Computational Radiometry Work Package

Document Number	Equipment or Sub-System

Subject

02-PythonWhirlwindCheatSheet

Distribution

Conclusions/Decisions/Amendments

Author	Signature
CJ Willers	

Date	Date	Date August 23, 2021
Previous Package No.	Superseding Package No.	Current Package No.

1 2 Python and Numpy whirlwind cheat sheet

This notebook forms part of a series on computational optical radiometry[1]. The notebooks can be downloaded from Github[2]. These notebooks are constantly revised and updated, please revisit from time to time.

The date of this document and module versions used in this document are given at the end of the file.

Feedback is appreciated: neliswillers at gmail dot com.

This notebook serves to give brief one-liner examples of key Python usage and idioms. The approach here is not to spend too much time on general Python constructs, but to move on to Numpy as soon as possible. Numpy is a critical base for any scientific calculation, providing powerful vectorised computation. I make no claim to any level of completeness in this document. You still need to read the books and reference manuals to get the big picture and the details. You still have to read the manuals, but this should help you move on to Numpy quickly.

```
from IPython.display import display
from IPython.display import Image
from IPython.display import HTML
from IPython.core.display import display, HTML  # display(HTML(df.↵
   to_html()))

import numpy as np
import os.path
```

1.1 The Zen of Python

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.2 Python: What and Why

"the reason python is the best numerical language is because it is not a numerical language"

- Nathaniel Smith, SciPy2016

10 years ago, Python was considered exotic in the analytics space – at best. Languages/packages like R and Matlab dominated the scene. Today, Python has become a major force in data analytics and visualization due to a number of characteristics: * **Multi-purpose**: prototyping, development, production, systems admin – one for all * **Libraries**: there is a library for almost any task or problem you face * **Efficiency**: speeds up IT development tasks for analytics applications * **Performance**: Python has evolved from a scripting language to a 'meta' language with bridges to all high performance environments (e.g. multi-core CPUs, GPUs, clusters) * **Interoperability**: Python integrates with almost any other language/ technology * **Interactivity**: Python allows domain experts to get closer to their business and financial data pools and to do real-time analytics * **Collaboration**: solutions like Wakari with IPython Notebook allow the easy sharing of code, data, results, graphics, etc.

A fundamental Python stack for interactive data analytics and visualization should at least contain the following libraries tools: * **Python** – the Python interpreter itself

- **NumPy** – high performance, flexible array structures and operations
- **SciPy** – scientific modules and functions (regression, optimization, integration)
- **pandas** – time series and panel data analysis and I/O
- **PyTables** – hierarchical, high performance database (e.g. for out-of-memory analytics)
- **matplotlib** – 2d and 3d visualization
- **Jupyter** (previously IPython) – interactive data analytics, visualization, publishing

1.3 Python for scientific work (instead of using Matlab)

Go make up your own mind:

`http://cyrille.rossant.net/why-using-python-for-scientific-computing/`[3] why use Python?

`http://cyrille.rossant.net/tag/ipython/`[4] Python weaknesses - a balanced view

`https://sites.google.com/site/pythonforscientists/python-vs-matlab`[5]

http://wiki.scipy.org/PerformancePython[6]

http://phillipmfeldman.org/Python/Advantages_of_Python_Over_Matlab.html[7]

http://www.stat.washington.edu/~hoytak/blog/whypython.html[8]

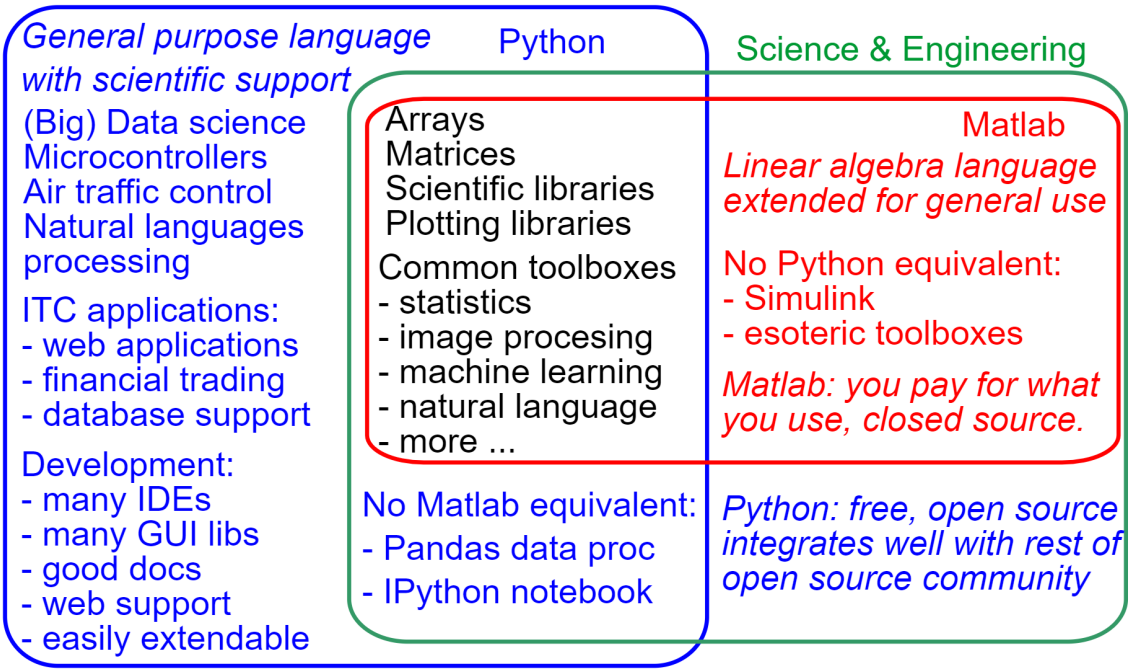
https://vnoel.wordpress.com/2008/05/03/bye-matlab-hello-python-thanks-sage/[9]

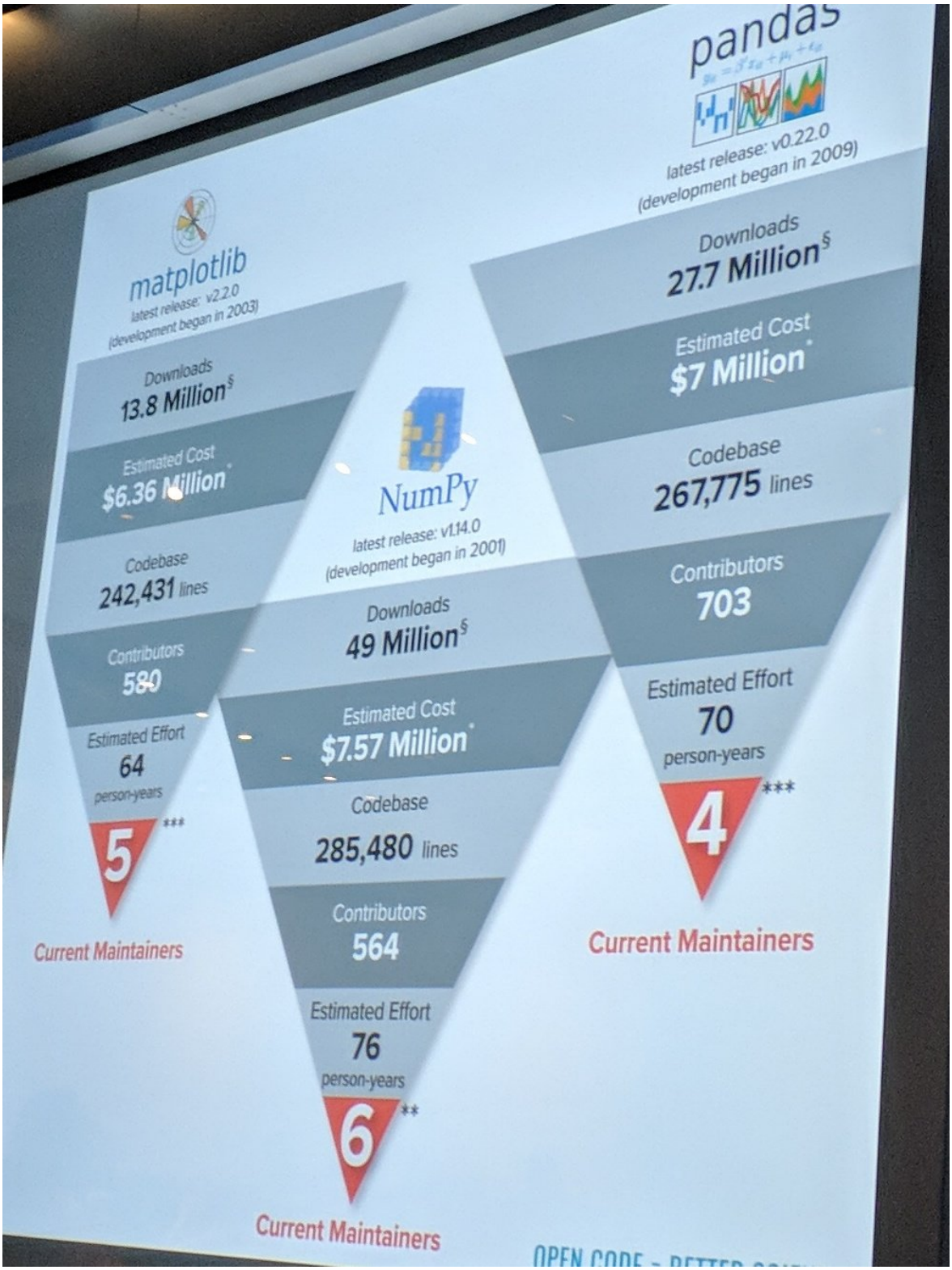
http://metarabbit.wordpress.com/2013/10/18/why-python-is-better-than-matlab-for-sc has some compelling evidence of the growing momentum of Python for scientific work - scroll to the top of this post for a very interesting graph.

https://github.com/jakevdp/2013_fall_ASTR599[11] Jake VanderPlas' comprehensive course for using Python for science, including using git and notebooks

http://www-personal.umich.edu/~kundeng/stats607/week_5_pysci-03-scipy.pdf[12] [nice presentation!]

http://nbviewer.ipython.org/github/Tooblippe/ipython_csir_30may/blob/master/csir_ipython.ipynb[13]





The graph above by Gina Helfrich and Kelle Cruz[14] shows shows the investment cost and scope of the *(only) three* main open source scientific libraries we are using today in Python:

- Total number of contributors: more than 1800 people.
- Lines of code : just under 800 000 lines of code.
- Person-years worth of work: 210 years!!
- total cost to repeat this work: almost USD21m!

The above covers only three of the scientific tools, it does not include the cost of * Python itself * Machine Learning tools * Web tools * Big Data tools * GUI design tools * Or any of the thousands of other specialised Python libraries.

It is probably fair to say that the total investment in all of the open source Python community exceeds USD50m, perhaps even USD100m!

This is serious effort and most certainly a safe mainstream road to be on. unless you are using Simulink, there is certainly no reason to use Matlab. Outside of Simulink Python and Friends can do much, much more than Matlab, is easier to use, is more powerful, and is free.

1.4 Learning Python

1.4.1 General/scientific introduction

Try to find the following books:

1. Python Data Science Handbook[15] good solid intro to data science in particular, but also general scientific work with Python.
2. Elegant SciPy[16] another good scientific intro to scientific work with Python.
3. Effective Computation in Physics[17] has nothing to do with physics, but is a very good general introduction to good programming practices in general, focuses on Python.
4. Python for Data Analysis Book[18] you have to use Pandas!
5. Effective Pandas[19] you have to use Pandas!
6. Fluent Python[20] teaches Python by focusing on on Python idioms and good style.
7. Effective Python[21] bring a deeper understanding into the language.
8. Python Machine Learning, 2nd Edition[22] not traditional scientific, but on machine learning.
9. Programming Python[23] if you come from another language.
10. Learning_Python[24] if you come from another language.

The following Python online resources may help you get going:

`https://newcircle.com/bookshelf/python\_fundamentals\_tutorial/index`[25]

`http://files.swaroopch.com/python/byte\_of\_python.pdf`[26]

`http://www.diveintopython.net/`[27]

`http://readwrite.com/2011/03/25/python-is-an-increasingly-popu#awesm=~oyGhXjmOOEXT`

`http://efytimes.com/e1/fullnews.asp?edid=117094`[29]

`http://folk.uio.no/hpl/scripting/book\_comparison.html`[30] (Python for scientific computation)

`https://mbakker7.github.io/exploratory\_computing\_with\_python/`[31]

A Crash Course in Python for Scientists[32], I don't agree with his recommendation to install PythonXY, I prefer Anaconda. https://github.com/Junnplus/awesome-python-books[33]

http://python-3-patterns-idioms-test.readthedocs.io/en/latest/[34]

There are many free books[35], classes[36], blogs[37] websites and tutorial videos[38], more videos[39] and conferences[40]. Material for Numpy is less bountiful, but the numpy reference[41] and StackOverflow[42] are good sources. Just google some variation of 'learning python[43]' and make your choice.

Beginner's guides:

https://docs.scipy.org/doc/numpy-dev/user/quickstart.html[44] https://wiki.python.org/moin/BeginnersGuide[45]

http://www.wikihow.com/Start-Programming-in-Python[46]

http://www.learnpython.org/[47]

http://www.sthurlow.com/python/[48]

See Steve Holden's Intermediate Python[49] page. HTML rendering of Steve's notebooks[50]. IPython notebooks here[51]. See the YouTube video[52].

http://sahandsaba.com/thirty-python-language-features-and-tricks-you-may-not-know.html[53]"

1.4.2 Python for science and engineering

The following books are great (but there are several others as well):

1. Scipy Lecture Notes[54] One document to learn numerics, science, and data with Python. The source code for the book is available here[55].
2. A very good introduction to Python for scientific work are the two books[56] by Hans Petter Langtangen: A Primer on Scientific Programming with Python[57], and Python scripting computational science[58].

Numpy/SciPy resources:

http://www.engr.ucsb.edu/~shell/che210d/numpy.pdf[59]

http://docs.scipy.org/doc/[60]

http://nbviewer.ipython.org/github/jrjohansson/scientific-python-lectures/tree/master/[61]

http://wiki.scipy.org/Tentative_NumPy_Tutorial[62]

http://matplotlib.org/gallery.html[63]

http://nbviewer.ipython.org/github/gumption/Python_for_Data_Science/blob/master/Python_for_Data_Science_all.ipynb[64] detailed introduction to Python for data science.

If you have some matlab background:

http://wiki.scipy.org/NumPy_for_Matlab_Users[65]

http://hyperpolyglot.org/numerical-analysis[66]

http://resources.sei.cmu.edu/asset_files/Presentation/2011_017_001_50519.pdf[67]

http://www.pyzo.org/python_vs_matlab.html[68]

1.5 Plotting packages and options

Scientific Plotting in Python[69]

Interactive Plotting in IPython Notebook (Part 1/2): Bokeh[70]

Interactive Plotting in IPython Notebook (Part 2/2): Plotly[71]

Overview of Python Visualization Tools [72]

Comparing Python web visualization libraries[73]

matplotlib matplotlib[74] Gallery[75] Examples[76] docs[77]

bokeh Welcome to bokeh[78] Gallery[79] Quickstart[80] Tutorials[81] User Guide[82]

plotly What is Plotly.js[83] Plotly.js Open-Source Announcement[84] Getting Started: Plotly for Python[85] Plotly Offline[86] User Guide[87] Plotly's Python API User Guide[88] Example 3-D plot[89]

1.6 Pandas

Pandas is roughly the Python equivalent of R's DataFrame, but somewhat different.

If you do any form of data analysis, using Pandas is greatly advised. Your life will never be the same again!

Python for Data Analysis[90], by Wes McKinney, the Pandas author.

Python Data Science Handbook[91], by Jake VanderPlas.

Effective Pandas[92] by Tom Augspurger.

Intro to pandas data structures[93], old but good introduction by Greg Reda.

https://www.dataschool.io/best-python-pandas-resources/

https://www.fullstackpython.com/pandas.html

https://github.com/NelisW/PythonNotesToSelf/tree/master/pandas

There are also several good tutorial videos on YouTube, recorded during some of the PyData or PyCon conferences over the years.

1.7 Installing Python

See the companion notebook 00-Installing-Python-pyradi.ipynb or on nbviewer at https://nbviewer.jupy
Installing-Python-pyradi.ipynb

1.8 Python Basics

1.8.1 Code structure

Python does not use open and closing braces to indicate structure, it uses indenting in the code to show structure. This feels awkward at first, but if you use a Python-aware editor, such as Notepad++, Sublime Text or IPython's editor, the editor should make it easy to manage the intents. The downside of the indenting method is that if you loose indenting, you lost the structure in the code. Be careful when you copy Python code from the web, sometimes you loose the indentation structure.

The preferred coding style is defined in PEP-8, see `http://legacy.python.org/dev/peps/pep-0008/`[94]

A single-line comment is indicated by a hasg symbol, which comments out everything until the end of the line. Multiline comments is started by three double quotes """ and ended likewise. By convension the opening and closing set of quotes are at the start of a line.

Python is particular about mixing indenting by space or by tab. It can be either one, but it must be consistent in one script. The preferred way is to indent by spaces, because tab-indenting can change if you move code from one editor to another.

Set your editor to automatically replace tabs by spaces, so that you always end up with spaces even if you press the tab key. PEP8 recommends using four spaces indenting, and a line length of 80 characters, but I find that you quickly run out of width on deeply indented code. My own preference is to use 2 spaces for indents and 100 character line lengths. IPython notebooks uses four spaces for indentation.

Long lines can break according to the following rules:

- Type a backslash \ anywhere on the line, followed by Enter and type the remaining text on the next line. It seems that spaces on the next line are included.
- Break the line anywhere where a comma is allowed. There is no need to enter a backslash when breaking after a comma.

```
print('Hello world'\
      )
print('{0} {1} {2:10,.2f}'.format('Hello',
                      'world', 127883.54355))
```

```
Hello world
Hello world 127,883.54
```

1.8.2 Docstrings and introspection

From PEP-0257: "A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `\_\_doc\_\_` special attribute of that object." Docstrings are considered the documentation for that module, function or class. It is normally written in a multiline comment. An example:

```
def add(x, y):
```

```
    """Return the sum of x and y.
    """
    return x + y
```

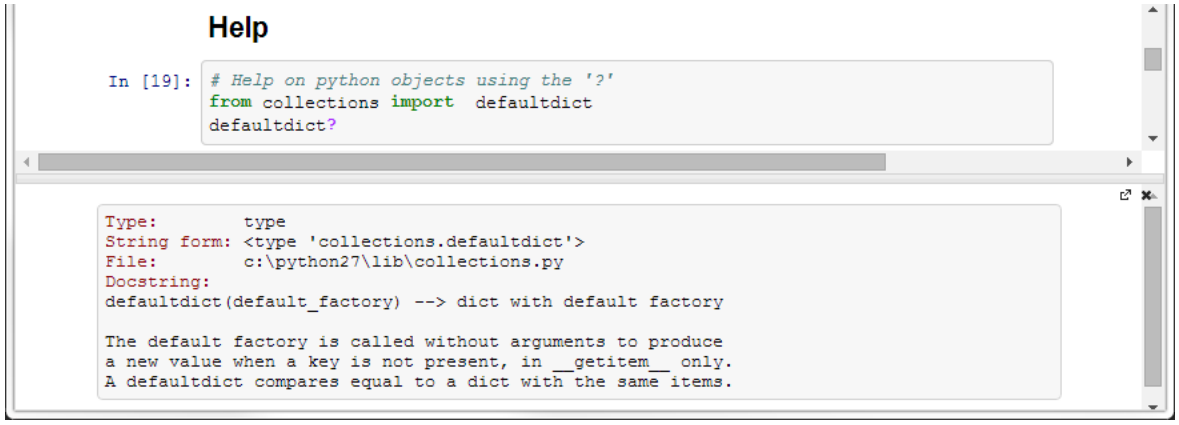
Introspection is when you access an objects docstring from within a development environment. For example, while you are using the function you want to read its documenation. Introspection is implemented in IPython by typing a question mark immediately after e.g., a function name and running the cell. The doc string will appear in a new panel at the bottom of the IPython browser page. For example when you type and run

```
import pyradi.ryutils as ryutils
ryutils.abshumidity?
```

The docstring is displayed as shown below.

http://legacy.python.org/dev/peps/pep-0257/[95] http://www.pythonforbeginners.com/basics/python-docstrings[96]

```
display(Image(filename='images/introspection.png'))
```



1.8.3 General overview

Python data types and how to determine the type of a Python variable.

Everything in Python is an object.

```
myInt = 5
print('{0} is object type {1}'.format(myInt, type(myInt)))
myFloat = np.pi
print('{0} is object type {1}'.format(myFloat, type(myFloat)))
lstInts = [1, 2, 3, 4, ]
print('{0} is object type {1}'.format(lstInts, type(lstInts)))
def myfunction(a):
    print(a)
print('{0} is object type {1}'.format(myfunction, type(myfunction)))
```

```
5 is object type <class 'int'>
3.141592653589793 is object type <class 'float'>
[1, 2, 3, 4] is object type <class 'list'>
<function myfunction at 0x000002251D5D4700> is object type <class '↵
   function'>
```

> In Python, (almost) everything is an object. What we commonly refer to as "variables" in Python are more properly called names. Likewise, "assignment" is really the binding of a name to an object. Each binding has a scope that defines its visibility, usually the block in which the name originates. https://www.jeffknupp.com/blog/2012/11/13/is-python-callbyvalue-or-callbyreference-neither/[97]

If an existing object is bound to a new name, changing the object by using the new name will also change the value object bound to the original name, because it is the same object (just bound to different names). To create a truly new object use the copy() function (for some data types). Slicing (:) lists and tuples creates a copy of the original structure (see later).

https://docs.python.org/2/library/copy.html[98]

http://effbot.org/pyfaq/how-do-i-copy-an-object-in-python.htm[99]

http://pymotw.com/2/copy/[100]

```
import copy

a = [1, 2, 3, ]
b = a
c = copy.copy(a) # shallow copy
# c = copy.deepcopy(a) # deep (recursive) copy
print('a={}'.format(a))
print('b={}'.format(b))
print('c={}'.format(c))
print('----------------------')
b[2] = 'z'
print('a={}'.format(a))
print('b={}'.format(b))
print('c={}'.format(c))
print('----------------------')
b = a[:]  # slicing all makes a copy of 'a' then bind 'b' to this copy
b[2] = 'b'
c[2] = 'c'
print('a={}'.format(a))
print('b={}'.format(b))
print('c={}'.format(c))
```

```
a=[1, 2, 3]
b=[1, 2, 3]
c=[1, 2, 3]
----------------------
a=[1, 2, 'z']
b=[1, 2, 'z']
c=[1, 2, 3]
----------------------
a=[1, 2, 'z']
b=[1, 2, 'b']
c=[1, 2, 'c']
```

1.8.4 Variable id(), is and ==

http://blog.lerner.co.il/why-you-should-almost-never-use-is-in-python/[101]

Every object (variable) has an ID, which can be determined using id(variable). This ID is a handle to the variable, not the value of the variable.

Assigning one variable to another does not create a new variable, the same ID is used, just with a second variable name.

The `is` keyword tests if the same ID is shared, whereas equality == tests the value of the variable.

In the example below `a` and `b` share the same ID handle and are hence the same variable storage location. `c` is a different storage location but with the same contents.

Always use == in logical tests to test for equality of value in the variable; if you use `is` you are testing the ID not the value.

You should only use `is` to test for ID or handle equality such as in `is None` or `is not None`.

```
a = [5, 9]
b = a
c = [5, 9]
print(a, type(a), id(a), a is b, a==b)
print(b, type(b), id(b), a is b, a==b)
print(c, type(c), id(c), a is c, a==c)
```

```
[5, 9] <class 'list'> 2358425454848 True True
[5, 9] <class 'list'> 2358425454848 True True
[5, 9] <class 'list'> 2358425094272 False True
```

Changing `a` or `b` changes the other as well, because they share the same ID and hence the same storage.

```
a[0] = 1
print(a, type(a), id(a), a is b, a==b)
print(b, type(b), id(b), a is b, a==b)
print(c, type(c), id(c), a is c, a==c)
print(' ')
b[0] = 5
print(a, type(a), id(a), a is b, a==b)
print(b, type(b), id(b), a is b, a==b)
print(c, type(c), id(c), a is c, a==c)
```

```
[1, 9] <class 'list'> 2358425454848 True True
[1, 9] <class 'list'> 2358425454848 True True
[5, 9] <class 'list'> 2358425094272 False False

[5, 9] <class 'list'> 2358425454848 True True
[5, 9] <class 'list'> 2358425454848 True True
[5, 9] <class 'list'> 2358425094272 False True
```

The above is however not always true. For example, it does not apply to simple integers. Assigning to `b` now created a new storage location with a new ID.

```
a = 5
b = a
print(a, type(a), id(a), a is b, a==b)
print(b, type(b), id(b), a is b, a==b)
b = 6
print(a, type(a), id(a), a is b, a==b)
print(b, type(b), id(b), a is b, a==b)
```

```
5 <class 'int'> 140709469632416 True True
5 <class 'int'> 140709469632416 True True
5 <class 'int'> 140709469632416 False False
6 <class 'int'> 140709469632448 False False
```

The same applies to `float` variables

```
a = 5.
b = a
print(a, type(a), id(a), a is b, a==b)
print(b, type(b), id(b), a is b, a==b)
b = 6.
print(a, type(a), id(a), a is b, a==b)
print(b, type(b), id(b), a is b, a==b)
```

```
5.0 <class 'float'> 2358429408592 True True
5.0 <class 'float'> 2358429408592 True True
5.0 <class 'float'> 2358429408592 False False
6.0 <class 'float'> 2358429408080 False False
```

Short string seem to share the same ID as well, but as pointed out in the link above: long strings do not share the same ID even if they have the same contents.

```
a = 'ABCD'
b = a
print(a, type(a), id(a), a is b, a==b)
print(b, type(b), id(b), a is b, a==b)
b = 'ABCD'
print(a, type(a), id(a), a is b, a==b)
print(b, type(b), id(b), a is b, a==b)
```

```
ABCD <class 'str'> 2358400150192 True True
ABCD <class 'str'> 2358400150192 True True
ABCD <class 'str'> 2358400150192 True True
ABCD <class 'str'> 2358400150192 True True
```

`None` is a singleton, meaning that there is only one instance of the variable, visible everywhere.

