Python programming guide for Earth Scientists

Maarten J. Waterloo and Vincent E.A. Post

Amsterdam Critical Zone Hydrology group
September 2015

Acacia Water
Solutions in Groundwater

Cover page: Meteorological tower in an abandoned agricultural field near Vagos, North Portugal, June 2008. Meteorological measurements are done for estimating evapotranspiration rates.

Contents

Chapter 1

Why use Python?

Students normally do most their calculations in spreadsheet programs such as MS Excel or, if you do not have an MS Office license, the alternative open source spreadsheet program *Calc* from Sun's *OpenOffice* suite or the The Document Foundation's LibreOffice suite. Working with spreadsheets is easy to learn and it works well when you deal with few data. However, suppose that you have long time series of meteorological data, or 10 Hz data wind/temperature from a Sonic anemometer (about 12 MB of data generated per day). In a spreadsheet program the maximum column length would then be reached very quickly so that you would have to divide your data over different sheets. This would be very inefficient. It is also complicated to build a complex computer model in a spreadsheet to simulate hydrological and micro-meteorological processes, such as groundwater flow, erosion, or release of greenhouse gases. Another issue is that when you get additional data, you will have to start copying all cells containing equations and you have to manually modify graphs that you made earlier.

Because of these limitations, professionals revert to writing dedicated computer programs or scripts that can do the processing of data, do model calculations automatically and generate updated graphs. The most common *compiled languages* for programming are the *C* programming language and its derivatives *C++* or *C#*, *Java* and *Fortran*, to name just a few. These languages are very powerful but have the disadvantage that commands are not directly evaluated. Instead, the following procedure has to be followed:

1. you have to write the program,
2. when you finish this, you need to compile it to create an executable,
3. you can then execute the program to get your results, but you may find some bugs and have to start over again.

The advantages of compiled languages are that they very efficient in coding, have little overhead and are therefore fast in execution. A disadvantage is that if you do not have the program source code, porting the program to a different operating system is difficult.

Another branch of programming languages are the *interpreted languages*. Interpreted programming languages are designed to be processed by an application that translates your plain text instructions into actions inside the target application. Here, your commands can be typed in and are directly evaluated by the interpreter when you press the ENTER key, thus without the need for prior compilation. A number of consecutive commands can be saved in an ASCII text file, which is then called a script. Examples of interpreted languages are Matlab, Python and R. This also means that as long as you have any of these programs installed on your computer, you can run the script independent of the operating system.

A Matlab license is quite costly and you may not have Matlab available at home. A good alternative is the open source Python language. Python is a very versatile open-source programming language that can be used to process data in a rather easy way. This document aims to teach you basic Python programming and give you hints and examples on how to use Python in your scientific career.

Chapter 2

Basics of Python

2.1 Starting up Python

There are several distributions of Python that install the most frequently used packages for you on a Microsoft Windows operating system machine. We use the *Python(x,y)* distribution (http://www.pythonxy.com). If you want to install Python on your computer, please visit http://python.hydrology-amsterdam.nl for download and installation instructions. Alternatively, you can download and use the *Anaconda* (https://store.continuum.io/cshop/anaconda/) or Enthought Python (http://www.enthought.com) editions. NOTE: if you intend to use equations in your graphs, you will need to install $\LaTeX$ (see http://thesistools.hydrology-amsterdam.nl/) on your computer as well.

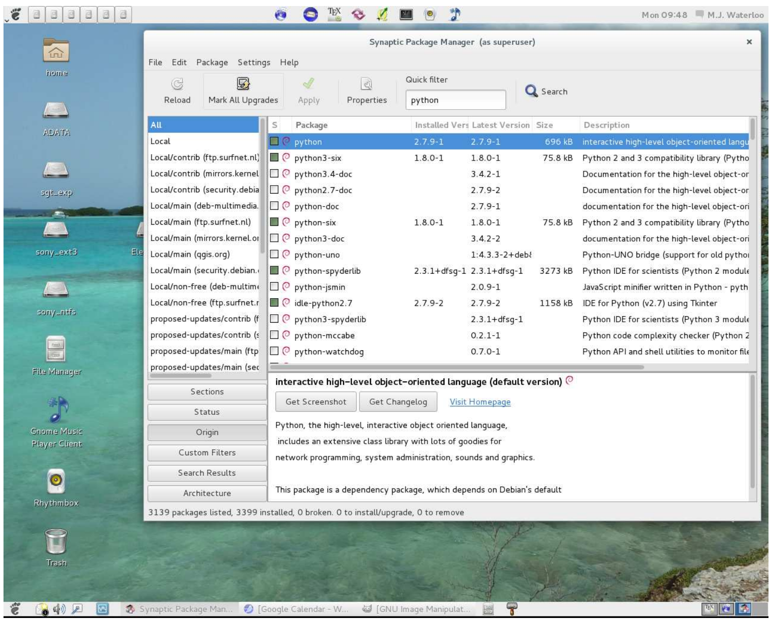
If you use one of the Linux operating system distributions, such as my preferred Debian distribution (https://www.debian.org/), Python comes pre-installed but you may need to install additional packages such as python-matplotlib, python-pylab or python-scipy. You can use *Synaptic* (as superuser, Figure 2.1) to see which python packages are available and to install these.

There are various ways to interface and program with Python and three of these are discussed below, appearing in order of increased complexity, flexibility and ease of use.

2.1.1 Python through a text terminal

The most simple method to work with Python – that is if you are familiar with Python already – is through a *text terminal* window. A text terminal, or *text console*, or terminal for short, is an interface to the computer that allows you to enter text commands on the *command line* and displays output from the computer. In Windows you open a command-line interpreter window by executing *cmd.exe*.

If you start up Python (or the more sophisticated IPython command shell window) from a terminal by entering *Python* (or *ipython*) at the terminal command-line prompt, Python will start up and you will see some text identifying the version of Python that is installed and the Python prompt (>>>), as shown in Figure 2.2.


Synaptic Package Manager (as superuser)
File Edit Package Settings Help
Reload
Mark All Upgrades
Apply
Properties
Quick filter
python
Search
All
Local
Local/contrib (ftp.surfnet.nl)
Local/contrib (mirrors.kernel
Local/contrib (security.debia
Local/main (deb-multimedia.
Local/main (ftp.surfnet.nl)
Local/main (mirrors.kernel.o
Local/main (qgis.org)
Local/main (security.debian.
Local/non-free (deb-multime
Local/non-free (ftp.surfnet.r
proposed-updates/contrib (f
proposed-updates/contrib (s
proposed-updates/main (ftp
proposed-updates/main (sec
Sections
Status
Origin
Custom Filters
Search Results
Architecture
Get Screenshot
Get Changelog
Visit Homepage

S	Package	Installed Vers	Latest Version	Size	Description
🟩	python	2.7.9-1	2.7.9-1	696 kB	interactive high-level object-oriented langu
🟩	python3-six	1.8.0-1	1.8.0-1	75.8 kB	Python 2 and 3 compatibility library (Pytho
☐	python3.4-doc		3.4.2-1		Documentation for the high-level object-or
☐	python2.7-doc		2.7.9-2		Documentation for the high-level object-or
☐	python-doc		2.7.9-1		documentation for the high-level object-ori
🟩	python-six	1.8.0-1	1.8.0-1	75.8 kB	Python 2 and 3 compatibility library (Pytho
☐	python3-doc		3.4.2-2		documentation for the high-level object-ori
☐	python-uno		1:4.3.3-2+deb8		Python-UNO bridge (support for old pytho
🟩	python-spyderlib	2.3.1+dfsg-1	2.3.1+dfsg-1	3273 kB	Python IDE for scientists (Python 2 module
☐	python-jsmin		2.0.9-1		JavaScript minifier written in Python - pyth
🟩	idle-python2.7	2.7.9-2	2.7.9-2	1158 kB	IDE for Python (v2.7) using Tkinter
☐	python3-spyderlib		2.3.1+dfsg-1		Python IDE for scientists (Python 3 module
☐	python-mccabe		0.2.1-1		Python code complexity checker (Python 2
☐	python-watchdog		0.7.0-1		Python API and shell utilities to monitor file

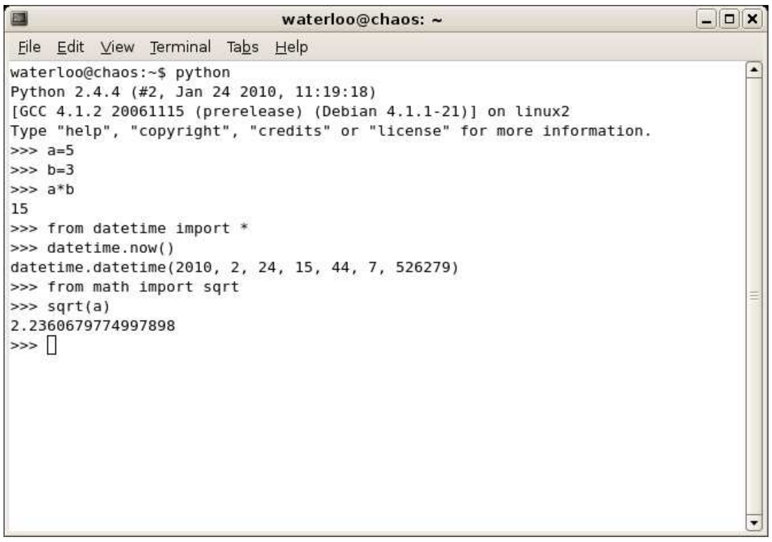
interactive high-level object-oriented language (default version)

Python, the high-level, interactive object oriented language,
includes an extensive class library with lots of goodies for
network programming, system administration, sounds and graphics.

This package is a dependency package, which depends on Debian's default

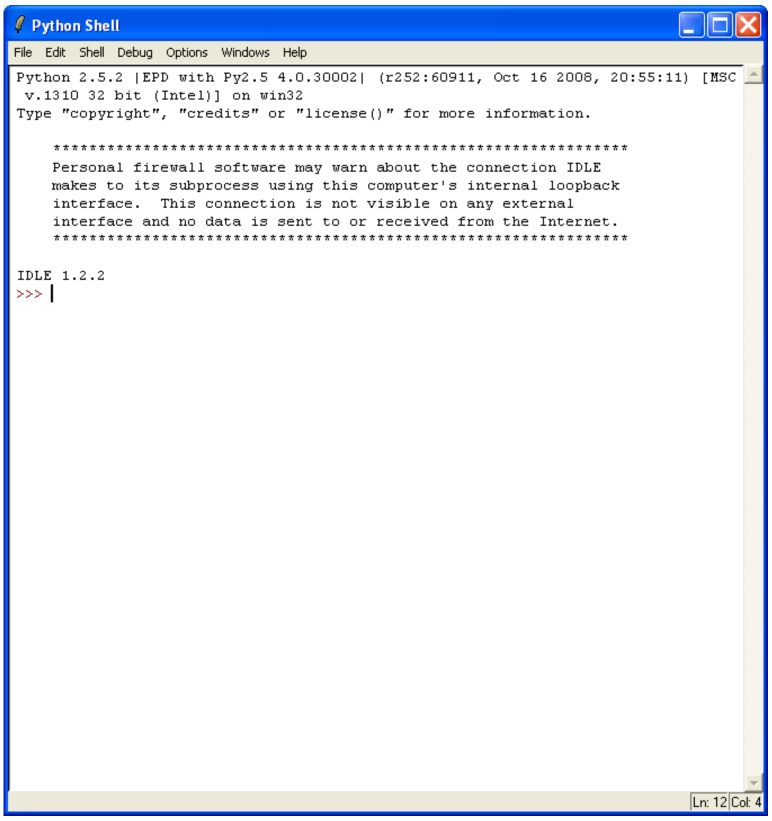
3139 packages listed, 3399 installed, 0 broken. 0 to install/upgrade, 0 to remove

Figure 2.1: Installing Python packages on a Debian OS distribution using the Synaptic package manager.



```
waterloo@chaos:~$ python
Python 2.4.4 (#2, Jan 24 2010, 11:19:18)
[GCC 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a=5
>>> b=3
>>> a*b
15
>>> from datetime import *
>>> datetime.now()
datetime.datetime(2010, 2, 24, 15, 44, 7, 526279)
>>> from math import sqrt
>>> sqrt(a)
2.2360679774997898
>>> 
```

Figure 2.2: Running Python from within a Debian Linux terminal window. Python has started up and several commands have been entered at the Python prompt (>>>).



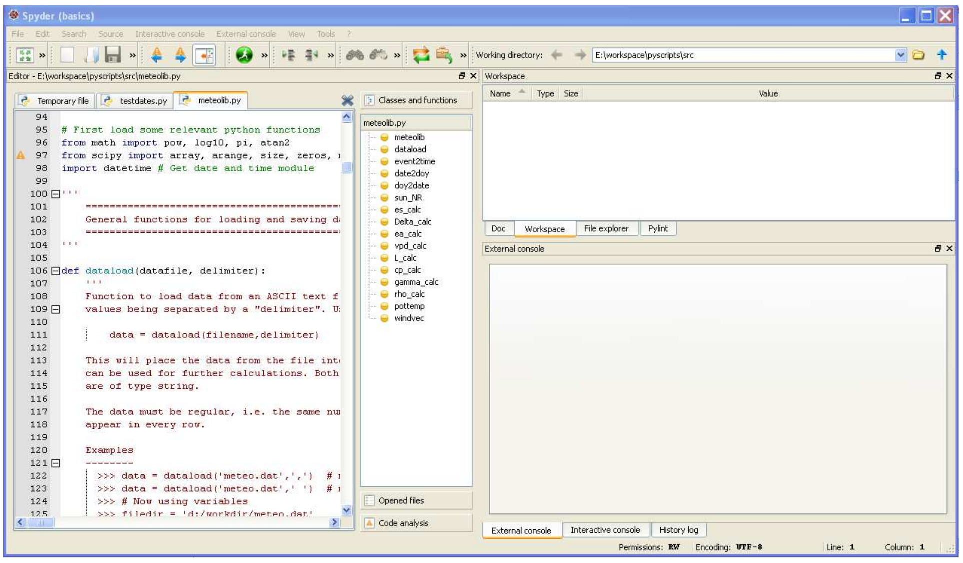
```
Python 2.5.2 |EPD with Py2.5 4.0.30002| (r252:60911, Oct 16 2008, 20:55:11) [MSC
 v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2.2
>>> 
```

Figure 2.3: IDLE shell window for working with Python.

