230*e**2 \
        + 0.17027663*e**3 - 1.43033468e-3*e**4 + 4.73886084e-6*e**5

T = Th + Tc

print('Temperature = {0:7.2f} degC'.format(T))
```

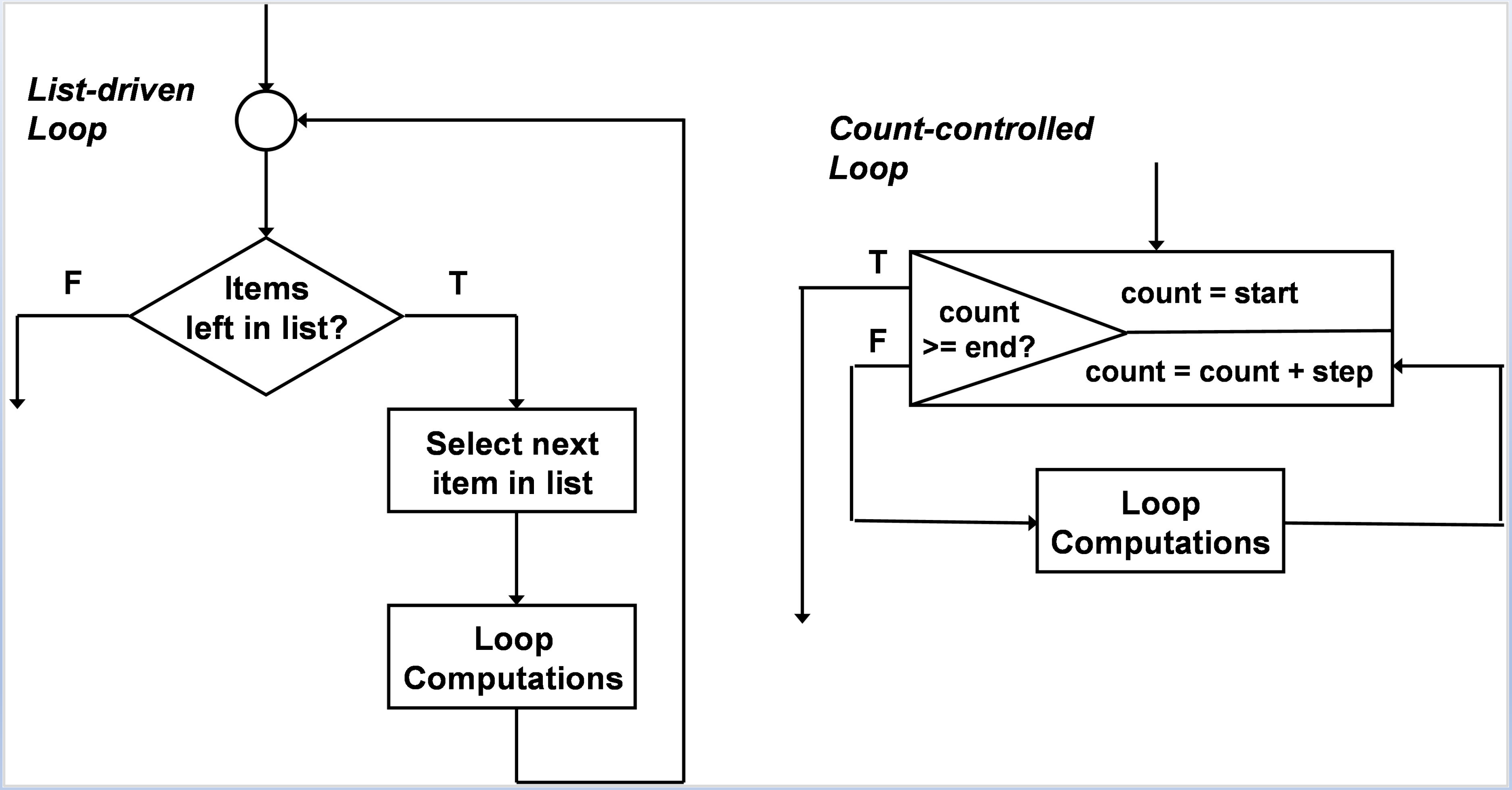


```
Enter emf in mV: -10
Voltage out of range. Please re-enter.
Enter emf in mV: 70
Voltage out of range. Please re-enter.
Enter emf in mV: 35
Enter cold junction temperature in degC: 20
Temperature =  652.16 degC
```

TypeJ_TC_with_input_validation.py

Program Structure and User-defined Functions

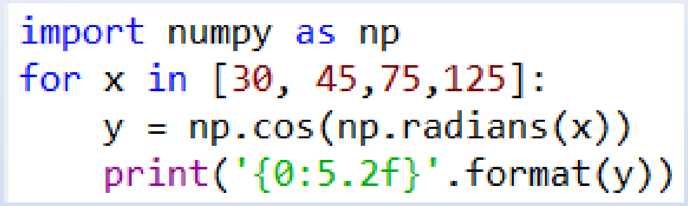
Repetition structures: list-driven and count-controlled loops



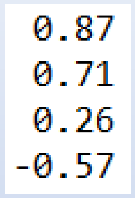
Program Structure and User-defined Functions

Repetition structures: list-driven loop

```
for variable in [list]:
    loop statements
```



```
import numpy as np
for x in [30, 45,75,125]:
    y = np.cos(np.radians(x))
    print('{0:5.2f}'.format(y))
```



```
 0.87
 0.71
 0.26
-0.57
```

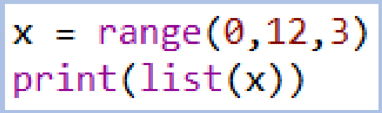
SimpleListLoop.py

Use of the **range** type to generate a list

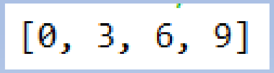
`range(start,end,step)`

If start left out, = 0

If step left out, = 1



```
x = range(0,12,3)
print(list(x))
```



```
[0, 3, 6, 9]
```

last item, 12 left out

Examples:

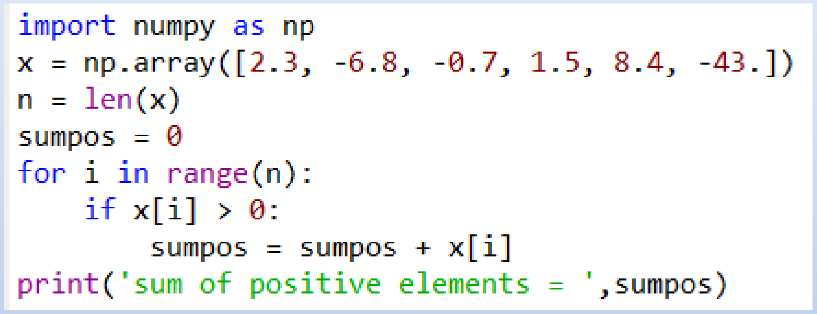
`range(10)`	10 integers 0,1,2,...,9
`range(n)`	n: integer variable, list from 0 to n-1
`range(1,11)`	10 integers from 1 to 10
`range(0,11,3)`	list of integers, [0, 3, 6, 9]

Program Structure and User-defined Functions

Repetition structures: count-controlled loop

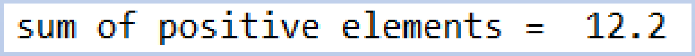
Examples:

for_loop_example.py

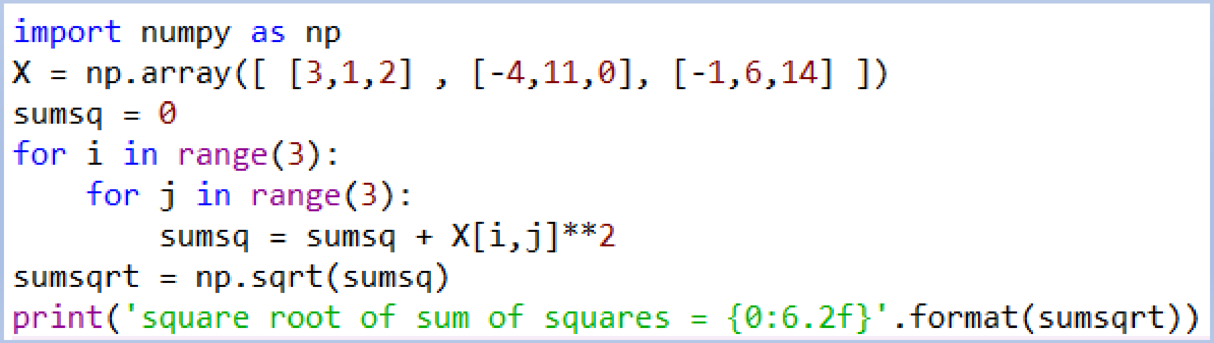


```python
import numpy as np
x = np.array([2.3, -6.8, -0.7, 1.5, 8.4, -43.])
n = len(x)
sumpos = 0
for i in range(n):
    if x[i] > 0:
        sumpos = sumpos + x[i]
print('sum of positive elements = ',sumpos)
```

Use of **for** loop variable as an array index (or subscript)



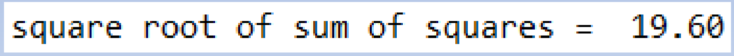
```
sum of positive elements =  12.2
```