You test for `None` by using `is` or `is not` because you are testing against the singleton handle.

```
a = None
b = 5
print(a is None, b is not None, b is None)
```

```
True True False
```

1.8.5 Strings

Short strings are indicated with a single or double quote (they have the same meaning) but must be used consistently to indicate start and end. Multiline strings are started with three double quotes and ended with three double quotes. Normally the closing three double quotes are on a line of their own. Strings can be formatted following the `format` function, details in

`http://docs.python.org/2/library/string.html#format-string-syntax`[102]

`https://stackoverflow.com/questions/5082452/python-string-formatting-vs-format`[103]

`http://pyformat.info/`[104]

```
strOne = 'one'
strTwo = "two"
strThree = ' one two "three" four'
strMulti = """Line 1
Line2
```

```
Line3
"""
print('-{0}- is object type {1}'.format(strOne, type(strOne)))
print('-{0}- is object type {1}'.format(strTwo, type(strTwo)))
print('-{0}- is object type {1}'.format(strThree, type(strThree)))
print('-{0}- is object type {1}'.format(strMulti, type(strMulti)))

strFormatted = '{0:04d} {1:7.2e} -{2:16s}- {1:16.12f}'.format(4, np.pi,↵
    'my string')
print(strFormatted)
```

```
-one- is object type <class 'str'>
-two- is object type <class 'str'>
- one two "three" four- is object type <class 'str'>
-Line 1
Line2
Line3
- is object type <class 'str'>
0004 3.14e+00 -my string       -   3.141592653590
```

Python's lists, tuples and dictionaries are very powerful structures.

Python uses zero-based indexing, meaning that the first element is indicated as element 0.

Lists lst=[1,2] are mutable, which means that you can change them in place, by appending and inserting elements.

```
#append to an existing list
lstNumbers = [4, 1, 2,  3, 4, ]
lstNumbers.append(5)
lstNumbers.append([2, 6, 7, 8])
print(lstNumbers)
#insert into specific locations into list
lstDiff = [1, 'a', 2]
lstDiff.insert(0,'zero')
lstDiff.insert(2,'wow')
print(lstDiff)
```

```
[4, 1, 2, 3, 4, 5, [2, 6, 7, 8]]
['zero', 1, 'wow', 'a', 2]
```

Get the one'th element of the list at six'th location in the list

```
print(lstNumbers[6][1])
```

```
6
```

Get the third element of the list just preceding it

```
print([1,0,1,2,3][3])
```

```
2
```

Python 2 has limited list unpacking capabilities. The following examples unpacks lists using the itertools module.

```
import itertools

list2d = [[1,2,3],[4,5,6], [7], [8,9]]
merged = list(itertools.chain.from_iterable(list2d))
```

```
print(merged)

list2d = [['foo']*3,['bar']*3,['ton']*3]
merged = list(itertools.chain.from_iterable([['foo']*3,['bar']*3,['ton'↵
   ]*3]))
print(merged)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
['foo', 'foo', 'foo', 'bar', 'bar', 'bar', 'ton', 'ton', 'ton']
```

```
# python 3
#a,*b,c = [0, 1, 2, 3, 4, 5]
```

Sections of lists can be extracted by slicing ':'. Slicing can indicate a start, end, and optionally a stride. Strides will be very important later in numpy arrays, but here we only consider its applications to lists and strings.

```
lstNumbers = [4, 1, 2,  3, 5, ]
print(lstNumbers[0])
print(lstNumbers[-1])
print(lstNumbers[:])
print(lstNumbers[2:])
print(lstNumbers[2:4])
print(lstNumbers[-2:])
print(lstNumbers[0:4:2])
```

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

Slicing can be used to reverse a string or list with a stride if -1, and if the stride is -2 every second letter is left out.

```
a = 'a string to be reversed'
print(a[::-1])
a = 'a string to be reversed'
print(a[::-2])
```

```
desrever eb ot gnirts a
dsee bo nrsa
```

Tuples tup=(1,2) are immutable (cannot change).

```
tupNumbers = (1, 2, 3, 4,)
print('the fourth element is {}'.format(tupNumbers[3]))
try:
    tupNumbers.append(5)
except:
    print('The append attempt raised an exception - it is not allowed!'↵
       )
```

```
the fourth element is 4
The append attempt raised an exception - it is not allowed!
```

Two lists can be used to construct combinations of all the elements in the two lists: use the `itertools.product` function to calculate the outer product of two lists.

```
print(lstNumbers)
```

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

```
import itertools
AA = [1, 2, 3]
BB = ['A','B']
print(AA)
print('')
print(BB)
print('')
for i,(aa,bb) in enumerate(list(itertools.product(AA,BB))):
    print(i,aa,bb)
```

```
[1, 2, 3]

['A', 'B']

0 1 A
1 1 B
2 2 A
3 2 B
4 3 A
5 3 B
```

Dictionaries dic=key1:val1, key2:val2 are like arrays, but the emements can be identified by any data type. In the example below we can extract values from the dictionary based on strings, but these key values can be any type type, even an object or a function. The key values of the dictionary are available by calling the `keys()` function of the dictionary.

The Collections module adds the capability to ease the use of dictionaries, see

`http://docs.python.org/2/library/collections.html`[105]

```
dicBand = {'NIR':'1-2', 'SWIR':'1.5-3', 'MWIR':'3-5',}
dicBand['LWIR']='8-12'
dicBand['VIS']='0.4-0.7'

print(dicBand)
print(dicBand['LWIR'])
print('-------------')

for key in dicBand:
    print('key={} value={}'.format(key,dicBand[key]))
print(' ')
print('-------------')

for key in sorted(dicBand.keys()):
    print('key={} value={}'.format(key,dicBand[key]))
```

```
{'NIR': '1-2', 'SWIR': '1.5-3', 'MWIR': '3-5', 'LWIR': '8-12', 'VIS': ↵
   '0.4-0.7'}
8-12
-------------
key=NIR value=1-2
key=SWIR value=1.5-3
key=MWIR value=3-5
```

```
key=LWIR value=8-12
key=VIS value=0.4-0.7

-------------
key=LWIR value=8-12
key=MWIR value=3-5
key=NIR value=1-2
key=SWIR value=1.5-3
key=VIS value=0.4-0.7
```

http://bugra.github.io/work/notes/2015-01-03/i-wish-i-knew-these-things-when-i-first-learned-python/

Python 3: You could do various interesting operations in keys and items of dictionaries. They are set-like.

```
#this works in Python 3
aa = {'mike': 'male', 'kathy': 'female', 'steve': 'male', 'hillary': '↵
   female'}
bb = {'mike': 'male', 'ben': 'male', 'hillary': 'female'}

aa.keys() & bb.keys() # {'mike', 'hillary'} # these are set-like
aa.keys() - bb.keys() # {'kathy', 'steve'}
# If you want to get the common key-value pairs in the two dictionaries
aa.items() & bb.items() # {('mike', 'male'), ('hillary', 'female')}
```

```
{('hillary', 'female'), ('mike', 'male')}
```

`defaultdict` allows you to assign to a dictionary[key] element even if the key'ed entry does not exist.

```
# from collections import defaultdict
# dictionary = defaultdict(list) # defaults to list
# for k, v in ls:
#     dictionary[k].append(v)
```

```
a = {'x': 1, 'y':2, 'z':3}
b = {'y': 5, 's': 10, 'x': 3, 'z': 6}
c = a
c.update(b)
print(c)
```

```
{'x': 3, 'y': 5, 'z': 6, 's': 10}
```

```
aa = {k: sum(range(k)) for k in range(10)}
print(aa)
# get the max value and the key where this occurs
print(max(zip(aa.values(), aa.keys())))
# traverse dict from max value to min value, with key
print(sorted(zip(aa.values(), aa.keys()), reverse=True))
```

```
{0: 0, 1: 0, 2: 1, 3: 3, 4: 6, 5: 10, 6: 15, 7: 21, 8: 28, 9: 36}
(36, 9)
[(36, 9), (28, 8), (21, 7), (15, 6), (10, 5), (6, 4), (3, 3), (1, 2), ↵
   (0, 1), (0, 0)]
```

Control flow in Python uses `for` or `while`.

The `enumerate` function is a very useful means to iterate over a list or tuple and get the index as well as the list value at that index.

The `while` loop executes as long as the conditional in the `while` statement is true. If the conditional is false, the `else` clause is executed.

```
a = [ 'a', 'b', 'c', ]
for item in a:
    print(item)
print('-------------')

strTmp = 'This string'
for i,item in enumerate(strTmp):
    print('character {} is {}'.format(i,item))
print('-------------')

for i,item in enumerate(a):
    print('index {} has value {}'.format(i,item))
print('-------------')

while False:
    print(a)
else:
    print('it is False!!')
print('-------------')

count = 0
while count < 5:
    print(f'{count} is  less than 5')
    count = count + 1
```

```
a
b
c
-------------
character 0 is T
character 1 is h
character 2 is i
character 3 is s
character 4 is
character 5 is s
character 6 is t
character 7 is r
character 8 is i
character 9 is n
character 10 is g
-------------
index 0 has value a
index 1 has value b
index 2 has value c
-------------
it is False!!
-------------
0 is  less than 5
1 is  less than 5
2 is  less than 5
3 is  less than 5
4 is  less than 5
```

Zipping pairs two lists element by element into a new list of matched tuples. You can even enumerate over zipped lists:

```
a = [ 'a', 'b', 'c', ]
```

```
b = [ '1', '2', '3', ]
print(zip(a,b))
```

```
<zip object at 0x000002251D5D35C0>
```

You can enumerate over the zipped list to extract the paired tuples

```
print('Enumerate over all zipped tuples')
for i,(ai,bi) in enumerate(zip(a,b)):
    print(i, ai, bi)

print('Alternative method, not quite as clear and instructive')
for i,aa in enumerate(a):
    print( 1, a[i], b[i])

print('C-Style programming to achieve the same effect')
if len(a)==len(b):
    for i in range(len(b)):
        print(i, a[i],b[i])
```

```
Enumerate over all zipped tuples
0 a 1
1 b 2
2 c 3
Alternative method, not quite as clear and instructive
1 a 1
1 b 2
1 c 3
C-Style programming to achieve the same effect
0 a 1
1 b 2
2 c 3
```

https://stackoverflow.com/questions/19777612/python-range-and-zip-object-type

https://www.journaldev.com/15891/python-zip-function

In Python 2, the objects zip and range behave as returning lists.

In Python 3 they were changed to generator-like objects which produce the elements on demand rather than expand an entire list into memory. One advantage was greater efficiency in their typical use-cases (e.g. iterating over them). To retrieve all the output at once into a familiar list object, you may simply call `list()` to iterate and consume the contents.

`list()` Convert an iterable (tuple, string, set, dictionary) to a list. It seems that the change is done in-place.

We can also extract data from the Python zip function. To extract zip, we have to use the same zip() function. But we have add an asterisk(*) in front of the name of the zip object.

```
data = [(1, 2, 3), ('a', 'b', 'c')]
print(data)
print('---------')
zipped = zip(*data)
print(zipped)
print(type(zipped))
print(list(zipped)) # ??? list(zipped) changes zipped to a list, no ↵
   longer zipped object
print(type(list(zipped)))
print(type(zipped))
```

```
print('---------')
# zipped = zip(*data)
unzipped = zip(*zipped)
print(unzipped)
print(type(unzipped))
print(list(unzipped))
```

```
[(1, 2, 3), ('a', 'b', 'c')]
---------
<zip object at 0x000002251D603AC0>
<class 'zip'>
[(1, 'a'), (2, 'b'), (3, 'c')]
<class 'list'>
<class 'zip'>
---------
<zip object at 0x000002251D603600>
<class 'zip'>
[]
```

From http://mastering-python-lists.blogspot.co.za/p/vectorized-code.html[106]. Given a function and a list of values [1, 2, 3, 4, 5] generate every sequential 3-tuple (1, 2, 3) (2, 3, 4) (3, 4, 5) and find the 3-tuple which produces the largest return value of some measure (the function dummyfn()). This requires some antics in an iterative loop form, but can be written in one line using zip.

Start with a list of numbers and take 3 slices where each slice is offset by 1 from the previous slice. Now, if you line up the slices they form a triangular matrix and the columns form the groups you are looking for. You can generate the groups by passing the slices to the zip function The zip function automatically stops when it reaches the end of the shortest slice. This prevents the "off by 1" errors that are so common in this type of code.

The max function works by comparing values with the < operator, but you can override this by specifying a key function. Each value is passed to this function and it is the results of the function that are compared. So in this example you would say max(groups, key=F). The max function automatically keeps track of the largest return value seen so far so you don't need to write any "test and swap" code. Note that F(value) is used internally for comparison purposes but this is not the value returned by max. Besides zip and max, there are other built in functions that operate on entire lists: all, any, filter, map, min, reduce, reversed, sorted, and sum.

```
num_list = [1, 2, 3, 4, 5]
def dummyfn(a): return True if a[0]+a[1]>a[2] else False
max( zip(num_list, num_list[1:], num_list[2:]), key=dummyfn )
```

```
(2, 3, 4)
```

```
# if function is written as lambda function
max( zip(num_list, num_list[1:], num_list[2:]), key=lambda a: True if a↵
   [0]+a[1]>a[2] else False )
```

```
(2, 3, 4)
```

List/dictionary comprehension is a shorthand way to define a list/dictionary. In the first example create a new list of square root values of items from an old list, but only for even values. In the second examples a dictionary is built using dictionary comprehension. Note that zip takes two lists and create a new list where elements of the new list are tuples with matching elements from the two input lists - pair them up element by element, much like a zip closing up.

http://carlgroner.me/Python/2011/11/09/An-Introduction-to-List-Comprehensions-in-P
html[107]

http://www.secnetix.de/olli/Python/list_comprehensions.hawk[108]

http://docs.python.org/2.7/library/functions.html[109]

http://howchoo.com/g/how-to-use-list-comprehension-in-python[110]

http://treyhunner.com/2015/12/python-list-comprehensions-now-in-color/[111]

```
import math
nlist = [0, 1, 2, 3, 4]
newl = [math.sqrt(i) for i in nlist if not i%2]
print(newl)

keys = ['a', 'n', 'z']
values = ['testing', 'this', 'dict']
d = {key: value for (key, value) in zip(keys, values)}
print(d)

print(zip(keys, values))
```

```
[0.0, 1.4142135623730951, 2.0]
{'a': 'testing', 'n': 'this', 'z': 'dict'}
<zip object at 0x000002251D616440>
```

Nested loops can also be used in list comprehensions, just be careful of the order of writing the nested for loops. Write the for loops in the same sequence at it was written as normal for loops.

```
flattened = []
unflat = [[1,2,3],[4,5,6],[7,8,9]]
for row in unflat:
    for n in row:
        flattened.append(n)
print(flattened)
```

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

```
flattenedlc = [n for row in unflat for n in row]
print(flattenedlc)
```

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

You can build lists of list or tuples to any depth and access them by indexing the elements. But be careful not to try and construct multi-dimensional arrays this way - numpy provides much better functionality.

```
a = [[0, 1, 2, 3], [4, 5, 6, 7]]
print(a)
print(a[0])
print(a[0][3])
```

```
[[0, 1, 2, 3], [4, 5, 6, 7]]
[0, 1, 2, 3]
3
```

1.8.6 Overloaded multiplication for non-numbers

```
print(20*'-')
```

```
--------------------
```

Watch out when using the `n *` construct with lists or dicts:

```
print(5 *[None])
```

```
[None, None, None, None, None]
```

```
# the same single dict object repeated with 3 x references in the list
dic = 3 * [{}]
print(dic)
# assigning to the single dict once,
# but all entries in the list reference the same dict
dic[0]['test'] = 'mystring'
print(dic)
```

```
[{}, {}, {}]
[{'test': 'mystring'}, {'test': 'mystring'}, {'test': 'mystring'}]
```

```
lst = 3 * [[]]
print(lst)
lst[0].append('mystring')
print(lst)
```

```
[[], [], []]
[['mystring'], ['mystring'], ['mystring']]
```

Rather create multiple instances add them one at a time

```
dic = []
lst = []
for i in [0,1,2]:
    dic.append({})
    lst.append([])

lst[0].append('mystring')
dic[0]['test'] = 'mystring'
print(lst)
print(dic)
```

```
[['mystring'], [], []]
[{'test': 'mystring'}, {}, {}]
```

1.8.7 Named Tuples: collections.namedtuple

`https://docs.python.org/2/library/collections.html#collections.namedtuple`[112]

`collections.namedtuple(typename, field\_names[, verbose=False][, rename=False])`

Returns a new tuple subclass named **typename**. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable.

The `field\_names` are a sequence of strings or single string with each fieldname separated by whitespace and/or commas.

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a keyword such as class, for, return, global, pass, print, or raise.

If `rename`is true, invalid fieldnames are automatically replaced with positional names. For example, `['abc', 'def', 'ghi', 'abc']` is converted to `['abc', '\_1', 'ghi', '\_3']`, eliminating the keyword `def` and the duplicate fieldname `abc`.

If `verbose` is true, the class definition is printed just before being built.

Standard tuples use integer indexes to access the tuple members. This is very easy to use, but it becomes cumbersome to remember which integer index associates with which member:

```
# http://pymotw.com/2/collections/namedtuple.html
bob = ('Bob', 30, 'male')
print('Representation:{}'.format(bob))

jane = ('Jane', 29, 'female')
print('\nField by index: {}'.format(jane[0]))

print('\nFields by index:')
for p in [ bob, jane ]:
    # use tuple unpacking to unpack the tuple in to its elements
    print('{} is a {} year old {}'.format(*p))
```

```
Representation:('Bob', 30, 'male')

Field by index: Jane

Fields by index:
Bob is a 30 year old male
Jane is a 29 year old female
```

A `namedtuple` is an object type in the `collections` module that allows you to name the members in the tuple. The members can then be accessed by name.

`http://pymotw.com/2/collections/namedtuple.html`[113]

`namedtuple` instances are just as memory efficient as regular tuples because they do not have per-instance dictionaries. Each kind of `namedtuple` is represented by its own class, created by using the `namedtuple()` factory function. The arguments are the name of the new class and a string containing the names of the elements/members.

The `namedtuple` class checks for standard Python keywords (not allowed as field names) and repeated/duplicate field member names within one `namedtuple`. The optional `rename=True` option allows the `namedtuple` class to rename the disallowed field names to other names.

`namedtuple` is backawards compatible, you can still use integer index values and unpacking, as well as the named members.

```
#http://pymotw.com/2/collections/namedtuple.html
import collections

#note that field names are all in a single string, with whitespace ↵
   separation
Person = collections.namedtuple('Person', 'name age gender')

print('Type of Person: {}'.format(type(Person)))

bob = Person(name='Bob', age=30, gender='male')
```

```
print('\nRepresentation: {}'.format(bob))

jane = Person(name='Jane', age=29, gender='female')
print('\nField by name: {}'.format(jane.name))
print('Field by name: {}'.format(jane[0]))

print('\nFields by index:')
for p in [ bob, jane ]:
    print('{} is a {} year old {}'.format(*p))
```

```
Type of Person: <class 'type'>

Representation: Person(name='Bob', age=30, gender='male')

Field by name: Jane
Field by name: Jane

Fields by index:
Bob is a 30 year old male
Jane is a 29 year old female
```

http://stackoverflow.com/questions/2970608/what-are-named-tuples-in-python[114]

namedtuples can be used similarly to struct or other common record types, except that they are immutable.

You should use named tuples instead of tuples anywhere you think object notation will make your code more pythonic and more easily readable. For example to represent very simple value types, particularly when passing them as parameters to functions. It makes the functions more readable, without seeing the context of the tuple packing.

Furthermore, you can also replace ordinary immutable classes that have no functions, only fields with them. You can even use your named tuple types as base classes:

```
class Point(namedtuple('Point', 'x y')):
    [...]
```

```
from collections import namedtuple
Point = namedtuple('Point', 'x y')
pt1 = Point(1.0, 5.0)
pt2 = Point(2.5, 1.5)

from math import sqrt
line_length = sqrt((pt1.x-pt2.x)**2 + (pt1.y-pt2.y)**2)
```

Named tuples can be converted to dictionaries using pt1._asdict() which returns an ordered dictionary. To get a standard dictionary use dict(pt1._asdict()). The members of the resulting dictionary are not immutable and can be changed if required. The example below also shows how to build a namedtuple from a dict.

```
print('{} {}'.format(pt1, type(pt1)))
print('{} {}'.format(pt1._asdict(), type(pt1._asdict())))
print('{} {}'.format(dict(pt1._asdict()), type(dict(pt1._asdict()))))
pt1d = pt1._asdict()
pt1d['x'] = 1000
print('{} {}'.format(pt1d, type(pt1d)))
```

```
# build a namedtuple from a dict:
pt1 = Point(**pt1d)
print('{} {}'.format(pt1, type(pt1)))

#get the field names from namedtuple
print('Point field names are {}'.format(pt1._fields))
```

```
Point(x=1.0, y=5.0) <class '__main__.Point'>
{'x': 1.0, 'y': 5.0} <class 'dict'>
{'x': 1.0, 'y': 5.0} <class 'dict'>
{'x': 1000, 'y': 5.0} <class 'dict'>
Point(x=1000, y=5.0) <class '__main__.Point'>
Point field names are ('x', 'y')
```

https://docs.python.org/2/library/collections.html#collections.namedtuple[115]

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```
class Point(namedtuple('Point', 'x y')):
    __slots__ = ()
    @property
    def hypot(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5
    def __str__(self):
        return 'Point: x={:6.3f}  y={:6.3f}  hypot={:6.3f}'.format(self↵
           .x, self.y, self.hypot)

for p in Point(3, 4), Point(14, 5/7.):
    print(p)
```

```
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the _fields attribute:

Point3D = namedtuple('Point3D', Point._fields ('z',))+

Default values can be implemented by using _replace() to customize a prototype instance:

```
Account = namedtuple('Account', 'owner balance transaction_count')
default_account = Account('<owner name>', 0.0, 0)
johns_account = default_account._replace(owner='John', balance=50)
print(johns_account)
```

```
Account(owner='John', balance=50, transaction_count=0)
```

```
# def preparefiles(filename):
#     if os.path.isfile(filename):
#         os.remove(filename)
#     conn = sqlite3.connect(filename)
#     conn.execute('''CREATE TABLE COMPANY
#            (ID INT PRIMARY KEY     NOT NULL,
#            NAME           TEXT    NOT NULL,
#            AGE            INT     NOT NULL,
#            ADDRESS        CHAR(50),
#            SALARY         REAL);''')
#     conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
#          VALUES (1, 'Paul', 32, 'California', 20000.00 )");
```

```
#     conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
#           VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");
#     conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
#           VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");
#     conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
#           VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");
#     conn.commit()
#     conn.close()
```

https://docs.python.org/2/library/collections.html#collections.namedtuple[116]

Named tuples are especially useful for assigning field names to result tuples returned by the csv or sqlite3 modules. Note that the table names and the name tuple member names are not the same:

```
from collections import namedtuple

EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, address, ↵
   paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open('data/test.csv', "↵
   rt"))):
    print('from csv: {} {} '.format(emp.name, emp.paygrade))

import sqlite3
conn = sqlite3.connect('data/test.db3')
cursor = conn.cursor()
cursor.execute('SELECT NAME,AGE,ADDRESS,SALARY FROM COMPANY')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print('from sqlite3: {} {} {} {}'.format(emp.name, emp.age, emp.↵
       address, emp.paygrade))
conn.close ;
```

```
from csv: Paul  20000.0
from csv: Allen  15000.0
from csv: Teddy  20000.0
from csv: Mark  65000.0
from sqlite3: Paul 32 California 20000.0
from sqlite3: Allen 25 Texas 15000.0
from sqlite3: Teddy 23 Norway 20000.0
from sqlite3: Mark 25 Rich-Mond  65000.0
```

1.8.8 Generators, iterators and yield — more detail to follow.

http://anandology.com/python-practice-book/iterators.html[117]

http://www.jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-

https://wiki.python.org/moin/Generators[119]

http://www.bogotobogo.com/python/python_generators.php[120]

http://zetcode.com/lang/python/itergener/[121]

http://nbviewer.ipython.org/github/empet/pytwist/blob/master/Generators-and-Dynami
ipynb[122]

1.8.9 Functions as objects

A function name is *just a name* bound to a function object and can be handled just as *any other name*. Study the following code to see how the predicate function is passed as a function parameter. This example counts the number of times that a certain condition ocurred.

https://www.jeffknupp.com/blog/2013/11/29/improve-your-python-decorators-explained

```
def is_even(value):
    """Return True if *value* is even.
    """
    #this is a comment
    return (value % 2) == 0

print(is_even(4))
print(is_even(5))

def count_occurrences(target_list, predicate):
    """Return the number of times applying the callable *predicate* to ↵
       a
    list element returns True.
    """
    return sum([1 for e in target_list if predicate(e)])

my_predicate = is_even
my_list = [2, 4, 6, 7, 9, 11]
result = count_occurrences(my_list, my_predicate)
print(result)
```

```
True
False
3
```

Here is another example, the function name is an entry into a dictionary.

http://code.activestate.com/recipes/65126-dictionary-of-methodsfunctions/[124]

```
import string

def function1(i):
    print('called function 1 with {}'.format(i))

def function2(i):
    print('called function 2 with {}'.format(i))

tokenDict = {"cat":function1, "dog":function2}

# simulate, say, lines read from a file
lines = ["cat","cat","dog"]

for i,line in enumerate(lines):
    tokenDict[line](i)
```

```
called function 1 with 0
called function 1 with 1
called function 2 with 2
```

1.8.10 Don't use mutable objects as default function parameters

Python will create the mutable object (default parameter) the first time the function is defined (i.e. the function is created or imported) - not when it is called.

http://nbviewer.ipython.org/github/rasbt/python_reference/blob/master/not_so_obvious_python_stuff.ipynb?create=1[125]

```
def append_to_list(value, def_list=[]):
    def_list.append(value)
    return def_list

my_list = append_to_list(1)
print(my_list)

my_other_list = append_to_list(2)
print(my_other_list)
```

```
[1]
[1, 2]
```

```
import time
def report_arg(my_default=time.time()):
    print(my_default)

report_arg()
time.sleep(1)
report_arg()
```

```
1629740885.3590853
1629740885.3590853
```

1.8.11 Reading and writing files

Use the `with` construct, it is more robust to errors. In this code we first create a random collection of words. It is then written to a file. This file is then read in and processed, printing the results each time.

http://effbot.org/zone/python-with-statement.htm[126]

```
import random, string, os

#the join function uses the given string to join the elements given in ↵
   the function parameter list
a = ['a','b','c']
print('*'.join(a))
print('-o0o-'.join(a))

#first create some random text - for extra marks figure out how this is↵
    done...
text = '\n'.join([' '.join(['' .join(random.sample(string.ascii_letters,
                random.randint(1,7))) for i in range(10)]) for i in ↵
                   range(4)])
print('Text to be written to file')
print(text)

#Write the text to file
```

```
filename = "x.txt"
with open(filename,'w') as f:
    f.write(text)

print('\nSingle string dump of all the text read in from the file')
with open("x.txt",'r') as f:
    data = f.read()
    print(data)

with open(filename,'r') as f:
    lines = f.readlines()
    print('\nRead in as lines from the file')
    print(lines)
    print('\nPrinting the lines one at a time after removing the ↵
       newline')
    for line in lines:
        print(line.strip())
    print('\nPrinting the words for each line')
    for line in lines:
        words = line.split()
        print(words)
#delete the file
try:
    os.remove(filename)
except OSError:
    pass
```

```
a*b*c
a-o0o-b-o0o-c
Text to be written to file
HKkU qjVBH KlAX LrexmV w YL RPH jLFRkiG xkjGWfA gUZduQ
JTj KLfzWk iWvNkJQ rb Z n ClX Y npINqBz aNhdZie
E cHT oC aHvEwiM PKsHh htxWvgT nhPNvY EI aCsm MsyBIUT
xa JHlc TdDZqw KEktBCJ v qbcOx MmYLS XcHV Satcp gYQ

Single string dump of all the text read in from the file
HKkU qjVBH KlAX LrexmV w YL RPH jLFRkiG xkjGWfA gUZduQ
JTj KLfzWk iWvNkJQ rb Z n ClX Y npINqBz aNhdZie
E cHT oC aHvEwiM PKsHh htxWvgT nhPNvY EI aCsm MsyBIUT
xa JHlc TdDZqw KEktBCJ v qbcOx MmYLS XcHV Satcp gYQ

Read in as lines from the file
['HKkU qjVBH KlAX LrexmV w YL RPH jLFRkiG xkjGWfA gUZduQ\n', 'JTj ↵
   KLfzWk iWvNkJQ rb Z n ClX Y npINqBz aNhdZie\n', 'E cHT oC aHvEwiM ↵
   PKsHh htxWvgT nhPNvY EI aCsm MsyBIUT\n', 'xa JHlc TdDZqw KEktBCJ v ↵
   qbcOx MmYLS XcHV Satcp gYQ']

Printing the lines one at a time after removing the newline
HKkU qjVBH KlAX LrexmV w YL RPH jLFRkiG xkjGWfA gUZduQ
JTj KLfzWk iWvNkJQ rb Z n ClX Y npINqBz aNhdZie
E cHT oC aHvEwiM PKsHh htxWvgT nhPNvY EI aCsm MsyBIUT
xa JHlc TdDZqw KEktBCJ v qbcOx MmYLS XcHV Satcp gYQ

Printing the words for each line
['HKkU', 'qjVBH', 'KlAX', 'LrexmV', 'w', 'YL', 'RPH', 'jLFRkiG', '↵
   xkjGWfA', 'gUZduQ']
['JTj', 'KLfzWk', 'iWvNkJQ', 'rb', 'Z', 'n', 'ClX', 'Y', 'npINqBz', '↵
   aNhdZie']
['E', 'cHT', 'oC', 'aHvEwiM', 'PKsHh', 'htxWvgT', 'nhPNvY', 'EI', 'aCsm↵
   ', 'MsyBIUT']
```

```
['xa', 'JHlc', 'TdDZqw', 'KEktBCJ', 'v', 'qbcOx', 'MmYLS', 'XcHV', '↵
   Satcp', 'gYQ']
```

1.8.12 Reading and writing raw binary files

Binary files are handled the same as text files, but with file open modes `rb` or `wb`:

```
# Read the entire file as a single byte string
with open('binaryfile.bin', 'rb') as f:
    data = f.read()

# Write binary data to a file
with open('binaryfile.bin', 'wb') as f:
    f.write(b'Hello World')
```

When reading binary all data returned will be in the form of byte strings, not text strings. Similarly, when writing, you must supply data in the form of objects that expose data as bytes.