The interfacing with Python is done in plain ASCII (American Standard Code for Information Interchange) text that is typed in. If you use a Python terminal, a separate *ASCII text editor*, such as the fancy *Xemacs* editor or the simple *Notepad* or *Wordpad* text editors can be used to store your commands for later re-use.

2.1.2 IDLE integrated development environment

An easier way to interface with Python than through a terminal window is by IDLE. IDLE is an Integrated Development Environment (IDE) for Python. This means that it incorporates an editor for Python scripts and a command window with menus for Python operations, such as running scripts or debugging (*i.e.* identifying errors in your script). You can start Idle up from the start menu through Python(x,y), or by typing ”idle” in a terminal command window. You will then see a *shell* window similar to that in Figure 2.3. The >>> sign that you see at the start of the last line in Figure 2.3 is called the *prompt* of the *command line* and indicates that Python is ready to receive commands. In the following sections a short explanation of Python programming is presented and we encourage you to try the commands out on your computer.


Spyder (basics)
File Edit Search Source Interactive console External console View Tools ?
Working directory: E:\workspace\pyscripts\src
Editor - E:\workspace\pyscripts\src\meteolib.py
Temporary file
testdates.py
meteolib.py
Classes and functions
meteolib.py
meteolib
dataload
event2time
date2doy
doy2date
sun_NR
es_calc
Delta_calc
ea_calc
vpd_calc
L_calc
cp_calc
gamma_calc
rho_calc
pottemp
windvec
Opened files
Code analysis
Workspace
Name Type Size Value
Doc Workspace File explorer Pylint
External console
External console Interactive console History log
Permissions: RW Encoding: UTF-8 Line: 1 Column: 1

```

# First load some relevant python functions
from math import pow, log10, pi, atan2
from scipy import array, arange, size, zeros, 1
import datetime # Get date and time module

'''
    =============================================
    General functions for loading and saving d
    =============================================
'''

def dataload(datafile, delimiter):
    '''
    Function to load data from an ASCII text f
    values being separated by a "delimiter". U:

        data = dataload(filename,delimiter)

    This will place the data from the file int
    can be used for further calculations. Both
    are of type string.

    The data must be regular, i.e. the same nu
    appear in every row.

    Examples
    --------
     >>> data = dataload('meteo.dat',',')   # 1
     >>> data = dataload('meteo.dat',' ')   # 1
     >>> # Now using variables
     >>> filedir = 'd:/workdir/meteo.dat'
```

Figure 2.4: Spyder integrated development environment for working with Python.

2.1.3 SPYDER integrated development environment

A more elaborate, Matlab-type of integrated development environment with advanced editing, interactive testing and debugging capabilities is provided in Python(x,y) and is called *Spyder*. Spyder opens a single window (Figure 2.4) with various menus and several screens for script editing, command interfacing, displaying your variables, functions, etc. and a *help function* search field. In most cases, you may prefer using Spyder, rather than the more austere methods discussed above, to use Python.

2.2 Working with Python

When you started up Python you can enter commands in the shell command window and look at the output. We will be storing commands sequentially in a text file, which is then called a Python script. You can find more about scripts in Chapter 3.

2.2.1 Commenting your code

If you insert a command in the *Python shell window* (Figure 2.3), you are writing a single line of *program code*. Often, you will need to type in several lines of commands, or *statements* to get the result that you need. Commenting your program, or code, is extremely important as it allows you, or somebody else who has to work with your code, to better understand what your code does. It is therefore very important to make a habit of explaining and commenting everything that you do in your code!

There are two ways to include comments in program code. A one line comment can be made using # at the beginning of the line, whereas a multi-line comment is made using matching pairs of three apostrophes (”' This is a multi-line comment ”').

```
>>> # This is a one line comment
>>> x = 3     # We assign a value of 3 to x
```

Multi-line comments are made by placing text between three single quotes (”') in a row:

```
>>> ''' This is a multiline comment
       that spans
       3 lines'''
' This is a multiline comment\n that spans\n 3 lines'
>>>
```

Assignment

Write two comment lines in the shell window. The first is a multi-line comment and contains a text explaining the purpose of your script (make something up) and the second line is a single line comment with the date and your name. Now open a new text file in IDLE (open > new window) and copy these lines into the text file and then save the file as *yourname.py*. This is now your *script* file. You can execute or run this file by selecting the Run > run module commands that are displayed in the top menu of your script file in Idle (see Figure 2.3), or pressing the ”run” button in Spyder. Note that running the file with comments does not produce any output in the shell window.

2.2.2 Operators

Python has *operators* that allow you to do basic calculations or work otherwise with variables numbers or variables. The most well-known ones are the standard mathematical operators (*e.g.* +, -, *, /), the ** power function operator, the % modulus (returns remainder of division) and the \\ floor operators (returns division result without decimal part).

Other operators are for comparison of values, such as == (equal to), != (not equal to), logical operators such as *and*) and *or*, bitwise operators (*e.g.* the binary & and | (or) operators, or membership operators (*in*, *not in*).

2.2.3 Using Python as a calculator

Python can be used as a very fancy calculator. You can do calculations with values or with variables containing data. For instance, you could make a division by typing:

```
>>> 3.0/4.0
0.75
>>>
```

We can also use variables in our calculations:

```
>>> a=3.0
>>> b=4.0
>>> a/b
0.75
>>>
```

Note that if we want to use scientific or mathematical functions, such as cos(), sin() or sqrt(), we get an error message telling us that these functions are not defined. In order to use these functions we need to load a module containing these functions first (see Section 2.2.4 below).

```
>>> cos(30)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'cos' is not defined
>>>
```

Assignment

Using Python as a calculator, try to calculate the following in Python. Take a good look at the output and keep the answers in mind while you read the next sections.

```
sin(3.14) # take the sine of 3.14
2**4 # Fourth power of 2
2.**4 # Fourth power of 2.0
3+4
3.0-4
3.0/4
3/4.0
3./4
3/4
```

2.2.4 Use of modules to load functions

You will have noticed from the assignment above that Python does know how to make a division, but does not know the command sin(x. Several of the basic mathematical functions are defined in Python. However, more advanced functions have often been defined in special *modules* that serve as *function libraries*. These modules are normal Python .py files that contain many function definitions and can be opened in your editor to see the contents. Popular modules are the *math* with mathematical functions, the *scipy* module with scientific Python functions, the *datetime* module that contains date and time functions and the *Pylab* module with the *matplotlib* library of functions for plotting of data. To use a function from such a module, you have to import the module first. There are several ways to import a module or a function in a module using the *import* and *from–import* statements:

1. `import X`: imports module X with all its functions and creates a reference to it in the current namespace. You can now use function `f`unc(), defined in module X, by calling `X.func()`. Functions with the same name can be defined in different libraries. For instance, the function *sin()* is defined in the module *math*, but also in the SCIentific PYthon *scipy* module. Using *import math* and *import scipy* would give you two options for the same function.This is the preferred way of importing functions as you always know for sure which implementation of the function you use. As an example for the sin(x) function, with x in radians, as defined in the math and scipy modules we see:

   ```
   >>> import math
   >>> import scipy
   >>> math.sin(2)
   0.90929742682568171
   >>> scipy.sin(2)
   0.90929742682568171
   ```

2. `from X import *`: imports module X, and creates references in the current name space to all public objects (*i.e.* functions) defined in the module, but not to objects starting with _ defined in the module. You can now use the name of the function, without the module name (which is not defined), to call the function but the `X.func()` command will not work here:

   ```
   >>> from math import *
   >>> cos(0.75)
   0.7316888688738209
   >>> math.cos(0.75)
   Traceback (most recent call last):
   NameError: name 'math' is not defined
   ```

   Note that if func() was already defined in another module imported earlier (for instance the scipy module), it will be overwritten and the latest imported version will be used. This may create some confusion as two different functions with the same name could potentially be available in your program.

3. `from X import func1, func2, func3`: This imports module X and only provides references to func1, func2 and func3 to the user, but not to the other functions defined in module X.

```
>>> from math import sin, cos, sqrt
>>> tan(0.5)
 Traceback (most recent call last):
 NameError: name 'tan' is not defined
>>> from math import tan
>>> tan(0.5)
 0.54630248984379048
```

4. `Y = __import__('X')`: this is like using `import X` but you can pass the module name X to an arbitrary string (called `Y` here) and assign the module to variable `Y`.

```
>>> N = __import__('numpy')
>>> N.cos(0.75)
 0.7316888688738209
```

As said before, the preferred way is to use the first option (`import X`) as the other options may create confusion, for instance when functions with the same name (cos, sin) are defined in multiple modules (defined in the modules math, numpy – numerical python module, scipy, etc.).

Two function libraries (modules) have been developed by M.J. Waterloo for use in Hydrology and micro-meteorology. These are the *meteolib.py* module with standard meteorological functions and the *evaplib.py* module that has several functions for calculating evaporation. See Section 5.3 for more on these modules.

Assignments

1. Import all functions from the the *scipy* module using method 2, then calculate the square root (sqrt() function) of 2
2. Import the *math* module using method 1 and calculate the square root of 2.
3. Import only the function sqrt() from the *numpy* module and calculate the square root of 2
4. Use method 4 to import the *math* module as M and calculate the square root of 2

2.2.5 Variables and data types

Variables can be used to store data, such as numbers (1, 20.4, etc.), character strings ('this is a text string') or date information ('2002-03-11', 2002, etc.). Variable names consist of upper or lower case characters and numbers. Names cannot contain spaces, commas, +, -, *, / and some other special characters. It is good practice to use variable names that are recognisable, for instance a, b, x, y, z_zero, alpha, LAI, Lambda_E, dummy_1, dummy_2, etc. are proper variable names. The lower case characters i, j, n and m are by convention used in counters. At the prompt we can assign values to variables a and b by typing the following commands:

```
>>> 3 * 5
15
>>> a = 5
>>> a
5
>>>
b = 3
>>> b
3
>>> a * b
15
>>>
```

We can look up the value of a by simply typing a at the prompt, as is done in the above. We can also do calculations with these variables and use Python as a calculator:

```
>>> c = a * b
>>> c
15
>>>d = a / b
>>>d
1
>>>
```

In the assignment in Section 2.2.3 you have calculates 3/4 and may have noticed that you got a value of 0 as the answer, rather than 0.75. Note again above that the division of a by b gives a value of 1, instead of 1.66666666. This has to do with how Python treats values. In the first case, the values 3 and 4 are both *integers*, and the resulting division is therefore also an integer (0). If either one of these values is a real value (*float*, 3., 3.0, 4., 4.0), then the division of 3.0 by 4.0 will also report a float (0.75). The *type* of value (integer, float, etc.) therefore is important for the outcome of your calculations!

The same holds for the variable types. We again defined a and b as *integer* data types by assigning them integer values (3, 5), instead of real or *float* data types (3.0, 5.0). Multiplying or dividing two integer values with each other will return an integer, but when one of these values is a float, the result of the calculation will also be a float. We can define a or b as floats by adding a decimal point to the value using statements:

```
>>> a = 5.0
>>> a
5.0
>>>d = a / b
```

```
>>>d
1.6666666666666667
>>>
```

Alternatively, you can convert an integer to a float or *vice versa* using the *float()* and *int()* functions:

```
>>> float(b)
3.0
>>>int(a)
5
>>>
```

Integer and float types are two of the simple data types defined in Python. Other simple data types dare the boolean *bool*, *i.e.* true or false) and *complex* (1+0j) data types. The above variables only held one value and were therefore assigned simple data types. You can see what type a variable is using the *type()* function:

```
>>> type(b)
<type 'float'>
>>>
```

To handle complex data, such as variables holding multiple values and/or text, Python also know a number of complex data types called *tuple*, *string*, *unicode*, *list*, *set*, *frozenset* and *dictionary*. The tuple can hold a mix of values and text that are separated by commas, and these values cannot be changed:

```
>>> test1 = (1, 2.0, 5, (3+2j), 'the dog is on the roof')
>>> test1
(1, 2.0, 5, (3+2j), 'the dog is on the roof')
>>> type(test1)
<type 'tuple'>
>>>
```

The string type (defined as a ”string of characters”) only holds text:

```
>>> test2 = 'the dog is on the roof'
>>> test2
'the dog is on the roof'
>>> type(test2)
<type 'str'>
>>>
```

Another useful data type in Python is the *dictionary* type. This data type consists of unordered key:value pairs that can for instance be used to use a kwargs dictionary variable to pass arguments to a function. Dictionary values can be of any type and can be duplicated in a dictionary, but the keys have to be unique and can consist of strings, numbers or tuples. A dictionary can be made using the *dict()* statement, for instance the following could be used to pass different arguments to the genfromtxt() function to import data from a file (Section 7.2.1):

```
kwargs = dict(delimiter=",",\
              missing_values={"\"NAN\""},\
              skip_header=5,\
              skip_footer=0,\
              filling_values={-9999},\
              converters = {0:strpdate2num('\"%Y-%m-%d %H:%M:%S\"'
                  )}\
              )
```

Another way is to make the dictionary using key:value pairs enclosed by a matching pair of {}:

```
musicrating={'John Denver':3.0,\
         'Frank Zappa':9.0,\
         'Jimi Hendrix':9.5,\
         'Janis Joplin':8.0,\
         'Lady Gaga':6.0,\
         'Rumer':6.5\
         }
>>>musicrating
{'Jimi Hendrix': 9.5, 'Rumer': 6.5, 'Lady Gaga': 6.0, 'Janis
    Joplin': 8.0, 'Frank Zappa': 9.0, 'John Denver': 3.0}
>>> type(musicrating)
<type 'dict'>
>>>
```

2.2.6 Formatting of numbers using string format conversion specifiers

We often do calculations with numbers. For instance, we have measured water level H and want to use a rating curve equation (Q=CH^x) to calculate discharge Q, where C and x are the rating curve coefficients. Suppose that $H = 0.35$ m, $C = 0.357$ and $x = 1.327$, we can then calculate Q:

```
>>> H=0.35
>>> C=0.357
>>> x=1.327
>>> Q=C*H**x
>>> print(Q)
0.088643421532
```