```python
import numpy as np
X = np.array([ [3,1,2] , [-4,11,0], [-1,6,14] ])
sumsq = 0
for i in range(3):
    for j in range(3):
        sumsq = sumsq + X[i,j]**2
sumsqrt = np.sqrt(sumsq)
print('square root of sum of squares = {0:6.2f}'.format(sumsqrt))
```

Nested for loops

i is the row index
j is the column index
for the 3x3 array **X**



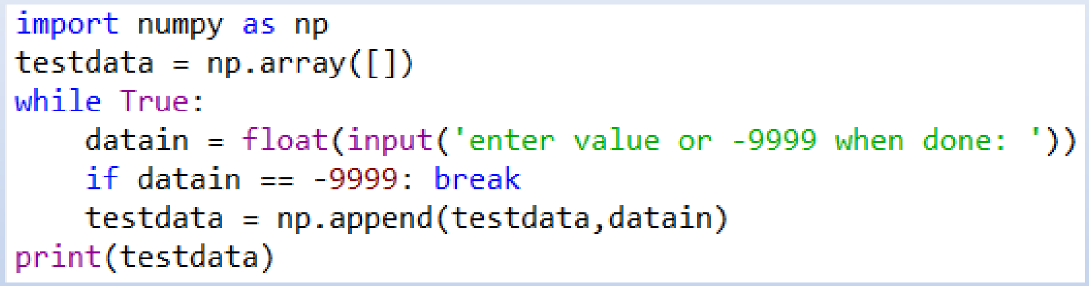
```
square root of sum of squares =  19.60
```

nested_for_loops_example.py

Program Structure and User-defined Functions

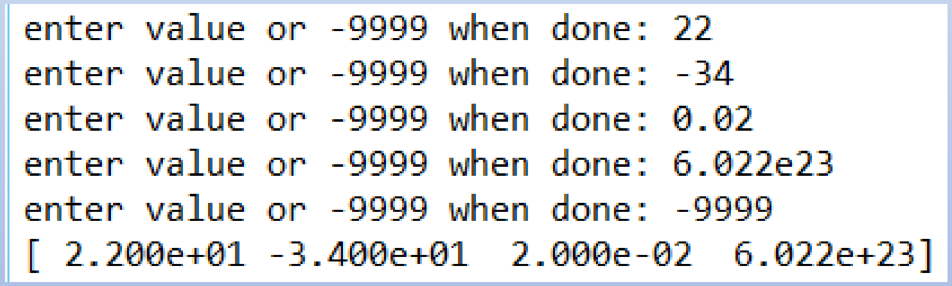
Repetition structures: the **break** statement

Example: Building an array with user input



```
import numpy as np
testdata = np.array([])
while True:
    datain = float(input('enter value or -9999 when done: '))
    if datain == -9999: break
    testdata = np.append(testdata,datain)
print(testdata)
```

Start with an empty array.
Exit loop when entry is = -9999.
Otherwise, expand **testdata** by **append**ing entry.



```
enter value or -9999 when done: 22
enter value or -9999 when done: -34
enter value or -9999 when done: 0.02
enter value or -9999 when done: 6.022e23
enter value or -9999 when done: -9999
[ 2.200e+01 -3.400e+01  2.000e-02  6.022e+23]
```

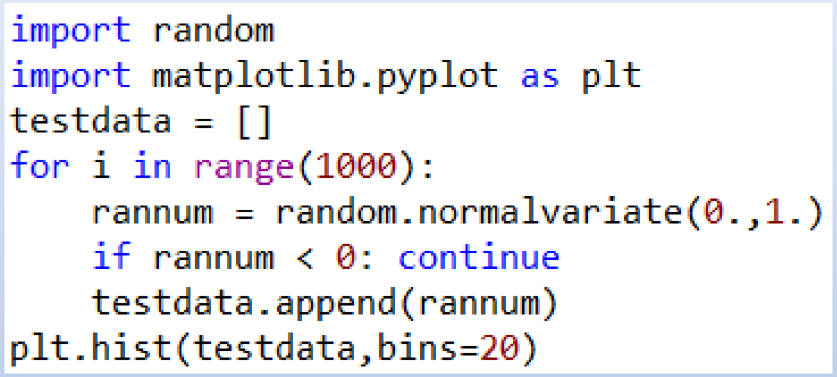
sentinel_break.py

Program Structure and User-defined Functions

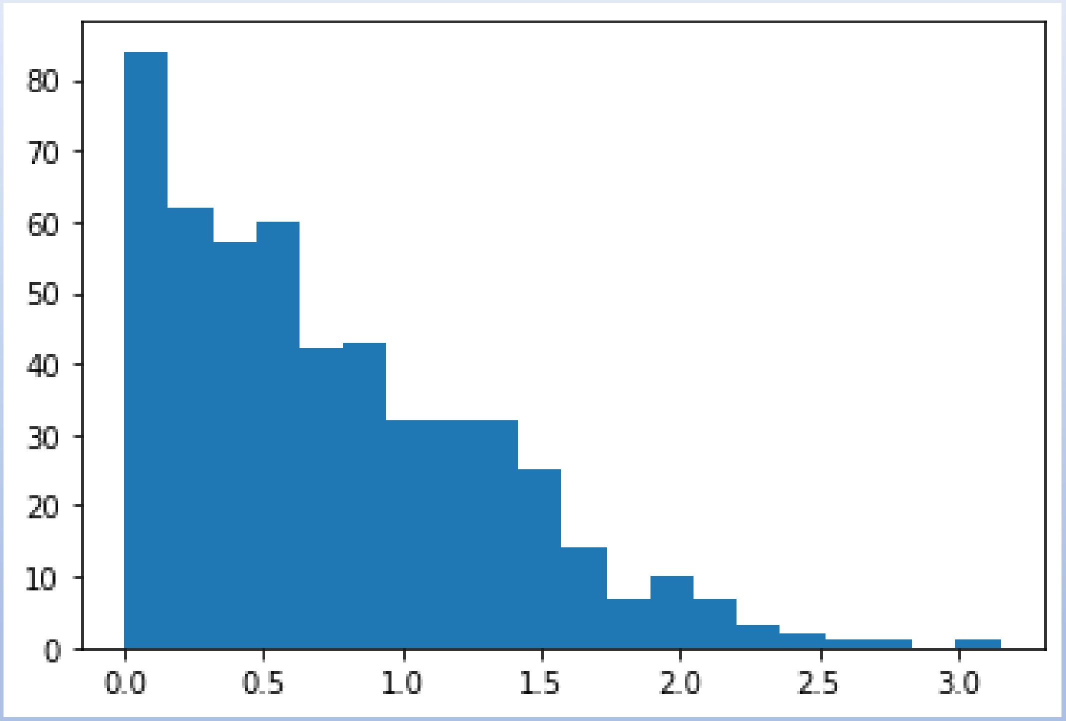
Repetition structures: the **continue** statement

Uses the **random** module

Example: Sifting positive random numbers



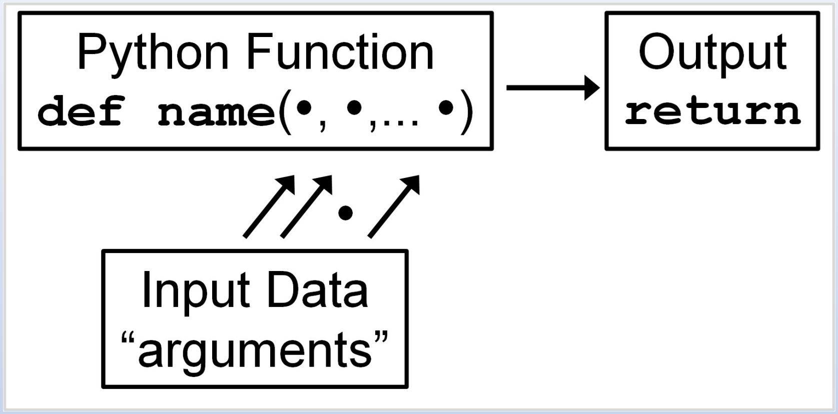
```
import random
import matplotlib.pyplot as plt
testdata = []
for i in range(1000):
    rannum = random.normalvariate(0.,1.)
    if rannum < 0: continue
    testdata.append(rannum)
plt.hist(testdata,bins=20)
```



Start with an empty **testdata** array.
In **for** loop, generate a random number (standard normal distribution).
If number is negative, do not store – **continue** next iteration.
If number is positive or zero, **append** to **testdata** array.
Create histogram of **testdata** array.

random_continue.py

Program Structure and User-defined Functions



```
def function_name(argument list):
    statements
    return results
```

Function must be “run” before it is invoked.
Can be invoked in the Console window or in statements below it.
Can store function(s) in separate **.py** file and import them.

```
def sgnsqr(x):
    sgnsq2 = x**2
    if x < 0: sgnsq2 = -sgnsq2
    return sgnsq2

print(sgnsqr(-2))
print(sgnsqr(2))
```

```
-4
4
```

sgnsqr_function.py

```
from sgnsqr_function import sgnsqr
y = float(input('enter a number: '))
print(sgnsqr(y))
```

```
enter a number: -44
-1936.0
```

Program Structure and User-defined Functions

lambda *anonymous* functions

lambda_example1.py

Abbreviating a **def** function