Using Python arrays, values can be written without conversion to a bytes object. The data type must be declared in the first parameter of the `array.array` function. These data types[127] include signed and unsigned shorts, ints, longs, doubles, and unicode.

```
import array
nums = array.array('i', [1, 2, 3, 4])
with open('data.bin','wb') as f:
    f.write(nums)
```

`https://docs.python.org/2/library/array.html`[128]

`http://chimera.labs.oreilly.com/books/1230000000393/ch05.html#\_discussion\_75`[129]

The following code writes two unsigned integers, two doubles and a numpy array to raw binary format:

```
import numpy as np
import array
img = np.ones((3,3))
with open('binfile.bin','wb') as fo:
    header = array.array('I',[512, 1024])
    fo.write(header)
    header = array.array('d',[700, 1250])
    fo.write(header)
    img.tofile(fo)
```

1.8.13 Watch out for integer numbers

It seems that Numpy interprets 400 as an integer int32. When this int32 value is raised to the fourth power, the value overflows, resulting in a *negative* number. This error can easily happen when entering integer temperature values, raised to a high power.

In [130], all temperature input values are always converted to float with `.astype(float)` (but only in [131]).

```
import numpy as np
tt = np.asarray([400, 500])
print(tt.dtype)
print(np.power(tt, 4))
print(tt**4)

tt = np.asarray([400, 500]).astype(float)
print(tt.dtype)
print(np.power(tt, 4))
```

```
int32
[ -169803776 -1924509440]
[ -169803776 -1924509440]
float64
[2.56e+10 6.25e+10]
```

1.8.14 Reading user keyboard input

Reading user input - this even works in the IPython notebook! Strings must be in quotes. You can even enter lists and dictionaries!

```
# foo=input('Please enter a value:')
# print(foo)
```

1.8.15 Accessing files on the internet

The Python [132] package makes is relatively easy to access a file on the internet. See also the functions [133], [134], and [135].

1.8.16 Python functions

```
def square(value):
    return value ** 2

print(square(5))
print(square(5.))
```

```
25
25.0
```

Functions can return anything, here it returns (1) a tuple of values, (2) a function that can be called. If you only want some of the values in the returned tuple, indicate those to ignore with underscore _.

```
def square(value):
    dicStr = {0:'zero',2:'two',5:'five',}
    rtnval = (value, value ** 2, dicStr[value])
    return rtnval

print(square(5))
print(square(5)[1])
print(square(5)[-1])
print('--------------')
```

```
def squarefn():
    return square
print(squarefn()(5))
print('-------------')

_, _, rtn = squarefn()(5)
print(rtn)
```

```
(5, 25, 'five')
25
five
-------------
(5, 25, 'five')
-------------
five
```

Functions can have named parameters where values are assigned by name. Furthermore, function parameters can define default values to be used if the user does not supply a value.

```
def power(value, power=2):
    return value ** power

print(power(value=5))
print(power(value=5, power=4))
```

```
25
625
```

1.9 Exception and Error Handling

Python has exception handling, with the try/exception structure. The code in the try block is evaluated and if an error occurred the code in the except block is executed to handle the error condition.

```
a = (1, 2)
try:
    a.append(4)
except:
    print('Appending to a tuple is not allowed')
```

```
Appending to a tuple is not allowed
```

1.10 Python Standard Library Logger

The `logging` module in the standard library provides a reasonably powerful logger capability. This examples shows how to log to two files at the same time. The case in point is for a session-level logfile, plus any number of lower level detailed log files.

The `logging` module provides a global-scope object, so there is no need to pass the object along between functions or objects.

The global `logging` object provides the functionality to have any number of individual loggers, which can be accessed by using a key (the filename) to indicate which logger you want to use. The file that is physically created on disk is named using the key, appended with '.log'.

The logger shown here opens handlers to cout and a disk file, each providing different levels of logging. The logger is created by the function `setUpLogFile` and is subsequently activated by code of the form `logging.getLogger(filename)`, where `filename` is the logfile filename (which also serves as the key to the logger).

The example code below provides the functionality where the user can provide a path to where the logfile must be created on disk (relative to where the current working directory).

The Python logging module provides the following threshold levels: CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET in order of severity. When logging, all messages marked with a level equal to, or higher than the threshold level, will be printed. At the highest level only critical messages will be printed, whereas at the debug level all higher level prints are also performed.

The class `SpecialFormatter` defines how the different levels of output are formatted.

This code is based on an example by Lynette van der Westhuizen.

```
import logging
import datetime as dt
import sys
import os.path

#↵
    ================================================================================

def setUpLogFile(logfilename, saveDir, logLevelCmnd=logging.WARNING, ↵
    logLevelFile=logging.DEBUG):
    """Set up the log file for IDA.

    Args:
        | logfilename (string) : Name of the log file - this is also ↵
          the key to access the logger.
        | saveDir (string) : Directory to save the log file, relative ↵
          to cwd.
        | logLevel (string): log level (logging.ERROR, logging.WARNING,↵
           logging.INFO, logging.DEBUG)

    Returns:
        | newLogger (logging file) : Retruns a log file with the name [↵
          logfilename]Log.
                              A log file [logfilename]Log.log is created↵
                                  in saveDir.
    Raises:
        | None
    """

    #--- Get the log file and to log all logLevel
    newLogger = logging.getLogger(logfilename)
    newLogger.setLevel(logLevelCmnd)
    newLogger.handlers=[] #Ensure no handlers so that duplicates aren't↵
        added in the event the code crashed and logger didn't close

    #--- Set a file handler log from level :
    filepath = os.path.join(saveDir, logfilename.lower())
    fileHdlr = logging.FileHandler("{}.log".format(filepath),'w')
    fileHdlr.setLevel(logLevelFile)
    fileHdlr.setFormatter(SpecialFormatter())
    newLogger.addHandler(fileHdlr)

    #--- Set a command window handler log from level WARNING
```

```
    cmdHdlr = logging.StreamHandler()
    cmdHdlr.setLevel(logLevelCmnd)
    cmdHdlr.setFormatter(SpecialFormatter())
    newLogger.addHandler(cmdHdlr)

    #--- Print the log file creation time
    printCreationTime = '\n↵
       ============================================================\n'+\
                        'Date of execution: {} {}\n'.format(dt.datetime↵
                           .now(), logfilename) +\
                        '↵
                           ============================================================
                           n\n'.format(logfilename)
    newLogger.info(printCreationTime)

    return newLogger

#↵
   ================================================================================

class SpecialFormatter(logging.Formatter):
    """Sets the formats of the levels to print for a logging file.

        All messages printing to the log file will have the format as ↵
           specified
        by the log file level.
        http://stackoverflow.com/questions/1343227/can-pythons-logging-↵
           format-be-modified-depending-on-the-message-log-level
    """
    #--Set format: format valid for python 2
    FORMATS = {
                logging.CRITICAL: "CRITICAL - module %(module)s line %(↵
                   lineno)d: %(message)s",
                logging.ERROR   : "ERROR - module %(module)s line %(↵
                   lineno)d: %(message)s",
                logging.WARNING : "WARNING - module %(module)s line %(↵
                   lineno)d: %(message)s",
                logging.INFO    : "INFO %(message)s",
                logging.DEBUG   : "DEBUG - module %(module)s line %(↵
                   lineno)d: %(message)s",
                'DEFAULT'       : "%(levelname)s: %(message)s"}

    #---Define format
    def format(self, record):
        """Defines the logging message format.
        """
        self._fmt = self.FORMATS.get(record.levelno, self.FORMATS['↵
           DEFAULT'])
        return logging.Formatter.format(self, record)
```

The logger is demonstrated here by writing one session-level logfile and two lower-level logfiles. Note that, because the `logging` module is global, you do not need to copy it around. The appropriate logger must be activated by the `logging.getLogger()` function - all subsequent logging with be to that specific logger.

```
#set up the session logger
analysisName = 'analysis'
saveDir = '.'
setUpLogFile(analysisName, '.', logLevelCmnd=logging.INFO, logLevelFile↵
```

```
   =logging.INFO)
anaLog = logging.getLogger(analysisName)
anaLog.error('this is an error message to anaLogFile')

#set up the specialist loggers
for i in ['a','b']:
    setUpLogFile(i, '.', logLevelCmnd=logging.INFO, logLevelFile=↵
       logging.INFO)

#use the session and specialist loggers.
for i in ['a','b']:
    anaLog.error('this is an error message to anaLogFile {}'.format(i) ↵
       )
    iLog = logging.getLogger(i)
    iLog.error('writing to logfile {} at ERROR level'.format(i) )
    iLog.info('writing to logfile {} at INFO level'.format(i) )
```

```
============================================================
Date of execution: 2021-08-23 19:48:06.526288 analysis
============================================================


this is an error message to anaLogFile

============================================================
Date of execution: 2021-08-23 19:48:06.527289 a
============================================================



============================================================
Date of execution: 2021-08-23 19:48:06.528288 b
============================================================


this is an error message to anaLogFile a
writing to logfile a at ERROR level
writing to logfile a at INFO level
this is an error message to anaLogFile b
writing to logfile b at ERROR level
writing to logfile b at INFO level
```

1.11 The switch/case Statement in Python

All of this comes from `http://www.pydanny.com/why-doesnt-python-have-switch-case.html`[136]

Python does not have a switch or case statement. To get around this fact, we use dictionary mapping:

```
def numbers_to_strings(argument):
    switcher = {
        0: "zero",
        1: "one",
        2: "two",
    }
    return switcher.get(argument, "nothing")
```

```
print(numbers_to_strings(1))
```

```
one
```

Dictionary Mapping for Functions

```
def zero():
    return "zero"

def one():
    return "one"

def numbers_to_functions_to_strings(argument):
    switcher = {
        0: zero,
        1: one,
        2: lambda: "two",
    }
    # Get the function from switcher dictionary
    func = switcher.get(argument, lambda: "nothing")
    # Execute the function
    return func()

print(numbers_to_strings(1), numbers_to_strings(2))
```

```
one two
```

If we don't know what method to call on a class, we can use a dispatch method to determine it at runtime.

```
print(str)
```

```
<class 'str'>
```

```
class Switcher(object):
    def numbers_to_methods_to_strings(self, argument):
        """Dispatch method"""
        # prefix the method_name with 'number_' because method names
        # cannot begin with an integer.
        method_name = 'number_' + str(argument)
        # Get the method from 'self'. Default to a lambda.
        method = getattr(self, method_name, lambda: "nothing")
        # Call the method as we return it
        return method()

    def number_0(self):
        return "zero"

    def number_1(self):
        return "one"

    def number_2(self):
        return "two"

sw = Switcher()

print(sw.numbers_to_methods_to_strings(0),sw.↵
   numbers_to_methods_to_strings(2))
```

```
zero two
```

1.11.1 Pathlib module

https://pbpython.com/pathlib-intro.html

https://pbpython.com/notebook-process.html

https://pymotw.com/3/pathlib/

https://towardsdatascience.com/10-python-file-system-methods-you-should-know-799f90ef13c2

https://realpython.com/python-pathlib/

https://github.com/chris1610/pbpython/blob/master/extras/Pathlib-Cheatsheet.pdf

What follows below are almost verbatim extracts from some of the above links, I do not claim originality in this code.

Pathlib is an object oriented interface to the filesystem and provides a more intuitive method to interact with the filesystem in a platform agnostic and pythonic manner. The pathlib library is included in all versions of python >= 3.4

It's a similar thought process to the os.path method of joining the current working directory (using Path.cwd()) with the various subdirectories and file locations. It is much easier to follow because of the clever overriding of the / to build up a path in a more natural manner than chaining many os.path.joins together. The / can join several paths or a mix of paths and strings (as above) as long as there is at least one Path object. If you do not like the special / notation, you can do the same thing with the .joinpath() method.

`Path.home()` is a path to your home directory.

The added benefit of these methods is that you are creating a Path object vs. just a string representation of the path. The actual string representation is the same but the variable type is a pathlib.PosixPath or pathlib.WindowsPath. The fact that the path is an object means we can do a lot of useful actions on the object. It's also interesting that the path object "knows" it is on a Linux system (aka Posix) and internally represents it that way without the programmer having to tell it. The benefit is that the code will run the same on a Windows machine and that the underlying library will take care of (m)any Windows eccentricities.

```
import pandas as pd
from pathlib import Path
from datetime import datetime
import os
import time
```

Independently of the operating system you are using, paths are represented in Posix style, with the forward slash as the path separator. On Windows, you will see something like the following. Still, when a path is converted to a string, it will use the native form, for instance with backslashes on Windows:

```
Path.cwd()
```

```
WindowsPath('K:/WorkN/ComputationalRadiometry')
```

```
str(Path.cwd())
```

```
'K:\\WorkN\\ComputationalRadiometry'
```

Possibly the most unusual part of the pathlib library is the use of the / operator. For a little peek under the hood, let us see how that is implemented. This is an example of operator overloading: the behavior of an operator is changed depending on the context. Python implements operator overloading through the use of double underscore methods (a.k.a. dunder methods).

The / operator is defined by the .**truediv**() method. If you take a look at the source code of pathlib, you'll see something like:

```
class PurePath(object):

    def \_\_truediv\_\_(self, key):
        return self.\_make\_child((key,))
```

```
today = datetime.today()
sales_file = Path.cwd() / "data" / "raw" / "Sales-History.csv"
pipeline_file = Path.cwd() / "data" / "raw" / "pipeline_data.xlsx"
summary_file = Path.cwd() / "data" / "processed" / f"summary_{today:%b↵
   -%d-%Y}.pkl"

parts = ["in", "input.xlsx"]
in_file_3 = Path.cwd().joinpath(*parts)

print( Path.cwd())
print(today)
print(sales_file)
print(pipeline_file)
print(summary_file)
print(in_file_3)

print(type(in_file_3))
print(Path.home().joinpath('python', 'scripts', 'test.py'))
```

```
K:\WorkN\ComputationalRadiometry
2021-08-23 19:48:06.858288
K:\WorkN\ComputationalRadiometry\data\raw\Sales-History.csv
K:\WorkN\ComputationalRadiometry\data\raw\pipeline_data.xlsx
K:\WorkN\ComputationalRadiometry\data\processed\summary_Aug-23-2021.pkl
K:\WorkN\ComputationalRadiometry\in\input.xlsx
<class 'pathlib.WindowsPath'>
C:\Users\nwillers\python\scripts\test.py
```

```
cwd = os.getcwd()
p = Path(cwd)
print(p.is_dir())
print(p.is_file())
print(p.parts)
print(p.absolute)
print(p.anchor)
print(p.as_uri())
print(p.parent)
print(p.name)
```

```
True
False
('K:\\', 'WorkN', 'ComputationalRadiometry')
<bound method Path.absolute of WindowsPath('K:/WorkN/↵
   ComputationalRadiometry')>
K:\
file:///K:/WorkN/ComputationalRadiometry
K:\WorkN
ComputationalRadiometry
```

Note that `.parent` returns a new Path object, whereas the other properties return strings. This means for instance that `.parent` can be chained as in the last example or even combined with / to create completely new paths. `.parent.parent` work back one more parent level.

```
p = Path(cwd) / 'README.md'
print(p)
print(p.name)
print(p.parent)
print(p.stem)
print(p.suffix)
print(p.anchor)
print(p.parent.parent / ('mynewname' + p.suffix))
```

```
K:\WorkN\ComputationalRadiometry\README.md
README.md
K:\WorkN\ComputationalRadiometry
README
.md
K:\
K:\WorkN\mynewname.md
```

Walking directories is not quite possible in pathlib, but glob may help a little.

If you want to recursively walk through all directories, use the `**/` glob syntax. `rglob` returns a list of all the filenames as Path objects. The `[!*]` syntax can be used to exclude portions of a file.

The `p.rglob('./data/**/*.[!png;!txt!ltn]*')` call returns Path objects in all sub directories below data, but excludes the files with the extensions shown.

glob and rglob are limited in the sense that these functions cannot support multiple file types.

Note that `rglob` and `glob` returns a generator that can be made into a list (if necessary) by

`list(p.glob('./*.md'))`

```
p.rglob('./data/**/*.[!png;!txt!ltn]*')
print(list)
```

```
<class 'list'>
```

```
p = Path(cwd)
print(type(p.glob('**/cmd-here/*.*')))

for i in p.glob('**/cmd-here/*.*'):
    print(i.name)
print(' ')
print(list(p.rglob('./cmd-here/*.*')),end='\n\n')
print(list(p.rglob('./data/**/*.[!png;!txt!ltn]*')),end='\n\n')
```

```
<class 'generator'>
cmd-window-here.reg

[WindowsPath('K:/WorkN/ComputationalRadiometry/cmd-here/cmd-window-here↵
   .reg')]

[WindowsPath('K:/WorkN/ComputationalRadiometry/data/test.csv'), ↵
   WindowsPath('K:/WorkN/ComputationalRadiometry/data/test.db3'), ↵
   WindowsPath('K:/WorkN/ComputationalRadiometry/data/well-fill/radio-↵
   MWIR Sensor-SensorA-gen.hdf5'), WindowsPath('K:/WorkN/↵
   ComputationalRadiometry/data/well-fill/MLS23km/modin.ontplt'), ↵
   WindowsPath('K:/WorkN/ComputationalRadiometry/data/well-fill/SW31km/↵
   modin.ontplt'), WindowsPath('K:/WorkN/ComputationalRadiometry/data/↵
   well-fill/SW31km/SW31kmMaritimePrad.ontplt')]
```

The following code collects all filenames in a dir and sub dirs and write the details to a data frame

```
import time
from pathlib import PureWindowsPath, PurePosixPath
p = Path(cwd)
pstr = './data/**/*'
all_files = []
for i in p.rglob(pstr):
    all_files.append((i.name, i.parent, time.ctime(i.stat().st_ctime)))



# # Windows -> Posix
# win = r'foo\bar\file.txt'
# posix = str(PurePosixPath(PureWindowsPath(win)))
# print(posix)  # foo/bar/file.txt

columns = ["File_Name", "Parent", "Created"]
df = pd.DataFrame.from_records(all_files, columns=columns)
# convert to posix and get rid of the drive
df['Parent'] = df.apply(lambda x:str(PurePosixPath(PureWindowsPath(x['↵
   Parent'])))[3:] , axis = 1)

display(HTML(df.head().to_html()))
display(HTML(df.tail().to_html()))
```

	File_Name	Parent	Created
0	.gitignore	/WorkN/ComputationalRadiometry/data	Thu Nov 12 17:43:39 2020
1	09d-narcsissus.xlsx	/WorkN/ComputationalRadiometry/data	Thu Nov 12 17:43:39 2020
2	aircraft-signatures.pptx	/WorkN/ComputationalRadiometry/data	Thu Nov 12 17:43:39 2020
3	emis-800K-MWIR.txt	/WorkN/ComputationalRadiometry/data	Thu Nov 12 17:43:39 2020
4	emis-MWIR-300K.png	/WorkN/ComputationalRadiometry/data	Thu Nov 12 17:43:39 2020

	File_Name	Parent	Create
57	modin	/WorkN/ComputationalRadiometry/data/well-fill/tropical5km	Thu Nov 12 17:
58	tape5	/WorkN/ComputationalRadiometry/data/well-fill/tropical5km	Thu Nov 12 17:
59	tape6	/WorkN/ComputationalRadiometry/data/well-fill/tropical5km	Thu Nov 12 17:
60	tape7	/WorkN/ComputationalRadiometry/data/well-fill/tropical5km	Thu Nov 12 17:
61	trop5kmUrbantauPrad.ltn	/WorkN/ComputationalRadiometry/data/well-fill/tropical5km	Thu Nov 12 17:

The next example defines a function, `tree()`, that will print a visual tree representing the file hierarchy, rooted at a given directory. Here, we want to list subdirectories as well, so we use the `.rglob()` method. Note that we need to know how far away from the root directory a file is located. To do this, we first use `.relative\_to()` to represent a path relative to the root directory. Then, we count the number of directories (using the `.parts` property) in the representation. When run, this function creates a visual tree

```
def tree(directory):
    print(f'+ {directory}')
    for path in sorted(directory.rglob('*')):
        depth = len(path.relative_to(directory).parts)
        spacer = '    ' * depth
        print(f'{spacer}+ {path.name}')

tree(Path.cwd() / 'data')
```

```
+ K:\WorkN\ComputationalRadiometry\data
    + .gitignore
    + 09d-narcsissus.xlsx
    + aircraft-signatures.pptx
    + emis-800K-MWIR.txt
    + emis-MWIR-300K.png
    + emis-MWIR-800K.png
    + FOV-optimisation-parameters.xlsx
    + InSb.txt
    + mask00.png
    + mask01.png
    + mirrorerror.npz
    + path100
        + modin
        + tape5
        + tape6
        + tape7
    + path1000
        + modin
        + tape5
        + tape6
        + tape7
    + path10000
        + modin
        + tape5
        + tape6
        + tape7
    + plume-015a.PNG
    + plume-015b.PNG
    + plume-032.PNG
    + plume-060.PNG
    + plume-116.PNG
    + plume-171.PNG
    + plume-282.PNG
    + plume-394a.PNG
    + plume-394b.PNG
    + test.csv
    + test.db3
    + unity3_5.txt
    + well-fill
        + MLS23km
            + MLS23kmUrbantauPrad.ltn
            + modin
```

```
            + modin.ontplt
            + tape5
            + tape6
            + tape7
        + radio-MWIR_Sensor-SensorA-gen.hdf5
        + sensor-definitions.xlsx
        + SW31km
            + modin
            + modin.ontplt
            + SW31kmMaritimePrad.ltn
            + SW31kmMaritimePrad.ontplt
            + tape5
            + tape6
            + tape7
        + tropical5km
            + modin
            + tape5
            + tape6
            + tape7
            + trop5kmUrbantauPrad.ltn
```

Opening files. For example, print the headers in the first markdown file in the current directory.

Note that `list(Path(cwd).glob('./*.md'))[0]` comprises the following: `glob` returns a generator to paths, which must be made to generate a list, of which the 0 element is used.

