

DRAFT Python for Science DRAFT

David J. Raymond
Physics Department and Geophysical Research Center
New Mexico Tech
Socorro, NM 87801
USA
raymond@kestrel.nmt.edu

November 1, 2009

Contents

1	Introduction	1
2	Standard Python	3
2.1	Using Python as a Desk Calculator	3
2.1.1	The math module	4
2.1.2	Variables	5
2.2	Scripts	6
2.2.1	Getting input	6
2.2.2	Making scripts executable	7
2.2.3	Operating system functions	9
2.3	Python Functions	9
2.3.1	Functions in modules	9
2.3.2	Keyword parameters	10
2.3.3	Complex data type	10
2.3.4	Scope of variables	11
2.4	Branching	12
2.5	Looping	13
2.5.1	the <i>while</i> loop	13
2.5.2	The <i>for</i> loop	15
2.5.3	Leaving loops early	17
2.6	Fun with sequences and dictionaries	17
2.6.1	Indexing and slicing of sequences	18
2.6.2	String methods	19
2.6.3	List methods	19
2.6.4	Sequence math	20
2.6.5	Dictionaries	21
2.6.6	Assignments and aliases	22
2.7	Object oriented programming in Python	23
2.7.1	Defining classes	23
2.7.2	Creating and working with class instances	24
2.7.3	Class inheritance	25
2.8	More input and output	26
2.8.1	General file handling	26
2.8.2	Standard input, output, and error	27
2.9	Further information	27

3	Numerical calculations with NumPy	29
3.1	Creating arrays	29
3.1.1	The <i>array</i> function	29
3.1.2	Creating one-dimensional sequences	30
3.1.3	The <i>zeros</i> and <i>ones</i> functions	31
3.1.4	Two-dimensional arrays with <i>meshgrid</i>	31
3.1.5	Array data types	32
3.2	Array indexing and looping	34
3.2.1	Basic indexing	34
3.2.2	Slicing	35
3.2.3	Views and copies	36
3.2.4	Reshaping arrays	37
3.2.5	Array looping	38
3.3	Array expressions and assignments	40
3.3.1	Broadcasting	41
3.4	Array methods	42
3.4.1	General methods	43
3.4.2	Mathematical methods	45
3.5	Array functions	46
3.6	Matrices	48
3.7	Masked arrays	48
3.8	Endianness	49
3.9	Further information	50
4	Graphics with Matplotlib	51
4.1	Line plots	51
4.1.1	Simple plots	51
4.1.2	Multiple lines and points	53
4.2	Log plots and histograms	54
4.3	Two-dimensional plots	56
4.3.1	Contour plots	57
4.3.2	Filled contours	59
4.3.3	Color mesh plots	59
4.3.4	Vector plots	60
4.3.5	Masked arrays and Matplotlib	62
4.4	Finer control of color and text	62
4.5	Combination plots	63
4.5.1	Multiple plots	63
4.5.2	Overlaying plots	65
4.6	Making plot output files	65
4.7	Further information	66

5	Accessing standard file formats	67
5.1	NetCDF3	67
5.1.1	Creating NetCDF files	68
5.1.2	Reading NetCDF files	70
5.2	NetCDF4	72
5.3	Candis	73
5.4	Further information	73

Chapter 1

Introduction

Python is an excellent language for doing many types of scientific (i. e., numerical) calculations. However, documentation for Python tends to be widely scattered and not very suitable for tutorial use. These notes attempt to document the scientific use of Python in a manner that allows the programmer with some knowledge of other computer languages to begin to do useful work quickly. The scientific use of Python is only presented for Unix-like systems, such as Unix itself and Linux. Apple's Mac OS-X mostly works the same way, but some gotchas may exist. Windows potentially presents many more difficulties. *Caveat emptor!*

This guide tries to present the features of a useful subset of Python in the rough order in which an increasingly sophisticated user will need them. The judgement of what to present comes largely from my own experience. Others may judge differently, so your mileage may vary.

The power of Python derives from the fact that external modules, some defined in computationally efficient compiled languages such as C and Fortran, can be accessed from Python. Since Python is an interpreted language, this speeds up computations over those done strictly in Python by an order of magnitude or more. It also provides access to some of the best numerical software which otherwise might remain inaccessible to most users. The modules *NumPy* (version 1.2 or greater) and *Matplotlib* (version 0.99 or greater) will be used extensively here. These modules respectively provide fast, array-based numerical tools and publication quality graphics.