```python
def gety(x,a,b):
    return a*x**b
```

```python
gety = lambda x,a,b: a*x**b

a = 4
b = -0.32
print(gety(0.5,a,b))
```

```
4.9933221956064475
```

Without the **a** and **b** arguments:

Then, **a** and **b** must be provided explicitly in the main script.

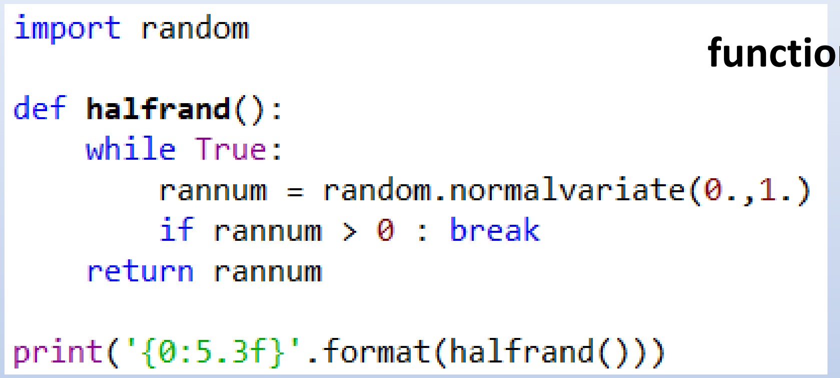
```python
gety = lambda x: a*x**b

a = 4
b = -0.32
print(gety(0.5))
```

```
4.9933221956064475
```

Program Structure and User-defined Functions

Function Arguments

A function with no arguments

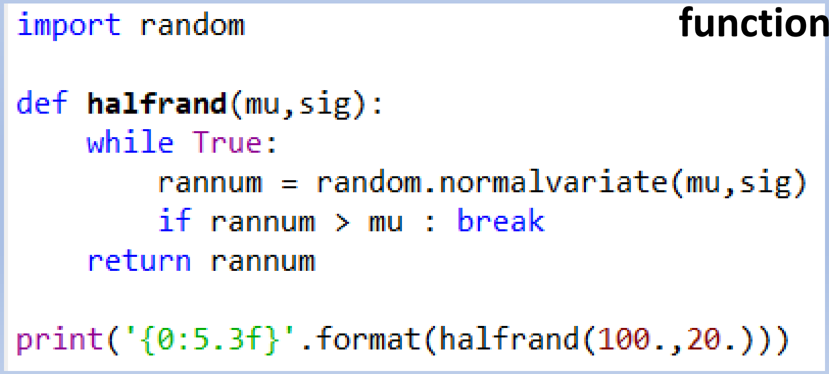


```python
import random

def halfrand():
    while True:
        rannum = random.normalvariate(0.,1.)
        if rannum > 0 : break
    return rannum

print('{0:5.3f}'.format(halfrand()))
```

functionwithnoarguments.py

```
1.881
```

Adding two arguments



```python
import random

def halfrand(mu,sig):
    while True:
        rannum = random.normalvariate(mu,sig)
        if rannum > mu : break
    return rannum

print('{0:5.3f}'.format(halfrand(100.,20.)))
```

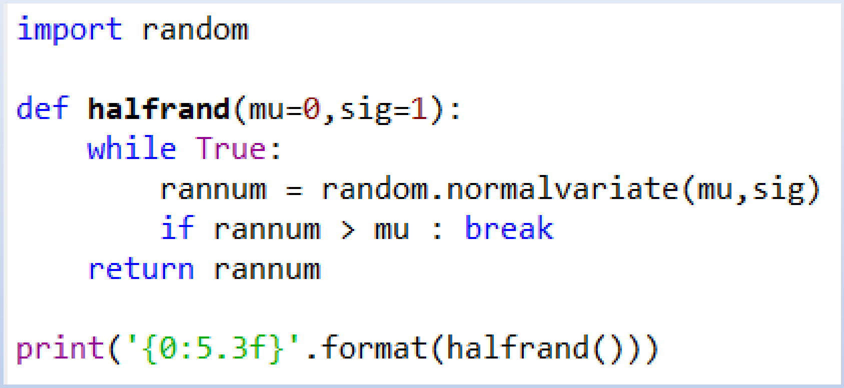
functionwithtwoarguments.py

```
102.743
```

Program Structure and User-defined Functions

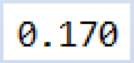
Function Arguments

Keyword arguments with default values



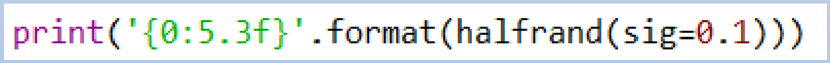
```python
import random

def halfrand(mu=0,sig=1):
    while True:
        rannum = random.normalvariate(mu,sig)
        if rannum > mu : break
    return rannum

print('{0:5.3f}'.format(halfrand()))
```

functionwithkeywordarguments.py

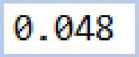


```
0.170
```

If no arguments specified, default values used.



```python
print('{0:5.3f}'.format(halfrand(sig=0.1)))
```



```
0.048
```

Can specify arguments by name in any order.
Default value used for argument left out.

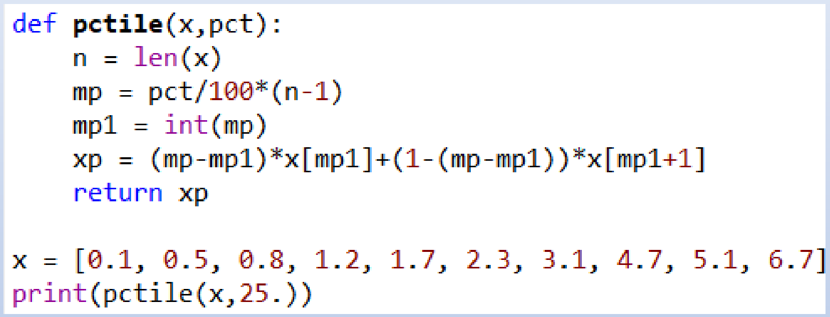
Program Structure and User-defined Functions

Function Arguments

Array arguments

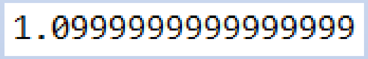
Example: percentile value of an ordered array

percentile1.py

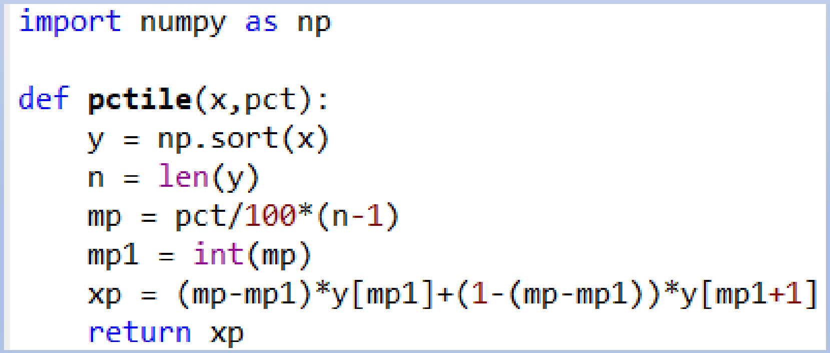


```python
def pctile(x,pct):
    n = len(x)
    mp = pct/100*(n-1)
    mp1 = int(mp)
    xp = (mp-mp1)*x[mp1]+(1-(mp-mp1))*x[mp1+1]
    return xp

x = [0.1, 0.5, 0.8, 1.2, 1.7, 2.3, 3.1, 4.7, 5.1, 6.7]
print(pctile(x,25.))
```

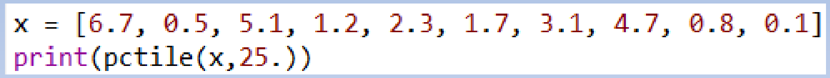
x in ascending order



```
1.0999999999999999
```



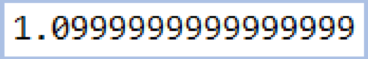
```python
import numpy as np

def pctile(x,pct):
    y = np.sort(x)
    n = len(y)
    mp = pct/100*(n-1)
    mp1 = int(mp)
    xp = (mp-mp1)*y[mp1]+(1-(mp-mp1))*y[mp1+1]
    return xp
```



```python
x = [6.7, 0.5, 5.1, 1.2, 2.3, 1.7, 3.1, 4.7, 0.8, 0.1]
print(pctile(x,25.))
```

x in any order
sorted into **y**

percentile2.py



```
1.0999999999999999
```

Program Structure and User-defined Functions

Function Arguments

Function invoked from within another function – interquartile range, **iqr**

```
import numpy as np

def pctile(x,pct):
    y = np.sort(x)
    n = len(y)
    mp = pct/100*(n-1)
    mp1 = int(mp)
    xp = (mp-mp1)*y[mp1]+(1-(mp-mp1))*y[mp1+1]
    return xp

def iqr(x):
    pct25 = pctile(x,25)
    pct75 = pctile(x,75)
    return pct75-pct25

x = [6.7, 0.5, 5.1, 1.2, 2.3, 1.7, 3.1, 4.7, 0.8, 0.1]
print('iqr = {0:5.2f}'.format(iqr(x)))
```

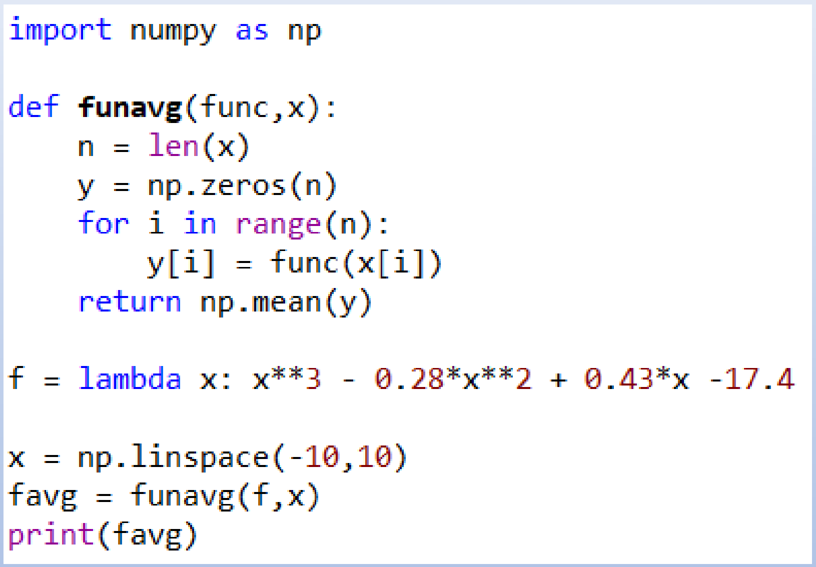
iqr.py

```
iqr =  2.40
```

Program Structure and User-defined Functions

Function Arguments

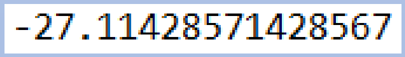
Arguments that are names of other functions



```
import numpy as np

def funavg(func,x):
    n = len(x)
    y = np.zeros(n)
    for i in range(n):
        y[i] = func(x[i])
    return np.mean(y)

f = lambda x: x**3 - 0.28*x**2 + 0.43*x -17.4

x = np.linspace(-10,10)
favg = funavg(f,x)
print(favg)
```

The first argument is a “dummy” name, **func**, for a function to be supplied.
The second argument, **x**, is an array of values.
The average of the **func(x)** values is returned.

The lambda function **f** is the first argument and 50 values of **x** from -10 to 10 are supplied.