When opening the file, the name is already present in `p`.

```
p = list(Path(cwd).rglob('./*.md'))[0]
print(type(p))
with p.open(mode='r') as fid:
    headers = [line.strip() for line in fid if line.startswith('#')]
print('\n'.join(headers))
```

```
<class 'pathlib.WindowsPath'>
### Computational Optical Radiometry with pyradi
```

There are a few different ways to list many files. The simplest is the `.iterdir()` method, which iterates over all files in the given directory. The following example combines `.iterdir()` with the `collections.Counter` class to count how many files there are of each filetype in the current directory. This code does not recurse into lower level folders, nor is is very accurate with files or folders without extensions: `.git` and `.gitignore` seem to be counted as file/folder with no extension/suffix.

```
import collections
collections.Counter(p.suffix for p in Path.cwd().iterdir())
```

```
Counter({'': 23,
         '.aux': 2,
         '.bbl': 1,
         '.bib': 5,
         '.blg': 1,
         '.ipynb': 29,
         '.log': 5,
         '.pdf': 5,
         '.tex': 16,
         '.cls': 2,
         '.pcl': 1,
         '.gz': 1,
         '.npy': 1,
```

```
         '.npz': 3,
         '.txt': 20,
         '.bin': 1,
         '.tar': 5,
         '.tgz': 5,
         '.png': 3,
         '.py': 5,
         '.fl7': 2,
         '.xlsx': 1,
         '.include': 1,
         '.md': 1,
         '.ltn': 1})
```

```
collections.Counter(p.suffix for p in Path.cwd().iterdir() if 'git' in ↵
   p.name)
```

```
Counter({'': 2})
```

```
collections.Counter(p.suffix for p in Path.cwd().iterdir() if '↵
   ipynb_checkpoints' in p.name)
```

```
Counter({'': 1})
```

To find the file in a directory that was last modified, you can use the .stat() method to get information about the underlying files. For instance, .stat().st_mtime gives the time of last modification of a file:

```
from datetime import datetime
directory = Path.cwd()
print(directory)
times, file_path = max((f.stat().st_mtime, f) for f in directory.↵
   iterdir())
print(datetime.fromtimestamp(times), file_path)
```

```
K:\WorkN\ComputationalRadiometry
2021-08-23 19:48:34.136029 K:\WorkN\ComputationalRadiometry\02-↵
   PythonWhirlwindCheatSheet.ipynb
```

You can even get the contents of the file that was last modified with a similar expression. Here we test for all `.txt` files and print the first ten lines of the most recently changed `.txt` file.

```
directory = Path.cwd()
filetuple = max((f.stat().st_mtime, f) for f in directory.iterdir() if ↵
   '.txt' in f.name)

print(filetuple)
print(filetuple[1].read_text().split('\n')[:10])
```

```
(1629721117.9883828, WindowsPath('K:/WorkN/ComputationalRadiometry/↵
   arrfile.txt'))
['## this is a header', '##Line two of header', ' 3.87250e-01  6.65286e↵
   -02  3.32997e-01  3.74701e-01', '-1.41658e+00  8.42898e-01  2.90373e↵
   -01  6.12334e-01', '-1.29467e+00 -1.57968e-01 -5.87222e-01  4.64316e↵
   -02', '']
```

This example shows how to construct a unique numbered file name based on a template. First, specify a pattern for the file name, with room for a counter. Then, check the existence of the file path created by joining a directory and the file name (with a value for the counter). If it already exists, increase the counter and try again

```
def unique_path(directory, name_pattern):
    counter = 0
    while True:
        counter += 1
        path = directory / name_pattern.format(counter)
        if not path.exists():
            return path

path = unique_path(Path.cwd(), 'test{:03d}.txt')
print(path)
with path.open('w') as fout:
    fout.write('This is a test file\n')
    fout.write('Line 2\n')
```

```
K:\WorkN\ComputationalRadiometry\test001.txt
```

Moving and deleting files

Through pathlib, you also have access to basic file system level operations like moving, updating, and even deleting files. For the most part, these methods do not give a warning or wait for confirmation before information or files are lost. Be careful when using these methods.

To move a file, use `.replace()`. Note that if the destination already exists, `.replace()` will overwrite it. To avoid possibly overwriting the destination path, the simplest is to test whether the destination exists before replacing:

```
if not destination.exists():
    source.replace(destination)
```

Directories and files can be deleted using `.rmdir()` and `.unlink()` respectively.

When you are renaming files, useful methods might be `.with\_name()` and `.with\_suffix()`. They both return the original path but with the name or the suffix replaced, respectively. Then call the `.replace()` method

```
path = Path.cwd() / 'test001.txt'
print(path)
if path.exists():
    print('executing .replace()')
    path.replace(path.with_suffix('.py'))
else:
    print(f'{path} does not exist')
```

```
K:\WorkN\ComputationalRadiometry\test001.txt
executing .replace()
```

1.12 Python Classes

Not presently covered here. Please google.

1.13 Function Objects and Operator Overloading

From wikipedia[137]: In Python, functions are first-class objects, just like strings, numbers, lists etc. This feature eliminates the need to write a function object in many cases. Any object with a **call**() method can be called using function-call syntax. An example is this accumulator class (based on Paul Graham's study on programming language syntax and clarity):

```
class Accumulator(object):
    def __init__(self, n):
        self.n = n

    def __call__(self, x):
        self.n += x
        return self.n
```

```
a = Accumulator(4)
print(a(3))
print(a(2))
```

```
7
9
```

From http://www.programiz.com/python-programming/operator-overloading[138]: Class functions that begins with double underscore (__) are called special functions in Python. To overload the + sign, we will need to implement __add__() function in the class. What actually happens is that, when you do p1 + p2, Python will call p1.__add__(p2) which in turn is Point.__add__(p1,p2). The reference gives a complete list of operators that can be overloaded, including: Addition + __add__, Subtraction - __sub__, Multiplication * __mul__, Power ** __pow__, Division / __truediv__, Floor Division // __floordiv__, Remainder (modulo) % __mod__, Bitwise Left Shift << __lshift__, Bitwise Right Shift >> __rshift__, Bitwise AND & __and__, Bitwise OR | __or__, Bitwise XOR ^ __xor__, and Bitwise NOT ~ __invert__.

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well. Comparison operators that can be overloaded include: Less than < __lt__, Less than or equal to <= __le__, Equal to == __eq__, Not equal to != __ne__, Greater than > __gt__, and

Greater than or equal to >= __ge__

```
class Point:
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)

    def __lt__(self,other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag

    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x,y)
```

```
p1 = Point(2,3)
p2 = Point(-1,2)
print(p1 + p2)
print(Point(1,1) < Point(-2,-3))
```

```
(1,5)
True
```

From here https://newcircle.com/bookshelf/python_fundamentals_tutorial/functional_programming[139]: Functions are first-class objects in Python, meaning they have attributes and can be referenced and assigned to variables. Python also supports higher-order functions, meaning that functions can accept other functions as arguments and return functions to the caller.

```
def i_am_an_object(myarg):
    '''I am a really nice function.
       Please be my friend.'''
    return myarg

an_object_by_any_other_name = i_am_an_object

print(i_am_an_object(1))
print(an_object_by_any_other_name(2))
print(i_am_an_object)
print(an_object_by_any_other_name)
print(i_am_an_object.__doc__)
print(i_am_an_object(i_am_an_object))
```

```
1
2
<function i_am_an_object at 0x000002251D634550>
<function i_am_an_object at 0x000002251D634550>
I am a really nice function.
       Please be my friend.
<function i_am_an_object at 0x000002251D634550>
```

From here https://newcircle.com/bookshelf/python_fundamentals_tutorial/functional_programming[140]:

In order to define non-default sorting in Python, both the sorted() function and the list's .sort() method accept a key argument. The value passed to this argument needs to be a function object that returns the sorting key for any item in the list or iterable. For example, given a list of tuples, Python will sort by default on the first value in each tuple. In order to sort on a different element from each tuple, a function can be passed that returns that element.

```
def second_element(t):
    return t[1]

zepp = [('Guitar', 'Jimmy'), ('Vocals', 'Robert'), ('Bass', 'John Paul'),
    ('Drums', 'John')]
print(sorted(zepp))
print(sorted(zepp, key=second_element))
```

```
[('Bass', 'John Paul'), ('Drums', 'John'), ('Guitar', 'Jimmy'), ('
    Vocals', 'Robert')]
[('Guitar', 'Jimmy'), ('Drums', 'John'), ('Bass', 'John Paul'), ('
    Vocals', 'Robert')]
```

See also here:

https://www.ics.uci.edu/~pattis/ICS-33/lectures/operatoroverloading1.txt[141]

https://docs.python.org/2/c-api/function.html[142]

http://python-3-patterns-idioms-test.readthedocs.io/en/latest/FunctionObjects.html[143]

https://newcircle.com/bookshelf/python_fundamentals_tutorial/functional_programm

1.14 Building function parameters as a **kwargs dictionary

Requirement:

- send function `fnpassed` as parameter to function `fncall`
- send the parameters to the passed to `fnpassed` also as parameters to `fncall`
- make it possible to add new `fnpassed`-parameters inside `fncall` before `fnpassed` is called.

We want this to have a calling function `fncall` that can call some other function `fnpassed` with a user-supplied list of parameters, but also adding some more parameters in `fncall`. In other words, the execution of `fnpassed` must be with parameters passed to `fncall` from outside, plus new parameters added inside `fncall`.

In the code below the function `testfn` must be executed with four parameters, three of which (1,2,3) are passed as `tryy` parameters , with the fourth (10) added within `fncall`, because the fourth is only available from within `fncall` (not beforehand).

The parameters passed to a function can be built up a dictionary and `.` is not an operator, so it doesn't really have a name, but it is defined as a "syntactic rule". So it should be called: "the keyword argument unpacking syntax". It unpacks a dictionary into a function parameter list.

So we can build up the dictionary in different places with new information and when complete, present this to the function using the **** unpacking.

https://pythontips.com/2013/08/04/args-and-kwargs-in-python-explained/

http://softwareengineering.stackexchange.com/questions/131403/what-is-the-name-of-in-python/131415

http://stackoverflow.com/questions/36901/what-does-double-star-and-star-do-for-parameters

http://stackoverflow.com/questions/1769403/understanding-kwargs-in-python

```
import numpy as np

def fnpassed(a,b,c,d):
    # also return the input parameters as a tuple
    return (a * b + c) / d, (a,b,c,d)


def fncall(fn,kwargs):
    # add the fourth parameter
    kwargs['d'] = 10.
    print('Building up kwargs: {}\n'.format(kwargs))
    aa = fn(**kwargs)
    return aa
```

```
rtnval,inpval = fncall(fnpassed,kwargs={'a':1,'b':2,'c':3})
print('Return value = {}, using input values {}'.format(rtnval,inpval))
```

```
Building up kwargs: {'a': 1, 'b': 2, 'c': 3, 'd': 10.0}

Return value = 0.5, using input values (1, 2, 3, 10.0)
```

1.15 Inspect

The inspect module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

You need to import the module file or create an instance of a class to be able to use inspect.

The getmembers() function retrieves the members of an object such as a class or module. The sixteen functions whose names begin with `is` are mainly provided as convenient choices for the second argument to getmembers().

https://docs.python.org/2/library/inspect.html

https://stackoverflow.com/questions/12994366/using-inspect-getmembers

https://stackoverflow.com/questions/33204160/listing-all-class-members-with-python-inspect-module

```
##
# get a list of all functions in a module file

import inspect

#import the module you want to inspect
import pyradi.rytarggen as rytarggen

# get all function members of the module
all_functions = inspect.getmembers(rytarggen, inspect.isfunction)

# build a string with all the functions in quotes with commas ↵
   separating
pstr = ""
for item in all_functions:
    pstr += "'{}', ".format(item[0])

print(pstr)
```

```
'analyse_HDF5_image', 'analyse_HDF5_imageFile', 'assigncheck', '↵
   calcLuxEquivalent', 'calcTemperatureEquivalent', 'create_HDF5_image↵
   ', 'hdf_Raw', 'hdf_Uniform', 'hdf_disk_photon', 'hdf_stairs',
```

1.16 Python: "is None" vs "==None"

PEP 8 says: "Comparisons to singletons like None should always be done with 'is' or 'is not', never the equality operators."

http://jaredgrubb.blogspot.co.za/2009/04/python-is-none-vs-none.html

https://stackoverflow.com/questions/2209755/python-operation-vs-is-not

https://stackoverflow.com/questions/23086383/how-to-test-nonetype-in-python

https://stackoverflow.com/questions/14247373/python-none-comparison-should-i-use-is-or

https://stackoverflow.com/questions/2020598/in-python-how-should-i-test-if-a-variable-is-none-true-or-false

https://www.appneta.com/blog/python-none-comparison/

https://graysonkoonce.com/always-use-none-for-default-args-in-python/

`is` is identity testing. It checks whether the right hand side and the left hand side are the very same object. No methodcalls are done, objects can't influence the is operation. (e.g. checking to see if var is None). None is a special singleton object, there can only be one. Just check to see if you have that object. The operators is and is not test for object identity: x is y is true if and only if x and y are the same object. x is not y yields the inverse truth value.

== is equality testing. (e.g. Is var equal to 3?). It checks whether the right hand side and the left hand side are equal objects (according to their **eq** or **cmp** methods.)

You use is (and is not) for singletons, like None, where you don't care about objects that might want to pretend to be None or where you want to protect against objects breaking when being compared against None.

```
lst = [1,2,3]
lst == lst[:]  # This is True since the lists are "equivalent"
lst is lst[:]  # This is False since they're actually different objects
```

PEP-0008: "beware of writing if x when you really mean if x is not None - e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type (such as a container) that could be false in a boolean context!"

1.17 What is difference between [None] and [] in python?

https://stackoverflow.com/questions/36928602/what-is-difference-between-none-and-in-python

[] is an empty list

[None] is a list with one element. that one element is None

```
## the following returns a single list with n elements, all None
n = 5
data = n * [None]
print(data)
data2 = [ [None] for i in [1,2,3]]
print(data2)
```

```
[None, None, None, None, None]
[[None], [None], [None]]
```

1.18 Upright micro symbol

In Python 3 all strings are unicode. The upright mu symbol (as required by the SI system to denote 0.000001) is ’ŏ0B5’

```
##
if sys.version_info[0] > 2:
    print('\u00B5m')
```

```
m
```

```
import matplotlib as mpl

mpl.rcParams['text.latex.preamble'] = [
       r'\usepackage{siunitx}',   # i need upright \micro symbols, but you need...
       r'\sisetup{detect-all}',   # ...this to force siunitx to actually use your f
       r'\usepackage{helvet}',    # set the normal font here
       r'\usepackage{sansmath}',  # load up the sansmath so that math -> helvet
       r'\sansmath'               # <- tricky! -- gotta actually tell tex to use!
]
mpl.rcParams['text.usetex'] = True

p = ryplot.Plotter(1,1,2,figsize=(12,6))
p.plot(1,spec[:,0],spec[:,1],'Normalised responsivity','Wavelength \si{\micro\metre
```

1.19 Esoteric Python tidbits

http://nbviewer.ipython.org/github/rasbt/python_reference/blob/master/not_so_obvious_python_stuff.ipynb?create=1[145]

http://sahandsaba.com/thirty-python-language-features-and-tricks-you-may-not-know.html[146]

Python variables can have data types[147], see the link for more details.

```
# a = int
# print(type(a))
# print a('3') + 4

# b = str
# print b(12) * 5

# class Foo:
#       def bar(self):
#           print "in Foo.bar()"

# f = Foo
# f().bar()
```

```
# Collect input and output arguments into one bunch
class Bunch(object):
    def __init__(self, **kwds): self.__dict__.update(kwds)

arg = Bunch(recurse=1, pattern_list='*.xml;*.py', return_folders=0, ↵
   results=[])

print(arg.__dict__)
```

```
{'recurse': 1, 'pattern_list': '*.xml;*.py', 'return_folders': 0, '↵
   results': []}
```

1.20 Numpy Basics

Before numpy can be used it must be imported into the workspace. Note that it is already pre-loaded in IPython, but not in other scipts. By general convention, it is normally imported as `np`.

Working with NumPy arrays[148].

Numpy performance tricks[149]

```
import numpy as np
```

1.20.1 Creating Numpy arrays

There are many more ways to create arrays we are only using one method. See `http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html`[150]

Numpy essentially provides arrays and matrices. A matrix is the numpy equivalent of a linear algebra matrix, while the array is just a structured set of data. Matrices and arrays both have N-dimensional structure, but the operations on these are different. Numpy arrays perform operations on element-wise level (unless otherwise told), which means that the multiplication of two numpy arrays yield a new array of the same size where each element is the multiplication of the elements in the same location of the two input arrays.

Numpy uses the linear algebra term, *rank* to denote the array number of dimensions. Rank-two arrays can only appear in one guise, that of an MxN array. A vector (one-dimensional array) can appear in three different guises, each of which is slightly different: (N,) (N,1) or (1,N) - and these different forms don't interoperate very well (as they seem to do in Matlab).

For example the transpose of (N,) is still (N,) because (N,) is a rank-one array, for which the transpose is not defined. So when using vectors (and we do use them extensively) be careful of this this difference.

Numpy arrays have a `shape` attribute that returns the size of the array. For rank-two arrays, the first value can be interpreted as the number of rows, then columns.

```
a = np.asarray([1, 2, 3]) # construct from a list
print(a.shape)
print(np.ones(a.shape).shape)
```

```
(3,)
(3,)
```

```
b = np.array(a) #construct from another array
print(b.shape)
print(np.zeros(a.shape).shape)
```

```
(3,)
(3,)
```

```
c = np.linspace(0,10,11) # 11 elements, ranging from 0 to 10
print(c.shape)
```

```
(11,)
```

```
d = np.eye(3)
print(d.shape)
print(d)
```

```
(3, 3)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

```
vis = np.asarray(4)
print(vis)
print(type(vis))
print(vis.shape)
visr = vis.reshape(-1,)
print(visr)
print(type(visr))
print(visr.shape)
```

```
4
<class 'numpy.ndarray'>
()
[4]
<class 'numpy.ndarray'>
(1,)
```

```
vis = np.asarray([4])
print(vis)
print(type(vis))
print(vis.shape)
```

```
[4]
<class 'numpy.ndarray'>
(1,)
```

```
vis = np.asarray([[4],[6]])
print(vis)
print(type(vis))
print(vis.shape)
```

```
[[4]
 [6]]
<class 'numpy.ndarray'>
(2, 1)
```

Arrays can be reshaped to different shapes, even ranks, provided the number of elements don't change. This code creates a rank-three array filled with zeros. The first parameter of this function must be a tuple of the required size, in this case 3x3x4. The array is initially flattened, made into a rank-one array. Using the `reshape` function it is then reshaped to different sizes (this is only of illustrative use, no real mathematical or physical meaning). If one of the reshape parameters is -1, it is interpreted that this axis must be made whatever value fits the reshaping.

```
a = np.zeros((3,3,4))
print(a.shape)
b = a.flatten()
print(b.shape)
b = b.reshape(18,-1)
print(b.shape)
b = b.reshape(3,-1,3)
print(b.shape)
b = b.reshape(-1,1)
print(b.shape)
```

```
(3, 3, 4)
(36,)
(18, 2)
(3, 4, 3)
(36, 1)
```

1.20.2 Sequence ascending or descending?

```
a= np.linspace(1,10,10)
print(np.all(a[1:] >= a[:-1]) )
a[5] = 2.
print(np.all(a[1:] >= a[:-1]) )
```

```
True
False
```

1.20.3 Modulus operator on floats

The modulus operator (returning the fraction of a division) also works for floats, returning the fractional remainder.

```
a = np.linspace(5.5, 9.5, 9)
b = a % 2
print(a)
print(b)
print(b==0)
print(a[b==0])
```

```
[5.5 6.  6.5 7.  7.5 8.  8.5 9.  9.5]
[1.5 0.  0.5 1.  1.5 0.  0.5 1.  1.5]
[False  True False False False  True False False False]
[6. 8.]
```

1.20.4 Operations on Numpy arrays

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape, as in the following example. Normally, broadcasting is transparent and operates as one would intuitively think.

Broadcasting is where numpy can figure out how to handle arrays of different shape, it the operation is well defined, e.g., adding a constant to an array - the constant is added to all elements. Broadcasting is very important and a rudimentary understanding is necessary.

See `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`[151]

Note that in this case the transpose had no effect.

```
e = (d + 2).T
print(e.shape)
print(e)
```

```
(3, 3)
[[3. 2. 2.]
 [2. 3. 2.]
 [2. 2. 3.]]
```

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
print('a * b  ={}'.format(a * b))
print('a - b  ={}'.format(a - b))
a = np.array([1.0, 2.0])
b = np.array([2.0, 2.0, 2.0])
try:
    print('a * b  ={}'.format(a * b))
except:
    print('{} * {} is not allowed, broadcasting not possible'.format(a,↵
       b))
```

```
a * b  =[2. 4. 6.]
a - b  =[-1.  0.  1.]
[1. 2.] * [2. 2. 2.] is not allowed, broadcasting not possible
```

The following code multiplies two rank-2 arrays, note the different results from different multiplications:

```
import numpy as np
a = np.asarray([0,1,2,3]).reshape(-1,1)
b = a.reshape(1,-1)
print('a={} a.shape={}\n'.format(a, a.shape))
print('b={} b.shape={}\n'.format(b, b.shape))


print('(N,1) x (N,1)  - element by element product')
aa = a * a
print('a*a={} aa.shape={}\n'.format(aa, aa.shape))
print('(1,N) x (1,N)  - element by element product')
bb = b * b
print('b*b={} bb.shape={}\n'.format(bb, bb.shape))


print('(N,1) x (1,N) - this is an outer product')
ab = a * b
print('a * b={} ab.shape={}\n'.format(ab, ab.shape))
```

```
print('(1,N) x (N,1) - this is an outer product')
ba = b * a
print('b * a={} ba.shape={}\n'.format(ba, ba.shape))


print('(N,1) . (N,1)  - dot product')
adb = np.dot(a,b)
print('a.b={} (a.b).shape={}\n'.format(adb, adb.shape))

print('(1,N) . (N,1)  - dot product')
bda = np.dot(b,a)
print('b.a={} (b.a).shape={}\n'.format(bda, bda.shape))
```

```
a=[[0]
 [1]
 [2]
 [3]] a.shape=(4, 1)

b=[[0 1 2 3]] b.shape=(1, 4)

(N,1) x (N,1)  - element by element product
a*a=[[0]
 [1]
 [4]
 [9]] aa.shape=(4, 1)

(1,N) x (1,N)  - element by element product
b*b=[[0 1 4 9]] bb.shape=(1, 4)

(N,1) x (1,N) - this is an outer product
a * b=[[0 0 0 0]
 [0 1 2 3]
 [0 2 4 6]
 [0 3 6 9]] ab.shape=(4, 4)

(1,N) x (N,1) - this is an outer product
b * a=[[0 0 0 0]
 [0 1 2 3]
 [0 2 4 6]
 [0 3 6 9]] ba.shape=(4, 4)

(N,1) . (N,1)  - dot product
a.b=[[0 0 0 0]
 [0 1 2 3]
 [0 2 4 6]
 [0 3 6 9]] (a.b).shape=(4, 4)

(1,N) . (N,1)  - dot product
b.a=[[14]] (b.a).shape=(1, 1)
```

In the following example a is a rank-one array and b is a rank-two array. The b array can be transposed, but transpose operator is not defined for the rank-one array.

```
a = np.asarray([1, 2, 3]) # construct from a list
print('array a has {} elements, shape {}'.format(a.shape[0],a.shape))
print('array a.T has {} elements, shape {}'.format(a.T.shape[0],a.T.↵
    shape))
print('a is equal to a.T: {}'.format(np.array_equal(a,a.T)))
print(' ')
```

```
b = a.reshape(-1,1)
print('array b has {} rows and {} columns, shape {}'.format(b.shape[0],↵
    b.shape[1], b.shape))
print('array b.T has {} rows and {} columns, shape {}'.format(b.T.shape↵
   [0], b.T.shape[1], b.T.shape))
print('b is equal to b.T: {}'.format(np.array_equal(b,b.T)))
```

```
array a has 3 elements, shape (3,)
array a.T has 3 elements, shape (3,)
a is equal to a.T: True

array b has 3 rows and 1 columns, shape (3, 1)
array b.T has 1 rows and 3 columns, shape (1, 3)
b is equal to b.T: False
```

1.20.5 Slicing and Stacking

Numpy arrays can be sliced to extract portions of the array. Start by creating a rank-two array with 15 elements, then reshape to a 3 row/5 column array. Then slice different parts from the array. A slice is specified in the form: start:end:stride, where the start value is *included* but the end value is *excluded* and the stride definest stride from te starting position (inclusive).

http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html[152]

Observe that the slice `[:,1]` returns a (N,) rank one vector whereas the slice `[:,:1]` returns a (N,1) rank two vector

```
a = np.arange(0,15).reshape(3,5)
print(a)
print('Original array: {} has shape {}'.format(a, a.shape))
print('All the rows of second column: {} has shape {}'.format(a[:,1],a↵
   [:,1].shape))
print('All the colums of second row: {} has shape {}'.format(a[1,:],a↵
   [1,:].shape))
print('Portion of the array: {} has shape {}'.format(a[0:2,1:3],a↵
   [0:2,1:3].shape))
print('Strided portion of the array: {} has shape {}'.format(a↵
   [0:2,0:5:2],a[0:2,0:5:2].shape))
print('---------------')
print('Sliced by {} all the rows of second column: {} has shape {}'.↵
   format('[:,1]',a[:,1],a[:,1].shape))
print('Sliced by {} all the rows of second column: {} has shape {}'.↵
   format('[:,:1]',a[:,1],a[:,:1].shape))
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
Original array: [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]] has shape (3, 5)
All the rows of second column: [ 1  6 11] has shape (3,)
All the colums of second row: [5 6 7 8 9] has shape (5,)
Portion of the array: [[1 2]
 [6 7]] has shape (2, 2)
Strided portion of the array: [[0 2 4]
 [5 7 9]] has shape (2, 3)
---------------
```

```
Sliced by [:,1] all the rows of second column: [ 1  6 11] has shape ↵
   (3,)
Sliced by [:,:1] all the rows of second column: [ 1  6 11] has shape ↵
   (3, 1)
```

Numpy makes provision to build new arrays by stacking arrays together. This stacking can be done horizontally or vertically. In both cases does the function require a tuple as input where the tuple contains the the two or more parts to be stacked. In order to stack the size of the two or more sub-arrays along the stacking direction must be the same. The results below show that for the purpose of stacking a (N,) array is considered to be the same as a (1,N) array - that is one row with many columns.

```
import numpy as np
a = np.asarray([1, 2, 3, 4])
b = np.asarray([5, 6, 7, 8])
c = np.asarray([9, 10, 11, 12])
d = np.hstack((a,b,c))
e = np.vstack((a,b,c))
print('a: value={} shape={}'.format(a, a.shape))
print('b: value={} shape={}'.format(b, b.shape))
print('c: value={} shape={}'.format(c, c.shape))
print('d: value={} shape={}'.format(d, d.shape))
print('e: value={} shape={}'.format(e, e.shape))
print('---------')

a = np.asarray([1, 2, 3, 4]).reshape(1,-1)
b = np.asarray([5, 6, 7, 8]).reshape(1,-1)
c = np.asarray([9, 10, 11, 12]).reshape(1,-1)
d = np.hstack((a,b,c))
e = np.vstack((a,b,c))
print('a: value={} shape={}'.format(a, a.shape))
print('b: value={} shape={}'.format(b, b.shape))
print('c: value={} shape={}'.format(c, c.shape))
print('d: value={} shape={}'.format(d, d.shape))
print('e: value={} shape={}'.format(e, e.shape))
print('---------')

a = np.asarray([1, 2, 3, 4]).reshape(-1,1)
b = np.asarray([5, 6, 7, 8]).reshape(-1,1)
c = np.asarray([9, 10, 11, 12]).reshape(-1,1)
d = np.hstack((a,b,c))
e = np.vstack((a,b,c))
print('a: value={} shape={}'.format(a, a.shape))
print('b: value={} shape={}'.format(b, b.shape))
print('c: value={} shape={}'.format(c, c.shape))
print('d: value={} shape={}'.format(d, d.shape))
print('e: value={} shape={}'.format(e, e.shape))
```

```
a: value=[1 2 3 4] shape=(4,)
b: value=[5 6 7 8] shape=(4,)
c: value=[ 9 10 11 12] shape=(4,)
d: value=[ 1  2  3  4  5  6  7  8  9 10 11 12] shape=(12,)
e: value=[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]] shape=(3, 4)
---------
a: value=[[1 2 3 4]] shape=(1, 4)
b: value=[[5 6 7 8]] shape=(1, 4)
c: value=[[ 9 10 11 12]] shape=(1, 4)
```

```
d: value=[[ 1  2  3  4  5  6  7  8  9 10 11 12]] shape=(1, 12)
e: value=[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]] shape=(3, 4)
---------
a: value=[[1]
 [2]
 [3]
 [4]] shape=(4, 1)
b: value=[[5]
 [6]
 [7]
 [8]] shape=(4, 1)
c: value=[[ 9]
 [10]
 [11]
 [12]] shape=(4, 1)
d: value=[[ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]
 [ 4  8 12]] shape=(4, 3)
e: value=[[ 1]
 [ 2]
 [ 3]
 [ 4]
 [ 5]
 [ 6]
 [ 7]
 [ 8]
 [ 9]
 [10]
 [11]
 [12]] shape=(12, 1)
```

1.20.6 Slicing with Ellipsis

Ellipsis is a Python object that can appear in slice notation. Ellipsis is used in numpy to indicate a placeholder for the rest of the array dimensions not specified. Think of it as indicating the full slice `[:]` for all the dimensions in the gap it is placed, so for a 3d array, `a[...,0]` is the same as `a[:,:,0]` and for 4d, `a[:,:,:,0]`. Similarly `a[0,...,0]` is `a[0,:,:,0]` (with however many colons in the middle make up the full number of dimensions in the array).

http://stackoverflow.com/questions/772124/what-does-the-python-ellipsis-object-do[

```
import numpy as np

a = np.zeros((2,3,4,5))
print('a.shape = {}'.format(a.shape))
print('a[...].shape = {}'.format(a[...].shape))
print('a[1,...].shape = {}'.format(a[1,...].shape))
print('a[1,...,1].shape = {}'.format(a[1,...,1].shape))
print('a[...,1].shape = {}'.format(a[...,1].shape))
```

```
a.shape = (2, 3, 4, 5)
a[...].shape = (2, 3, 4, 5)
a[1,...].shape = (3, 4, 5)
a[1,...,1].shape = (3, 4)
```

```
a[...,1].shape = (2, 3, 4)
```

In NumPy, there are several flavors of array containing one element. 1. A rank-0 array with shape (). 1. A rank-1 array with shape (1,). 1. A rank-2 array with shape (1,1).