Table 2.1: String format conversion specifiers

Specifier	Output format
d	Signed integer decimal
i	Signed integer decimal
o	Unsigned octal
u	Unsigned decimal (obsolete)
x	Unsigned hexadecimal (lowercase)
X	Unsigned hexadecimal (uppercase)
e	Floating point exponential format (lowercase)
E	Floating point exponential format (uppercase)
f	Floating point decimal format
F	Floating point decimal format
g	Same as ”e” if exponent is greater than -4 or less than precision, ”f” otherwise
G	Same as ”E” if exponent is greater than -4 or less than precision, ”F” otherwise
c	Single character (accepts integer or single character string)
r	String (converts any Python object using repr()[1])
s	String (converts any Python object using str())
%	No argument is converted, results in a '%' character in the result

You will note that when printing, we get a lot of decimals for the calculated Q. There are several ways to control the *output format*. To format a number for printing or saving to file, we convert it to a string and we can use string format conversion specifiers to choose between decimal notation, scientific notation, etc., and also control how many decimals we want to print. In other ways, there are controls to format the printing of a number. We do this by converting the number into a *string* using the *str()* command. There are several *format specifiers* that control the output as shown in Table 2.1. Several examples for formatting of the value of Q are given below. The general syntax is the format specifier (for instance '%.4g' for general with four decimals) followed by a % operator and the variable Q:

```
>>> print(str('%.4g' % Q))
0.08864
>>> print(str('%.2f' % Q))
0.09
>>> print(str('%.4f' % Q))
0.0886
>>> print(str('%.2e' % Q))
8.86e-02
>>> print(str('%.2E' % Q))
8.86E-02
>>> print(str('%.4f' % Q))
0.0886
>>> print(str('%.4i' % Q))
```

[1]repr(object) returns a string containing a printable representation of an object, i.e. the same value yielded by conversions (surrounded by reverse quotes)

```
0000
>>> str('%0.3f' % Q)
'0.089'
>>> str('%10.3f' % Q)
'     0.089'
```

The string formatting operator, % (the percent sign) uses two inputs, i.e. the string with formatting characters '%.2E' and the data Q to be formatted, separated by the percent sign, thus '%.4i' **%** Q. The number before the dot (.) in the string with formatting characters specifies the total number of characters to be used, whereas the number following the dot in the string with formatting characters specifies the number of decimals (e.g. %5.1f would give a value of five characters (can be filled by spaces for values lower than 1000) with 1 decimal '1234.5' or ' 34.5'). Note that the conversion to integer '(%.4i' %) of this float gives four zeros due to the .4 in the format specifier (%.4i). Note that the str() statement changes the type of the variable to string (str) type, in this case from *float* to *str* as shown below.

```
>>> type(Q)
<type 'float'>
>>> type(str('%.2f' % Q))
<type 'str'>
>>>
```

We can also use the str() statement end-format specifiers to format two or more numbers at a time:

```
>>> print(str('%0.4f \%.2e'  % (Q,H)))
0.0886 3.50e-01
```

2.2.7 Arrays

Most of our hydrological and meteorological data consist of measurements made over time, such as water level data, sapflow data, conductivity measurements, etc. With meteorological data collected by our Campbell 21X datalogger, we have one row of data, such as radiation, temperature, relative humidity, air pressure, *etc* at every time interval. In most cases individual data values will be of type *float*, for instance a temperature of 27.82 °C, or net radiation of 256.9 W m^{-2}. The various rows and columns of data can be considered an *array* or *matrix*, where every row represents a line of data collected at a certain date and time. This means that this array would be of the *tuple* type.

Each value in an array can be accessed by its index numbers. Say that we define the array t_data as an array of 5 rows and 6 columns containing year, day, time, temperature and temperature variance and standard deviation data:

```
>>> # Import the scipy module to work with arrays
>>> from scipy import *
>>> # Now define the array t_data
>>> t_data = array([[2007,121,1435,22.95,.287,.536],
                    [2007,121,1440,22.45,.188,.434],
                    [2007,121,1445,21.84,.207,.455],
                    [2007,121,1450,21.90,.265,.514],
                    [2007,121,1455,22.34,.476,.69]])
>>> t_data
array([[  2.00700000e+03,   1.21000000e+02,   1.43500000e+03,
          2.29500000e+01,   2.87000000e-01,   5.36000000e-01],
       [  2.00700000e+03,   1.21000000e+02,   1.44000000e+03,
          2.24500000e+01,   1.88000000e-01,   4.34000000e-01],
       [  2.00700000e+03,   1.21000000e+02,   1.44500000e+03,
          2.18400000e+01,   2.07000000e-01,   4.55000000e-01],
       [  2.00700000e+03,   1.21000000e+02,   1.45000000e+03,
          2.19000000e+01,   2.65000000e-01,   5.14000000e-01],
       [  2.00700000e+03,   1.21000000e+02,   1.45500000e+03,
          2.23400000e+01,   4.76000000e-01,   6.90000000e-01]])
>>>
```

Each value in a two-dimensional array is identified by indices [x,y], where x represents the row number and y the column number. The first row and column have indexes of zero (0). Hence, the upper left value (2007) is t_data[0,0], whereas the lower right value (row 5, column 6, value = 0.69) is t_data[4,5].

```
>>> t_data[0,0]    # first value in array
2007.0
>>> t_data[4,5]    # last value in array
0.69
```

We can select an entire column or row of data using the colon (:). Say that we want to assign the fourth column with temperature data to the variable T, then we would use the statement t = t_data[:,3].

```
>>> t_data[:,3]        # fourth column in array
array([ 22.95,  22.45,  21.84,  21.9 ,  22.34])
>>> t_data[1,:]       # Second row in array
array([  2.00700000e+03,   1.21000000e+02,   1.44000000e+03,
         2.24500000e+01,   1.88000000e-01,   4.34000000e-01])
>>> T = t_data[:,3]   # Put temperature values in variable T
```

If we use a minus (-) sign in the index, we start counting from the final rows or columns of the array. As such t_data[-1,-1] gives us the last, lower right value, in the array (0.69) and t_data[:,-2] gives us the column before the last one in the array.

```
 >>> t_data[:,-1]
array([ 0.536,  0.434,  0.455,  0.514,  0.69 ])
>>> t_data[:,-2]
```

```
array([ 0.287,  0.188,  0.207,  0.265,  0.476])
```

We can also place a colon (:) before an array index to make more selective choices. If we want only the first three columns of an array, we can use t_data[:,:3] to select these:

```
 >>> t_data[:,:3]
array([[ 2007.,   121.,  1435.],
       [ 2007.,   121.,  1440.],
       [ 2007.,   121.,  1445.],
       [ 2007.,   121.,  1450.],
       [ 2007.,   121.,  1455.]])
>>>
```

Using a negative sign will provide everything except the columns/rows counted from the end of the array. With t_data having six columns, we could select only the first column by:

```
>>> t_data[:,:-5]
array([[ 2007.],
       [ 2007.],
       [ 2007.],
       [ 2007.],
       [ 2007.]])
>>>
```

Finally, we could also select everything except the first and last columns by:

```
>>> t_data[:,1:-1]
array([[ 2007.],
       [ 2007.],
       [ 2007.],
       [ 2007.],
       [ 2007.]])
>>>
```

These tricks are of good use when providing exact index values is difficult because the size of the array is not known, or changes during program execution.

2.2.8 Combining or splitting arrays of data

It is sometimes necessary to combine two arrays in a single array, for instance if you have a 1-dimensional array with soil depth, and another array with soil chemical data. The merging of two arrays is called *stacking* and can be done using the *hstack* or *vstack()* instructions.

For example, if we wish to add one row to an existing matrix, *i.e.* stack the matrices vertically or row on row, we can use *vertical stack* with the vstack() statement. All arrays in the sequence must have the same shape along all but the first axis.

The vstack() or hstack() statements take exactly one argument called *tup*, which contains the matrices to be stacked. For example, if we have two matrices a and b of 3 columns, we can stack them as follows with the argument being (a,b):

```
>>> from scipy import *
>>> a = array([0,0,0])
>>> b = array([[1,1,1],[2,2,2]])
>>> c = vstack((a,b))
>>> c
array([[0,  0,  0],
       [1,  1,  1],
       [2,  2,  2]])
```

Splitting a data array is also possible, using *vsplit()* or *hsplit()*. These commands have two arguments, being the array to be slit and the indices or sections where the split is desired.

You can use vstack() and hstack() statements to rebuild arrays divided by *vsplit()* and *hsplit()*, respectively.

Assignment

Make three variables a, b and c with respective values of 10.3, 5 and 137.289. Now calculate:

- a+b+c
- a/b-c
- (b+c)/a
- 2*b+5.1/c+a*15.9
- $\sqrt{a}$
- $\log b$

NOTE: The log and square root functions are defined in the *math* module or the *scipy* (*sci*entific *py*thon). To access the sqrt() and log() functions you need to load one of these modules first by executing the following command in IDLE:

```
>>> from scipy import *
>>>
```

Store these commands in the script file that you made earlier. If you run the script file, no output will be produced. You can produce output using the *print* command by specifying

```
print a+b+c
print sqrt(a)
```

in your script. Please try this for the above calculations...

Now define a variable called dummy that contains an array with two rows of three values each. The first row is 2.0, 5.6, 3.13 and the second row 17.3, 25.8, 1.6.

- What answers do you get when you type dummy[0,2], dummy[:,0] and dummy[1,:] in the shell window?
- Which column is selected if you do dummy[:,-1]?
- What data do you select with dummy[:-1,:]?
- What data do you select with dummy[:,:-2]?
- What data do you select with dummy[1:-1,:]?
- Multiply the first column of dummy by 3.0.
- Make a new variable dummy2 that holds the second row of dummy divided by 2.0

Chapter 3

Python scripts

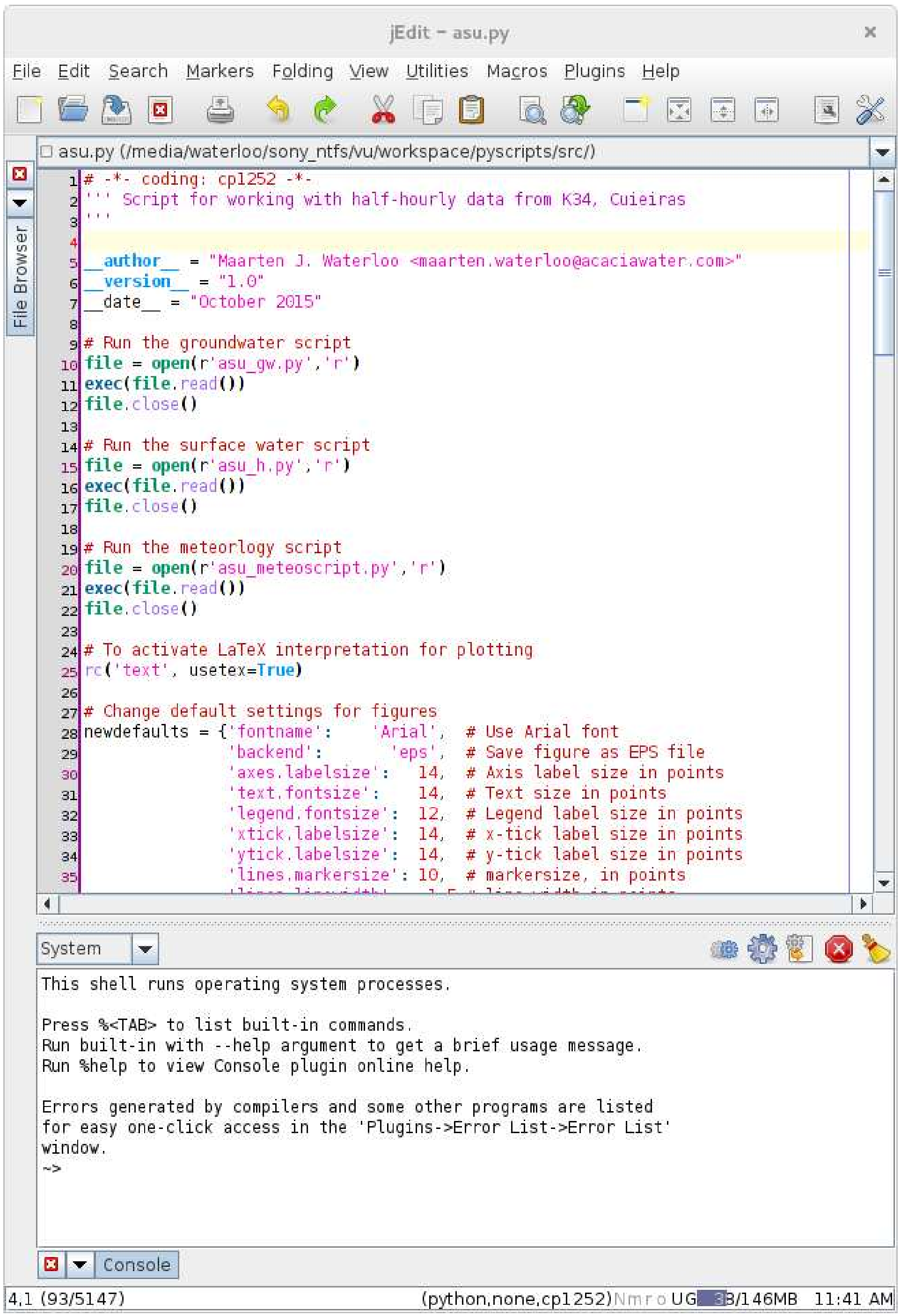
3.1 What is a script?

Using the command line in Idle to execute statements works fine if we only have a few commands to give, and we do not have to repeat these often. However, for some processing tasks, we have many commands that we want to issue in sequence. In addition, we might like to repeat our calculations on different data at a later time, or have a web application processing data for publishing on-line. When we start up IDLE, we open a *shell window* (Figure reffig:idle), in which we see a >>> prompt that Idle is ready to receive commands. To make things easier, we could also collect a sequence of these same commands in a text file, which is then called a *script* file. Such a script file serves as a kind of program and always has a name with a *.py* extension by convention.

Python scripts are just text files and can therefore be created in any text editor. However, programming is made easier if the editor recognises Python statements, does automatic indenting and colourises text, such as comments or flow control text, for clarity. Here we shall use the rather simple built-in Idle editor window, but an example of using the very versatile *JEdit programmer's text editor* (http://www.jedit.org) for editting script files for different programming languages is shown in Figure 3.1. To open an editor window in IDLE, choose File > New. An editor window similar to that in Figure 3.2 will then open.