```
-27.11428571428567
```

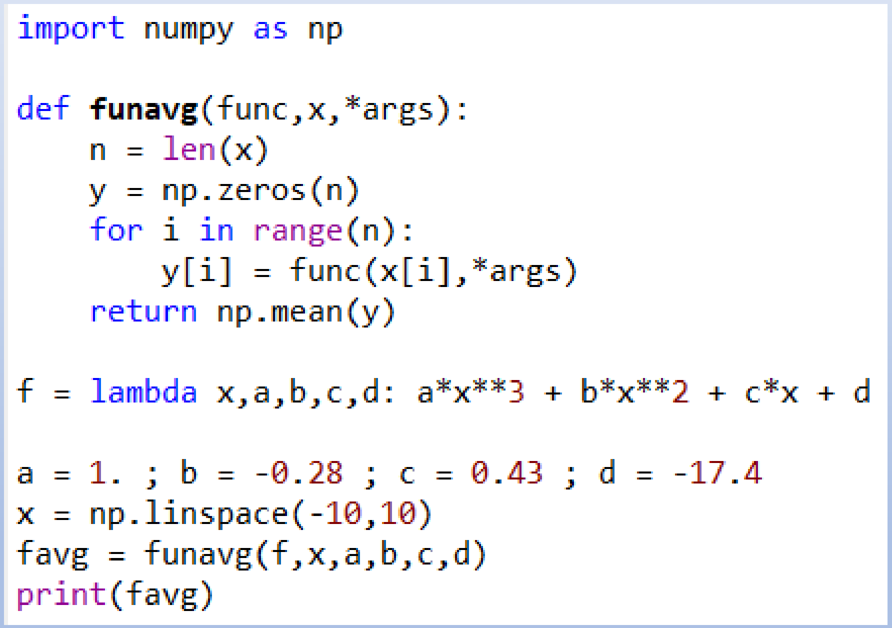
functionfunction.py

Program Structure and User-defined Functions

Function Arguments

Arguments that are names of other functions

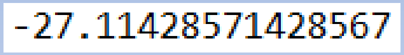
with pass-through arguments - ***args**



```
import numpy as np

def funavg(func,x,*args):
    n = len(x)
    y = np.zeros(n)
    for i in range(n):
        y[i] = func(x[i],*args)
    return np.mean(y)

f = lambda x,a,b,c,d: a*x**3 + b*x**2 + c*x + d

a = 1. ; b = -0.28 ; c = 0.43 ; d = -17.4
x = np.linspace(-10,10)
favg = funavg(f,x,a,b,c,d)
print(favg)
```

The **funavg** function is still generic but allows for one or more extra arguments to be passed through to the **func** evaluations.

The extra arguments are included in the call to **funavg** and defined in the **lambda** definition.



```
-27.11428571428567
```

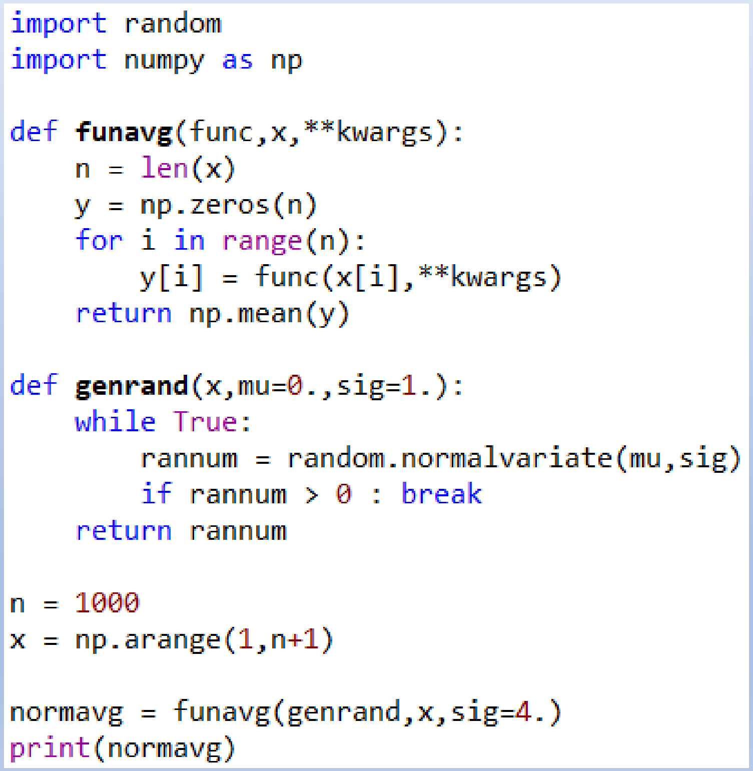
functionfunctionwithpassthrough.py

Program Structure and User-defined Functions

Function Arguments

Arguments that are names of other functions

with pass-through keyword arguments - ****kwargs**



```
import random
import numpy as np

def funavg(func,x,**kwargs):
    n = len(x)
    y = np.zeros(n)
    for i in range(n):
        y[i] = func(x[i],**kwargs)
    return np.mean(y)

def genrand(x,mu=0.,sig=1.):
    while True:
        rannum = random.normalvariate(mu,sig)
        if rannum > 0 : break
    return rannum

n = 1000
x = np.arange(1,n+1)

normavg = funavg(genrand,x,sig=4.)
print(normavg)
```

```
3.077226211257881
```

Identify potential keyword arguments with ****kwargs**
Note: two asterisks, **

Include keyword arguments for **mu** and **sig** with default values.

Specify only the **sig** value. The **mu** value will be its default, 0.

functionfunctionwithpassthroughkeywords.py

Program Structure and User-defined Functions

Function Arguments

Generally, the order of arguments supplied to a function must agree with the order in the function definition (**def** or **lambda**).

Pass-through arguments, ***args** and ****kwargs**, must follow other arguments.

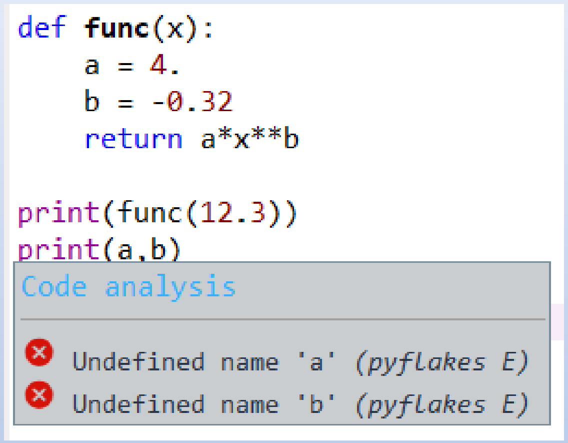
Any ***args** must agree in order with their function definition.

Keyword arguments can be supplied partially and in any order.

Common practice is to define ***args** and then ****kwargs**.

Program Structure and User-defined Functions

Variable scope



```
def func(x):
    a = 4.
    b = -0.32
    return a*x**b

print(func(12.3))
print(a,b)
```

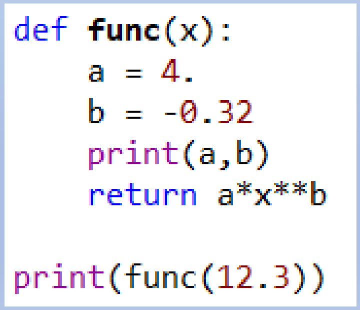
```
Undefined name 'a' (pyflakes E)
Undefined name 'b' (pyflakes E)
```

The variables **a** and **b** are local to function **func**.
They can’t be “seen” from the main script.
For this reason, there are errors.

scope1.py

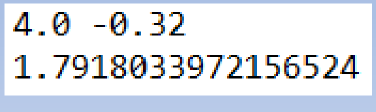
So, variables local to a function have *local scope*, and can’t be referenced from outside the function.

Note: arguments, also called *formal parameters*, are not local variables. They refer to values/variables from where the function is invoked.