Chapter 2

Standard Python

We touch first on those aspects of “standard” Python (including “standard” modules) which we need to do scientific work. Standard Python is a clean but powerful language and is relatively easy to learn.

2.1 Using Python as a Desk Calculator

As noted above, Python is an interpreted language, which means that you can type code directly into it and get the result immediately. This makes Python an excellent desk calculator. Start the Python interpreter at your prompt by typing `python`:

```
swallow$ python
Python 2.5.2 (r252:60911, Jan  4 2009, 17:40:26)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The prompt `>>>` tells you that python is ready for input. Type in any numeric expression and you should get an answer. For instance

```
>>> 3*(2 + 5.5)
22.5
>>>
```

The number 22.5 is the result of the expression. To finish with Python, type `^D` or `quit()`. The `^D` means hold down the *control* key and type `d`.

```
>>> quit()
swallow$
```

Here is one quirk of Python. Let’s try division:

```
>>> 5/3
1
>>> 5%3
```

```
2
>>> 4./3.
1.3333333333333333
>>>
```

Notice that the ratio of integers yields an integer, not the (possibly expected) floating point result. To do floating point division you need at least one of the numbers to be floating point to begin with. The `%` operator provides the remainder in integer division.

2.1.1 The math module

Let's try some trig:

```
>>> sin(0.01)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>>
```

Oops! Something went wrong. To use trigonometric or other common math functions, you need to load the *math* module:

```
>>> from math import *
>>> sin(0.01)
0.00999998333341666645
>>>
```

The `*` loads all the functions defined in the *math* module into the interpreter. Now your common math functions work:

```
>>> exp(3.)
20.085536923187668
>>> log(3.)
1.0986122886681098
>>> log10(3.)
0.47712125471966244
>>> abs(-2.)
2.0
>>> atan(1.)
0.78539816339744828
>>> atan2(-3.,-4.)
-2.4980915447965089
>>>
```

If the *math* module were loaded with the `import math` statement rather than `from math import *`, the math functions would all need the prefix `math.` to work:

```
>>> import math
>>> log(3.)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'log' is not defined
>>> math.log(3.)
1.0986122886681098
>>>
```

Note that angles are in radians, as in compiled languages such as C and Fortran. Also, by default, floating point numbers (i. e., numbers with decimal points) are “double precision” or 64 bits. Integers depend on the architecture of the machine being used, but are at least 32 bits. Floats and ints can be converted explicitly from one to the other:

```
>>> float(3)
3.0
>>> int(3.1)
3
>>> int(-3.1)
-3
>>>
```

Conversion to ints rounds toward zero for both positive and negative floats on this computer, but this is not guaranteed since it depends on computer hardware.

2.1.2 Variables

Sometimes one wants to use a certain number repeatedly. In this case it is possible to assign it to a *variable*. So, instead of typing

```
>>> 2*23.5
47.0
>>> 3*23.5
70.5
>>> 4*23.5
94.0
>>>
```

do instead

```
>>> const = 23.5
>>> 2*const
47.0
>>>
```

etc.

By the way, the last value returned by the interpreter may be retrieved, as it is stored in the variable `_`:

```
>>> "frog"
'frog'
>>> _
'frog'
>>>
```

2.2 Scripts

Sometimes one would like to package a set of commands to Python so that repeated typing of these commands is not needed. This can be done by creating a Python *script*. Using your favorite text editor, create a file called `trivial.py` with the following content:

```
# trivial1.py -- This is a trivial script.
a = 3.
result = a**3 + 2.*a**2 - 5.
print a, result
```

Then type `python trivial1.py` at the command line to execute the statements in this script, just as if you typed them directly into Python:

```
swallow$ python trivial1.py
3.0 40.0
```

The `print` statement prints out the values of the variables `a` and `result`. The operator `**` raises the variable before it to the power following it. Note everything following a `#` is treated as a *comment*. Comments are there for humans to read – Python ignores them.

You could make the output more informative by adding some descriptive information to the print statement:

```
print 'for a =', a, '    a**3 +2a**2 - 5 =', result
yields
    for a = 3.0    a**3 +2a**2 - 5 = 40.0
```

The modified print statement introduces a new data type in Python; any text enclosed between (single or double) quotes is called a *string*. More on this later.

2.2.1 Getting input

Actually, this script is pretty useless, since to get a different result, one would have to edit the script to change the value of `a`. However, a slight modification causes the script to input a value of `a` from the command line each time it is run:

```
# trivial2.py -- This is a trivial script.
import sys
print 'type a: ',
text = sys.stdin.readline()
```