3.2 Example script

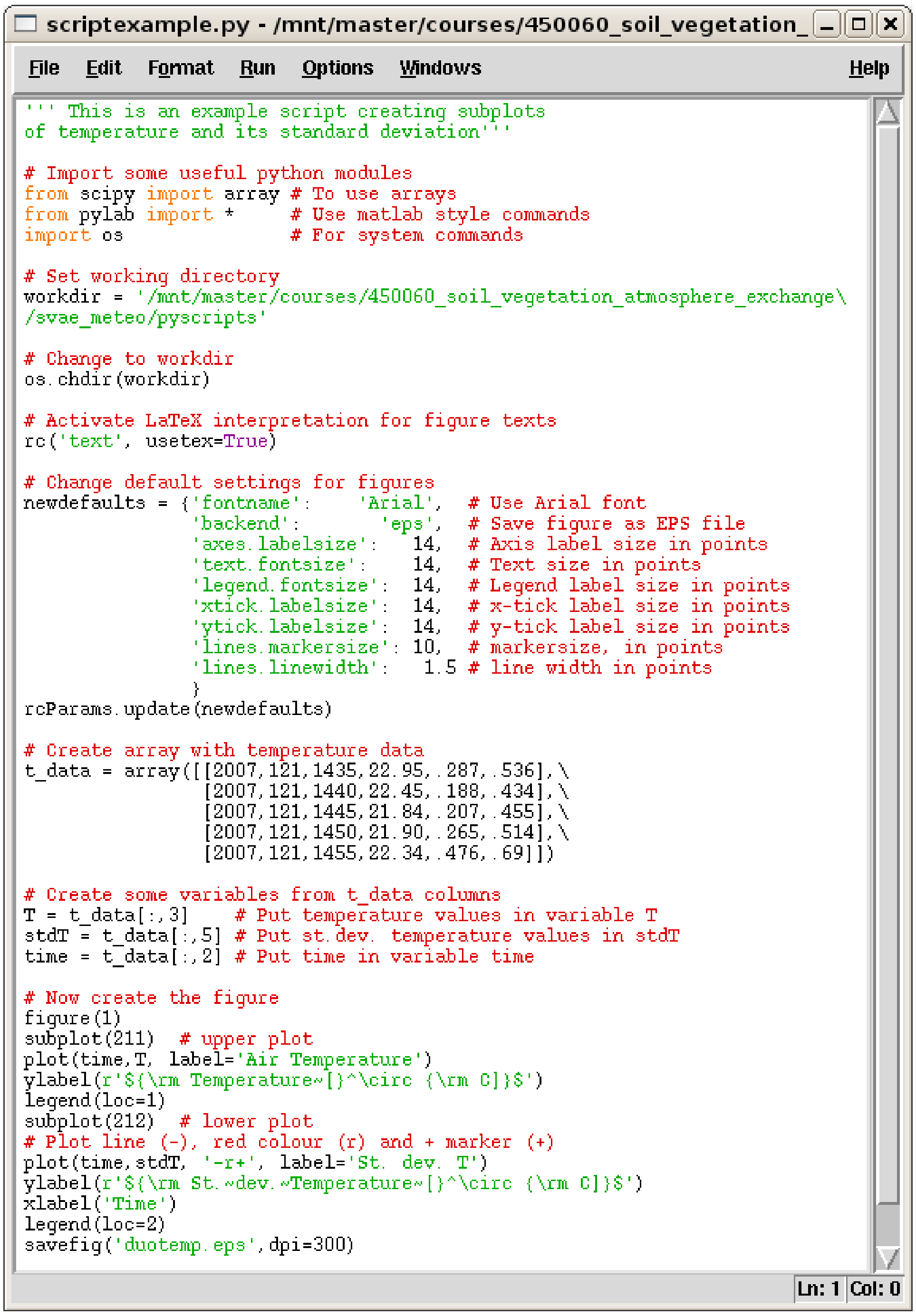
An example of a simple script file listing the sequence of commands to create the sub-plot in Figure 9.5 is given in Figure 3.2. This script can be run from the editor by opening the examplescript.py file (Download this file for experimenting with Python from http://python.hydrology-amsterdam.nl) and then from the menu select Run > Run module. You will have to type show() in the Idle command window to display the figure. You now have the basis to start programming in Python.


jEdit - asu.py
File Edit Search Markers Folding View Utilities Macros Plugins Help
asu.py (/media/waterloo/sony_ntfs/vu/workspace/pyscripts/src/)
File Browser
System
Console
4,1 (93/5147)
(python,none,cp1252)Nmro UG 3B/146MB 11:41 AM

```python
# -*- coding: cp1252 -*-
''' Script for working with half-hourly data from K34, Cuieiras
'''

__author__ = "Maarten J. Waterloo <maarten.waterloo@acaciawater.com>"
__version__ = "1.0"
__date__ = "October 2015"

# Run the groundwater script
file = open(r'asu_gw.py','r')
exec(file.read())
file.close()

# Run the surface water script
file = open(r'asu_h.py','r')
exec(file.read())
file.close()

# Run the meteorlogy script
file = open(r'asu_meteoscript.py','r')
exec(file.read())
file.close()

# To activate LaTeX interpretation for plotting
rc('text', usetex=True)

# Change default settings for figures
newdefaults = {'fontname':    'Arial',  # Use Arial font
               'backend':       'eps',  # Save figure as EPS file
               'axes.labelsize':   14,  # Axis label size in points
               'text.fontsize':    14,  # Text size in points
               'legend.fontsize':  12,  # Legend label size in points
               'xtick.labelsize':  14,  # x-tick label size in points
               'ytick.labelsize':  14,  # y-tick label size in points
               'lines.markersize': 10,  # markersize, in points
```

```
This shell runs operating system processes.

Press %<TAB> to list built-in commands.
Run built-in with --help argument to get a brief usage message.
Run %help to view Console plugin online help.

Errors generated by compilers and some other programs are listed
for easy one-click access in the 'Plugins->Error List->Error List'
window.
~>
```

Figure 3.1: JEdit programmer's text editor showing the formatting for a python script file.


scriptexample.py - /mnt/master/courses/450060_soil_vegetation_
File
Edit
Format
Run
Options
Windows
Help
Ln: 1
Col: 0

```
''' This is an example script creating subplots
of temperature and its standard deviation'''

# Import some useful python modules
from scipy import array # To use arrays
from pylab import *      # Use matlab style commands
import os                # For system commands

# Set working directory
workdir = '/mnt/master/courses/450060_soil_vegetation_atmosphere_exchange\
/svae_meteo/pyscripts'

# Change to workdir
os.chdir(workdir)

# Activate LaTeX interpretation for figure texts
rc('text', usetex=True)

# Change default settings for figures
newdefaults = {'fontname':     'Arial',  # Use Arial font
               'backend':        'eps',  # Save figure as EPS file
               'axes.labelsize':   14,   # Axis label size in points
               'text.fontsize':    14,   # Text size in points
               'legend.fontsize':  14,   # Legend label size in points
               'xtick.labelsize':  14,   # x-tick label size in points
               'ytick.labelsize':  14,   # y-tick label size in points
               'lines.markersize': 10,   # markersize, in points
               'lines.linewidth':   1.5 # line width in points
               }
rcParams.update(newdefaults)

# Create array with temperature data
t_data = array([[2007,121,1435,22.95,.287,.536],\
                [2007,121,1440,22.45,.188,.434],\
                [2007,121,1445,21.84,.207,.455],\
                [2007,121,1450,21.90,.265,.514],\
                [2007,121,1455,22.34,.476,.69]])

# Create some variables from t_data columns
T = t_data[:,3]     # Put temperature values in variable T
stdT = t_data[:,5] # Put st.dev. temperature values in stdT
time = t_data[:,2] # Put time in variable time

# Now create the figure
figure(1)
subplot(211)  # upper plot
plot(time,T, label='Air Temperature')
ylabel(r'${\rm Temperature~[}^\circ {\rm C]}$')
legend(loc=1)
subplot(212)  # lower plot
# Plot line (-), red colour (r) and + marker (+)
plot(time,stdT, '-r+', label='St. dev. T')
ylabel(r'${\rm St.~dev.~Temperature~[}^\circ {\rm C]}$')
xlabel('Time')
legend(loc=2)
savefig('duotemp.eps',dpi=300)
```

Figure 3.2: Idle editor window showing example script.

3.3 Debugging

The process of removing errors (bugs) in your script is called *debugging*. There are two forms of errors, those that impact the syntax of the code (forgotten :, }, etc.) and those that are syntactically correct but produce wrong output. The latter ones are usually much more difficult to find. These would include, for instance, erroneous use of a variable name in an equation or use of wrong unit conversions (*e.g.* multiply by 100 instead of 1000). Most program codes contain such errors, which sometimes only appear under certain conditions, or are never detected at all because the output seems reasonable.

The Python compiler will stop when an error in the program code syntax is detected. When you run a script and an error occurs, a message will be displayed in the Python shell window. The error may have been caused in your script, but may also be transferred to Pylab, Math, Scipy or other function libraries that you have imported in your script. When variables or functions have been defined in the code that has been executed before the error occurred, these will remain available in memory and can be therefore accessed/used by typing in commands at the prompt in the shell window. This is very useful for debugging your script, as you can often detect why a bug occurred by looking at the variables that have loaded.

3.4 Useful snippets of code

When you are using Python regularly, you will often be recycling snippets of code. A few examples have been given earlier already, such as how to load an ASCII data file and assign variables to the columns:

```
# Use some operating system and scipy routines
from os import chdir,getcwd,listdir
from scipy import loadtxt
# Assign working directory to variable workdir
workdir = 'i:/svae_meteo/pyscripts'
# Load data from datafile.txt in variable data
data = loadtxt(datafile.txt, comments='#', delimiter = '\t')
# Assign first column to variable data1
data1 = data[:,0]
```

You can use the Run > Run module menu entries in the Idle editor to run a script. Alternatively, within the Idle shell window, or any other Python shell window, a script can be executed using the following command:

```
>>> execfile("script.py")
```

A script file can also be executed independently from Idle or Python using the following command in a terminal window:

```
watm@pc06-091:~\$ python script.py
```

Chapter 4

Program flow control

4.1 Boolean tests

Suppose that we want to know which values in the t_data array defined in Section 2.2.7 are non-temperatures. We know that temperatures will not become higher than 50 °C, hence we could say that values over 50 °C would not be true temperatures, but erroneous values that would need correction. The *conditional statement* t_data> 50 would then indicate these values as Boolean types, *i.e. True* for those that fulfil the condition and and *False* for those that do not satisfy the condition. Below some examples of conditional statements.

```
>>> # Set variable a to 1
>>> a=1
>>> # Check if a equals 1
>>> a==1
True
>>> a==2
False
>>> # Now check the t_data array for data larger than 50
>>> t_data > 50.0
>>> array([[True, True, True, False, False, False],
    [True, True, True, False, False, False],
    [True, True, True, False, False, False],
    [True, True, True, False, False, False],
    [True, True, True, False, False, False]], dtype=bool)
>>>
```

The following statements use conditional statements to execute some part of code depending on whether the condition is True or False. Note that the conditional statement is followed by a colon (:) and that the code to be executed needs to be indented. The statement continues during indentation and only ends when the indentation is stopped.

4.2 The IF statement

If statements are useful to perform an action based on whether a condition is met or not. For instance, the code below sets a solar radiation Rs variable at -10.0 W m^{-2} and then tests if $Rs < 0$ W m^{-2}. If the condition is met (true, $R_s < 0$) it will set $Rs = 0.0$ and print a short text and the value of Rs. If not true, it prints "Solar radiation larger than or equal to zero.":

```
>>> Rs = -10.0
>>> if Rs < 0:
        Rs = 0.0
        print 'Nighttime solar radiation =',Rs
    else:
        print 'Solar radiation larger than or equal to zero.'

Nightime solar radiation = 0.0
>>>
```

Note that the conditional statement Rs<0 is ended with a colon (:). You could also include *elif :* statements between the if and else statements to make further distinctions, for instance for very bright sunlight conditions:

```
>>> Rs = -10.0
>>> if Rs < 0:
        Rs = 0.0
        print 'Nighttime solar radiation =',Rs
    elif Rs > 1000:
      print 'Very bright sunlight!'
    else:
        print 'Solar radiation larger than or equal to zero.'

Nightime solar radiation = 0.0
>>>
```

4.3 The FOR and WHILE statements

The *for* or *while* statements are used to loop through some code. The *for loop* is a looping statement that iterates over a sequence of objects. We can create a set of sequential data using the *range()* function.

```
>>> help(range)
Help on built-in function range in module __builtin__:

range(...)
    range([start,] stop[, step]) -> list of integers

    Return a list containing an arithmetic progression
    of integers. range(i, j) returns [i, i+1, i+2, ...,
    j-1]; start (!) defaults to 0. When step is given,
    it specifies the increment (or decrement). For
    example, range(4) returns [0, 1, 2, 3]. The end
    point is omitted! These are exactly the valid
    indices for a list of 4 elements.

>>> range(5)
[0, 1, 2, 3, 4]
>>>
```

Now let's use this in a *for loop*, the *else* statement is optional and is executed when the i becomes outside the defined range:

```
>>> for i in range(1,10,2):
        print 'i=',i
  else:
        print 'We reached the end of the loop'

i= 1
i= 3
i= 5
i= 7
i= 9
We reached the end of the loop
>>>
```

Suppose that we do not want to print these values but store them in a list, together with a second value that is twice as large as i. We can define a list data type variable with the statement testlist = []. We would then have to *append* each line of new values created in the loop and temporarily stored in the "dummy" variable to the list of previous values stored in the variable *testlist* using a *testlist.append(dummy)* instruction. The resulting list can then be converted to an array data type using the *array()* function. This can be done as follows:

```
>>> from scipy import *
>>> testlist = [] # Define a variable to store your values
>>> for i in range(1, 10, 2):
        j = 2.0 * i
        dummy = i, j
        # append the dummy variable to previous values
        testlist.append(dummy)

>>> testlist
[(1, 2.0), (3, 6.0), (5, 10.0), (7, 14.0), (9, 18.0)]
>>> testlist = array(testlist) # convert list of values to array
>>> testlist
array([[  1.,   2.],
       [  3.,   6.],
       [  5.,  10.],
       [  7.,  14.],
        [   9.,    18.]])
>>>
```

This procedure is very useful for calculating daily averages, sums, standard deviations, etc.