```
def func(x):
    a = 4.
    b = -0.32
    print(a,b)
    return a*x**b

print(func(12.3))
```

But there is no problem when we place the **print** command inside the function.

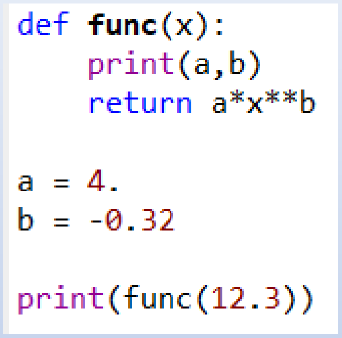


```
4.0 -0.32
1.7918033972156524
```

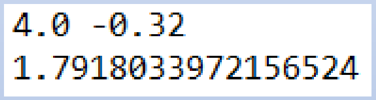
scope2.py

Program Structure and User-defined Functions

Variable scope



```python
def func(x):
    print(a,b)
    return a*x**b

a = 4.
b = -0.32

print(func(12.3))
```

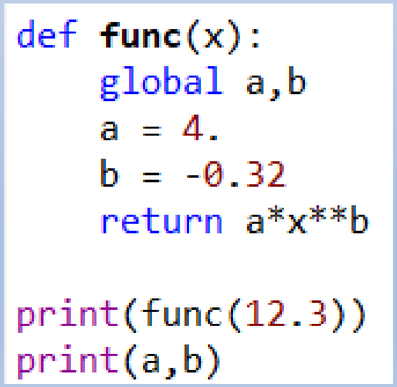


```
4.0 -0.32
1.7918033972156524
```

When **a** and **b** are assigned in the main script, their values are seen and available from within the function.
They have *global scope*.

scope3.py

The scope of a local variable can be extended by the **global** declaration.

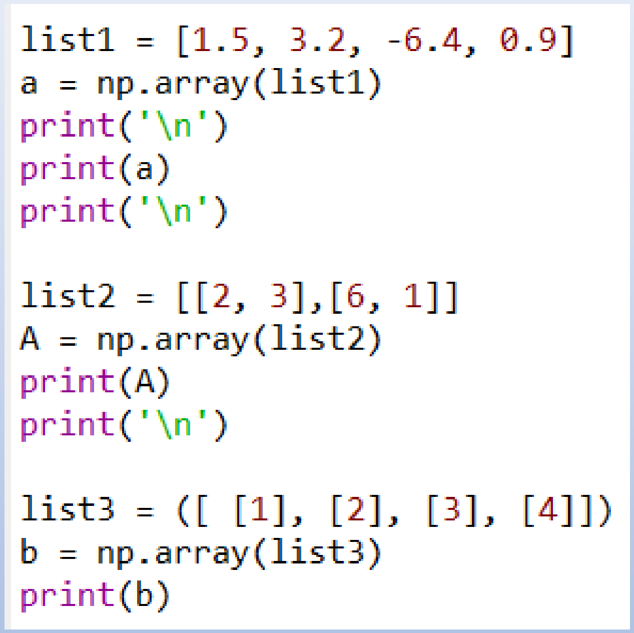


```python
def func(x):
    global a,b
    a = 4.
    b = -0.32
    return a*x**b

print(func(12.3))
print(a,b)
```

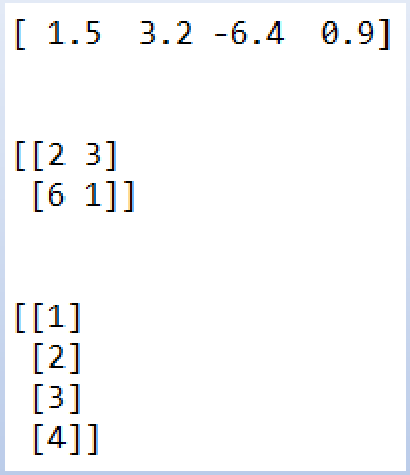
scope4.py

Arrays and Matrix Operations

Creating NumPy arrays from lists and tuples.



```
list1 = [1.5, 3.2, -6.4, 0.9]
a = np.array(list1)
print('\n')
print(a)
print('\n')

list2 = [[2, 3],[6, 1]]
A = np.array(list2)
print(A)
print('\n')

list3 = ([ [1], [2], [3], [4]])
b = np.array(list3)
print(b)
```



```
[ 1.5  3.2 -6.4  0.9]


[[2 3]
 [6 1]]


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

1x4 row vector

2x2 matrix

4x1 row vector

convert_list_to_array.py

In Python terminology, the dimensions of an array are called *axes*.

a has a single axis

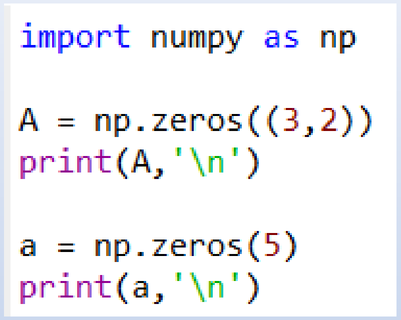
A has two axes

b has four axes

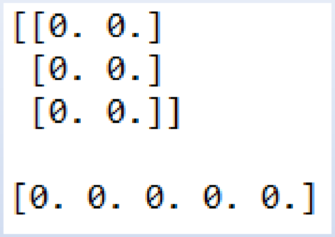
Arrays and Matrix Operations

Creating special arrays – **zeros** and **ones** functions

zeros_function.py

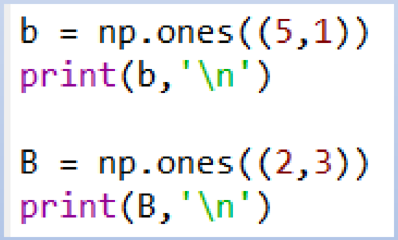


```python
import numpy as np

A = np.zeros((3,2))
print(A,'\n')

a = np.zeros(5)
print(a,'\n')
```

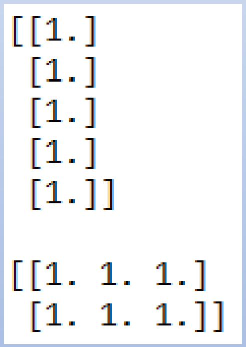


```
[[0. 0.]
 [0. 0.]
 [0. 0.]]

[0. 0. 0. 0. 0.]
```

Notice that only one set of parentheses
is required for a row array of zeros (a single axis).



```python
b = np.ones((5,1))
print(b,'\n')

B = np.ones((2,3))
print(B,'\n')
```

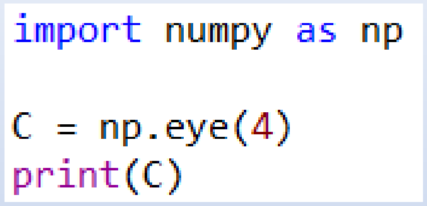


```
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]]

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

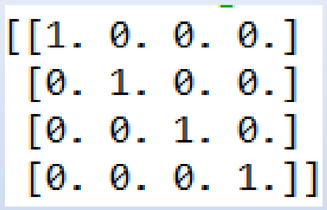
(5,1) is required for a column array of ones

Arrays and Matrix Operations

Creating special arrays – **eye** function



```
import numpy as np

C = np.eye(4)
print(C)
```



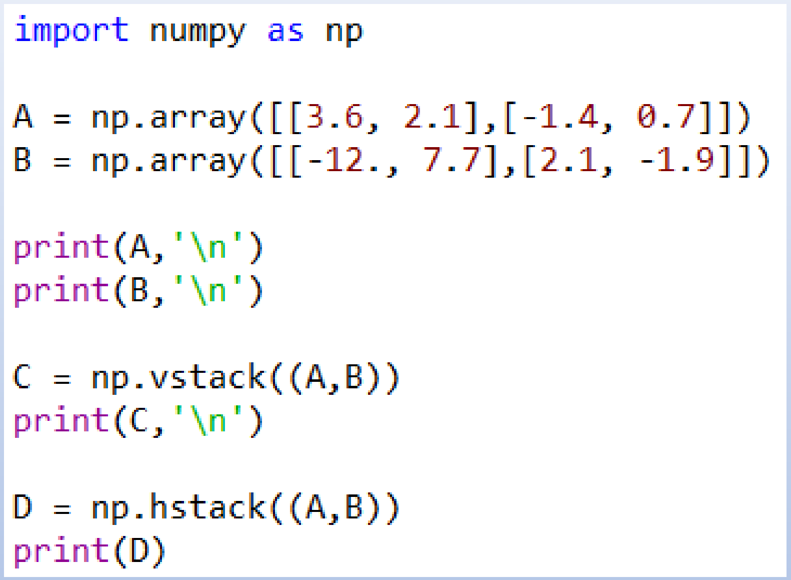
```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

Identity matrix

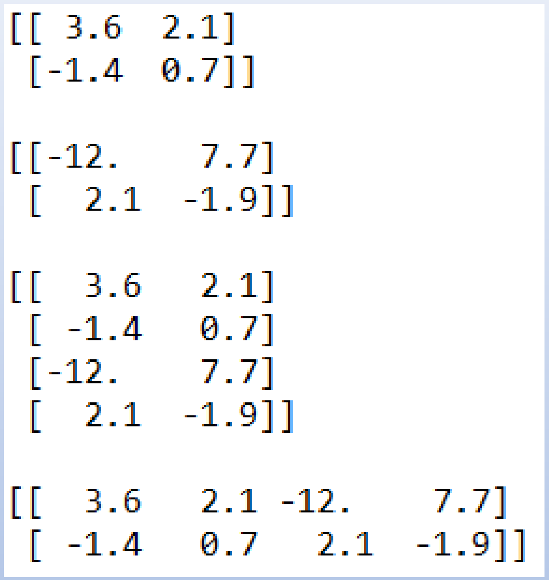
eye_function.py

Arrays and Matrix Operations

Combining and stacking arrays



```python
import numpy as np

A = np.array([[3.6, 2.1],[-1.4, 0.7]])
B = np.array([[-12., 7.7],[2.1, -1.9]])

print(A,'\n')
print(B,'\n')

C = np.vstack((A,B))
print(C,'\n')

D = np.hstack((A,B))
print(D)
```



```
[[ 3.6  2.1]
 [-1.4  0.7]]

[[-12.    7.7]
 [  2.1  -1.9]]

[[  3.6   2.1]
 [ -1.4   0.7]
 [-12.    7.7]
 [  2.1  -1.9]]

[[  3.6   2.1 -12.    7.7]
 [ -1.4   0.7   2.1  -1.9]]
```