Note that using an empty tuple `()` as index/slice gives you all the elements in the dataset, as an array object for rank-1 and higher arrays, and as a scalar element for rank-0 arrays.

The form `[...]` or `[()]` is in fact the only way you can get the value of a rank-0 array, because [0] is not allowed for a rank-0 array.

Expanded from the original writeup in Python and HDF5[154] by Andrew Collette.

```
a = 3 * np.array(1) #rank 0
print('a.shape of a rank-0 array with single element: {}'.format(a.↵
    shape))
print('a[0] is not allowed for rank-0 with shape ()')
print('a returns the full array with all dimensions included (a scalar)↵
    : {}'.format(a))
print('a[...] returns the full array with all dimensions included (a ↵
    scalar): {}'.format( a[...]))
print('a[()] returns the full array with all dimensions included (a ↵
    scalar): {}\n'.format( a[()]))

a = 5 * np.array([1])#rank 1
print('a.shape of a rank-1 array with single element: {}'.format(a.↵
    shape))
print('a[0] returns the value of the 0th element: {}'.format( a[0]))
print('a[...] returns the full array with all dimensions included: {}'.↵
    format( a[...]))
print('a[()] returns the full array with all dimensions included: {}\n'↵
    .format( a[()]))

a = 7 * np.ones((1,1))#rank 2
print('a.shape of a rank-2 array with single element: {}'.format(a.↵
    shape))
print('a[0] returns the value of the 0th row: {}'.format( a[0]))
print('a[0,0] returns the value of the 0th col of the 0th row: {}'.↵
    format( a[0,0]))
print('a[...] returns the full array with all dimensions included: {}'.↵
    format( a[...]))
print('a[()] returns the full array with all dimensions included: {}'.↵
    format( a[()]))
```

```
a.shape of a rank-0 array with single element: ()
a[0] is not allowed for rank-0 with shape ()
a returns the full array with all dimensions included (a scalar): 3
a[...] returns the full array with all dimensions included (a scalar): ↵
    3
a[()] returns the full array with all dimensions included (a scalar): 3

a.shape of a rank-1 array with single element: (1,)
a[0] returns the value of the 0th element: 5
a[...] returns the full array with all dimensions included: [5]
a[()] returns the full array with all dimensions included: [5]

a.shape of a rank-2 array with single element: (1, 1)
a[0] returns the value of the 0th row: [7.]
a[0,0] returns the value of the 0th col of the 0th row: 7.0
a[...] returns the full array with all dimensions included: [[7.]]
```

```
a[()] returns the full array with all dimensions included: [[7.]]
```

In Python 3, you can use the Ellipsis literal ... as a 'nop' placeholder for code:

```
def will\_do\_something():
    ...
```

Finally a tongue-in-the-cheek application of ellipsis is found here[155]

```
# Portable way to get the You-Know-Which object without naming it
class __:
 def __getitem__(__, _):
  return _
___ = __()[...]

# An Ellipsobinary-to-ASCII convertor
class __:
 def __getitem__(__, _):
  return chr(sum(1<<i if _[-i-1] is ___ else 0 for i in range(len(_))))
_ = __()

# Finally, use the That-Which-Must-Not-Be-Named object
print (
 _[...,_,_,...,_,_,_] +
  _[...,...,_,_,...,_,...] +
   _[...,...,_,...,...,_,_] +
    _[...,...,_,...,...,_,_] +
     _[...,...,_,...,...,...,...] +
      _[...,_,...,...,_,_] +
       _[...,_,_,_,_,_] +
        _[...,...,...,_,...,...,...] +
         _[...,...,_,...,...,...,...] +
          _[...,...,...,_,_,...,_] +
           _[...,...,_,...,...,_,_] +
            _[...,...,_,_,...,_,_] +
             _[...,_,_,_,_,...])
```

```
Hello, world!
```

1.20.7 Changing the shape of an array

https://docs.scipy.org/doc/numpy-dev/user/quickstart.html#changing-the-shape-of-an

The order of the elements in the array resulting from ravel() is normally 'C-style', that is, the rightmost index 'changes the fastest', so the element after a[0,0] is a[0,1]. If the array is reshaped to some other shape, again the array is treated as 'C-style'. Numpy normally creates arrays stored in this order, so ravel() will usually not need to copy its argument, but if the array was made by taking slices of another array or created with unusual options, it may need to be copied. The functions ravel() and reshape() can also be instructed, using an optional argument, to use FORTRAN-style arrays, in which the leftmost index changes the fastest.

```
import numpy as np
a = np.floor(10*np.random.random((3,4)))
print(a)
print(a.shape)
```

```
b = a.ravel() # flatten the array
print(b)
a.shape = (6, 2)
print(a.T)
a = a.reshape(4,3)
print(a)
```

```
[[0. 0. 7. 9.]
 [5. 2. 5. 5.]
 [0. 3. 7. 8.]]
(3, 4)
[0. 0. 7. 9. 5. 2. 5. 5. 0. 3. 7. 8.]
[[0. 7. 5. 5. 0. 7.]
 [0. 9. 2. 5. 3. 8.]]
[[0. 0. 7.]
 [9. 5. 2.]
 [5. 5. 0.]
 [3. 7. 8.]]
```

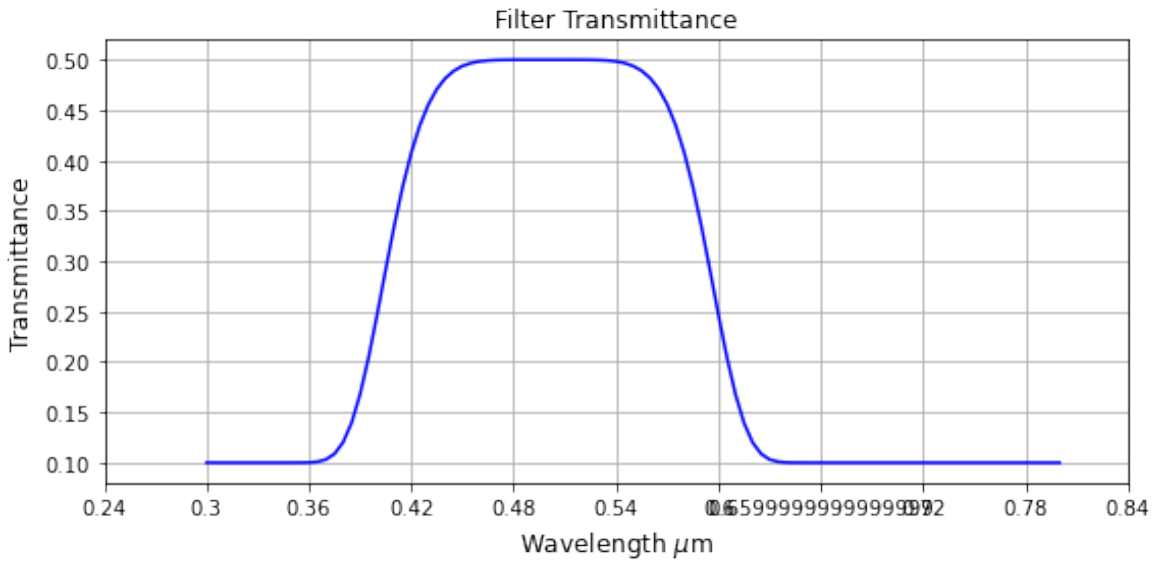
1.20.8 Numpy array application for spectral calculations

Numpy is much more than just array-like containers, there is also a comprehensive number of mathematical and logical operators to process the array contents. Numpy arrays are used as a basis for other packages such as the image processing library scikit-image and the data analysis package pandas. Numpy also forms the basis of the pyradi package, but serves mainly as a container for spectral values.

In pyradi spectral variables are respresented as arrays (of all ranks). For example, a set of wavelength values can be represented as an array. Spectral variables such as optics transmittance have values at specific wavelengths. So the transmittance vector and wavelength vector correspond on an element-by-element basis: the value of transmittance at the specific transmittance. The example below defines a wavelength band from 300 nm to 800 nm in 101 samples, and then calculates a filter transmittance using pyradi. The filter response is then plotted. IPython presents the graph in the document itself because the \%matplotlib inline magic is used.

```
%matplotlib inline
import numpy as np
import pyradi.ryutils as ryutils
import pyradi.ryplot as ryplot

wl = np.linspace(0.3, 0.8, 101)
sfilter = ryutils.sfilter(wl, 0.5, 0.2, 6, 0.5, 0.1)
p = ryplot.Plotter(1,figsize=(9,4))
p.plot(1,wl,sfilter,'Filter Transmittance', 'Wavelength $\mu$m', '↵
   Transmittance')
p.saveFig('filter.png')
```

```
****  If saveFig does not work inside the notebook please comment out ↵
   the line "%matplotlib inline"
To disable ryplot warnings, set doWarning=False
```



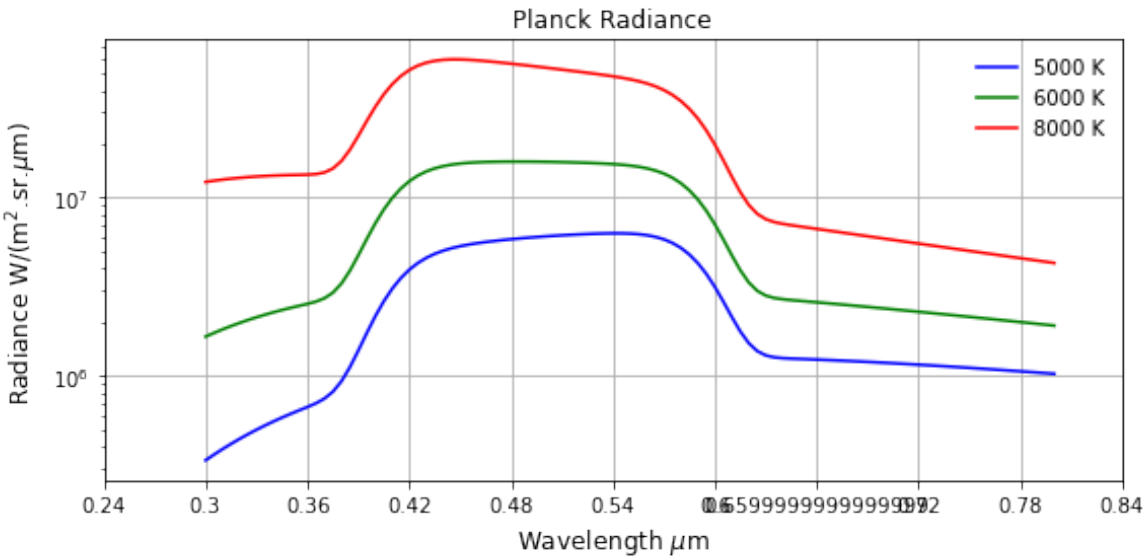
The concept of a single spectral variable, such as the filter above, can be extended to many spectral variables, all sharing the same wavelength values. One such example is the Planck law radiation at different temperatures. In the next example, the radiation is calculated at a range of temperatures and plotted on the graph. Note in this case that the value returned from the `ryplanck.planck()` function is a rank-two array. Along the first dimension is the spectral axis and along the second dimension is the radiance for the different temperatures.

The plotting functions can plot more than one line if a rank-two array is passed. However many columns there are in the array, so many lines will be drawn. The only requirement is that the size of the radiance (y) array along the zero axis much match the size of the wavelength (x) array along its zero axis.

Observe how the line labels are created using list comprehension, properly formatted with units.

```
import pyradi.ryplanck as ryplanck

temperatures = [5000, 6000, 8000]
radiance = ryplanck.planck(wl,temperatures, 'el') / np.pi

print('wl.shape = {}'.format(wl.shape))
print('len(temperatures) = {}'.format(len(temperatures)))
print('radiance.shape = {}'.format(radiance.shape))

labels = ['{} K'.format(tmp) for tmp in temperatures]
print('labels = {}'.format(labels))

p = ryplot.Plotter(1,figsize=(9,4))
p.semilogY(1,wl,radiance * sfilter.reshape(-1,1),'Planck Radiance', '↵
   Wavelength $\mu$m', 'Radiance W/(m$^2$.sr.$\mu$m)',label=labels)
p.saveFig('filtered-planck.png')
```

```
wl.shape = (101,)
len(temperatures) = 3
radiance.shape = (101, 3)
labels = ['5000 K', '6000 K', '8000 K']
****  If saveFig does not work inside the notebook please comment out ↵
   the line "%matplotlib inline"
To disable ryplot warnings, set doWarning=False
```




```
for i,temperature in enumerate(temperatures):
    integral = np.trapz((radiance[:,i]).reshape(-1,1) * sfilter.reshape↵
       (-1,1), wl, axis=0)
    print('Temperature = {} K,  Integrated radiance = {:.3e} W/(m2.sr)'↵
       .format(temperature, integral[0]))
```

```
Temperature = 5000 K,  Integrated radiance = 1.394e+06 W/(m2.sr)
Temperature = 6000 K,  Integrated radiance = 3.618e+06 W/(m2.sr)
Temperature = 8000 K,  Integrated radiance = 1.259e+07 W/(m2.sr)
```

1.20.9 File input/output

Numpy can import data from file and write to a file. The simplest import is `numpy.loadtxt()` and `numpy.savetxt()`. In the following section the spectral data and radiance data from the previous calculation are stacked together, such that the first column is the spectral vector, followed by the radiance in columns 1 and following. For this purpose the horisontal stack function is used: stacking arrays side by side. But there is a problem in that the wavelength variable is rank one (N,) , whereas a rank-two array (N,1) is required - it is reshaped before stacking. The stacked array is then saved to an ASCII file. Note that the format string requires that the numbers are written to five decimal places. The data can also be saved in binary and compressed binary format.

In the next step the data is read back in from the ASCII file. To confirm that the data is correctly saved, the first five rows are printed: first for the original array and then for the data read in.

```
data = np.hstack((wl.reshape(-1,1), radiance))
print(data[:5,:])
filename = 'planck.txt'
np.savetxt(filename, data, fmt='%.5e', delimiter=' ', newline='\n',
               header='First column is wavelength, rests are radiance ↵
                  values',
               footer='', comments='# ')
fdata = np.loadtxt(filename, comments='#', usecols=None)
print(' ')
print(fdata[:5,:])
#delete the file
try:
    os.remove(filename)
except OSError:
    pass
```

```
[[3.00000000e-01 3.34707227e+06 1.65601053e+07 1.22419595e+08]
 [3.05000000e-01 3.60634396e+06 1.73819698e+07 1.24380201e+08]
 [3.10000000e-01 3.87124455e+06 1.81921849e+07 1.26143776e+08]
 [3.15000000e-01 4.14096682e+06 1.89884520e+07 1.27714942e+08]
 [3.20000000e-01 4.41469987e+06 1.97686663e+07 1.29098956e+08]]

[[3.00000e-01 3.34707e+06 1.65601e+07 1.22420e+08]
 [3.05000e-01 3.60634e+06 1.73820e+07 1.24380e+08]
 [3.10000e-01 3.87124e+06 1.81922e+07 1.26144e+08]
 [3.15000e-01 4.14097e+06 1.89885e+07 1.27715e+08]
 [3.20000e-01 4.41470e+06 1.97687e+07 1.29099e+08]]
```

It is however quite difficult to check every single element in the array we want to compare the two arrays, using Numpy to do the checking. The `array\_equal()` function indicates that the array values differ.

Problem! Why are the two arrays not the same?

The values written to the file has only five decimal places resolution - some of the information is lost on writing to the file!

There is another comparison function that allows us to specify a tolerance to be used when comparing two arrays

```
print('The two arrays are exactly the same {}'.format(np.array_equal(↵
    data, fdata)))
tolerance = 1e-6
print('The two arrays are the same {} to within {} relative tolerance'↵
    .\
        format(np.allclose(data, fdata,tolerance),tolerance))
```

```
The two arrays are exactly the same False
The two arrays are the same False to within 1e-06 relative tolerance
```

Numpy arrays can be written to raw binary format using the `array.tofile('filename')` function. This is not the binary numpy format, but raw bytes with no header. These file are not portable across endian boundaries.

1.20.10 Array pre-allocation instead of stacking

Stacking is a slow process, use with care for large arrays. Another way would be to calculate the size of the array beforehand and pre-allocate an array of that size. Once the array exists, the values can be assigned by slicing. In this case the number of rows must remain the same, but a new column will be added to allow space for the spectral vector and the radiance array.

```
newsize = (radiance.shape[0], radiance.shape[1] + 1)
ndata = np.zeros(newsize)

print('Original data shape is {}'.format(data.shape))
print('Newly assigned data shape is {}'.format(ndata.shape))

ndata[:,:1] = wl.reshape(-1,1) #first column
ndata[:,1:] = radiance # second column onwards

print('The stacked and pre-allocated arrays are exactly the same {}'.↵
    format(np.array_equal(data, ndata)))
```

```
Original data shape is (101, 4)
Newly assigned data shape is (101, 4)
The stacked and pre-allocated arrays are exactly the same True
```

1.20.11 Save multiple numpy arrays to binary file

```
import numpy as np
import os.path
arrA = np.asarray([[1,2],[3,4]])
arrB = np.asarray([[11,22],[33,44]])
npzfilename = 'savearray.npz'
np.savez_compressed(npzfilename, arrA=arrA, arrB=arrB)
#see if the file exists
if os.path.isfile(npzfilename):
    arr = np.load(npzfilename)
    print(arr['arrA'])
    print(arr['arrB'])
```

```
[[1 2]
 [3 4]]
[[11 22]
 [33 44]]
```

1.21 Element-wise conditionals with Numpy

1.21.1 Avoid divide-by-zero errors in np.log

Use the `where` parameter of numpy's ufuncs:

```
numpy.log(x,
    out=None, where=True, casting='same\_kind', order='K', dtype=None,
        subok=True[, signature, extobj])
            = <ufunc 'log'>
```

out ndarray, None, or tuple of ndarray and None, optional

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where array_like, optional

This condition is broadcast over the input. At locations where the condition is True, the out array will be set to the ufunc result. Elsewhere, the out array will retain its original value. Note that if an uninitialized out array is created via the default out=None, locations within it where the condition is False will remain uninitialized.

Use it like this:

```
res = np.log2(m, out=np.zeros\_like(m), where=(m!=0))
```

No RuntimeWarning is raised, and - log(m) [the ufunc] is used for elements where the condition is True - out is assigned to `out=np.zeros\_like(m)` for elements where the condition is False (the log is not computed)

`out=np.zeros\_like(m)` is quite important. The output might look alright when you forget this, but you will be using uninitialized memory.

https://stackoverflow.com/questions/21752989/numpy-efficiently-avoid-0s-when-taking-logmatrix/52209380

https://numpy.org/doc/stable/reference/ufuncs.html

```
import numpy as np
m = np.array([[1., 0], [2, 3]])
res = np.log2(m)
```

```
C:\Users\nwillers\AppData\Local\Temp/ipykernel_16528/3823898964.py:3: ↵
   RuntimeWarning:

divide by zero encountered in log2
```

```
m = np.array([[1., 0], [2, 3]])
res = np.log2(m, out=np.zeros_like(m), where=(m!=0))
```

1.21.2 Test for NaN in array

NaNs may interfere with some operations, e.g. plotting a mesh grid in Matplotlib. Working with NaNs in Numpy is well supported.

```
a = np.asarray([[1, np.nan], [3, 4], [4,5]])
print(a)
print(np.isnan(a))
print(np.isnan(a).any())
```

```
[[ 1. nan]
 [ 3.  4.]
 [ 4.  5.]]
[[False  True]
 [False False]
 [False False]]
True
```

1.21.3 Selections into array

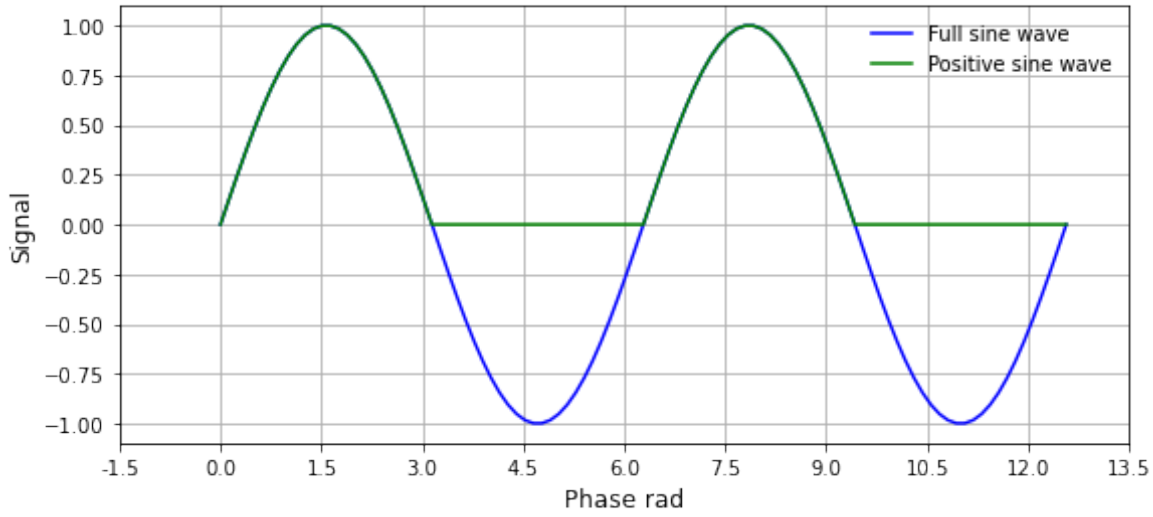
Numpy has a rich set of functions to support conditional testing on element level. This section only touches on the surface.

In the example below a signal is created that only contains the positive values of a sine wave, being zero elsewhere.

```
phase = np.linspace(0, 4 * np.pi, 101)
signal = np.sin(phase)
signalpos = np.where(signal>0, signal, 0)

p = ryplot.Plotter(1,figsize=(9,4))
p.plot(1,phase,signal,'', 'Phase rad', 'Signal', label=['Full sine wave↵
   '])
```

```
p.plot(1,phase,signalpos,'', 'Phase rad', 'Signal', label=['Positive ↵
    sine wave'])
```

```
<AxesSubplot:xlabel='Phase rad', ylabel='Signal'>
```



1.22 Selecting elements with array indices

Numpy provides the facility to extract only certain array indices, based on conditional values.

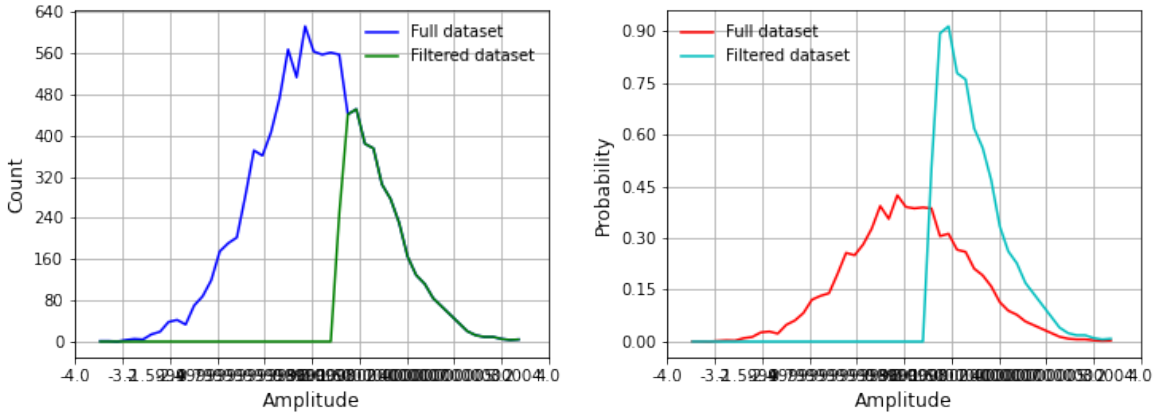
Some experiment requires a gaussian-distributed random number sequence, but with a twist: only values in a specific amplitude range is required. The following script creates a large number of gaussian random variables, then filters them based on amplitude by selecting a sub set of all values by the logical test `rndAll > 0.4`. This test creates a logical array which selects elements from rndAll. The histograms of the two distributions are then calculated and displayed.