The *while* is another looping statement that repeatedly executes statements as long as a condition is true. It can also have an optional ELSE clause. In the following example we assign a value of 5 to a. In the while statement we increase a with one and print out the text a is lower than 10 every time an iteration is made. As you can see below, the loop stops when a reaches 10.

```
>>> a = 5 # Set a to 5
>>> while a < 10:
        print a,' is lower than 10'
        a = a + 1  # Increase a by 1

5  is lower than 10
6  is lower than 10
7  is lower than 10
8  is lower than 10
9  is lower than 10
>>>
```

The *break* and *continue* statements also allow looping but will not be discussed here.

Chapter 5

Functions in Python

We have already been introduced to the float(), int(), type(), sqrt() and $\log()$ functions in the above, but there are many more (predefined) functions in Python. Below are some examples.

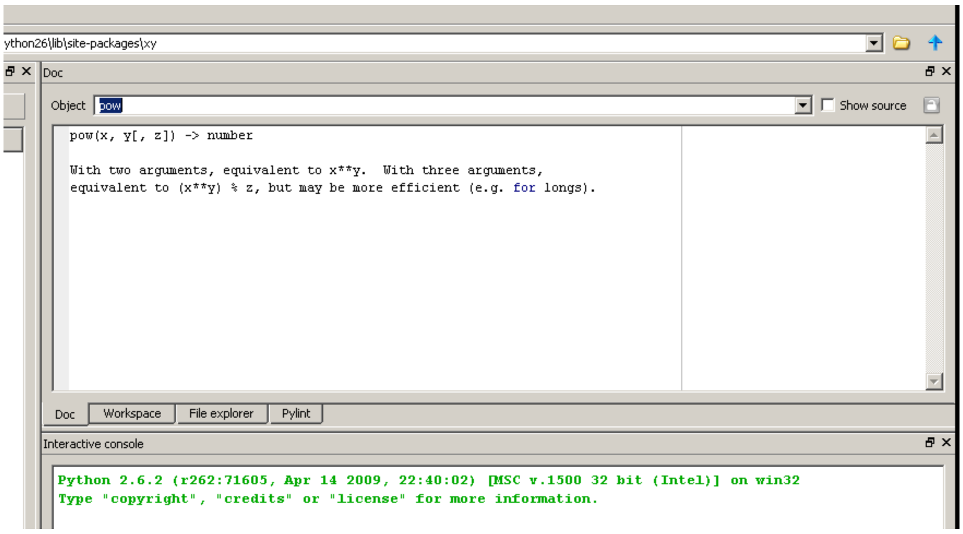
5.1 Basic functions

5.1.1 Help() function

An important function is the *help()* function, that provides you information about a function. Let's take the *pow()* function as an example:

```
>>> help(pow)
Help on built-in function pow in module __builtin__:

pow(...)
    pow(x, y[, z]) -> number

    With two arguments, equivalent to x**y.  With three
    arguments, equivalent to (x**y) \% z, but may be
    more efficient (e.g. for longs).

>>> pow(3,2)
9
>>> 3**2.0
9.0
>>>
```

This tells you that pow(x,y) will give you x to the power of y (as does $x * *y$) and that the function is built into Python and in this case returns an integer (9). As mentioned earlier a lot of functions are defined in function libraries, such as in the *math* module (sin(), cos(), tan(), pow() and sqrt() functions, pi, e constants, etc.). A function library is called a *module*. The *pow()* function is also defined in the



```
pow(x, y[, z]) -> number

With two arguments, equivalent to x**y.  With three arguments,
equivalent to (x**y) % z, but may be more efficient (e.g. for longs).
```

```
Python 2.6.2 (r262:71605, Apr 14 2009, 22:40:02) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
```

Figure 5.1: Documentation screen in Spyder that provides explanation of functions in Python.

module *math*, that has to be loaded before you can use the power function using *from math import pow*, or load all functions in *math* by using *from math import **.

```
>>> from math import pow
>>> help(pow)
Help on built-in function pow in module math:

pow(...)
    pow(x,y)

    Return x**y (x to the power of y).

>>> pow(3,2)
9.0
>>>
```

You can see which functions the *math* module holds by typing in *help(math)*.

In Spyder, there is a separate screen with a tab labelled *doc* (Figure 5.1) for documentation, that provides documentation for functions. In addition, next to the *tools* menu entry a question mark (?) provides access to addition documentation of several important Python modules.

Assignment

Consult help for the functions *sqrt*, *log*, *rand*, *len* and *math* and write down what each of these functions does.

5.1.2 Find() function

The *find()* function is defined in the *pylab* module, which should be imported before you can use this function. The find() function returns the indices from an array where the condition that you use in the function is TRUE. It will do so in an 1-dimensional array. Remember that we assigned temperature to the variable T before. Suppose that we want to know when the temperature is higher than 22.0 °C, we could then issue the following commands:

```
>>> T
array([ 22.95,  22.45,  21.84,  21.9 ,  22.34])
>>> Thigh = find(T > 22.0)
>>> Thigh
array([0, 1, 4])
>>>
```

This shows you that T[0], T[1] and T[4] satisfy the condition of being larger than 22.0 °C. Suppose that we only want these values to be converted to K. We would then do:

```
>>> T
array([ 22.95,  22.45,  21.84,  21.9 ,  22.34])
>>> Tkelvin = T[find(T > 22.0)] + 273.16
>>> Tkelvin
array([ 296.1 ,  295.6 ,  295.49])
>>>
```

The above shows that we can use the *find()* function within other functions, which is very useful.

An alternative to *find()* is to use a condition as an array index. If we want to find the temperatures after, for instance, 14:45 h, we could do:

```
>>> # Find temperatures for time after 14:45 h
>>> t_data[t_data[:,2]>1445,3]
array([ 21.9 ,  22.34])
>>>
```

This is often more elegant than using the find() function.

An application would be the following. Suppose that you have measured solar radiation (short-wave) at half-hourly intervals over several days and that you know that these values should be zero at night, but are sometimes small negative or positive numbers instead. If you would like to set values of R_s smaller than 5 W m^{-1} to zero, you could do the following, where the R_s variable holds an array of half-hourly solar radiation values:

```
>>> Rs[Rs<5]=0
```

or

```
>>> Rs[find(Rs<5)]=0
```

5.1.3 Mod() and floor() functions

Sometimes there is the need to separate the decimal part from the integer part of a number. One such example is the conversion of data logger time, stored as year, day of year and hour (2008, 157, 1430), to a decimal time that is needed for plotting of the data.

The *floor()* function in the *math* module will return the integer part of a number, i.e. the largest integer value less than or equal to that number, whereas the *mod()* in the *scipy* module returns the modulo, *i.e.* the decimal part. The mod() function takes two arguments, the number (n) and a divisor (d). Mod(n,d) returns the difference between n and the largest multiple of d smaller than n. With a divisor d of 1, mod(n,1) will return the decimal part of the number n. You can also use the modulo operator (%) instead of the mod() function. :

```
>>> from math import floor
>>> from scipy import mod
>>> floor(3.5)
3.0
>>> mod(3.5,1)
0.5
>>> 3.5 % 1
0.5
>>>
```

5.1.4 Range functions

Ranges of values can be created using the *range()*, *arange()* and the *linspace()* functions. The range() function creates a list type, whereas the arange() function creates an array of integers and linspace() function an array of float values:

```
>>> x=range(0, 10)
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> type(x)
<type 'list'>
>>> y = arange(0, 10)
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> type(y)
<type 'numpy.ndarray'>
>>> z = linspace(1, 6, 5)
>>> z
array([ 1.  ,  2.25,  3.5 ,  4.75,  6.  ])
```

```
>>> type(z)
<type 'numpy.ndarray'>
>>>
```

You can also use a third, additional argument in the range() and arange() functions to influence the steps:

```
>>> x=arange(0, 10, 2)
>>> x
array([0, 2, 4, 6, 8])
>>>
```

5.2 Statistical functions

We often need to describe the statistics of our dataset. We can use the *average()*, *std()*, *sum() max()* and *min()* functions to get the average, standard deviation, sum, maximum and minimum values, respectively. These functions are defined in the module *scipy* so you have to import this module first. Let's do this for the variable T again.

```
>>> from scipy import *
>>> average(T)
22.295999999999999
>>> std(T)
0.40450463532572778
>>> sum(T)
111.47999999999999
>>> max(T)
22.949999999999999
>>> min(T)
21.84
>>>
```

It is also possible to do linear regression using the *polyfit()* function from the *scipy* module. The polyfit() function is based on least-square analysis and takes three arguments, x-data, y-data and a number n describing the polynomial order (1 = linear regression, 2 = quadratic, 3 = cubic, etc.). In the linear case ($y = ax + b$) it returns two coefficients, the slope a and the intercept b. You call the function for linear analyses as: a,b = polyfit(xdata,ydata,1). This function will not provide you with any further statistics. If you need such detailed statistics (r^2, F-test, etc.) then you could download and use the module *ols.py*, which you can get at http://www.scipy.org/Cookbook/OLS.

5.3 Meteorological and hydrological functions

We have defined several meteorological functions and evaporation formulae in the *meteolib.py* and *evaplib.py* modules. You can download these modules from http://python.hydrology-amsterdam.nl/.

5.3.1 General meteorological functions: meteolib.py

The following functions are defined in meteolib.py:

- date2doy(): converts a date (dd, mm, yyyy) to a "day of year" (1-366) value.
- sun_NR(): calculates daylength and extraterrestrial radiation input, used for calculating evaporation
- es_calc(): Calculate saturation vapour pressures
- ea_calc(): Calculate actual vapour pressures
- vpd_calc(): Calculate vapour pressure deficits
- Delta_calc(): Calculate slope of vapour pressure curve
- L_calc(): Calculate latent heat of vapourisation
- cp_calc(): Calculate specific heat
- gamma_calc(): Calculate psychrometric constant
- rho_calc(): Calculate air density
- pottemp(): Calculate potential temperature (1000 hPa reference pressure)

The latent heat of vapourisation (λ) depends only on temperature. To calculate λ from temperature variable T we would do:

```
>>> from meteolib import *
>>> L = L_calc(T)
>>> L
array([ 2445736.025975 ,  2446919.4761   ,  2448363.2852525,
        2448221.2712375,  2447179.8351275])
>>>
```

Use the help() function to get information about the other functions in the meteolib module.

5.3.2 Evaporation functions: evaplib.py

A second library, *evaplib.py* was developpped by M.J. Waterloo and contains functions to calculate daily evaporation rates according to the Penman open water evaporation [E0(); Penman, 1948, 1956, 1963, Valiantzas, 2006], the FAO Penman Monteith reference evaporation [ET0pm(); Allen et al., 1998], the Dutch Makkink reference crop evaporation [Em(); de Bruin, 1987] and the Priestley Taylor evaporation [Ept(); Priestley and Taylor, 1972]. The following statement calculates the Priestley Taylor evaporation for a temperature of 21.65 $^{\circ}\mathrm{C}$, a relative humidity of 67%, air pressure of 101300 Pa, net radiation input of 18200000 J m^{-2} day^{-1} and a soil heat flux of 600000 J m^{-2} day^{-1} .

```
>>> from evaplib import *
>>> Ept(21.65,67.0,101300,18200000,600000)
6.3494561161280778
>>>
```

The evaplib.py library can be downloaded from *ecohydro.falw.vu.nl*.

5.4 Defining your own functions

If you need to execute several statements or calculations more than once, it may be a good idea to define your own function. You can define two types of functions in Python. The first type is named and is defined using the *def – return* keywords. You can store these function scripts in a function module from which you can import these for repeated use within a second script. The second type are anonymous functions, called *lambda* functions. Both function types will be discussed below.

5.4.1 Defining a named function

The definition of a traditional named function starts with the *def* statement followed by the function name and, optionally between parenthesis, a list of its arguments called the *function argument list*. The arguments are any input values that you want to pass on to the function. For example, lets define a function *linreservoir()*. The discharge recession can often be described by the following equation:

$$Q_t = Q_0 e^{-\frac{1}{k}t} \tag{5.1}$$

Now suppose that you want to calculate Q_t, for which you would need Q_0, k and t as the functions arguments. You could then define the function *linreservoir()* as follows, where we import the *exponential function* from the *scipy* within the function and we state in the function argument list that Q_0 and k are of type *float*:

```
def linreservoir( q0 = float(), k = float(), t ):
  from scipy import exp  # import exponential function
    qt = q0 * exp(-1 / k * t)
    return qt
```

Note that the function statements are all indented and that the final statement (return qt) provides output of the function. If we save this function in a script file called linres.py, we can import the function linreservoir() as follows:

```
>>> from linres import linreservoir
>>> linreservoir(100.0, 30.0, 67)
10.717059875230674
>>>
```

The above works only with single values as input. If you want to work with arrays of time values as input, you can specify this in the function argument list:

```
# import modules
from scipy import array

def linreservoir(q0=float(), k=float(), t = array([]) ):
   from scipy import exp
   qt = q0 * exp(-1 / k * t)
   return qt
```

If we now use the function with *t* as an array, we als get an array of *qt* as output:

```
>>> t=array([1, 2, 3, 4])
>>> t
array([1, 2, 3, 4])
>>> q = linreservoir(100.0, 50.0, t)
>>> q
array([ 98.01986733,  96.07894392,  94.17645336,  92.31163464])
>>>
```

It is always very important to document your programs well so that other people may use or modify them. The first thing that springs to mind is adding a *help entry* text in your function. This means that if you would type *help(linreservoir)* it would show you how to use the function and what its inputs and outputs are. Adding a help entry is easy. Just insert the help text as a multi-line comment after the function definition, as shown by the example below.

```
# import modules
from scipy import array

def linreservoir(float(q0), float(k), t = array([]) ):
   '''The function linreservoir() calculates the
      discharge at time t from an initial
      discharge Q(t=0) and a recession
      constant
```

```
    Input:
    ----------
        t: (array of) time
        q0: initial discharge value
        k: recession constan

    Output:
    ----------
        qt: (array of) discharge values at time t

    Example:
    ----------
        >>> linreservoir(15.0, 30.0, 20 )
        2.6359713811572676
    '''

    # Import relevant Python modules
    from scipy import exp

    # Do the calculations
    qt = q0 * exp(-1 / k * t)
    return qt
```

If we now invoke the help function (*e.g.* help(linreservoir)) we get the following output with our *help text*:

```
>>> help(linreservoir)
Help on function linreservoir in module __main__:

linreservoir(q0=0.0, k=0.0, t=array([], dtype=float64))
    The function linreservoir() calculates the
    discharge at time t from an initial
    discharge Q(t=0) and a recession constant

    Input:
    ----------
        t:  time
        q0: initial discharge value
        k:  recession constant

    Output:
    ----------
        qt: discharge at time t

    Examples:
    ----------
        >>> linreservoir(15.0, 30.0, 20 )
    7.70125678548888

>>>
```

Many meteorological functions are defined like this in the *meteolib* and *evaplib* modules (see Sections 5.3.1 and 5.3.2).