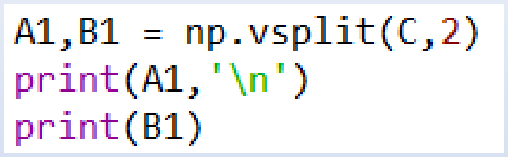
Vertical stack

Horizontal stack

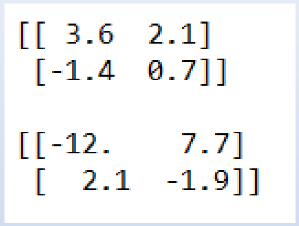
stack_examples.py

Arrays and Matrix Operations

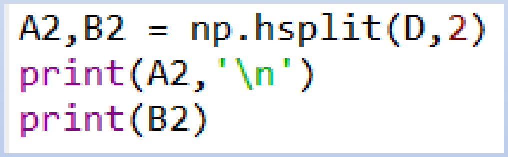
Splitting arrays



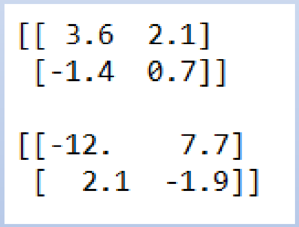
```
A1,B1 = np.vsplit(C,2)
print(A1,'\n')
print(B1)
```



```
[[ 3.6  2.1]
 [-1.4  0.7]]

[[-12.    7.7]
 [  2.1  -1.9]]
```



```
A2,B2 = np.hsplit(D,2)
print(A2,'\n')
print(B2)
```



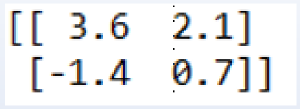
```
[[ 3.6  2.1]
 [-1.4  0.7]]

[[-12.    7.7]
 [  2.1  -1.9]]
```

Split arrays using **vsplit** and **hsplit** must be of equal size.

Can carry out splits of unequal size using indexing (later).

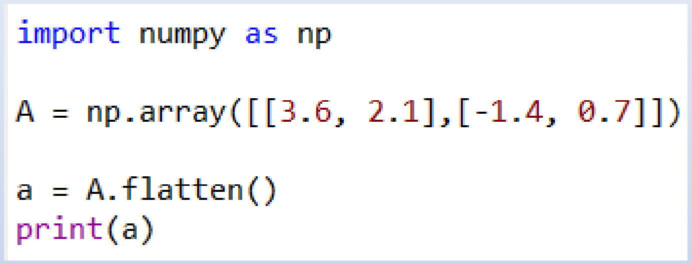
Arrays and Matrix Operations

Reshaping arrays

A 

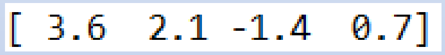
```
[[ 3.6  2.1]
 [-1.4  0.7]]
```

Flattening an array onto a single axis



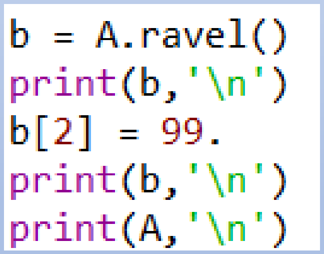
```python
import numpy as np

A = np.array([[3.6, 2.1],[-1.4, 0.7]])

a = A.flatten()
print(a)
```

flatten is a method that is applied to the array object. It is Python-based, not NumPy. The dimensions of **A** are not changed.

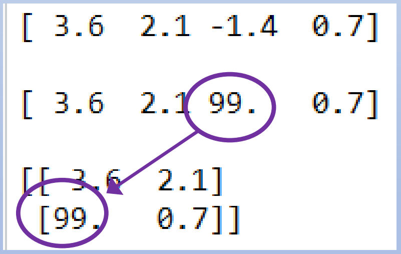


```
[ 3.6  2.1 -1.4  0.7]
```

Providing a different *view* of an array using **ravel** method.



```python
b = A.ravel()
print(b,'\n')
b[2] = 99.
print(b,'\n')
print(A,'\n')
```



```
[ 3.6  2.1 -1.4  0.7]

[ 3.6  2.1 99.   0.7]

[[ 3.6  2.1]
 [99.   0.7]]
```

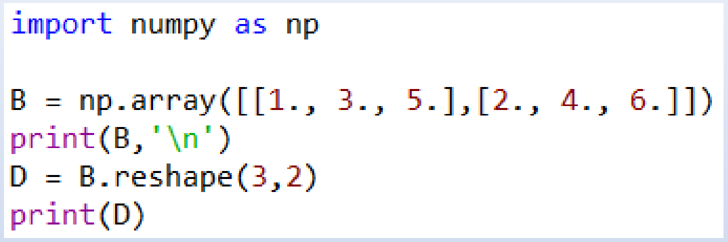
Modifying an element of **b** also modifies the corresponding element of **A**. **b** is just a different view of **A**, but they share the same memory.

This is tricky! (and we don't use it that frequently)

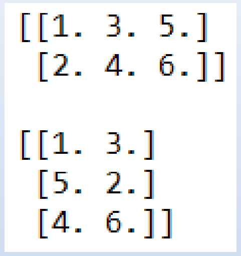
reshape_examples.py

Arrays and Matrix Operations

Reshaping arrays



```
import numpy as np

B = np.array([[1., 3., 5.],[2., 4., 6.]])
print(B,'\n')
D = B.reshape(3,2)
print(D)
```



```
[[1. 3. 5.]
 [2. 4. 6.]]

[[1. 3.]
 [5. 2.]
 [4. 6.]]
```

Notice that this is <u>not</u> a transpose. **reshape** essentially flattens the array and then reshapes it.

There is a **resize** method that changes the view of the array, similar to the **ravel** method.

reshape_examples1.py

Arrays and Matrix Operations

Indexing arrays

typical mathematical subscripts

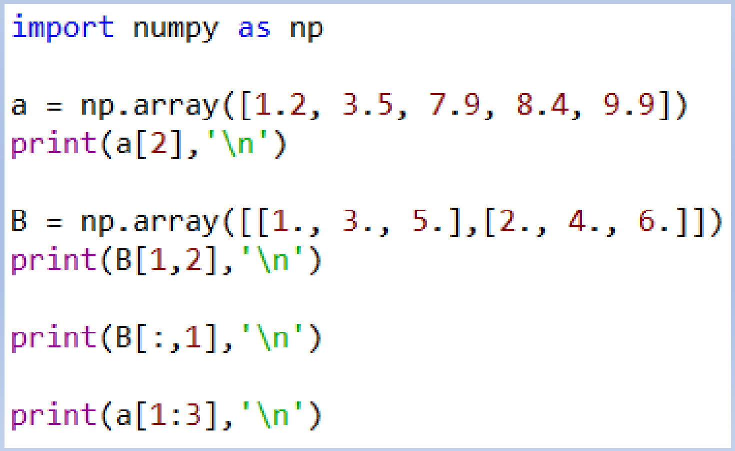
$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{bmatrix}$$

corresponding Python indices

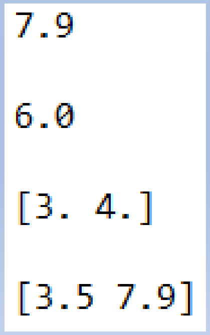
$$\begin{bmatrix} x_{[0,0]} & x_{[0,1]} & x_{[0,2]} & x_{[0,3]} \\ x_{[1,0]} & x_{[1,1]} & x_{[1,2]} & x_{[1,3]} \\ x_{[2,0]} & x_{[2,1]} & x_{[2,2]} & x_{[2,3]} \end{bmatrix}$$

Examples

IndexingExamples.py



```
import numpy as np

a = np.array([1.2, 3.5, 7.9, 8.4, 9.9])
print(a[2],'\n')

B = np.array([[1., 3., 5.],[2., 4., 6.]])
print(B[1,2],'\n')

print(B[:,1],'\n')

print(a[1:3],'\n')
```



```
7.9

6.0

[3. 4.]

[3.5 7.9]
```

3rd element

2nd row, 3rd element

Note: result shown as a row

2nd and 3rd element (not the 4th)

Arrays and Matrix Operations

Indexing arrays

IndexingExamples2.py

```
import numpy as np

Z = np.array([[ 3, -2,  7,  1],
              [ 4, -1,  6,  5],
              [ 9, -8,  4,  2],
              [-5,  8, -7, -3]])
print(Z[1:3,1:3])
```

```
[[-1  6]
 [-8  4]]
```

rows 2 and 3
columns 2 and 3

Indexing with for loop variables

```
import numpy as np

B = np.array([[1., 3., 5.],[2., 4., 6.]])
print(B,'\n')

sumsqrtB = 0
for i in range(2):
    for j in range(3):
        sumsqrtB = sumsqrtB + np.sqrt(B[i,j])
print('{0:5.2f}'.format(sumsqrtB))
```

```
[[1. 3. 5.]
 [2. 4. 6.]]

10.83
```

subscripts align properly
with the use of the **range** type

range(2) $\longrightarrow$ [0,1]

range(3) $\longrightarrow$ [0,1,2]

IndexingExamples3.py

Arrays and Matrix Operations

Array operations

```python
import numpy as np

a = np.array([1.2, 3.5, 7.9, 8.4, 9.9])
b = a/3
print(b)
```

divide by 3 applies to each element of the array

```
[0.4        1.16666667 2.63333333 2.8        3.3       ]
```

array_operations_1.py

```python
import numpy as np

a = np.array([1, 3, 5])
b = np.array([2, 4, 6])
print(a+b)
```

arrays added, item by item
size must be the same

array_operations_2.py

```
[ 3  7 11]
```

```python
import numpy as np

x = np.array([0.1, 0.2, 0.3])
y = x**3 -2*x**2 + 0.4*x + 3
print(y)
```