The `np.histogram` function can return either the counts in the bins (`density=False`), or the estimated probability function (`density=True`).

```
import numpy as np
import pyradi.ryplot as ryplot
%matplotlib inline

rndAll = np.random.randn(10000)
rndSel = rndAll[rndAll > 0.4]
print('Number of random numbers in input set is {}'.format(rndAll.shape↵
    [0]))
print('Number of random numbers in filtered set is {}'.format(rndSel.↵
    shape[0]))

hAll, bAll = np.histogram(rndAll,bins=50)
hSel, bSel = np.histogram(rndSel,bins=bAll) # use the bins

p = ryplot.Plotter(1,1,2,figsize=(12,4))
p.plot(1,bAll[1:],hAll,'', 'Amplitude', 'Count', label=['Full dataset'↵
    ])
p.plot(1,bSel[1:],hSel,'', 'Amplitude', 'Count', label=['Filtered ↵
    dataset'])
```

```
hAll, bAll = np.histogram(rndAll,bins=50, density=True)
hSel, bSel = np.histogram(rndSel,bins=bAll, density=True) # use the ↵
    bins
p.plot(2,bAll[1:],hAll,'', 'Amplitude', 'Probability', label=['Full ↵
    dataset'])
p.plot(2,bSel[1:],hSel,'', 'Amplitude', 'Probability', label=['Filtered↵
     dataset'])

print('Integrated PDF for all data = {}'.format(np.trapz(hAll,bAll[1:])↵
    ))
print('Integrated PDF for selected data = {}'.format(np.trapz(hSel,bSel↵
    [1:])))
```

```
Number of random numbers in input set is 10000
Number of random numbers in filtered set is 3420
Integrated PDF for all data = 0.9997499999999999
Integrated PDF for selected data = 0.9994152046783624
```



Further experimentation with array indexes with more complex logical patterns. We first create some data with no real physical significance, but all arrays have the same length, as if these are all time samples on the same timeline `a`. Two selections are made, the first based only on the value of `b` and the second based on the value of `b` and `c`. These selections are index arrays with zeros where the conditions are not met, and ones where the conditions are met. The `selecx` index arrays are then used to extract only the required values from the `a`, `b`, `c` arrays when plotting. To make the selected regions easier to spot, they are slightly shifted up or down. The plots are for the following selection filtering definitions:

- All values
- b in the range (18,75)
- b in the range (18,30) and c in the range [0,infinity]
- b in the range (15,75) and c in the range [0,infinity], but at intervals of 1 on a axis.

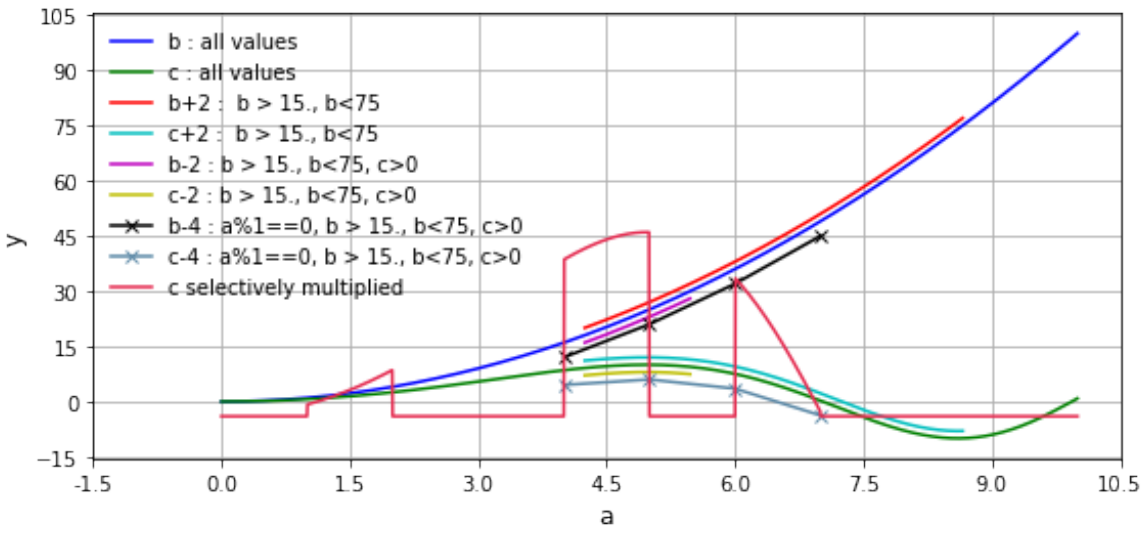
You can stick any number of logical tests inside the `np.all()`.

```
%matplotlib inline
import numpy as np
import pyradi.ryutils as ryutils
import pyradi.ryplot as ryplot
```

```
a = np.linspace(0,10,2001)
b = a ** 2.
c = 10 * np.sin(b/(5 * np.pi))
select0 = np.all([ b > 18., b<75], axis=0)
select1 = np.all([ b > 18., b<30, c>=0], axis=0)
select2 = np.all([ a%1==0, b > 15., b<75, c>0], axis=0)
#the next line builds a complex selection of three sections from the larger set.
select3 = np.logical_or(np.logical_or(np.all([ a>1, a<2], axis=0), np.all([ a>4, a<5], axis=0) ),
                        np.all([ a>6, a<7], axis=0))

print(select3)
print('a size {}'.format(a.shape))
print('a for (b > 18., b<75) size {}'.format(a[select0].shape))
print('a for (b > 18., b<30, c>=0) size {}'.format(a[select1].shape))
print('a for (a%1==0, b > 15., b<75, c>0) size {}'.format(a[select2].shape))
print('a for (a>1, a<2, d>4, d<5 ) size {}'.format(a[select3].shape))

p = ryplot.Plotter(1,figsize=(9,4))
p.plot(1,a,b,'', 'a', 'y', label=['b : all values'])
p.plot(1,a,c,'', 'a', 'y', label=['c : all values'])
p.plot(1,a[select0],b[select0]+2, label=['b+2 :  b > 15., b<75'])
p.plot(1,a[select0],c[select0]+2,label=['c+2 :  b > 15., b<75'])
p.plot(1,a[select1],b[select1]-2,label=['b-2 : b > 15., b<75, c>0'])
p.plot(1,a[select1],c[select1]-2,label=['c-2 : b > 15., b<75, c>0'])
p.plot(1,a[select2],b[select2]-4,label=['b-4 : a%1==0, b > 15., b<75, c>0'],markers='x')
p.plot(1,a[select2],c[select2]-4,label=['c-4 : a%1==0, b > 15., b<75, c>0'],markers='x')

p.plot(1,a,5 * select3 * c-4,label=['c selectively multiplied'])
```

```
[False False False ... False False False]
a size (2001,)
a for (b > 18., b<75) size (884,)
a for (b > 18., b<30, c>=0) size (247,)
a for (a%1==0, b > 15., b<75, c>0) size (4,)
a for (a>1, a<2, d>4, d<5 ) size (597,)
```

```
<AxesSubplot:xlabel='a', ylabel='y'>
```



1.23 2D selection after interpolation

The code below takes a low-resolution 2D input and interpolates the data to a higher resolution. The nature of the data, with high value at the edge of the domain, results in a poor interpolation output. To obtain a better result, I mirrored the data across $x = 0$ to obtain better values around $x = 0$ (first graph). The new data set is interpolated but now has values for $x < 0$, which were required for interpolation but are not part of the valid result. These additional values had to be truncated from the data. This was done in the following code:

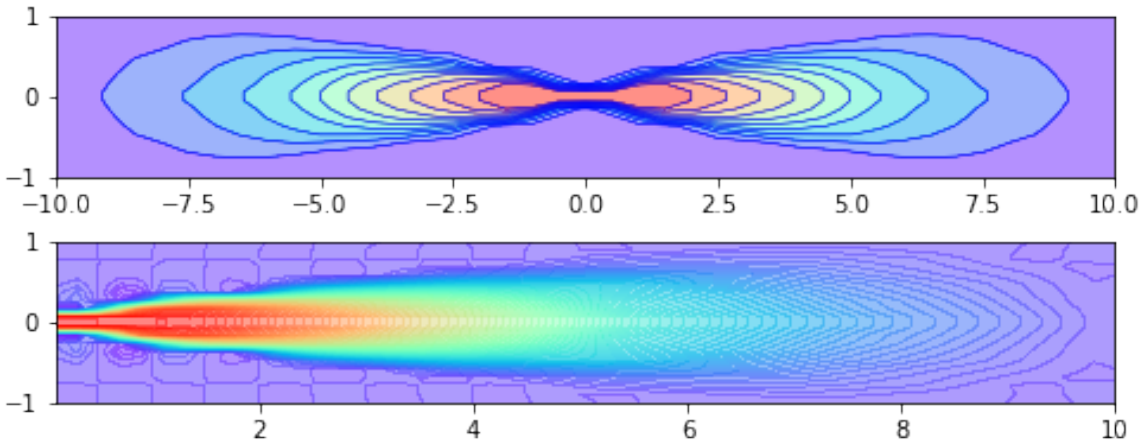
```
select = xnew.\_\_ge\_\_(0)
xnew = xnew[select].reshape(znew.shape[0],-1)
ynew = ynew[select].reshape(znew.shape[0],-1)
znew = znew[select].reshape(znew.shape[0],-1)
```

where the first line returns a binary array with ones where the condition is true and zeros elsewhere. This binary array is used to slice the mirrored data; but the returned value is a `(-1,)`-shaped array. The array had to be reshaped to obtain the 2D form required. The result is shown in the second graph.

```
from scipy import interpolate
p = ryplot.Plotter(1,2,1,figsize=(8,3));
z = np.asarray([
    [300, 300, 300, 300, 300, 300, 300, 300, 300, 2400, 2400, 300, 300,↵
        300, 300, 300, 300, 300, 300, 300],
    [300, 300, 300, 300, 300, 300, 300, 300, 700, 2400, 2400, 700, 300,↵
        300, 300, 300, 300, 300, 300, 300],
    [300, 300, 300, 300, 300, 300, 300, 730, 2152, 2400, 2400, 2152, ↵
       730, 300, 300, 300, 300, 300, 300, 300],
    [300, 300, 300, 300, 300, 300, 300, 1521, 2200, 2350, 2350, 2200, ↵
       1521, 300, 300, 300, 300, 300, 300, 300],
    [300, 300, 300, 300, 300, 300, 510, 1620, 2050, 2200, 2200, 2050, ↵
       1620, 510, 300, 300, 300, 300, 300, 300],
    [300, 300, 300, 300, 300, 485, 1000, 1620, 1900, 2050, 2050, 1900, ↵
       1620, 1000, 485, 300, 300, 300, 300, 300],
    [300, 300, 300, 300, 300, 638, 1120, 1550, 1753, 1895, 1895, 1753, ↵
       1550, 1120, 638, 300, 300, 300, 300, 300],
    [300, 300, 300, 300, 403, 750, 1150, 1450, 1620, 1728, 1728, 1620, ↵
       1450, 1150, 750, 403, 300, 300, 300, 300],
```

```
    [300, 300, 300, 300, 495, 800, 1100, 1336, 1486, 1580, 1580, 1486, ↵
       1336, 1100, 800, 495, 300, 300, 300, 300],
    [300, 300, 300, 350, 540, 805, 1009, 1200, 1325, 1393, 1393, 1325, ↵
       1200, 1009, 805, 540, 350, 300, 300, 300],
    [300, 300, 331, 400, 580, 805, 920, 1025, 1129, 1200, 1200, 1129, ↵
       1025, 920, 805, 580, 400, 331, 300, 300],
    [300, 300, 360, 429, 600, 760, 830, 900, 950, 1015, 1015, 950, 900,↵
        830, 760, 600, 429, 360, 300, 300],
    [300, 315, 380, 440, 570, 680, 760, 807.8, 840, 894, 894, 840, ↵
       807.8, 760, 680, 570, 440, 380, 315, 300],
    [300, 330, 385, 440, 540, 610, 680, 717.3, 750, 790, 790, 750, ↵
       717.3, 680, 610, 540, 440, 385, 330, 300],
    [300, 335, 380, 430, 488, 540, 595, 630, 663, 695, 695, 663, 630, ↵
       595, 540, 488, 430, 380, 335, 300],
    [300, 335, 370, 404, 440, 490, 520, 553, 577.6, 613, 613, 577.6, ↵
       553, 520, 490, 440, 404, 370, 335, 300],
    [300, 320, 354, 374, 400, 440, 470, 500, 513, 538, 538, 513, 500, ↵
       470, 440, 400, 374, 354, 320, 300],
    [300, 310, 331, 347, 370, 400, 420, 440, 460, 477, 477, 460, 440, ↵
       420, 400, 370, 347, 331, 310, 300],
    [300, 300, 300, 320, 340, 360, 370, 385, 400, 414.6, 414.6, 400, ↵
       385, 370, 360, 340, 320, 300, 300, 300],
    [300, 300, 300, 300, 309, 317.9, 325, 330, 340, 350, 350, 340, 330,↵
        325, 317.9, 309, 300, 300, 300, 300],
    [300, 300, 300, 300, 300, 300, 300, 300, 300, 300, 300, 300, 300, ↵
       300, 300, 300, 300, 300, 300, 300]
    ]).T
z = np.hstack((np.fliplr(z)[:,:-1],z))
xi = np.linspace(-10,10,z.shape[1])
yi = np.linspace(-1,1,z.shape[0])
x,y = np.meshgrid(xi,yi)
p.meshContour(1, x,y,z);

aa = np.hstack((y.reshape(-1,1),x.reshape(-1,1),z.reshape(-1,1) ))
ynew, xnew = np.mgrid[-1:1:20j, -10:10:100j]
tck = interpolate.bisplrep(aa[:,0], aa[:,1], aa[:,2], s=0 )
znew = interpolate.bisplev(ynew[:,0], xnew[0,:], tck)

select = xnew.__ge__(0)
xnew = xnew[select].reshape(znew.shape[0],-1)
ynew = ynew[select].reshape(znew.shape[0],-1)
znew = znew[select].reshape(znew.shape[0],-1)

p.meshContour(2, xnew, ynew, znew, levels=100, contourLine=False);
```

```
C:\Users\nwillers\Anaconda3\lib\site-packages\scipy\interpolate\↵
   _fitpack_impl.py:977: RuntimeWarning:

No more knots can be added because the number of B-spline
coefficients already exceeds the number of data points m.
Probable causes: either s or m too small. (fp>s)
    kx,ky=3,3 nx,ny=26,43 m=820 fp=221.139521 s=0.000000
```



1.23.1 Piecewise functions and interpolation

Suppose you want to construct a lookup table for q given by

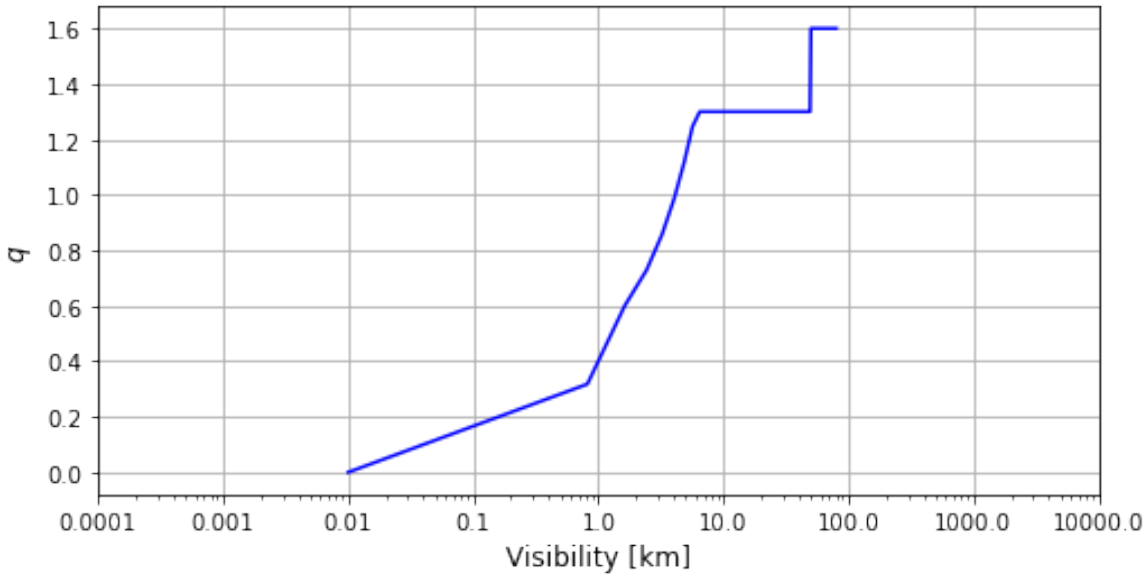
q	V
1.6	V>50 km
1.3	6 km > V > 50 km
0.16V+0.34	1 km > V > 6 km
V-0.5	0.5 > V > 1 km
0	V < 0.5 m

Note this this data is based on a horribly incorrect model of the atmosphere, the data and model are wrong, but serve to illustrate the numpy concepts.

The code below constructs the table (`vis`,`q`), plots the results and create a linear interpolation lookup function.

```
vis = np.linspace(0.01,80,100)
q = np.zeros(vis.shape)
q += np.where(vis>50,1.6,0)
q += np.where((vis>6)&(vis<=50),1.3,0)
q += np.where((vis>1)&(vis<=6),0.16*vis+0.34,0)
q += np.where((vis>0.5)&(vis<=1),vis-0.5,0)
q += np.where(vis<=0.5,0,0)

import pyradi.ryplot as ryplot
%matplotlib inline
p = ryplot.Plotter(1,1,1,figsize=(8,4))
p.semilogX(1,vis,q,'','Visibility [km]','$q$')

from scipy import interpolate
f = interpolate.interp1d(vis,q)
vs = [1,5,10,60]
for v in vs:
    print('Lookup value at vis={} is {}'.format(v,f(v)))
```

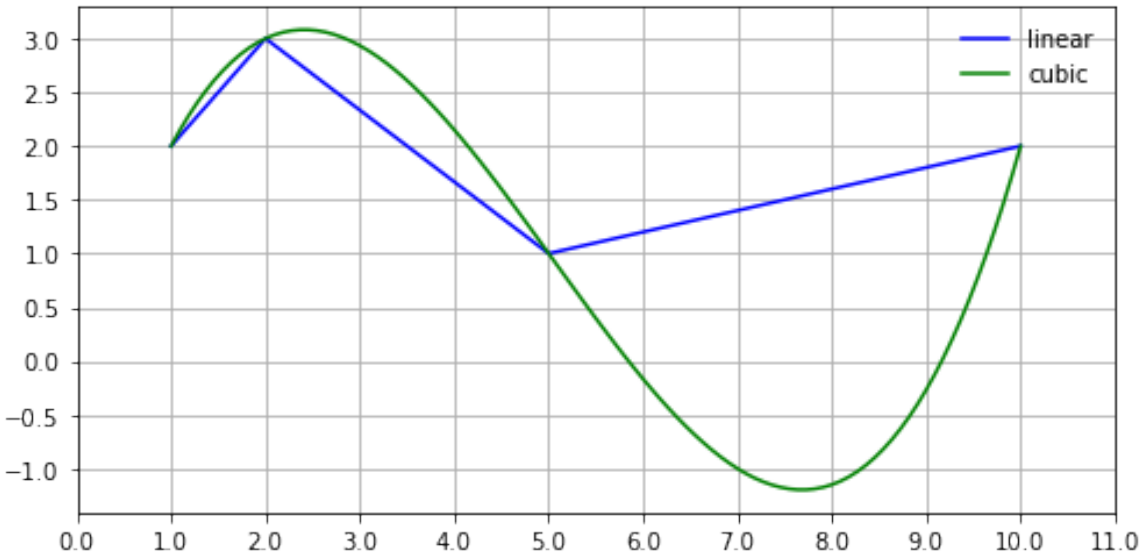
```
Lookup value at vis=1 is 0.3815473782707687
Lookup value at vis=5 is 1.1400000000000001
Lookup value at vis=10 is 1.3
Lookup value at vis=60 is 1.6
```



SciPy has a number of interpolation options in `scipy.interpolate`. The one-dimensional interpolation scheme enables the definition of the lookup table as a function by using `scipy.interpolate.interp1d` The user can define the interpolation scheme to be used - the default is `kind='linear'`, to do cubic interpolation use `kind='cubic'`. In the example below obsserve the difference between the two schemes. This is an ill-posed problem: normally one would have a much denser input data set. The purpose with using this poor data set it to illustrate the dangers of interpolation.

```
from scipy import interpolate
import pyradi.ryplot as ryplot
import numpy as np
%matplotlib inline

# create input data in wavelength and wavenumber domain
wl = np.asarray([1, 2, 5, 10])
fn = np.asarray([2,3,1, 2])

# create the plotting samples - much tighter than input data
wli = np.linspace(1,10,100)

#create the lookup functions
fwll = interpolate.interp1d(wl,fn)
fwlc = interpolate.interp1d(wl,fn, kind='cubic')

p = ryplot.Plotter(1,1,1,figsize=(8,4))
p.plot(1,wli,fwll(wli),label=['linear'])
p.plot(1,wli,fwlc(wli),label=['cubic'])
```

```
<AxesSubplot:>
```



1.23.2 Selective operations on elements in an array: Poisson distribution for large λ

The Poisson probability distribution describe the arrival/generation of discrete events, e.g., generation or arrival of photons in optical flux. The noise variance σ^2 in such a photon stream is equal to the mean value λ. The Poisson distribution is a discrete distribution and is defined only for integers. A practical problem arises when the photon count is large: the numerical limits of integer arithmetic is exceeded in the Numpy `np.random.poisson` function.

For mean values exceeding 1000 the Normal distribution with mean and variance equal to λ is a good approximation[157] to the Poisson distribution. The Normal distribution is a real valued distribution and is not restricted by integer computation limits. So we can use the Normal distribution for large λ. However, for small λ it is a poor approximation, so we must use both, each in the appropriate case. This can be done with array code of the form shown here, where `in` is the value of λ:

```
out =  (inp<=1000) * np.random.poisson(inp * (inp<=1000) )\
     + (inp>1000) * np.random.normal(loc=inp, scale=np.sqrt(inp))
```

where `(inp<=1000)` and `(inp>1000)` are boolean arrays of the same size as the input, but with values corresponding with the element-wise conditional truth. The result therefore comprises a combination of Poisson and Normal elements, element-wise conditional on the value of λ. To prevent the integer overflow for large λ we also have to scale the value of λ with the conditional in the code `in * (inp<=1000)`, reducing the value to zero where it exceeds the threshold of 1000.

The above code works well, provided that λ (`inp`) exceeds zero, because the Normal distribution is not kind to zero variance. Conceptually it should just return the mean value, but computationally the implementation seems to want to divide by zero. So we must prevent passing a zero standard deviation value to `np.random.normal`. This can be done by adding a small value to the standard deviation, say `1e-10`. This should have negligible effect when $\lambda > 1000$, but not when λ is small, near zero. For catch this condition we modify the code as follows:

```
out =  (inp<=1000) * np.random.poisson(inp * (inp<=1000) )\
     + ((inp>tpoint) & (inp!=0)) * np.random.normal(loc=inp, scale=np.sqrt(inp+1e
```

Now the Normal distribution value is only added for values exceeding zero.

```
import numpy as np
tpoint = 1000
sdelta = 1e-10
asize = 100000 # you need values of more than 10000000 to get really ↵
   good stats
for lam in [0, 10, tpoint-5, tpoint-1, tpoint, tpoint+1, tpoint+5, ↵
   20000]:
    inp = lam * np.ones((asize,1))
    out =  (inp<=tpoint) * np.random.poisson(inp * (inp<=tpoint) )\
         + ((inp>tpoint) & (inp!=0)) * np.random.normal(loc=inp, scale↵
            =np.sqrt(inp+sdelta))

    print('lam={} mean={} var={} err-mean={} err-var={}'.format(lam,
       np.mean(out),np.var(out), (lam-np.mean(out))/lam, (lam-np.var(↵
          out))/lam))
```

```
lam=0 mean=0.0 var=0.0 err-mean=nan err-var=nan
lam=10 mean=10.0104 var=9.998891839999999 err-mean↵
   =-0.001040000000000063 err-var=0.00011081600000011349
lam=995 mean=995.09094 var=1006.4988299163997 err-mean↵
   =-9.139698492466933e-05 err-var=-0.011556612981306232
lam=999 mean=999.00183 var=992.0658266510998 err-mean↵
   =-1.8318318318727272e-06 err-var=0.006941114463363595
lam=1000 mean=1000.02757 var=995.4013098951001 err-mean↵
   =-2.7569999999968787e-05 err-var=0.004598690104899902
lam=1001 mean=1001.0345116617314 var=1002.8937435509258 err-mean↵
   =-3.4477184546811076e-05 err-var=-0.0018918516992265928
lam=1005 mean=1004.9608075063601 var=1008.7897370896733 err-mean↵
   =3.899750610936954e-05 err-var=-0.0037708826762918217
lam=20000 mean=19999.917310582656 var=19891.047335329156 err-mean↵
   =4.1344708672113485e-06 err-var=0.005447633233542183
```

```
C:\Users\nwillers\AppData\Local\Temp/ipykernel_16528/68833440.py:11: ↵
   RuntimeWarning:

invalid value encountered in double_scalars
```

1.23.3 Numpy meshgrid

The `numpy.meshgrid` function takes two vectors and returns two two-dimensional arrays that loop the vector values along the two orthogonal dimensions. The returned arrays are the same for any form of input vectors, (N,), (1,N), or (N,1). This is best illustrated by an example

```
valCols = np.asarray([1, 2, 3, 4, 5])
valRows = np.asarray([10, 20, 30])
varCol, varRow = np.meshgrid(valCols, valRows)
print('valCols vector is  (shape={}) \n{}'.format(valCols.shape,valCols↵
   ))
print('valRows vector is  (shape={}) \n{}'.format(valRows.shape,valRows↵
   ))
print('varCol array is (shape={})\n{}'.format(varCol.shape,varCol))
print('varRow array is  (shape={})\n{}'.format(varRow.shape,varRow))
```

```
valCols vector is  (shape=(5,))
[1 2 3 4 5]
valRows vector is  (shape=(3,))
```

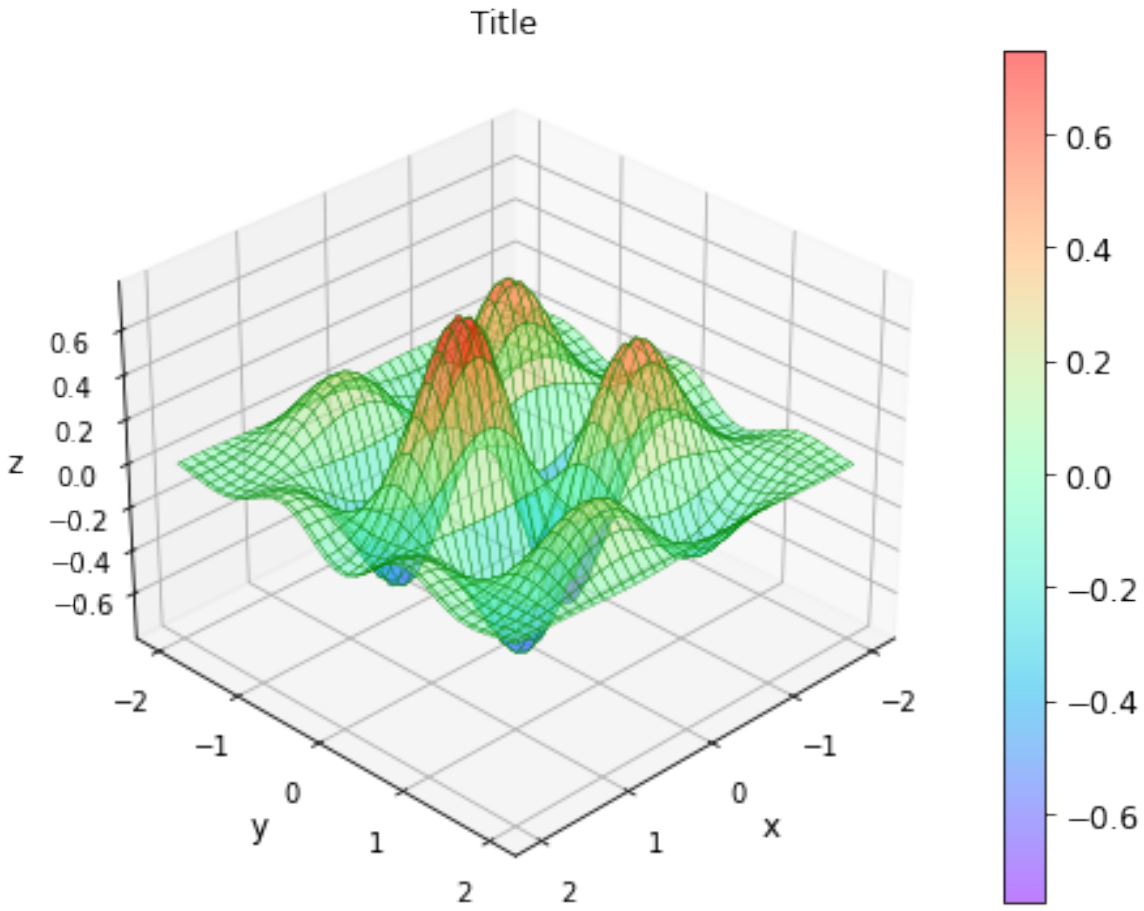
```
[10 20 30]
varCol array is (shape=(3, 5))
[[1 2 3 4 5]
 [1 2 3 4 5]
 [1 2 3 4 5]]
varRow array is  (shape=(3, 5))
[[10 10 10 10 10]
 [20 20 20 20 20]
 [30 30 30 30 30]]
```

The `numpy.meshgrid` function was conceived to create a two-dimensional grid, e.g., for a cartesian grid. Because the two arrays returned contains variations along rows and along columns only, these arrays can be used in any function that requires to operate on two parameters.