5.4.2 Lambda functions

In contrast to the named functions, *lambda functions* are anonymous functions that are not attached to a name during runtime of your script. They are defined as follows using the lambda construct where the function outputs the argument (x) to the power 3:

```
>>> power3 = lambda x: x**3
>>> power3
<function <lambda> at 0xb9f0ca4>
>>> power3(3)
27
>>>
```

You can use more than one argument in a lambda function:

```
>>> powerxy =  lambda x, y: x**3 + y**2
>>> powerxy(3,4)
43
>>>
```

These lambda functions can be part of named functions.

Chapter 6

Object oriented programming in Python – Classes

In the previous chapters you have learned the basics of Python. This included data types, use of variables and function definitions that combined in a script form a program that can be used to perform certain actions. In many computer models, parts of code are implemented with minor changes at different places in the program. This implies that if such a part of code is changed or used differently, the change in function code should be implemented in different places in the program, making it subject to errors. It is much more efficient to develop code as an *object* in one place, that serves as a recipe use elsewhere in the program. A computer language that can do this, such as Python, is called an *object-oriented* language.

For instance, if you want to develop a program that defines functions for soils, you have to deal with differences between soil types, such as for example course grained, clay and peat soils. If you have very simple functions you could use a dictionary data type (Section 2.2.5) to achieve this:

```
>>> soiltype = {'course':'execute program code for course soils',\
   'clay':'execute program code for clay soils',\
   'peat':'execute program code for peat soils',\
   }
>>> soiltype.get('peat','soiltype unknown')
'execute program code for peat soils'
>>> soiltype.get('silt','soiltype unknown')
'soiltype unknown'
>>>
```

You could also achieve the same using if – elif –else statements (Section 4.2) that would select the appropriate function code:

```
>>> soil='sand'
>>> if soil =='course':
   print 'execute function code for course soils'
 elif soil =='clay':
  print 'execute function code for clay soils'
elif soil =='peat':
  print 'execute function code for peat soils'
else:
  print 'unknown soil type'

unknown soil type
>>>
```

It is very likely that the functions used for these different soils show similarity and you therefore have to edit three functions if you make modifications.

Object oriented programming makes it possible to develop a kind of general recipe or prototype for a set of functions that can have several *instances* with different functioning, *i.e.* for each of the soil types, in your program. This recipe can contain constants, variables and functions. The latter are then called the *methods* of the *object*.

A prototype of an object can be created in Python by defining a *Class* type. Such a class type contains variables, properties and methods that characterize the objects of the class. The attributes of a class are called *data members* and *methods* that can be accessed through a . (dot) notation. For instance if a class is constructed with the name of *address*, *address.street* might show the street name variable, whereas *address.city* would yield the name of the city.

Below is a simple program using a *soil()* class with two functions for calculating arbitrary soil values K and H:

```
''' This is a class for handling soil data

    An instance of this class can be created by calling soil(C),
    where C is a constant used in the calculations. C varies
    between soil types.

    Written by M.J. Waterloo as an example.

'''

class soil(object):

    # Define an initial variable that is related to each class
    # instance
    def __init__(self,C):
        # Get a constant C for each class instance created
        self.C = C

    def calcK(self, theta):
        # Calculate a K variable according to moisture status
        # theta (0-1) and the constant C
        K = 1/theta*self.C
        return K

    def calcH(self, theta):
        # Calculate water pressure based on soil moisture theta
        H = self.C / theta * 100
        return H

# Make several instances of the class for each soil type, note
# that C varies
clay = soil(0.05)
sand = soil(0.7)

# Calculate soil properties K and H
kclay = clay.calcK(0.4)
ksand = sand.calcK(0.4)
print 'K for clay at theta = 0.4 is: ', kclay
print 'K for sand at theta = 0.4 is: ', ksand
```

In the above first a soil() class and functions in the class are defined, where def __init__(self,C): is done when the instance of the soil class is initiated. In this case C varies between soil types and is used to calculate K and H based on a θ input variable. Then an instance of soil is made for clay with $C = 0.05$ by calling clay = soil(0.05) (and also for sand with a different C value) and K is calculated for $\theta = 0.4$. The same is done for sand. When we run the script we get:

```
>>>
K for clay at theta = 0.4 is:  0.125
K for sand at theta = 0.4 is:  1.75
>>>
```

Note that the outcome differs between soil types for the same value of θ. We could now easily add an instance for peat by including:

```
peat = soil(0.1)
```

We then have three instances, one for each soil type, that we can call to calculate corresponding K values.

Chapter 7

File and directory manipulations

7.1 Directory manipulations

When you start you have to create a directory to work in. Please create a directory called *svae_python* on one of the drives of your computer. Now start Python by opening Idle. To change change your working directory you have to import the OS module first. The OS (Miscellaneous operating system interfaces) module contains functions that pertain to commands for your operating system, such as changing directories, making directories, listing files, etc. Some examples are given below. To import this module type on the command prompt:

```
>>> import os
>>>
```

Now you can use the *os.getcwd()* function to see where you are (/home/watm is my home directory under unix):

```
>>> os.getcwd()
'/home/watm'
>>>
```

You can change to another directory using the *os.chdir()* function. To change to another directory use the *os.chdir()* command, for instance type os.chdir('C:\svae_python') in MS Windows, or if you work under unix os.chdir('/home/watm/svae_python'). It is often easy to define a variable, for instance called "dataroot", holding the name of your working directory. For unix:

```
>>> dataroot = '/home/watm/svae_python'
>>> os.chdir(dataroot)
>>> os.getcwd()
'/home/watm/svae_python'
>>>
```

or in MS windows:

```
>>> dataroot = 'c:\svae_python'
>>> os.chdir(dataroot)
>>> os.getcwd()
'c:\svae_python'
>>>
```

To list all files in the current working directory use os.listdir(os.getcwd()):

```
>>> os.listdir(os.getcwd())
['meteolib.py','meteo2008.txt','meteoscript.py']
>>>
```

If you have a long directory path and file name, such as /home/watm/svae_python/data.txt the *os.path* module provides possibilities to split or join these paths and file names or file name extensions. For instance, if you have the variable /home/watm/svae_python/data.txt and you want to separate the filename from the directory path, you can do:

```
>>> import os
>>> pathname,filename=os.path.split('/home/watm/svae_python/data.
    txt')
>>> pathname
'/home/watm/svae_python'
>>> filename
'data.txt'
>>>
```

If you want to split the path (/home/watm/svae_python), file name (data) and its extension you could use os.path.splitext()

```
>>>pathfilename,extname=os.path.splitext('/home/watm/svae_python/
    data.txt')
>>> pathfilename
'/home/watm/svae_python/data'
>>> extname
'.txt'
>>>
```

These commands are very useful for in case you want to place files in a different directory than where these were originally, or to change file names.

Assignment

Based on the above, add a few lines of program code to your personal script file that sets your working directory and gives a list of the files in this directory.

7.2 Working with ASCII data files

7.2.1 Reading data from file with loadtxt()

Often we get our data from automatic data registration units, or data loggers. These data loggers store the values from the instruments attached to them in an ACSCII or text file. The values in this file are separated by a *delimiter*, which usually is a space ( ), a tab (	) or a comma (,). The latter file is often called a *comma separated values* file and has .csv as its file name extension. Comments in such files are often preceded by the pound (#) symbol. Output for a VU water level data logger is, for example:

```
# DOY day month year hr min sec H
124 04 05 09 10 15 30 +00002
124 04 05 09 10 30 30 −00005
124 04 05 09 10 45 30 −00010
124 04 05 09 11 00 30 −00020
```

The data on the first line represents the *day of year* (124, doy), day (04), month (05), year (09), hour (10), minutes (15), seconds (30) and the water level in $1/100$ of a cm (+00002). Note that all values are separated by spaces.

If these data were stored in a file called *wl.txt*, we can read these into an array variable (here called wldata) using the loadtxt() function that is defined in the *scipy* (Scientific Python) or *pylab* modules:

```
>>> from scipy import loadtxt
>>> wldata = loadtxt('wl.txt', comments='#', delimiter= ' ')
>>>
```

Note that you cannot have missing values in the file, so every column must have the same length and you should not have empty lines at the end of the file. If you have missing data in one or more columns, you could fill these locations first with -9999, a value commonly used to indicate missing data. Now we have an array in which the different columns are stored. The first header lines (#) was skipped due to the *comments='#'* argument in the loadtxt() function. The first column of the array (wldata[:,0]) holds the DOY, and the last column (wldata[:,7]) the water level data. You can now perform calculations with the data:

```
>>> wl = []
>>> wl = wldata[:,7]/10000.0
```

The water level is now stored in the variable *wl* and its unit is now m, rather than in tenths of a mm.

Assignment

1. Use the loadtxt() function (defined in the *pylab* or *scipy* modules) to read the file *wl.txt* (available at http://ecohydro.falw.vu.nl/python) and place the data in a variable named *wldata*. Note that the first line in wl.txt contains a comment with the header for each column.

2. Now assign each column in wldata to individual variables as defined in the header, *i.e.* JD, DD, MM, etc.

3. Calculate the average value of your variable H (the water level)

4. Select all H values after day 160

5. The actual water level H was 0.31 m at the beginning of the measurements when the logger H was set to zero and the logger H values are in 0.1 mm units. Correct H so that it represents the water level in m.

The Campbell Scientific CR1000 data logger that we use for our meteorological measurements stores data differently than the VU water level data logger. Here is an example of the data output of the logger:

```
"TOA5","MyDatalogger","CR1000","E3686","CR1000.Std.27","CPU:
    cr1000_meterology_v_1.12_Portugal_2014.CR1","28966","fasttable
    "
"TIMESTAMP","RECORD","AvgTCa","stdTCa"
"TS","RN","Deg C","Deg C"
"","","Avg","Std"
"2014-06-05 15:05:00",01,"NAN","NAN"
"2014-06-05 15:10:00",02,"NAN","NAN"
"2014-06-05 15:15:00",03,27.01,1.798
"2014-06-05 15:20:00",04,24.25,0.98
```

We see that it is a *comma separated values* (csv) file, the first few lines identify the logger and the meaning of the data in the four columns ("TIMESTAMP", "RECORD", "AvgTCa", "stdTCa"), their units, etc., followed by data lines, which all start with a date string ("2014-06-05 15:05:00"), float values (27.01) and in some cases "NAN", which is the abbreviation for Not A Number for when the sensor had not yet been connected.

If you use the loadtxt() function as above, it will give you an error telling you that it can only read float values. As such we have to extend loadtxt() arguments to tell it:

1. that we should skip the first four lines of the file because these contain a header with text

2. that the first column should be translated to a date

3. that if it encounters ”NAN” it should convert it to a number indicating missing values, usually -9999.

Loadtxt() has different arguments to do this. To skip the first four lines we use the *skiprows* argument in loadtxt():

```
>>> from scipy import loadtxt
>>> data = loadtxt('filename.dat',\
        comments='#',\
        delimiter=',',\
        skiprows=4)
>>>
```

Loadtxt() also has an argument to translate certain input it reads to a different output. In this case, we want to convert the date string in the first column (column 0) to a date number that Python can use for plotting. This is done with the loadtxt() *converters*= argument that we apply to column 0 and telling it what format the date string is (”%Y-%m-%d %H:%M:%S”) and how to translate it using the matplotlib strpdate2num() function.

```
from matplotlib.dates import strpdate2num  # matplotlib date
    function

meteodata = loadtxt('filename.dat',\
                    comments='#',\
                    delimiter=',',\
                    skiprows=4,
                    converters = {0:strpdate2num('\"%Y-%m-%d %H:%M
                        :%S\"')}\
                    )
```

Finally we have to make sure that the ”NAN” values are converted to indicate missing values (-9999).

7.2.2 Reading data from file with genfromtxt()

The *genfromtxt()* function in the scipy is a more versatile alternative for scipy.loadtxt() for importing data from a text file.

Suppose that we are working with an HBV model input file ptq.txt, which is an ASCII text file that starts with a 2-line header followed by input data in four tab-separated columns being a date, precipitation P, temperature T and discharge Q:

```
Dinkel catchment 1976-2010
Date    P T Q
19760101    9.4 3.7 0.260
19760102    16.7    4.3 0.430
19760103    13.3    5.0 1.418
19760104    4.3 2.0 1.625
```

```
19760105    1.3  5.1  1.708
19760106    15.9    3.7  1.739
19760107    0.3  7.5  1.875
19760108    0.1  6.5  NaN
19760109    0.0  6.9  1.668
```

We could read the file as follows:

```
import scipy
import matplotlib.dates

kwargs=dict(delimiter='\t',\
   deletechars=' ',\
   converters = {0:matplotlib.dates.strpdate2num('%Y%m%d')},\
   skip_header=1,\
   names=True,\
   skip_footer=0,\
   filling_values=-9999,\
   missing_values={'\"NaN\"'}\
   )
ptqdata=scipy.genfromtxt(datafile, **kwargs)
```

We have first stored the arguments for scipy.genfromtxt in the kwargs *dict* dictionary data container type variable (Section 2.2.5) that can store keys and their corresponding values (*e.g.* delimiter=','). The converters key is used to convert the date to a matplotlib.dates date object (pylab numerical date value) and missing values are identified as *NaN* (Not a Number).

In the statement *scipy.genfromtxt(datafile, **kwargs)* statement point to the previously declared kwargs dictionary using **. We have seen in Section 2.2.3 that the ** operator is used as a power operator (x**4 raises x to the power of four), but now ** is used for *keyword argument unpacking*. In this statement ** therefore unpacks the kwargs dictionary into separate keyword options as defined in the kwargs dict() statement. Note that a singe ** before a variable (e.g. *args) would mean *argument unpacking*. So both * and ** are used to unpack data structures (e.g. lists of arguments or keywords) of the arguments that these precede.