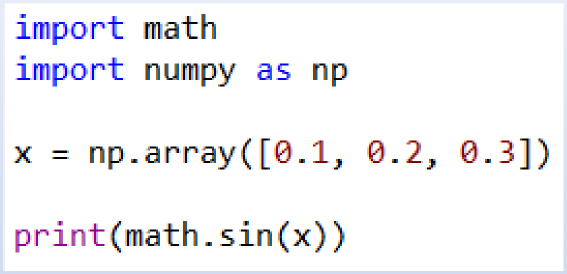
"vectorizing" a polynomial calculation

array_operations_3.py

```
[3.021 3.008 2.967]
```

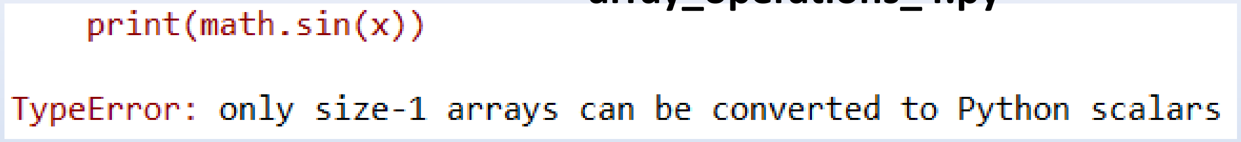
Arrays and Matrix Operations

Array operations



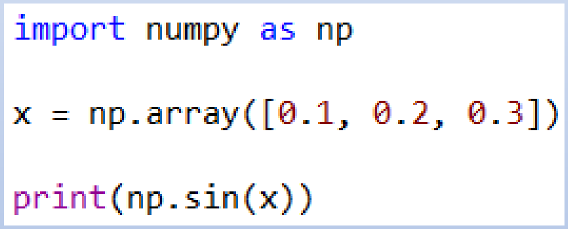
```
import math
import numpy as np

x = np.array([0.1, 0.2, 0.3])

print(math.sin(x))
```

array_operations_4.py



```
    print(math.sin(x))

TypeError: only size-1 arrays can be converted to Python scalars
```

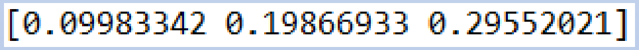
Math module functions do not accept arrays.



```
import numpy as np

x = np.array([0.1, 0.2, 0.3])

print(np.sin(x))
```

NumPy module functions do.

array_operations_5.py

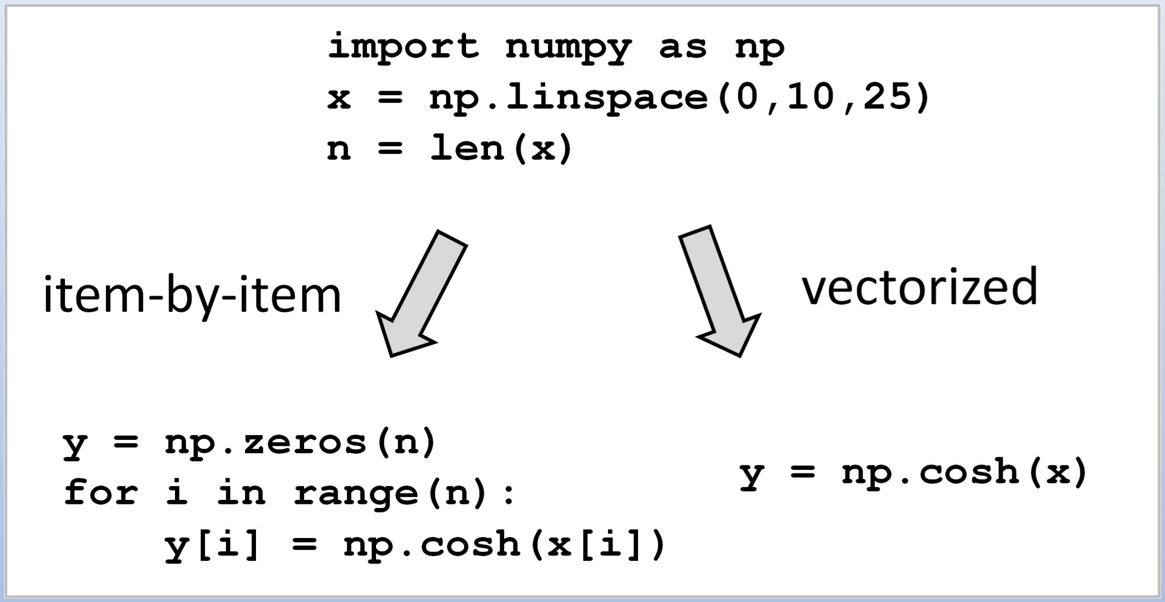


```
[0.09983342 0.19866933 0.29552021]
```

Arrays and Matrix Operations

Array operations

for loop with item-by-item versus vectorized calculation



```
import numpy as np
x = np.linspace(0,10,25)
n = len(x)
```

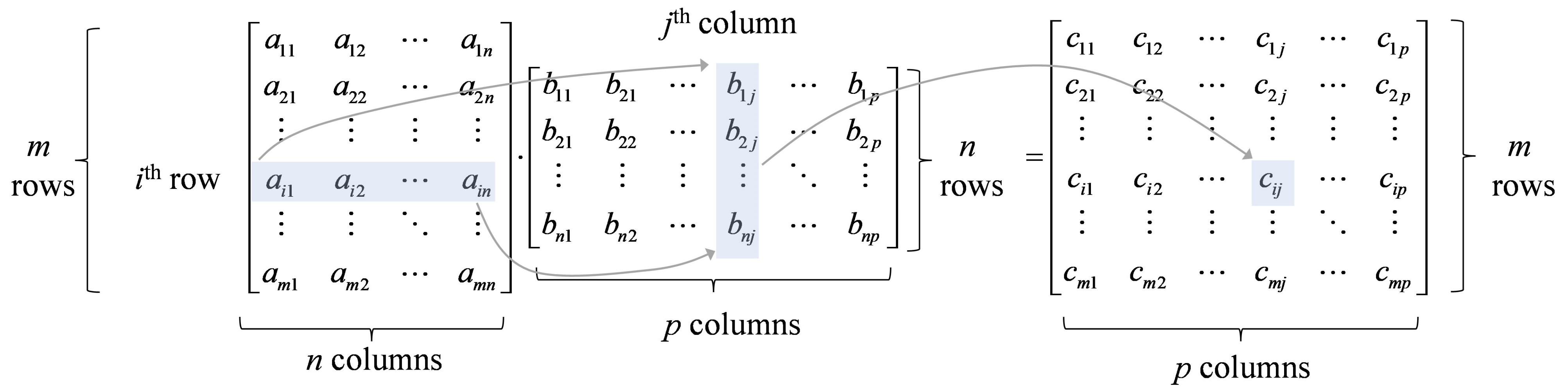
```
y = np.zeros(n)
for i in range(n):
    y[i] = np.cosh(x[i])
```

```
y = np.cosh(x)
```

Arrays and Matrix Operations

Vector/matrix operations

Matrix multiplication



$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{21} & \cdots & b_{1j} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2j} & \cdots & b_{2p} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nj} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1j} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2j} & \cdots & c_{2p} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ c_{i1} & c_{i2} & \cdots & c_{ij} & \cdots & c_{ip} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mj} & \cdots & c_{mp} \end{bmatrix}$$

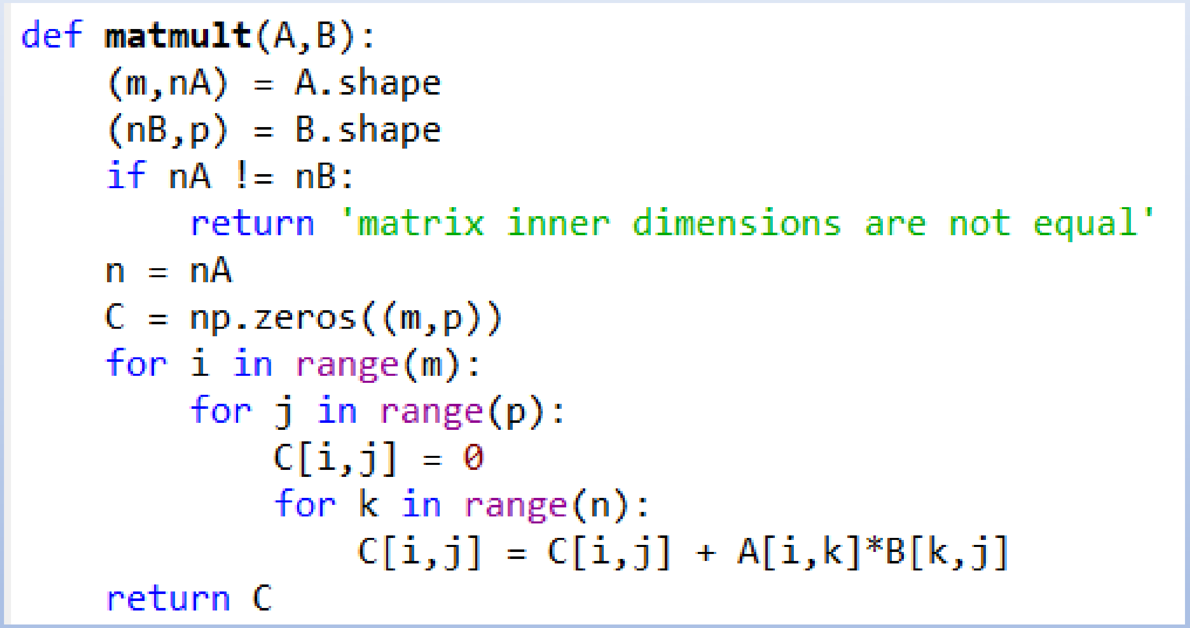
$$\begin{aligned} c_{ij} &= a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots a_{in}b_{nj} \\ &= \sum_{k=1}^{n} a_{ik}b_{kj} \end{aligned}$$

$$[m \times \cancel{n}] \cdot [\cancel{n} \times p] \Rightarrow [m \times p]$$

Arrays and Matrix Operations

Vector/matrix operations

Matrix multiplication – home-grown function



```python
def matmult(A,B):
    (m,nA) = A.shape
    (nB,p) = B.shape
    if nA != nB:
        return 'matrix inner dimensions are not equal'
    n = nA
    C = np.zeros((m,p))
    for i in range(m):
        for j in range(p):
            C[i,j] = 0
            for k in range(n):
                C[i,j] = C[i,j] + A[i,k]*B[k,j]
    return C
```

Use the **shape** property to get numbers of rows and columns.