In the following example a square cartesian grid on [-2,2] is formed. `numpy.meshgrid` is used to create two arrays for varying x and varying y. The two arrays are flattened (made into (N,)) and then a relatively complex function is applied to the data. After the function call the data is reshaped back to the original shape. Note that the calculation over two dimensions is done without any `for` loops.

```
import numpy as np
import pyradi.ryplot as ryplot
from matplotlib import cm

def myFunc(x,y):
  scale = np.sqrt(np.exp(-(x**2 +y**2)))
  return np.sin(2 * x) * np.cos(4 * y) * scale

x = np.linspace(-2, 2, 101)
y = np.linspace(-2, 2, 101)
varx, vary = np.meshgrid(x, y)
zdata = myFunc(varx.flatten(), vary.flatten()).reshape(varx.shape)

p = ryplot.Plotter(1,1,1,figsize=(10,6))
p.mesh3D(1, varx, vary, zdata, ptitle='Title', xlabel='x', ylabel='y', ↵
   zlabel='z',
  rstride=3, cstride=3, linewidth= 0.3, maxNX=5, maxNY=5, maxNZ=0,
  drawGrid=True, cbarshow=True, alpha=0.5)
```

```
<Axes3DSubplot:title={'center':'Title'}, xlabel='x', ylabel='y'>
```



1.23.4 Ranges with complex / imaginary step sizes

`numpy.mgrid()` can be used to construct 1d ranges as a convenient substitute for arange. It also allows the use of complex-numbers in the step-size to indicate the number of points to place between the (inclusive) end-points. The real purpose of this function however is to produce N, N-d arrays which provide coordinate arrays for an N-dimensional volume.

If the step length is not a complex number, then the stop is not inclusive and the step size is interpreted as a step increment. However, if the step length is a complex number (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value is inclusive.

http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.mgrid.html

http://docs.scipy.org/doc/scipy/reference/tutorial/basic.html

```
import numpy as np
a = np.mgrid[-5:5:5]
print(a)
ai = np.mgrid[-5:5:5j]
print(ai)
```

```
[-5  0]
[-5.  -2.5  0.   2.5  5. ]
```

1.23.5 Numpy integration and meshgrid

Integration in Numpy is quite easy, the function `numpy.trapz` does all the work:

```
for n in [4, 11, 101, 1001, 100001]:
    t = np.linspace(0,  np.pi/2, n)
    s = np.sin(t)
    int = np.trapz(s, t, axis=0)
    print('Integral of sin over [0,pi/2] in {} samples is {} '.format(n,int))
```

```
Integral of sin over [0,pi/2] in 4 samples is 0.9770486166568533
Integral of sin over [0,pi/2] in 11 samples is 0.9979429863543573
Integral of sin over [0,pi/2] in 101 samples is 0.9999794382396074
Integral of sin over [0,pi/2] in 1001 samples is 0.999999794383233
Integral of sin over [0,pi/2] in 100001 samples is 0.999999999979438
```

A recurring pattern in the use of pyradi is the integration of a two-dimensional dataset along one of the dimensions. In most programming languages, this would require at least one or possibly two loops over the dimensions. The `numpy.meshgrid` function has the potential of trading the complexity of looping against the complexity of loop abstraction.

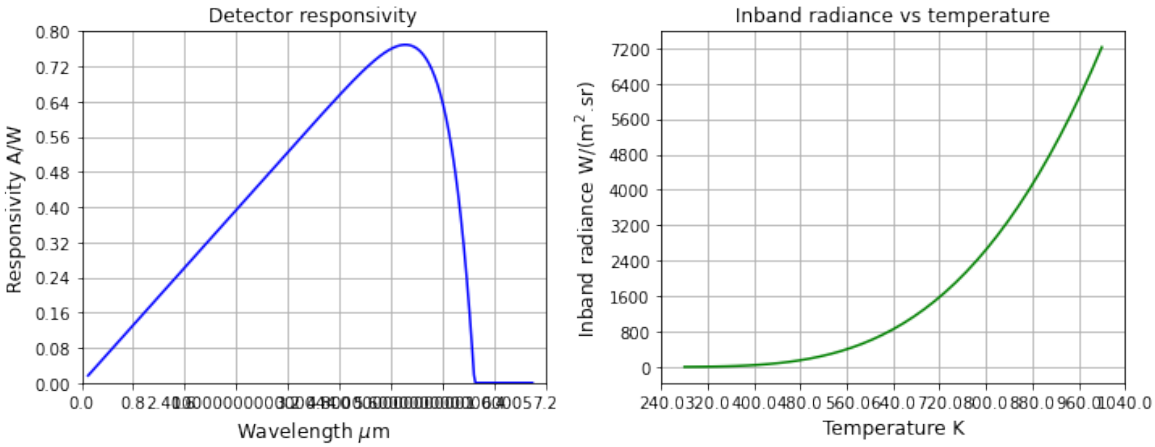
In the example below the inband blackbody radiance over a spectral detector is calculated using

$L_{\text{inband}} = \int_0^{\infty} R_\lambda L_\lambda(T) d\lambda \approx \sum_{0.1\mu\text{m}}^{7\mu\text{m}} R_\lambda L_\lambda(T) d\lambda$

The wavelength spectral domain (shape (S,1)) is calculated from a short wavelength up to 7 μm, covering the detector spectral band. The detector spectral response (shape (S,1)) is calculated using using a function from `pyradi.ryutils`. A range of source temperatures is defined (shape (T,1)). The same spectral range used to calculate the detector is used to calculate the blackbody radiance, using `pyradi.ryplanck.planck`. The value returned by `ryplanck.planck` is a two-dimensional (shape (S,T)) array with spectral variation across the rows (axis=0) and temperature variation across the columns (axis=1). The challenge is now to multiply each column in the radiance table with the spectral responsivity, without using a loop. The `numpy.meshgrid` function is used to calculate an array with the responsivity variation along the rows, but with exactly the same values in all columns. Once this new array is available, it is multiplied with the radiance array. After multiplication the columns in the new array contain the product of $R_\lambda L_\lambda(T)$, with each column at a different temperature. The summation/integral is then performed by using `numpy.trapz` and integrating along axis=0 (rows), which is the direction of spectral variation. The resulting vector is the integral over all wavelengths, for the different temperatures.

```
import pyradi.ryutils as ryutils
import pyradi.ryplanck as ryplanck
wavelength = np.linspace(0.1, 7, 305).reshape(-1, 1)
detector = ryutils.responsivity(wavelength, 6.1, 1, 15, 1.0)
temperatures = np.linspace(280, 1000, 41)
spectralBaseline = ryplanck.planck(wavelength,temperatures,'el') / np.pi
_, detVar = np.meshgrid(temperatures, detector)
spectralWeighted = spectralBaseline * detVar
inbandRadiance = np.trapz(spectralWeighted, wavelength, axis=0 )

p = ryplot.Plotter(2,1,3,figsize=(18,4))
p.plot(1,wavelength,detector,'Detector responsivity','Wavelength $\mu$m', 'Responsivity A/W',pltaxis=[0.5,7,0,0.8])
p.plot(2,temperatures,inbandRadiance,'Inband radiance vs temperature','Temperature K', 'Inband radiance W/(m$^2$.sr)')
```

```
<AxesSubplot:title={'center':'Inband radiance vs temperature'}, xlabel↵
   ='Temperature K', ylabel='Inband radiance W/(m$^2$.sr)'>
```



1.24 Row-wise dot product

http://stackoverflow.com/questions/26168363/elegant-expression-for-row-wise-dot-product-of-two-matrices

Suppose you have two arrays of vectors, both with shape (N,3). Each row in each array respresents a 3D coordinate. The angle between the corresponding 3D vectors can be determined from the dot product along along axis=1.

```
#create random arrays and normalise
def make3darrays(shp):
    a = np.random.random(shp)
    a = a / np.linalg.norm(a,axis=1).reshape(-1,1)
    return (a)

#make two arrays
a = make3darrays((4,3))
b = make3darrays((4,3))

# conform apprpriate shape
print(a, np.linalg.norm(a,axis=1))
print(b, np.linalg.norm(b,axis=1))

ang = np.arccos(np.clip(np.sum(a*b,axis=1),-1,1))

print(ang)
```

```
[[0.52523594 0.4210046  0.73951493]
 [0.14665794 0.55280291 0.82030506]
 [0.44925517 0.65201453 0.6107756 ]
 [0.79155146 0.28446877 0.5408547 ]] [1. 1. 1. 1.]
[[0.73927677 0.66421933 0.11082659]
 [0.73776046 0.66406452 0.12135822]
 [0.97548154 0.09171993 0.20005803]
 [0.45630827 0.75253848 0.47483534]] [1. 1. 1. 1.]
[0.72289745 0.9583804  0.90175609 0.58794526]
```

1.25 Overflow errors in Numpy

Thanks to Bertus Theron for this input.

It refers to the fact that (for older versions of Python) neither exception nor warning is raised by Numpy when an integer overflow is encountered. Later versions of Python cause an error. Test with this example:

```
#Calculation without neither receiving warning nor exception ↵
   encountered
# a = np.array( [400,  300], dtype = np.int32 )
# print(a)
# a**4
#note that IPython does warn you
```

How to make Numpy raise an exception for the same calculation

```
# np.seterr( invalid='raise' )
# a**4
```

The default NumPy error handling setting is: invalid = "warn"

I notice that this warning is correctly trapped for scalar NumPy integers and for single-element NumPy arrays, BUT not trapped for NumPy arrays with 2 or more elements.

Regarding numpy.seterr(), the NumPy documentation says

```
numpy.seterr(all=None, divide=None, over=None, under=None, invalid=None)[source]
 Set how floating-point errors are handled.
```

and

```
Note that operations on integer scalar types (such as int16) are handled like flo
```

1.26 Algebraic Manipulation with SymPy

SymPy performs symbolic mathematical manipulation and evaluation. The output from SymPy can be displayed in the IPython notebook.

http://nbviewer.ipython.org/github/jrjohansson/scientific-python-lectures/blob/master/Lecture-5-Sympy.ipynb[158]

http://docs.sympy.org/dev/tutorial/printing.html[159]

https://stackoverflow.com/questions/19470796/ipython-notebook-output-latex-or-math

```
from sympy import *
init_printing()
e0 = Symbol('e0')
e1 = Symbol('e1')
a = e1 ** 2 * (1- e0) - e0 ** 2 * (1 - e1) + \
     (e1**2-e0**2)*(1-e0) *(1-e1)/ (1-(1-e0)*(1-e1))
b = a.simplify()
print('$'+latex(b)+'$')
```

```
$\frac{e_{0}^{3} e_{1}^{2} - e_{0}^{3} e_{1} - e_{0}^{2} e_{1}^{3} + e_↵
   {0}^{2} + e_{0} e_{1}^{3} - e_{1}^{2}}{e_{0} e_{1} - e_{0} - e_{1}}$
```

```
# interactive.printing.init_printing(use_latex=True)
a
```

$$-e_0^2\left(1-e_1\right)+e_1^2\left(1-e_0\right)+\frac{\left(1-e_0\right)\left(1-e_1\right)\left(-e_0^2+e_1^2\right)}{-\left(1-e_0\right)\left(1-e_1\right)+1}$$

```
b
```

$$\frac{e_0^3e_1^2-e_0^3e_1-e_0^2e_1^3+e_0^2+e_0e_1^3-e_1^2}{e_0e_1-e_0-e_1}$$

```
from sympy.solvers.solveset import linsolve
from sympy import Matrix, S
from sympy import symbols
init_printing()
x, y, z = symbols("x, y, z")
A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
b = Matrix([3, 6, 9])
c = linsolve((A, b), [x, y, z])
print(A)
print(b)
print(c)
print('$'+latex(A)+'$')
print('$'+latex(b)+'$')
print('$'+latex(c)+'$')
```

```
Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
Matrix([[3], [6], [9]])
FiniteSet((-1, 2, 0))
$\left[\begin{matrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 10\end{matrix}\right↵
   ]$
$\left[\begin{matrix}3\\6\\9\end{matrix}\right]$
$\left\{\left( -1, \  2, \  0\right)\right\}$
```

```
A
```

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 10\end{bmatrix}$$

```
b
```

$$\begin{bmatrix}3\\6\\9\end{bmatrix}$$

```
c
```

$$\left\{(-1,\ 2,\ 0)\right\}$$

```
from sympy.solvers.solveset import linsolve
from sympy import Matrix, S
from sympy import symbols

x0, x1, x2, x3, y0, y1, y2, y3 = symbols('x0, x1, x2, x3, y0, y1, y2, ↵
   y3')
a11, a21, a31, a12, a22, a32, a13, a23, a33 = symbols('a11, a21, a31, ↵
   a12, a22, a32, a13, a23, a33')

A = Matrix([
```

```
    [0,0,1,0,0,0,  0,  0],
    [1,0,1,0,0,0,-x1,  0],
    [1,1,1,0,0,0,-x2,-x2],
    [0,1,1,0,0,0,  0,-x3],
    [0,0,0,0,0,1,  0,  0],
    [0,0,0,1,0,1,-y1,  0],
    [0,0,0,1,1,1,-y2,-y2],
    [0,0,0,0,1,1,  0,-y3]
    ])
b = Matrix([a33*x0,a33*x1,a33*x2,a33*x3,a33*y0,a33*y1,a33*y2,a33*y3])
# b = Matrix([x0,x1,x2,x3,y0,y1,y2,y3])
a = linsolve((A, b), [a11,a21,a31,a12,a22,a32,a13,a23])
print(A)
print('')
print(b)
print('')
print(a)
print('')
print('$'+latex(A)+'$')
print('')
print('$'+latex(b)+'$')
print('')
print('$'+latex(a)+'$')
```

```
Matrix([[0, 0, 1, 0, 0, 0, 0, 0], [1, 0, 1, 0, 0, 0, -x1, 0], [1, 1, 1,↵
    0, 0, 0, -x2, -x2], [0, 1, 1, 0, 0, 0, 0, -x3], [0, 0, 0, 0, 0, 1, ↵
   0, 0], [0, 0, 0, 1, 0, 1, -y1, 0], [0, 0, 0, 1, 1, 1, -y2, -y2], [0,↵
    0, 0, 0, 1, 1, 0, -y3]])

Matrix([[a33*x0], [a33*x1], [a33*x2], [a33*x3], [a33*y0], [a33*y1], [↵
   a33*y2], [a33*y3]])

FiniteSet(((a33*x0*x2*y1 - a33*x0*x2*y3 - a33*x0*x3*y1 + a33*x0*x3*y2 -↵
    a33*x1*x2*y0 + a33*x1*x2*y3 + a33*x1*x3*y0 - a33*x1*x3*y2)/(x1*y2 -↵
    x1*y3 - x2*y1 + x2*y3 + x3*y1 - x3*y2), (-a33*x0*x1*y2 + a33*x0*x1*↵
   y3 + a33*x0*x2*y1 - a33*x0*x2*y3 - a33*x1*x3*y0 + a33*x1*x3*y2 + a33↵
   *x2*x3*y0 - a33*x2*x3*y1)/(x1*y2 - x1*y3 - x2*y1 + x2*y3 + x3*y1 - ↵
   x3*y2), a33*x0, (a33*x0*y1*y2 - a33*x0*y1*y3 - a33*x1*y0*y2 + a33*x1↵
   *y0*y3 - a33*x2*y0*y3 + a33*x2*y1*y3 + a33*x3*y0*y2 - a33*x3*y1*y2)↵
   /(x1*y2 - x1*y3 - x2*y1 + x2*y3 + x3*y1 - x3*y2), (a33*x0*y1*y3 - ↵
   a33*x0*y2*y3 - a33*x1*y0*y2 + a33*x1*y2*y3 + a33*x2*y0*y1 - a33*x2*↵
   y1*y3 - a33*x3*y0*y1 + a33*x3*y0*y2)/(x1*y2 - x1*y3 - x2*y1 + x2*y3 ↵
   + x3*y1 - x3*y2), a33*y0, (a33*x0*y2 - a33*x0*y3 - a33*x1*y2 + a33*↵
   x1*y3 - a33*x2*y0 + a33*x2*y1 + a33*x3*y0 - a33*x3*y1)/(x1*y2 - x1*↵
   y3 - x2*y1 + x2*y3 + x3*y1 - x3*y2), (a33*x0*y1 - a33*x0*y2 - a33*x1↵
   *y0 + a33*x1*y3 + a33*x2*y0 - a33*x2*y3 - a33*x3*y1 + a33*x3*y2)/(x1↵
   *y2 - x1*y3 - x2*y1 + x2*y3 + x3*y1 - x3*y2)))

$\left[\begin{matrix}0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\1 & 0 & 1 & 0 & 0 &↵
    0 & - x_{1} & 0\\1 & 1 & 1 & 0 & 0 & 0 & - x_{2} & - x_{2}\\0 & 1 &↵
    1 & 0 & 0 & 0 & 0 & - x_{3}\\0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\0 & 0 &↵
    0 & 1 & 0 & 1 & - y_{1} & 0\\0 & 0 & 0 & 0 & 1 & 1 & 1 & - y_{2} & - y_↵
   {2}\\0 & 0 & 0 & 0 & 1 & 1 & 0 & - y_{3}\end{matrix}\right]$

$\left[\begin{matrix}a_{33} x_{0}\\a_{33} x_{1}\\a_{33} x_{2}\\a_{33} ↵
   x_{3}\\a_{33} y_{0}\\a_{33} y_{1}\\a_{33} y_{2}\\a_{33} y_{3}\end{↵
   matrix}\right]$

$\left\{\left( \frac{a_{33} x_{0} x_{2} y_{1} - a_{33} x_{0} x_{2} y_↵
   {3} - a_{33} x_{0} x_{3} y_{1} + a_{33} x_{0} x_{3} y_{2} - a_{33} ↵
```

```
x_{1} x_{2} y_{0} + a_{33} x_{1} x_{2} y_{3} + a_{33} x_{1} x_{3} y_{0} - a_{33} x_{1} x_{3} y_{2}}{x_{1} y_{2} - x_{1} y_{3} - x_{2} y_{1} + x_{2} y_{3} + x_{3} y_{1} - x_{3} y_{2}}, \  \frac{- a_{33} x_{0} x_{1} y_{2} + a_{33} x_{0} x_{1} y_{3} + a_{33} x_{0} x_{2} y_{1} - a_{33} x_{0} x_{2} y_{3} - a_{33} x_{1} x_{3} y_{0} + a_{33} x_{1} x_{3} y_{2} + a_{33} x_{2} x_{3} y_{0} - a_{33} x_{2} x_{3} y_{1}}{x_{1} y_{2} - x_{1} y_{3} - x_{2} y_{1} + x_{2} y_{3} + x_{3} y_{1} - x_{3} y_{2}}, \  a_{33} x_{0}, \  \frac{a_{33} x_{0} y_{1} y_{2} - a_{33} x_{0} y_{1} y_{3} - a_{33} x_{1} y_{0} y_{2} + a_{33} x_{1} y_{0} y_{3} - a_{33} x_{2} y_{0} y_{3} + a_{33} x_{2} y_{1} y_{3} + a_{33} x_{3} y_{0} y_{2} - a_{33} x_{3} y_{1} y_{2}}{x_{1} y_{2} - x_{1} y_{3} - x_{2} y_{1} + x_{2} y_{3} + x_{3} y_{1} - x_{3} y_{2}}, \  \frac{a_{33} x_{0} y_{1} y_{3} - a_{33} x_{0} y_{2} y_{3} - a_{33} x_{1} y_{0} y_{2} + a_{33} x_{1} y_{2} y_{3} + a_{33} x_{2} y_{0} y_{1} - a_{33} x_{2} y_{1} y_{3} - a_{33} x_{3} y_{0} y_{1} + a_{33} x_{3} y_{0} y_{2}}{x_{1} y_{2} - x_{1} y_{3} - x_{2} y_{1} + x_{2} y_{3} + x_{3} y_{1} - x_{3} y_{2}}, \  a_{33} y_{0}, \  \frac{a_{33} x_{0} y_{2} - a_{33} x_{0} y_{3} - a_{33} x_{1} y_{2} + a_{33} x_{1} y_{3} - a_{33} x_{2} y_{0} + a_{33} x_{2} y_{1} + a_{33} x_{3} y_{0} - a_{33} x_{3} y_{1}}{x_{1} y_{2} - x_{1} y_{3} - x_{2} y_{1} + x_{2} y_{3} + x_{3} y_{1} - x_{3} y_{2}}, \  \frac{a_{33} x_{0} y_{1} - a_{33} x_{0} y_{2} - a_{33} x_{1} y_{0} + a_{33} x_{1} y_{3} + a_{33} x_{2} y_{0} - a_{33} x_{2} y_{3} - a_{33} x_{3} y_{1} + a_{33} x_{3} y_{2}}{x_{1} y_{2} - x_{1} y_{3} - x_{2} y_{1} + x_{2} y_{3} + x_{3} y_{1} - x_{3} y_{2}}\right)\right\}$
```

A

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & -x_1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & -x_2 & -x_2 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & -x_3 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & -y_1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & -y_2 & -y_2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & -y_3 \end{bmatrix}$$

b

$$\begin{bmatrix} a_{33} x_0 \\ a_{33} x_1 \\ a_{33} x_2 \\ a_{33} x_3 \\ a_{33} y_0 \\ a_{33} y_1 \\ a_{33} y_2 \\ a_{33} y_3 \end{bmatrix}$$

a

$$\left\{\left(\frac{a_{33} x_0 x_2 y_1 - a_{33} x_0 x_2 y_3 - a_{33} x_0 x_3 y_1 + a_{33} x_0 x_3 y_2 - a_{33} x_1 x_2 y_0 + a_{33} x_1 x_2 y_3 + a_{33} x_1 x_3 y_0 - a_{33} x_1 x_3 y_2}{x_1 y_2 - x_1 y_3 - x_2 y_1 + x_2 y_3 + x_3 y_1 - x_3 y_2}, \frac{-a_{33} x_0 x_1 y_2}{}\right.\right.$$

1.26.1 Euler angle sequence rotation

The next few cells calculate the rotation matrix for a sequence of Euler angle object-fixed (passive) rotations. In these rotations the axes or frame for the rotation moves with the object that rotates.

These rotation matrices assume the coordinate to be a row vector that is post-multiplied by the transformation matrix:

$$v' = (v_x, v_y, v_z) \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{1.1}$$

```
##
import sympy
# Symbolic version of rotation matrices
def Rx_s(sym=r'psi'):
    ''' Symbolic rotation matrix about the x-axis, by an angle sym '''
    symR = sympy.Symbol(sym)
    return sympy.Matrix([[1,0,0],
    [0, sympy.cos(symR), -sympy.sin(symR)],
    [0, sympy.sin(symR), sympy.cos(symR)]])

def Ry_s(sym=r'phi'):
    ''' Symbolic rotation matrix about the y-axis, by an angle sym '''
    symR = sympy.Symbol(sym)
    return sympy.Matrix([[sympy.cos(symR),0, sympy.sin(symR)],
    [0,1,0],
    [-sympy.sin(symR), 0, sympy.cos(symR)]])

def Rz_s(sym=r'theta'):
    ''' Symbolic rotation matrix about the z-axis, by an angle sym '''
    symR = sympy.Symbol(sym)
    return sympy.Matrix([[sympy.cos(symR), -sympy.sin(symR), 0],
    [sympy.sin(symR), sympy.cos(symR), 0],
    [0, 0, 1]])
```

Rotation around the x axis:

```
Rx_s('psi')
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\psi) & -\sin(\psi) \\ 0 & \sin(\psi) & \cos(\psi) \end{bmatrix}$$

Rotation around the y axis:

```
Ry_s('phi')
```

$$\begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix}$$

Rotation around the z axis:

```
Rz_s('theta')
```

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The Fick rotation sequence rotates first around the z axis, then the y axis and finally the x axis.