Executing the above statements gives:

```
>>> ptqdata
array([(721354.0, 9.4, 3.7, 0.26), (721355.0, 16.7, 4.3, 0.43),
       (721356.0, 13.3, 5.0, 1.418), ..., (733983.0, 8.7, 17.2,
           nan),
       (733984.0, 0.0, 19.5, nan), (733985.0, 0.1, 19.3, nan)],
      dtype=[('Date', '<f8'), ('P', '<f8'), ('T', '<f8'), ('Q', '<
         f8')])
>>>
```

The data is now in a *ndarray* with formats according to dtype and we can address, for instance, the entire date column, the first date value only, and the fifth T value by:

```
>>> ptqdata['Date']
array([ 721354.,  721355.,  721356., ...,  733983.,  733984.,
        733985.])
>>> ptqdata['Date'][0]
721354.0
>>> ptqdata['T'][4]
5.0999999999999996
>>>
```

7.2.3 Saving Python variables to file

Suppose that we have one or more variables that we would like to save in files, such as calculated daily evaporation rates. Python allows saving such variables to an ASCII text file using the *open()* function to open a text file and the *close()* function after we have written the data. The following example opens the file *d:/mydata/asu_pe.txt*, as defined in the variable *pe_outfile* for writing ('w'), then writes two lines with comments starting with the # character and ending with a newline (\n). Then we create single line lists (filelist=[]) in which we append the day number dayno[n], a tab (filelist.append('\t')), sum of rainfall (dsumP[n]) as a string number with format xxxxx.xx ('%5.2f' %), then another tab and the evaporation E0[n]. These are written to the file with *fileout.writelines(filelist)*. When all the lines have been written (n = len(dayno)) we close the file with *fileout.close()*. The *len()* function provides the length of a string, list or array as a number. In this case it provides the number of rows in the dayno variable.

```
pe_outfile = 'd:/mydata/asu_pe.txt'
fileout = open(pe_outfile,'w')
fileout.write('# Cuieiras Reserve, Manaus\n')
fileout.write('# Date, Rainfall [mm], Penman E0 [mm]\n')
for i in range(0,len(dayno)):
    filelist=[]
    filelist.append(str(int(dayno[i])))
    filelist.append('\t')
    filelist.append(str('%5.2f' % float(dsumP[i])))
    filelist.append('\t')
    filelist.append(str('%5.2f' % float(E0[i])))
    filelist.append('\n')
    fileout.writelines(filelist)
fileout.close()
```

Assignment

Save the variables from wldata, with the corrected H value, in a text file named wlm.txt.

7.3 Working with NetCDF files

Network Common Data Form (netCDF) machine-independent binary data format, with software libraries that support the creation, access, and sharing of array-oriented scientific data. The format is maintained by the University Corporation for Atmospheric Research (UCAR) and is often used for storing scientific data in a uniform way.

Python knows various software libraries (modules) that have commands to create and access data stored in this format. One of these is PyNIO (http://www.pyngl .ucar.edu/Nio.shtml), developed by NCAR's Computational and Information Systems Laboratory, which also reads other common formats (GRIB, HDF, etc.). There is also a visualisation module (PyNGL).

An alternative is the netcdf4-python module, which has recently been integrated in Python(x,y). To make a new NetCDF file you can use the *Dataset()* function and variables can be added to the file and accessed using the *createVariable()* function, where the *time* variable in the example below is a one-dimensional array of values (1 column) of type float:

```
>>> from netCDF4 import Dataset
>>> testcdffile = Dataset('test.nc', 'w', format='NETCDF4')
>>> print testcdffile.file_format
NETCDF4
>>> times = testcdffile.createVariable('time','f8',('time',))
>>> testcdffile.close() # Close the file
```

Consult the netcdf4-python documentation for more information.

A simple library with netCDF functions has been developed by Ko van Huisstееden and is available at http://ecohydro.falw.vu.nl/python.

Chapter 8

Working with dates and times

In hydrology we often work with time series. The *datetime* module provides the tools to conveniently work with dates and times. Detailed information about the functions in the datetime module is given in http://docs.python.org/library/datetime.html. We have to import the module first. If we need the current date or date and time we can do this using the *datetime.date()* and *datetime.now()* functions:

```
>>> from datetime import *
>>> date.today()
datetime.date(2009, 6, 1)
>>> datetime.now()
datetime.datetime(2009, 6, 1, 16, 54, 44, 395137)
```

In the above example, the year (2009) is followed by the month (6), day (1), hour (16), minutes (54), seconds (44) and microseconds (395137). We can also have an ordinal value of the date, which in Python is the number of days since 01-01-0001. For the first of June 2009 this would be:

```
>>> date.toordinal(date.today())
733559
>>>date.fromordinal(733559)
datetime.date(2009, 6, 1)
```

We often have data where the year, month, day, and perhaps time (in hour, minutes, seconds) are provided in separate columns (*e.g.* 2009 10 29 15 31 03). There are several ways to get the day of year (doy, 1–366) from these date values. The first one is to use the date2doy() function that is defined in the *meteolib function library*. For instance if we have a number of rows with year (YYYY), month (MM) and day of month (DD) values, we could call the following to assign day of year values to the variable doy:

```
doy= date2doy(YYYY,MM,DD)
```

We can also use the *datetime* module to combine these date values into a datetime object using the following code, where the year is again given in the variable YYYY, the month as MM and the day as DD.

```
>>> import datetime
>>>YYYY=2000
>>>MM=9
>>>DD=18
>>>datum=datetime.date(YYYY,MM,DD)
>>>datum
 datetime.date(2000, 9, 18)
```

With the function *strftime()* we can now convert these dates to other values, such as the day of year as a decimal number (doy, 0-366, %j format), the week number (%U), weekday as a decimal number (%w, 0 (Sunday) – 6), etc. This is shown below:

```
>>>datum.strftime('%j')
 '262'
>>> datum.strftime('%U')
 '38'
>>> datum.strftime('%w')
 '1'
```

So let's do something useful. This script reads in a file with three columns containing the year, month and day of the month. It then creates a datetime object that is stored in an array called alldata and converts the date to the day of year using the strftime() function. We use the *int* to convert the string that results from the strftime() function to an integer doy number. We use *append* to add data to the end of the arrays.

```
>>>from scipy import loadtxt
>>>import datetime
>>>datafile = 'test.txt'
>>># Define delimiter
>>>delimiter = ','
>>># Load data from file
>>>data=loadtxt(datafile,comments='#',delimiter=delimiter)
>>># Place each column in its own variable
>>>YYYY=data[:,0]
>>>MM=data[:,1]
>>>DD=data[:,2]
>>># Define arrays holding data
>>>alldata = []
>>>doy=[]
>>># Loop through data to convert to datetime object
>>># and doy values
>>>for i in range(0,len(data)):
>>>     datum = datetime.date(YYYY[i],MM[i],DD[i])
>>>     alldata.append(datum)
>>>     doy.append(int(datum.strftime('%j')))
>>> print doy
[335, 336, 337]
```

Chapter 9

Plotting data in a graph

9.1 Basics of plotting graphs

The *pylab* module allows you to plot your data in many different ways and save the image for later use in a report. In order to do so you first have to import the pylab module

```
>>> import pylab
>>>
```

This allows you to create very nice graphics. For meteorological data, it is often useful to plot a variable, such as wind speed, temperature or radiation against time. In order to do so we use the *plot()* function in the pylab module. Let's use the temperature data that were introduced in Section 2.2.7 as an example for plotting. We assigned the temperature data to a variable T, whereas the time data was given in the third column of the array (*i.e.* t_data[:,2]). We can plot the data by simply defining a figure (*pylab.figure(1)*, giving the *pylab.plot()* command followed by the *pylab.show()* command:

```
>>> pylab.figure(1)
<matplotlib.figure.Figure instance at 0xb6d571ec>
>>> pylab.plot(t_data[:,2],T)
[<matplotlib.lines.Line2D instance at 0xb5ee42ec>]
>>> pylab.show()
```

This will create a figure window as in Figure 9.1. If you want to issue additional commands in Idle, close the figure window first. We can fancy things up a bit by giving the graph a label for the x-axis:

```
>>> xlabel('Time')
<matplotlib.text.Text instance at 0xb5de64cc>
```

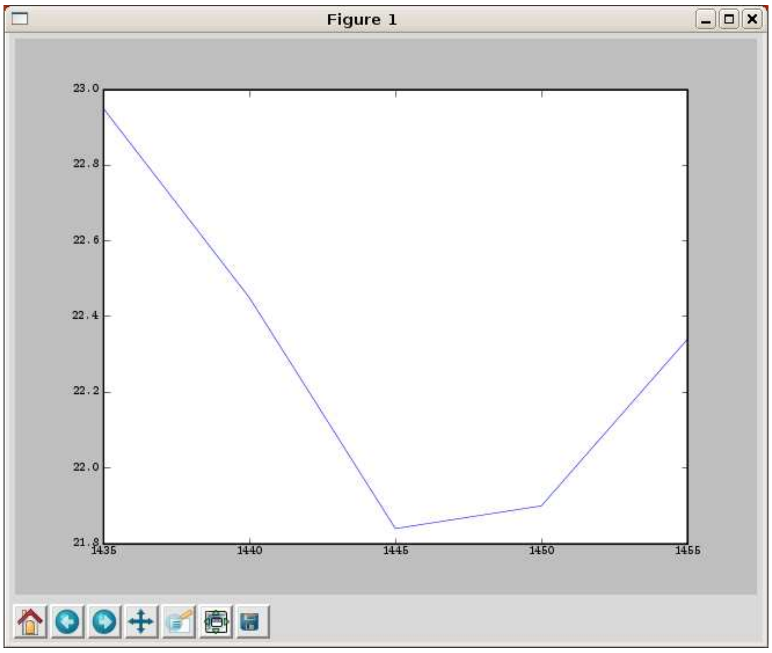


Figure 9.1: Plot of temperature data *versus* time using the plot command of the pylab module.

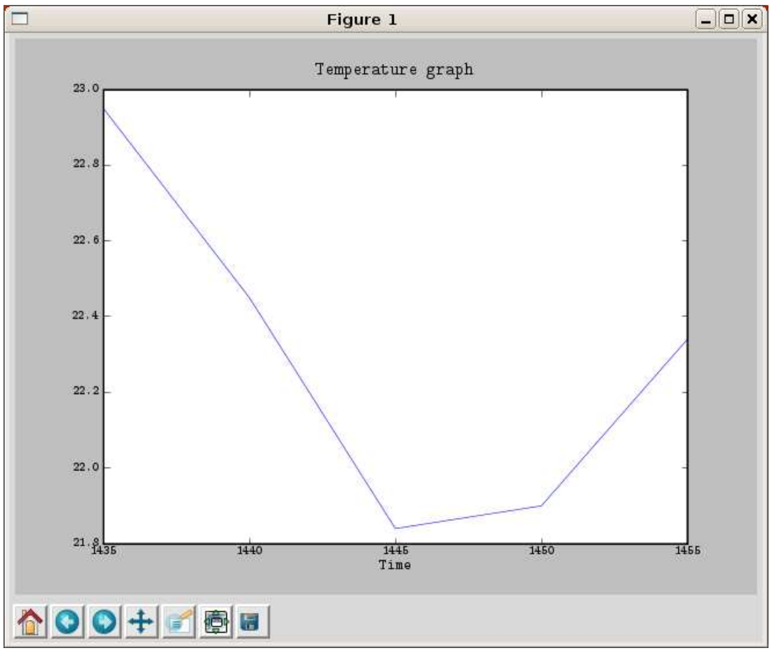


Figure 9.2: Plot of temperature data *versus* time, now with x-axis label and title.

This will result in the graph shown in Figure 9.2. For the y-axis label we need a superscript in the °C unit for temperature. For this we need to invoke LaTeX style code, where the \circ command will create the small circle and adding the ∧ will put format it as superscript. We need to set the following first to activate LaTeX text interpretation:

```
>>> rc('text', usetex=True)
>>>
```

Then we need to use the *ylabel()* command with LaTeX text. In LaTeX, a formula starts and ends with the $ symbol. Text between the $ signs will be formatted as an equation, which means that spaces are ignored. If you need a space you should use the tilde (∼) symbol between two words. We often want normal text in the label, and then use a superscript, for instance, in the unit. We can explicitly state that the text should be in normal roman-font text font by placing it in a \rm environment, *i.e.* between accolades. We also have to place an *r* before the label text to tell Python that this is a *raw* string so it does not interpret Python escape commands, such as \n that would make a new line, or \t that would insert a tab in Python. So the ylabel() statement could look like this:

```
>>> ylabel(r'${\rm Temperature~[}^\circ {\rm C]}$',fontsize=16)
<matplotlib.text.Text instance at 0xb4e93bec>
>>>
```

to get Temperature [°C], or

```
>>> ylabel(r'${\rm Wind~speed~[m~s}^{-1} {\rm ]}$',fontsize=16)
<matplotlib.text.Text instance at 0xb4e93bec>
>>>
```

to get Wind speed [m s^{-1}]. Note again the use of ∼ where we need spaces in the text. In these examples we also increased the font to 16 points by the fontsize statement in the ylabel() command, as this makes the figure better readable in your report. The result is shown in Figure 9.3. We also need to increase the font sizes of the x- and y-ticks, using *xticks(fontsize=15)* and *yticks(fontsize=15)* commands, and increase the linewidth, include a legend label and specify a legend. We should then save the figure in the image directory of our report. All commands to create Figure 9.4 are listed below (without Python responses).

```
>>> figure(1)
>>> plot(t_data[:,2],T, label='Air Temperature',linewidth=1.5)
>>> xlabel('Time',fontsize=16)
>>> ylabel(r'${\rm Temperature~[}^\circ {\rm C]}$',fontsize=16)
>>> xticks(fontsize=15)
>>> yticks(fontsize=15)
>>> legend()
>>> savefig('temperature.eps',dpi=300)
```

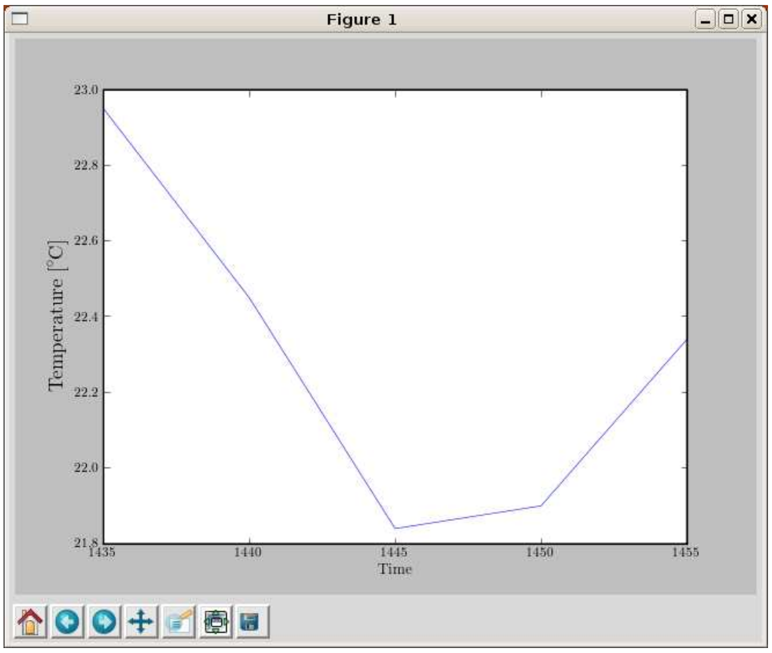


Figure 9.3: Plot of temperature data *versus* time, now with x- and y-axis labels, the latter with fontsize=16.

```
>>> show()
```

Inserting the saved figure (as encapsulated postscript, eps, file) in your report results in Figure 9.5.