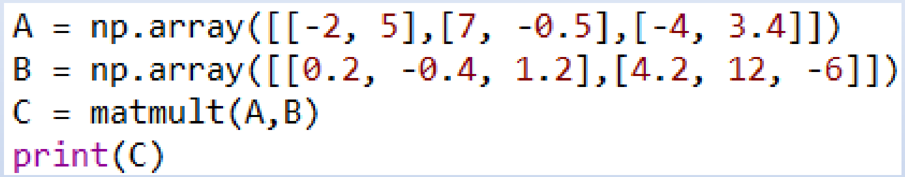
Nested **for** loops:
i^th row of A
j^th column of B
sum the A*B products

matmult_example.py

Arrays and Matrix Operations

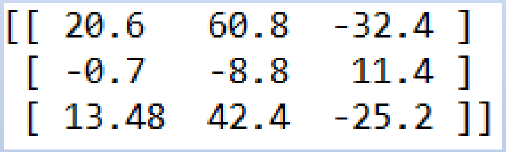
Vector/matrix operations

Matrix multiplication – home-grown function



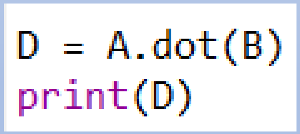
```
A = np.array([[-2, 5],[7, -0.5],[-4, 3.4]])
B = np.array([[0.2, -0.4, 1.2],[4.2, 12, -6]])
C = matmult(A,B)
print(C)
```

3 x 2 * 2 x 3 ⇨ 3 x 3

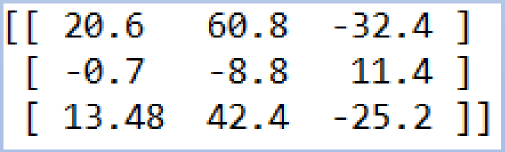


```
[[ 20.6    60.8   -32.4 ]
 [ -0.7    -8.8    11.4 ]
 [ 13.48   42.4   -25.2 ]]
```

Matrix multiplication – using the **dot** method



```
D = A.dot(B)
print(D)
```



```
[[ 20.6    60.8   -32.4 ]
 [ -0.7    -8.8    11.4 ]
 [ 13.48   42.4   -25.2 ]]
```

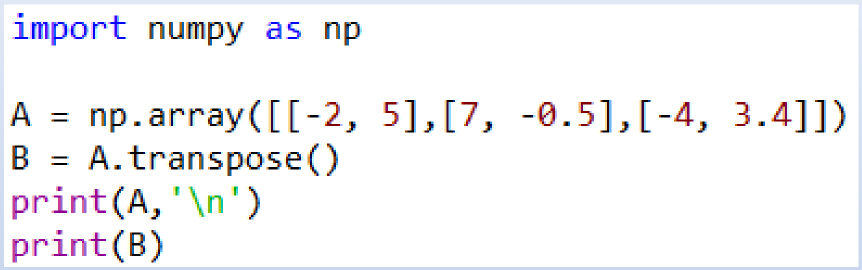
same result

compact, preferred

Arrays and Matrix Operations

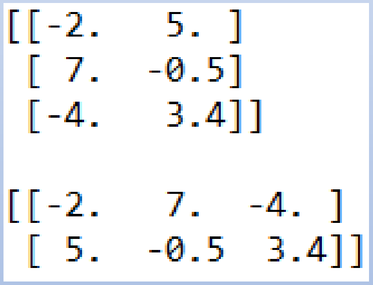
Vector/matrix operations

Matrix transpose



```python
import numpy as np

A = np.array([[-2, 5],[7, -0.5],[-4, 3.4]])
B = A.transpose()
print(A,'\n')
print(B)
```

using the **transpose** method



```
[[-2.   5. ]
 [ 7.  -0.5]
 [-4.   3.4]]

[[-2.   7.  -4. ]
 [ 5.  -0.5  3.4]]
```

rows become columns
and vice versa

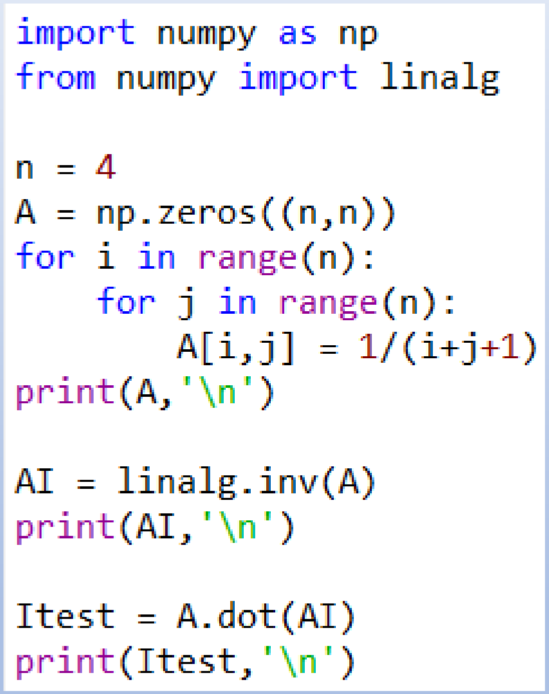
transpose_example.py

Arrays and Matrix Operations

matrix_inverse_example.py

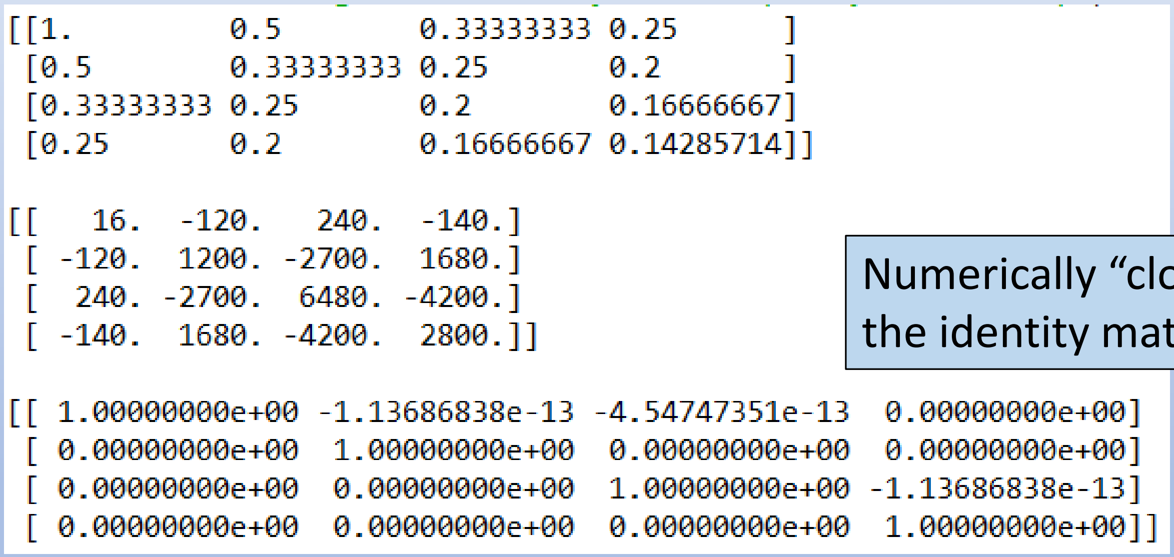
Vector/matrix operations

Matrix inversion using the **inv** function in the NumPy **linalg** submodule



```python
import numpy as np
from numpy import linalg

n = 4
A = np.zeros((n,n))
for i in range(n):
    for j in range(n):
        A[i,j] = 1/(i+j+1)
print(A,'\n')

AI = linalg.inv(A)
print(AI,'\n')

Itest = A.dot(AI)
print(Itest,'\n')
```

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}$$



```
[[1.         0.5        0.33333333 0.25      ]
 [0.5        0.33333333 0.25       0.2       ]
 [0.33333333 0.25       0.2        0.16666667]
 [0.25       0.2        0.16666667 0.14285714]]

[[   16.   -120.    240.   -140.]
 [ -120.   1200.  -2700.   1680.]
 [  240.  -2700.   6480.  -4200.]
 [ -140.   1680.  -4200.   2800.]]

[[ 1.00000000e+00 -1.13686838e-13 -4.54747351e-13  0.00000000e+00]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00 -1.13686838e-13]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.00000000e+00]]
```

Numerically “close” to the identity matrix

The **A** matrix created above is called a Hilbert matrix of order **n**.
Computation of the inverse of a Hilbert matrix is notably difficult.

Reference:

Introduction to Engineering and Scientific Computing with Python

David E. Clough

Steven C. Chapra

CRC Press, Taylor & Francis, 2023.

"Prof. Clough, may I be excused? My brain is full."

What's next?

Python Bootcamps 1, 2 and 3

- 1: Getting up to speed with Python
- 2: Learning to use Python to solve typical problem scenarios
- 3: Detailed modeling of packed-bed and plug-flow reactors