```
Rz_s()*Ry_s()*Rx_s()
```

$$\begin{bmatrix} \cos(\phi)\cos(\theta) & \sin(\phi)\sin(\psi)\cos(\theta)-\sin(\theta)\cos(\psi) & \sin(\phi)\cos(\psi)\cos(\theta)+\sin(\psi)\sin(\theta) \\ \sin(\theta)\cos(\phi) & \sin(\phi)\sin(\psi)\sin(\theta)+\cos(\psi)\cos(\theta) & \sin(\phi)\sin(\theta)\cos(\psi)-\sin(\psi)\cos(\theta) \\ -\sin(\phi) & \sin(\psi)\cos(\phi) & \cos(\phi)\cos(\psi) \end{bmatrix}$$

Calculate the inverse of the Fick matrix. Rotation matrices are symmetrical, hence the inverse is the same as the transpose.

The product of three inverse matrices does not calculate correctly, so it was necessary to simplify the inverse. A second simplification was necessary to simplify the equations to the results shown below.

```
R3I =   sympy.trigsimp(sympy.trigsimp(sympy.Inverse(Rz_s())))
R2I =   sympy.trigsimp(sympy.trigsimp(sympy.Inverse(Ry_s())))
R1I =   sympy.trigsimp(sympy.trigsimp(sympy.Inverse(Rx_s())))


R1I * R2I * R3I
```

$$\begin{bmatrix} \left(-\frac{\sin^2(\phi)}{\cos(\phi)}+\frac{1}{\cos(\phi)}\right)\left(-\frac{\sin^2(\theta)}{\cos(\theta)}+\frac{1}{\cos(\theta)}\right) & \left(-\frac{\sin^2(\phi)}{\cos(\phi)}+\frac{1}{\cos(\phi)}\right)\sin(\theta) \\ -\left(-\frac{\sin^2(\psi)}{\cos(\psi)}+\frac{1}{\cos(\psi)}\right)\sin(\theta)+\left(-\frac{\sin^2(\theta)}{\cos(\theta)}+\frac{1}{\cos(\theta)}\right)\sin(\phi)\sin(\psi) & \left(-\frac{\sin^2(\psi)}{\cos(\psi)}+\frac{1}{\cos(\psi)}\right)\cos(\theta)+\sin(\phi)\sin(\psi)\sin(\theta) \\ \left(-\frac{\sin^2(\theta)}{\cos(\theta)}+\frac{1}{\cos(\theta)}\right)\sin(\phi)\cos(\psi)+\sin(\psi)\sin(\theta) & \sin(\phi)\sin(\theta)\cos(\psi)-\sin(\psi)\cos(\theta) \end{bmatrix}$$

The array can also be output as LaTeX for inclusion in a report:

```
sympy.latex(R1I * R2I * R3I)
```

```
'\\left[\\begin{matrix}\\left(- \\frac{\\sin^{2}{\\left(\\phi \\right)↵
   }}{\\cos{\\left(\\phi \\right)}} + \\frac{1}{\\cos{\\left(\\phi \\↵
   right)}}\\right) \\left(- \\frac{\\sin^{2}{\\left(\\theta \\right)↵
   }}{\\cos{\\left(\\theta \\right)}} + \\frac{1}{\\cos{\\left(\\theta ↵
   \\right)}}\\right) & \\left(- \\frac{\\sin^{2}{\\left(\\phi \\right)↵
   }}{\\cos{\\left(\\phi \\right)}} + \\frac{1}{\\cos{\\left(\\phi \\↵
   right)}}\\right) \\sin{\\left(\\theta \\right)} & - \\sin{\\left(\\↵
   phi \\right)}\\\\- \\left(- \\frac{\\sin^{2}{\\left(\\psi \\right)↵
   }}{\\cos{\\left(\\psi \\right)}} + \\frac{1}{\\cos{\\left(\\psi \\↵
   right)}}\\right) \\sin{\\left(\\theta \\right)} + \\left(- \\frac{\\↵
   sin^{2}{\\left(\\theta \\right)}}{\\cos{\\left(\\theta \\right)}} + ↵
   \\frac{1}{\\cos{\\left(\\theta \\right)}}\\right) \\sin{\\left(\\phi↵
    \\right)} \\sin{\\left(\\psi \\right)} & \\left(- \\frac{\\sin↵
   ^{2}{\\left(\\psi \\right)}}{\\cos{\\left(\\psi \\right)}} + \\frac↵
   {1}{\\cos{\\left(\\psi \\right)}}\\right) \\cos{\\left(\\theta \\↵
   right)} + \\sin{\\left(\\phi \\right)} \\sin{\\left(\\psi \\right)} ↵
   \\sin{\\left(\\theta \\right)} & \\sin{\\left(\\psi \\right)} \\cos↵
   {\\left(\\phi \\right)}\\\\\\left(- \\frac{\\sin^{2}{\\left(\\theta ↵
   \\right)}}{\\cos{\\left(\\theta \\right)}} + \\frac{1}{\\cos{\\left↵
   (\\theta \\right)}}\\right) \\sin{\\left(\\phi \\right)} \\cos{\\↵
   left(\\psi \\right)} + \\sin{\\left(\\psi \\right)} \\sin{\\left(\\↵
   theta \\right)} & \\sin{\\left(\\phi \\right)} \\sin{\\left(\\theta ↵
   \\right)} \\cos{\\left(\\psi \\right)} - \\sin{\\left(\\psi \\right)↵
   } \\cos{\\left(\\theta \\right)} & \\cos{\\left(\\phi \\right)} \\↵
   cos{\\left(\\psi \\right)}\\end{matrix}\\right]'
```

1.27 Regular Expressions

https://www3.ntu.edu.sg/home/ehchua/programming/howto/Regexe.html[161]

1.28 Optimizing and performance tricks

The Top Mistakes Developers Make When Using Python for Big Data Analytics[162]

1.29 References and garbage collection

Python has built-in garbage collection, which for most tasks appear to work reasonably well. In some cases however the standard garbage collection methodology may not suffice and user intervention may be required. Python's garbage collection and its intricacies requires a complete, separate notebook, but here is a summary. What follows is my present understanding, which is incomplete and may even be wrong in places.

Python's memory model uses a reference-to-a-data-item with associated reference counting[163] approach. Some data item is allocated memory and a reference to that memory is given when you 'assign' the data item to a name. So,

`a = ['gg','hh']`

creates the data item (the list) and allocates a reference to this data item to the variable `a`. Note that the data item and the variable are two different things, `a` merely refers to the data item, it isn't the data item.

An object is automatically marked to be collected when its reference count drops to zero.

gc – Python's Garbage Collector[164]

Memory Management and Limits[165]

http://stackoverflow.com/questions/9617001/python-garbage-collection-fails

http://python.6.x6.nabble.com/question-about-garbage-collection-td1661865.html

http://stackoverflow.com/questions/16261240/releasing-memory-of-huge-numpy-array-in-ipython

The `sys.getrefcount()` function shows the number of references attached/assigned to the data item. The count is actually one higher[166] than expected because there is a temporary reference to the object held by getrefcount() itself.

References are removed by assigning the value `None` to the variable.

```
import sys
a = None
print('before assigning ref  a  {}'.format(sys.getrefcount(a)))
a = ['gg','hh']
print('after  assigning ref  a  {}'.format(sys.getrefcount(a)))
b = a
print('after  assigning ref  b  {}'.format(sys.getrefcount(a)))
b = None
print('after  removing  ref  b  {}'.format(sys.getrefcount(a)))
a = None
print('after  removing  ref  a  {}'.format(sys.getrefcount(a)))
```

```
before assigning ref  a  144928
after  assigning ref  a  2
after  assigning ref  b  3
after  removing  ref  b  2
after  removing  ref  a  144938
```

When all references to a data item are removed, the data item still remains in memory, until removed by the Python garbage collector. This may take place at any time (perhaps never at all during a short session) when the Python runtime feels like it. The garbage collector is not able to remove all data items (I seem to recall that lists of floats and ints are not removed).

You can force the garbage collector to clean up the memory by using `gc`, Python's Garbage Collector

```
import gc
gc.collect()
```

644

1.30 Context Managers

From `http://jeffknupp.com/blog/2016/03/07/python-with-context-managers/`[167]:

"We need a convenient method for indicating a particular variable has some cleanup associated with it, and to guarantee that cleanup happens, no matter what. Given that requirement, the syntax for using context managers makes a lot of sense:

```
with something\_that\_returns\_a\_context\_manager() as my\_resource:
    do\_something(my\_resource)
    ...
    print('done using my\_resource')
```

That's it! Using with, we can call anything that returns a context manager (like the built-in `open()` function). We assign it to a variable using `... as <variable\_name>`. Crucially, the variable only exists within the indented block below the with statement. Think of with as creating a mini-function: we can use the variable freely in the indented portion, but once that block ends, the variable goes out of scope. When the variable goes out of scope, it automatically calls a special method that contains the code to clean up the resource."

1.31 HTTP server

The following starts a simple HTTP server in the current working directory.

```
# Python 3.x
$ python3 \-m http\.server    \# Python 2\.x    $ python -m SimpleHTTPServer 8000
```

See also here:

https://docs.python.org/3/library/http.server.html

https://docs.python.org/2.7/library/basehttpserver.html

and here:

https://gist.github.com/bradmontgomery/2219997

https://www.acmesystems.it/python_httpd

For more sophisticated applications consider one of the many full stack Python web frameworks such as Django, or a microframework such as Flask.

https://wiki.python.org/moin/WebFrameworks

https://www.airpair.com/python/posts/django-flask-pyramid

1.32 Hardware interfacing

1.32.1 Serial ports

Python can be used to interface to hardware, one such example investigated here is using the serial port to communicate. The `pyserial` module provides this capability, see the documentation[168]. It is on the Anaconda distribution but not included in the standard download. To install type `conda install pyse` at the prompt in a command window.

A simple example is shown here[169]. In another example[170] `pyserial` is used to communicate with an ESP8266. A detailed example is given here[171]. See this example to read and plot data[172]. This example[173] gets data via a serial interface and then plot it in a GUI.

It turns out that you need to run the Python code with Administrator rights, otherwise it will not grant you access to the COM port.

1.32.2 Communicating with an Arduino /ESP

See these pages:

1. http://fab.cba.mit.edu/classes/863.14/people/guy_zyskind/notebook.html
2. http://lucsmall.com/2015/02/23/homebrew-esp8266-breakout/
3. http://fab.cba.mit.edu/classes/863.14/people/guy_zyskind/week12.html
4. https://github.com/guyz/interactive-arduino-programming

1.33 Python and module versions, and dates

```
try:
    import pyradi.ryutils as ryutils
    print(ryutils.VersionInformation('matplotlib,numpy,pyradi,scipy,↵
       pandas'))
except:
    print("pyradi.ryutils not found")
```

```
Software versions
Python:   3.8.3 64bit [MSC v.1916 64 bit (AMD64)]
IPython:   7.26.0
OS:   Windows 10 10.0.19041 SP0
matplotlib:   3.4.3
numpy:   1.20.3
pyradi:   1.1.4
scipy:   1.7.1
pandas:   1.3.2
Mon Aug 23 19:48:30 2021 South Africa Standard Time
```

BIBLIOGRAPHY

[1] [Online]. Available: https://github.com/NelisW/ComputationalRadiometry#computational-optical-radiometry-with-pyradi

[2] [Online]. Available: https://github.com/NelisW/ComputationalRadiometry#computational-optical-radiometry-with-pyradi

[3] [Online]. Available: http://cyrille.rossant.net/why-using-python-for-scientific-computing/

[4] [Online]. Available: http://cyrille.rossant.net/tag/ipython/

[5] [Online]. Available: https://sites.google.com/site/pythonforscientists/python-vs-matlab

[6] [Online]. Available: http://wiki.scipy.org/PerformancePython

[7] [Online]. Available: http://phillipmfeldman.org/Python/Advantages_of_Python_Over_Matlab.html

[8] [Online]. Available: http://www.stat.washington.edu/~hoytak/blog/whypython.html

[9] [Online]. Available: https://vnoel.wordpress.com/2008/05/03/bye-matlab-hello-python-thanks-sage/

[10] [Online]. Available: http://metarabbit.wordpress.com/2013/10/18/why-python-is-better-than-matlab-for-scientific-software/

[11] [Online]. Available: https://github.com/jakevdp/2013_fall_ASTR599

[12] [Online]. Available: http://www-personal.umich.edu/~kundeng/stats607/week_5_pysci-03-scipy.pdf

[13] [Online]. Available: http://nbviewer.ipython.org/github/Tooblippe/ipython_csir_30may/blob/master/csir_ipython.ipynb

[14] [Online]. Available: https://twitter.com/Sydonahi/status/991797172877381632

[15] [Online]. Available: http://shop.oreilly.com/product/0636920034919.do

[16] [Online]. Available: https://www.amazon.in/Elegant-SciPy-Juan-Nunez-iglesias/dp/1491922877

[17] [Online]. Available: http://shop.oreilly.com/product/0636920033424.do

[18] [Online]. Available: http://wesmckinney.com/pages/book.html

[19] [Online]. Available: https://leanpub.com/effective-pandas

[20] [Online]. Available: http://shop.oreilly.com/product/0636920032519.do

[21] [Online]. Available: http://www.effectivepython.com/

[22] [Online]. Available: https://sebastianraschka.com/books.html

[23] [Online]. Available: http://shop.oreilly.com/product/9780596158118.do

[24] [Online]. Available: http://shop.oreilly.com/product/9780596158071.do

[25] [Online]. Available: https://newcircle.com/bookshelf/python_fundamentals_tutorial/index

[26] [Online]. Available: http://files.swaroopch.com/python/byte_of_python.pdf

[27] [Online]. Available: http://www.diveintopython.net/

[28] [Online]. Available: http://readwrite.com/2011/03/25/python-is-an-increasingly-popu#awesm=~oyGhXjmOOEXT1e

[29] [Online]. Available: http://efytimes.com/e1/fullnews.asp?edid=117094

[30] [Online]. Available: http://folk.uio.no/hpl/scripting/book_comparison.html

[31] [Online]. Available: https://mbakker7.github.io/exploratory_computing_with_python/

[32] [Online]. Available: http://nbviewer.ipython.org/gist/rpmuller/5920182

[33] [Online]. Available: https://github.com/Junnplus/awesome-python-books

[34] [Online]. Available: http://python-3-patterns-idioms-test.readthedocs.io/en/latest/

[35] [Online]. Available: http://docs.python-guide.org/en/latest/intro/learning/

[36] [Online]. Available: https://developers.google.com/edu/python/

[37] [Online]. Available: http://www.python.org/about/gettingstarted/

[38] [Online]. Available: https://www.youtube.com/playlist?list=PLEA1FEF17E1E5C0DAvideos

[39] [Online]. Available: http://pyvideo.org/

[40] [Online]. Available: http://www.python.org/community/workshops

[41] [Online]. Available: http://docs.scipy.org/doc/numpy/numpy-ref-1.8.0.pdf

[42] [Online]. Available: http://stackoverflow.com/

[43] [Online]. Available: https://www.google.com/search?q=learning+python&ie=utf-8&oe=utf-8

[44] [Online]. Available: https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

[45] [Online]. Available: https://wiki.python.org/moin/BeginnersGuide

[46] [Online]. Available: http://www.wikihow.com/Start-Programming-in-Python

[47] [Online]. Available: http://www.learnpython.org/

[48] [Online]. Available: http://www.sthurlow.com/python/

[49] [Online]. Available: http://holdenweb.blogspot.com/2014/04/intermediate-python-open-source.html

[50] [Online]. Available: http://nbviewer.ipython.org/github/DevTeam-TheOpenBastion/int-py-notes/tree/master/nbsource/

[51] [Online]. Available: https://github.com/DevTeam-TheOpenBastion/int-py-notes/tree/master/nbsource

[52] [Online]. Available: https://www.youtube.com/watch?v=JYK_8iK2zsw

[53] [Online]. Available: http://sahandsaba.com/thirty-python-language-features-and-tricks-you-may-not-know.html

[54] [Online]. Available: http://www.scipy-lectures.org/

[55] [Online]. Available: https://github.com/scipy-lectures/scipy-lecture-notes

[56] [Online]. Available: http://folk.uio.no/hpl/scripting/book_comparison.html

[57] [Online]. Available: http://www.springer.com/la/book/9783642549588

[58] [Online]. Available: http://www.springer.com/la/book/9783540739159

[59] [Online]. Available: http://www.engr.ucsb.edu/~shell/che210d/numpy.pdf

[60] [Online]. Available: http://docs.scipy.org/doc/

[61] [Online]. Available: http://nbviewer.ipython.org/github/jrjohansson/scientific-python-lectures/tree/master/

[62] [Online]. Available: http://wiki.scipy.org/Tentative_NumPy_Tutorial

[63] [Online]. Available: http://matplotlib.org/gallery.html

[64] [Online]. Available: http://nbviewer.ipython.org/github/gumption/Python_for_Data_Science/blob/master/Python_for_Data_Science_all.ipynb

[65] [Online]. Available: http://wiki.scipy.org/NumPy_for_Matlab_Users

[66] [Online]. Available: http://hyperpolyglot.org/numerical-analysis

[67] [Online]. Available: http://resources.sei.cmu.edu/asset_files/Presentation/2011_017_001_50519.pdf

[68] [Online]. Available: http://www.pyzo.org/python_vs_matlab.html

[69] [Online]. Available: http://www.pycon.se/2014/assets/slides/Plotly-Pycon-Sweden.pdf

[70] [Online]. Available: https://pyscience.wordpress.com/2014/09/01/interactive-plotting-in-ipython-notebook-part-12-bokeh/

[71] [Online]. Available: https://pyscience.wordpress.com/2014/09/02/interactive-plotting-in-ipython-notebook-part-22-plotly-2/

[72] [Online]. Available: http://pbpython.com/visualization-tools-1.html

[73] [Online]. Available: http://nipunbatra.github.io/2014/04/comparing-python-web-visualization/

[74] [Online]. Available: http://matplotlib.org/

[75] [Online]. Available: http://matplotlib.org/gallery.html

[76] [Online]. Available: http://matplotlib.org/examples/index.html

[77] [Online]. Available: http://matplotlib.org/contents.html#

[78] [Online]. Available: http://bokeh.pydata.org/en/latest/

[79] [Online]. Available: http://bokeh.pydata.org/en/latest/docs/gallery.html

[80] [Online]. Available: http://bokeh.pydata.org/en/latest/docs/quickstart.html

[81] [Online]. Available: http://nbviewer.ipython.org/github/bokeh/bokeh-notebooks/blob/master/index.ipynb#Tutorial

[82] [Online]. Available: http://bokeh.pydata.org/en/latest/docs/user_guide.html

[83] [Online]. Available: https://plot.ly/javascript/

[84] [Online]. Available: https://plot.ly/javascript/open-source-announcement/

[85] [Online]. Available: https://plot.ly/python/getting-started/

[86] [Online]. Available: https://plot.ly/python/offline/

[87] [Online]. Available: https://plot.ly/python/user-guide/

[88] [Online]. Available: http://nbviewer.ipython.org/github/plotly/python-user-guide/blob/master/Index.ipynb

[89] [Online]. Available: https://plot.ly/~chriddyp/1780.embed

[90] [Online]. Available: http://shop.oreilly.com/product/0636920023784.do

[91] [Online]. Available: http://shop.oreilly.com/product/0636920034919.do

[92] [Online]. Available: https://leanpub.com/effective-pandas

[93] [Online]. Available: http://gregreda.com/2013/10/26/intro-to-pandas-data-structures/

[94] [Online]. Available: http://legacy.python.org/dev/peps/pep-0008/

[95] [Online]. Available: http://legacy.python.org/dev/peps/pep-0257/

[96] [Online]. Available: http://www.pythonforbeginners.com/basics/python-docstrings

[97] [Online]. Available: https://www.jeffknupp.com/blog/2012/11/13/is-python-callbyvalue-or-callbyreference-neither/

[98] [Online]. Available: https://docs.python.org/2/library/copy.html

[99] [Online]. Available: http://effbot.org/pyfaq/how-do-i-copy-an-object-in-python.htm

[100] [Online]. Available: http://pymotw.com/2/copy/

[101] [Online]. Available: http://blog.lerner.co.il/why-you-should-almost-never-use-is-in-python/

[102] [Online]. Available: http://docs.python.org/2/library/string.html#format-string-syntax

[103] [Online]. Available: https://stackoverflow.com/questions/5082452/python-string-formatting-vs-format

[104] [Online]. Available: http://pyformat.info/

[105] [Online]. Available: http://docs.python.org/2/library/collections.html

[106] [Online]. Available: http://mastering-python-lists.blogspot.co.za/p/vectorized-code.html

[107] [Online]. Available: http://carlgroner.me/Python/2011/11/09/An-Introduction-to-List-Comprehensions-in-Python.html

[108] [Online]. Available: http://www.secnetix.de/olli/Python/list_comprehensions.hawk

[109] [Online]. Available: http://docs.python.org/2.7/library/functions.html

[110] [Online]. Available: http://howchoo.com/g/how-to-use-list-comprehension-in-python

[111] [Online]. Available: http://treyhunner.com/2015/12/python-list-comprehensions-now-in-color/

[112] [Online]. Available: https://docs.python.org/2/library/collections.html#collections.namedtuple

[113] [Online]. Available: http://pymotw.com/2/collections/namedtuple.html

[114] [Online]. Available: http://stackoverflow.com/questions/2970608/what-are-named-tuples-in-python

[115] [Online]. Available: https://docs.python.org/2/library/collections.html#collections.namedtuple

[116] [Online]. Available: https://docs.python.org/2/library/collections.html#collections.namedtuple

[117] [Online]. Available: http://anandology.com/python-practice-book/iterators.html

[118] [Online]. Available: http://www.jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/

[119] [Online]. Available: https://wiki.python.org/moin/Generators

[120] [Online]. Available: http://www.bogotobogo.com/python/python_generators.php

[121] [Online]. Available: http://zetcode.com/lang/python/itergener/

[122] [Online]. Available: http://nbviewer.ipython.org/github/empet/pytwist/blob/master/Generators-and-Dynamical-Systs.ipynb

[123] [Online]. Available: https://www.jeffknupp.com/blog/2013/11/29/improve-your-python-decorators-explained/

[124] [Online]. Available: http://code.activestate.com/recipes/65126-dictionary-of-methodsfunctions/

[125] [Online]. Available: http://nbviewer.ipython.org/github/rasbt/python_reference/blob/master/not_so_obvious_python_stuff.ipynb?create=1

[126] [Online]. Available: http://effbot.org/zone/python-with-statement.htm

[127] [Online]. Available: https://docs.python.org/2/library/array.html

[128] [Online]. Available: https://docs.python.org/2/library/array.html

[129] [Online]. Available: http://chimera.labs.oreilly.com/books/1230000000393/ch05.html#_discussion_75

[130] [Online]. Available: http://nelisw.github.io/pyradi-docs/_build/html/ryplanck.html#module-pyradi.ryplanck

[131] [Online]. Available: http://nelisw.github.io/pyradi-docs/_build/html/ryplanck.html#module-pyradi.ryplanck

[132] [Online]. Available: http://docs.python.org/2/howto/urllib2.html

[133] [Online]. Available: http://nelisw.github.io/pyradi-docs/_build/html/ryfiles.html#pyradi.ryfiles.unzipGZipfile

[134] [Online]. Available: http://nelisw.github.io/pyradi-docs/_build/html/ryfiles.html#pyradi.ryfiles.untarTarfile

[135] [Online]. Available: http://nelisw.github.io/pyradi-docs/_build/html/ryfiles.html#pyradi.ryfiles.downloadFileUrl

[136] [Online]. Available: http://www.pydanny.com/why-doesnt-python-have-switch-case.html

[137] [Online]. Available: https://en.wikipedia.org/wiki/Function_object#In_Python

[138] [Online]. Available: http://www.programiz.com/python-programming/operator-overloading

[139] [Online]. Available: https://newcircle.com/bookshelf/python_fundamentals_tutorial/functional_programming

[140] [Online]. Available: https://newcircle.com/bookshelf/python_fundamentals_tutorial/functional_programming

[141] [Online]. Available: https://www.ics.uci.edu/~pattis/ICS-33/lectures/operatoroverloading1.txt

[142] [Online]. Available: https://docs.python.org/2/c-api/function.html

[143] [Online]. Available: http://python-3-patterns-idioms-test.readthedocs.io/en/latest/FunctionObjects.html

[144] [Online]. Available: https://newcircle.com/bookshelf/python_fundamentals_tutorial/functional_programming

[145] [Online]. Available: http://nbviewer.ipython.org/github/rasbt/python_reference/blob/master/not_so_obvious_python_stuff.ipynb?create=1

[146] [Online]. Available: http://sahandsaba.com/thirty-python-language-features-and-tricks-you-may-not-know.html

[147] [Online]. Available: http://jugad2.blogspot.com/2014/04/python-variables-can-have-types-as.html

[148] [Online]. Available: http://nbviewer.ipython.org/github/LUMC/programming-course/blob/master/numpy.ipynb

[149] [Online]. Available: http://cyrille.rossant.net/numpy-performance-tricks/

[150] [Online]. Available: http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html

[151] [Online]. Available: http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

[152] [Online]. Available: http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

[153] [Online]. Available: http://stackoverflow.com/questions/772124/what-does-the-python-ellipsis-object-do

[154] [Online]. Available: http://shop.oreilly.com/product/0636920030249.do

[155] [Online]. Available: http://blog.brush.co.nz/2009/05/ellipsis/

[156] [Online]. Available: https://docs.scipy.org/doc/numpy-dev/user/quickstart.html#changing-the-shape-of-an-array

[157] [Online]. Available: http://en.wikipedia.org/wiki/Poisson_distribution#Related_distributions

[158] [Online]. Available: http://nbviewer.ipython.org/github/jrjohansson/scientific-python-lectures/blob/master/Lecture-5-Sympy.ipynb

[159] [Online]. Available: http://docs.sympy.org/dev/tutorial/printing.html

[160] [Online]. Available: https://stackoverflow.com/questions/19470796/ipython-notebook-output-latex-or-mathjax

[161] [Online]. Available: https://www3.ntu.edu.sg/home/ehchua/programming/howto/Regexe.html

[162] [Online]. Available: https://www.airpair.com/python/posts/top-mistakes-python-big-data-analytics

[163] [Online]. Available: http://edcjones.tripod.com/refcount.html

[164] [Online]. Available: https://pymotw.com/2/gc/

[165] [Online]. Available: https://pymotw.com/2/sys/limits.html

[166] [Online]. Available: https://docs.python.org/2/library/sys.html#sys.getrefcount

[167] [Online]. Available: http://jeffknupp.com/blog/2016/03/07/python-with-context-managers/

[168] [Online]. Available: https://pyserial.readthedocs.org/en/latest/

[169] [Online]. Available: http://www.varesano.net/blog/fabio/serial%20rs232%20connections%20python>

[170] [Online]. Available: http://lucsmall.com/2015/02/23/homebrew-esp8266-breakout/

[171] [Online]. Available: http://petrimaki.com/2013/04/28/reading-arduino-serial-ports-in-windows-7/

[172] [Online]. Available: https://www.lebsanft.org/?p=48

[173] [Online]. Available: https://github.com/mba7/SerialPort-RealTime-Data-Plotter