You can also use markers, where the 'o' represents a closed circle, a '+' gives you a plus marker, etc. By default, markers are filled but you can create open dots and determine the fill and edge colours using the *facecolors* and *edgecolors* options. For example we can create a scatter plot graph displaying x and y values that we obtain from the *random* module in *scipy*. The *scipy.random.randn()* function gives you random samples from the standard normal distribution (a normal distribution whit average value of 0 and variance of 1). The s= option in the *pylab.scatter* function represents the size of the markers.

```
>>> import pylab
>>> import scipy
>>> x=scipy.random.randn(100)
>>> y=scipy.random.randn(100)
>>> a=scipy.random.randn(100)
>>> b=scipy.random.randn(100)
>>> figure(1)
>>> pylab.scatter(x, y, s=50, facecolors='none', edgecolors='b')
>>> pylab.scatter(a, b, s=70, facecolors='y', edgecolors='g')
```

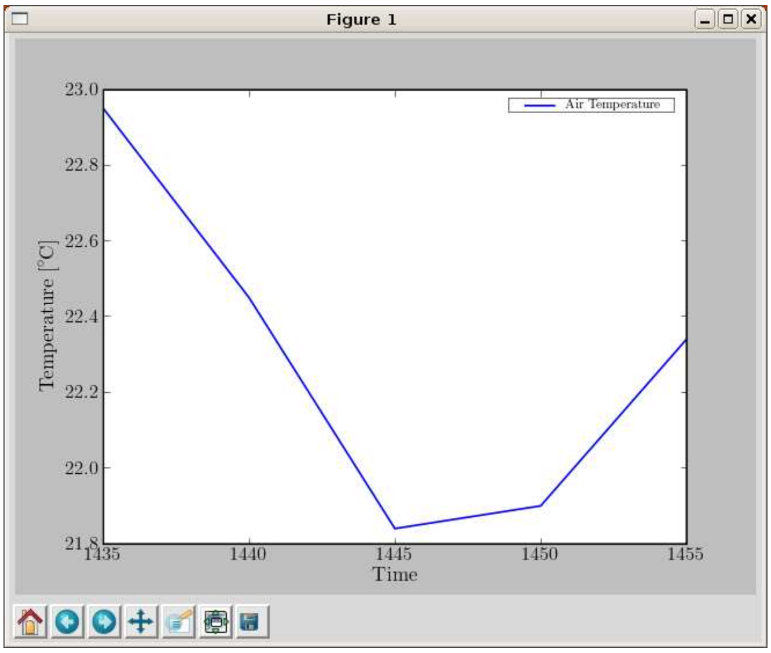


Figure 9.4: Plot of temperature data *versus* time, with x- and y-axis labels (fontsize 16), x- and y-ticks fontsize 15, a legend and a linewidth of 1.5 instead of the default 1.0.

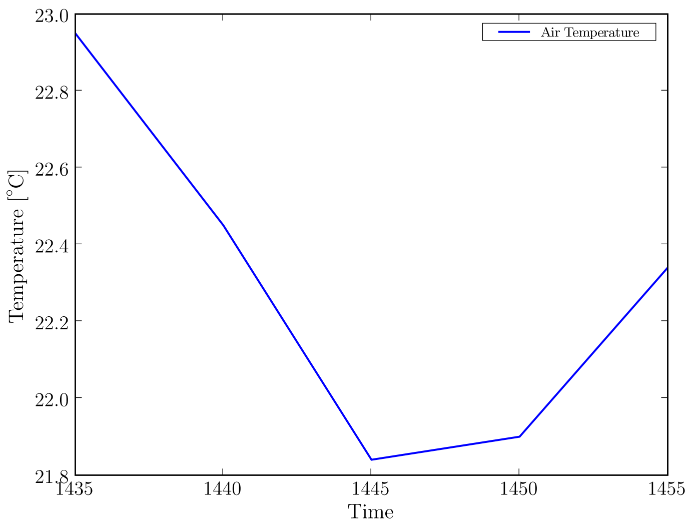


Figure 9.5: Saved EPS figure of temperature data *versus* time, with x- and y-axis labels (fontsize 16), x- and y-ticks fontsize 15, a legend and a linwidth of 1.5 instead of the default 1.0.

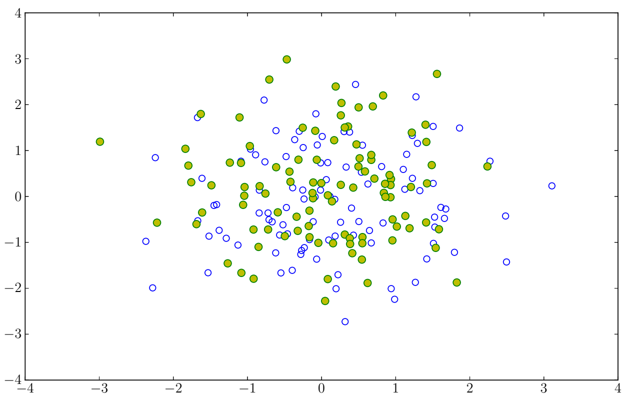


Figure 9.6: Example of the use of the pylab.scatter function with open and closed markers of different colours.

```
>>> pylab.show()
```

The above commands would result in a plot similar to that shown in Figure 9.6.

9.2 More complicated graphs

It is sometimes necessary to present your data in more complicated graphs. For instance, if you have spatially varying data (*e.g.* topography, groundwater head variations, variations in NO_3 concentrations) you often want to plot contours using these data. In these cases the data consist of x and y coordinates that determine the location, and a third variable (z) that varies spatially.

A contour plot can be made using the meshgrid() and contour() commands.

9.3 Interactive plotting of figures

Instead of the *show()* command that displays figures, we can also use the *ion()* function for plotting figures interactively while running your Python script. The *ion()* function is defined in the pylab and this module should therefore be imported before you can include the ion() statement in your script.

9.4 Changing text fonts in a figure

We can also change the default font and other parameters in one go by issueing the following command before we start creating the figure. We do this with the *rcParams.update()* function. In the script below we define a variable *newdefaults* that lists how figures are saved (.png, .eps, etc) by specifying the *backend* variable, and the font sizes of axis labels (*axes.labelsize* variable, ticks (*xtick.labelsize*, *ytick.labelsize*, text (*text.fontsize* and legend (*legend.fontsize*. We then use *rcParams.update(newdefaults)* to effectuate the update.

```
>>> newdefaults = {'fontname': 'Roman',
          'backend': 'eps',
          'axes.labelsize': 16,
          'text.fontsize': 16,
          'legend.fontsize': 15,
          'xtick.labelsize': 14,
          'ytick.labelsize': 14}
>>> rcParams.update(newdefaults)
```

9.5 Creating subplots and a second y-axis

If we want subplots in a figure, we can use the subplot() command:

```
>>> figure(2)   # Create second figure
>>> subplot(211)  # Plot temperature data in upper plot
>>> plot(t_data[:,2],T, label='Air Temperature',linewidth=1.5)
>>> ylabel(r'${\rm Temperature~[}^\circ {\rm C]}$')
>>> subplot(212) # Plot standard deviations in lower plot
>>> plot(t_data[:,2],t_data[:,4], label='St. dev. T',linewidth
    =1.5)
>>> ylabel(r'${\rm St.~dev.~Temperature~[}^\circ {\rm C]}$')
>>> xlabel('Time')
>>> savefig('temp2.eps',dpi=300)
```

This creates Figure 9.7.

For plotting time series with a large difference between y-values, such as for temperature (0–30 °C) and relative humidity (60–100%), it is often convenient to use a second y-axis for plotting the latter. This can be done with the *twinx()* command. Basically, twinx() creates a new set of axes over the existing figure. The temperature data would then plot on the first set of axes (left y-axis), whereas the relative humidity would be plotted on the second created set of axes (right y-axis).

The following script code would create a single figure with the temperature plotted on the left y-axis, and its standard deviation on the right axis as in Figure 9.8.

```
# Create figure 3, now with two y-axes
figure(3)
# create labels for the x- and left y-axis
xlabel(r'${\rm Time}$')
ylabel(r'${\rm Temperature~[}^\circ {\rm C]}$')
# Plot T versus time, use red colour (r) and round marker (o)
```

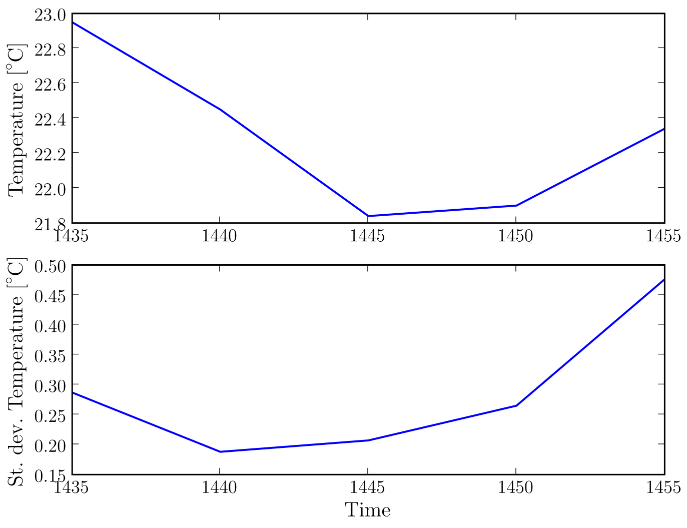


Figure 9.7: Saved EPS figure of temperature and standard deviation data *versus* time using the subplot() function.

```
plot(time,T,'ro',label='Temperature')
# Place a legend
legend(loc=1)
# Use twinx() to create a second set of axes on top of the first
    set
twinx()
# Create label for the right y-axis
ylabel(r'${\rm St.~dev.~Temperature~[}^\circ {\rm C]}$')
# Now plot standard deviation of T versus time
plot(time,stdT,label='St.dev')
# Place a legend for the second set of axes
legend(loc=2)
# Save the figure as EPS file
savefig('twoaxes.eps',dpi=300)
```

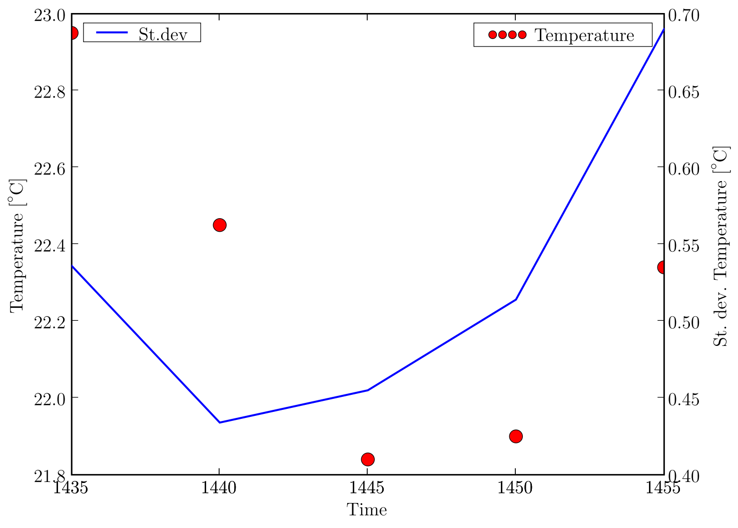


Figure 9.8: Saved EPS figure of time series of temperature and standard deviation of temperature using the twinx() function to create two y-axes.

Bibliography

R. Allen, L.S. Pereira, D. Raes, and M. Smith. *Crop Evaporation - Guidelines for computing crop water requirements*, volume 56 of *FAO Irrigation and Drainage Paper*. FAO, Rome, Italy, 1998. URL `http://www.fao.org/docrep/x0490e/x0490e00.htm`.

H.A.R. de Bruin. From Penman to Makkink. In J.C. Hooghart, editor, *Evaporation and Weather*, pages 5–31, The Hague, 1987. TNO Committee on Hydrological Research. Proceedings and Information 39.

H. L. Penman. Natural evaporation from open water, bare soil and grass. *Proceedings of the Royal Society*, 193:120–146, 1948. Series A.

H. L. Penman. Evaporation: An introductory survey. *Neth. J. Agric. Sci.*, 4:9–29, 1956.

H. L. Penman. Vegetation and hydrology. Technical communication 53, Commonwealth Agricultural Bureaux, Farnham Royal, Bucks, UK, 1963.

C. H. B. Priestley and R. J. Taylor. On the assessment of surface heat flux and evaporation using large-scale parameters. *Monthly Wheater Review*, 100:81–92, 1972.

J.D. Valiantzas. Simplified versions for the Penman evaporation equation using routine weather data. *Journal of Hydrology*, 331(3–4):690–702, 2006. doi: 10.1016/j.jhydrol.2006.06.012.

Index