```
a = float(text)
result = a**3 + 2.*a**2 - 5.
print 'for a =', a, '    a**3 +2a**2 - 5 =', result
```

Running this results in

```
swallow$ python trivial2.py
type a: 4
  for a = 4.0    a**3 +2a**2 - 5 = 91.0
swallow$
```

where the 4 is typed in response to the prompt `type a:` and stored as a string in the variable `text`. The call to the function `float()` converts this string to a float variable.

Information is read by the system on the standard input, which necessitates importing the module `sys` to enable this capability. Notice that we have used the import form `import sys` rather than the form we used earlier for the `math` module, which would have been `from sys import *`. Which form is used depends on whether one wants to include the name of the module as a prefix to calls to module functions or not, e. g., `sys.stdin` versus `stdin`. The advantage of the latter is less typing. The advantage of the former is that a possible pre-existing function `stdin` doesn't get clobbered.

The `stdin` method of `sys` has many sub-methods, of which `readline()` is just one. Each call reads an entire line of typing. The sub-method `readlines()` grabs all lines in the form of a *list* of lines:

```
linelist = sys.stdin.readlines()
nlines = len(linelist)
print linelist[0]
print linelist[1]
...
print linelist[nlines - 1]
print linelist[-1]
```

The list is a compound Python data type. The list can hold a bunch of individual elements consisting of constants or variables, including other lists. The number of elements is returned by the function `len()`, as shown above. The elements of a list can be accessed by the square bracket notation illustrated above. Indexing starts at zero and the last element of the list has an index equal to the number of elements minus one. The last element can also be indexed by `-1`, the second to the last by `-2`, etc. This indexing convention holds for compound objects throughout Python.

2.2.2 Making scripts executable

If you get tired of typing

```
swallow$ python trivial2.py
```

you can add the line

```
#!/usr/bin/python
```

to the beginning of the script and then make the script executable by typing

```
swallow$ chmod +x trivial.py
```

at the system prompt. Then the script can be run by simply typing

```
swallow$ trivial2.py
```

Change `/usr/bin/python` to whatever the full path is for python on your system.

An advantage of an executable Python script is that it presents an alternate way of entering input data into the script. For example, we can rewrite our trivial script as follows:

```
#!/usr/bin/python
# trivial3.py -- This is a trivial script.
import sys
print sys.argv # Included so we can see what is going on.
a = float(sys.argv[1])
result = a**3 + 2.*a**2 - 5.
print 'for a =', a, '    a**3 +2a**2 - 5 =', result
```

The variable `sys.argv` from the `sys` package is a list of the words typed on the command line, starting with the command itself. Several examples follow:

```
swallow$ trivial3.py
['./trivial3.py']
Traceback (most recent call last):
  File "./trivial3.py", line 5, in <module>
    a = float(sys.argv[1])
IndexError: list index out of range
```

In this case just the command is typed, and the returned list just contains the command as its first and only element. Indexing in Python starts from zero, so there is no `sys.argv[1]`, a fact noted and commented upon by the Python interpreter. Let us try instead

```
swallow$ trivial3.py 3
['./trivial3.py', '3']
for a = 3.0    a**3 +2a**2 - 5 = 40.0
```

The script converts the string `'3'` into the corresponding floating point number and proceeds as before. If we type

```
swallow$ trivial3.py "frogmorton"
['./trivial3.py', 'frogmorton']
Traceback (most recent call last):
  File "./trivial3.py", line 5, in <module>
    a = float(sys.argv[1])
ValueError: invalid literal for float(): frogmorton
```

the float conversion becomes problematic and Python again reports its distress. However, if we type

```
swallow$ trivial3.py 3 "frogmorton"
['./trivial3.py', '3', 'frogmorton']
for a = 3.0      a**3 +2a**2 - 5 = 40.0
```

Python is happy even though there is an extra command line argument which remains unused. Note that the print statement in this script is not really needed; it is there only to make it clear how the command line arguments are transmitted to the script.

2.2.3 Operating system functions

Various connections with the underlying operating system are available through the *os* module. The most interesting of these functions is the ability to run arbitrary commands in a sub-shell. For example:

```
>>> import os
>>> os.system("date")
Sat Oct  3 17:22:19 MDT 2009
0
>>>
```

The output of the command is printed and the exit status of the system call is returned.

2.3 Python Functions

In this section we begin to consider Python statements which control branching, looping and modularization of code. An important point to remember here is that Python uses indentation of code to define groups of related statements, or clauses. Less indentation ends the clause. Furthermore, the statements which introduce clauses always end with a colon. Examples will clarify this as we go along. It is better to use tabs rather than spaces for indentation as it is easier to make the indentation consistent.

2.3.1 Functions in modules

A function is a clause which can be called by the user or by other functions to execute a particular task. Functions can be defined interactively, but they are most useful when placed in special files called *modules*, which can be imported by the interpreter. Let's take as an example a function which solves the quadratic equation

$$ax^2 + bx + c = 0. \tag{2.1}$$

We all know that this equation has two solutions

$$x = \frac{-b \pm (b^2 - 4ac)^{1/2}}{2a}. \tag{2.2}$$

Consider the following script, named `quadratic1.py`:

```
# quadratic1.py -- Quadratic equation solver.
def qsolns(a,b,c):
    temp = b**2 - 4.*a*c
    x1 = (-b + temp**0.5)/(2.*a)
    x2 = (-b - temp**0.5)/(2.*a)
    return [x1, x2]
```

This script contains the definition of a single function `qsolns`. The three variables `a`, `b`, and `c` are transmitted to the body of the function, which computes the two solutions `x1` and `x2`. The two solutions are then combined into a list `[x1, x2]` and returned to the caller. To use this script, which is called a *module*, We use it from within Python as follows:

```
>>> import quadratic1
>>> quadratic1.qsolns(1.,4.,0.)
[0.0, -4.0]
>>>
```

Direct substitution verifies 0.0 and -4.0 as solutions to the original equation.

2.3.2 Keyword parameters

There is another way to call the function `qsolns`:

```
>>> quadratic1.qsolns(a = 1., b = 4., c = 0.)
[0.0, -4.0]
>>>
```

This makes the relationship between the function arguments and the values assigned to them more obvious. However, it also is more verbose. The first instance uses *positional parameters* the second uses *keyword parameters*. The two methods can be mixed as long as keyword parameters follow positional parameters:

```
>>> quadratic1.qsolns(1., b = 4., c = 0.)
[0.0, -4.0]
>>> quadratic1.qsolns(a = 1., b = 4., 0.)
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>> quadratic1.qsolns(c = 0., a = 1., b = 4.)
[0.0, -4.0]
>>>
```

As the last case shows, keyword parameters can appear in any order as long as all positional parameters appear first.

2.3.3 Complex data type

So far, so good. However, suppose we try the following:

```
>>> quadratic1.qsolns(1,4,6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "quadratic1.py", line 4, in qsolns
    x1 = (-b + temp**0.5)/(2.*a)
ValueError: negative number cannot be raised to a fractional power
>>>
```

Clearly the problem is that for the choice of (a, b, c) the solutions to the quadratic equation are complex rather than real.

One solution to this problem is to make the variable `temp` complex:

```
# quadratic3.py -- Quadratic equation solver.
def qsolns(a,b,c):
    temp = complex(b**2 - 4.*a*c, 0.)
    x1 = (-b + temp**0.5)/(2.*a)
    x2 = (-b - temp**0.5)/(2.*a)
    return [x1, x2]
```

Running this yields

```
>>> import quadratic3
>>> solns = quadratic3.qsolns(1.,4.,6.)
>>> print solns
[(-2+1.4142135623730951j), (-2-1.4142135623730951j)]
>>>
```

A complex number with a negative real part can be raised to a fractional power, the result of which is also complex. The occurrence of one complex variable in an expression makes all calculations complex, including the answers. Note the use of the `complex(real,imag)` function to construct a complex number or variable. If `c` is complex, the real and imaginary parts are given by `c.real` and `c.imag`. The complex conjugate of `c` is given by `c.conjugate()`. Finally, the absolute value is computed by `abs(c)`. For complex trig, log, and exponential functions, load the `cmath` module, which provides complex versions of most of the functions of the `math` module.

2.3.4 Scope of variables

Several comments about functions are worth making. Variables defined inside a function are generally invisible outside the function. However, variables outside the function (but in the same script or module) can be seen inside the function, but not assigned to, with one exception; if a variable is defined as *global*, it can be read and assigned to in both locations. Globals are generally discouraged, so we do not discuss them further here.

The values of the variables passed from the calling environment are available inside the function via the list in parentheses in the function definition line. Thus, the function `qsolns` has access to the values of `a`, `b`, and `c`.

The results of a function calculation can be accessed by the calling environment via the `return` statement in the function. In the case of `qsolns`, the list containing the roots of the quadratic equation is returned to the calling environment, as illustrated above. This highly channeled flow of information to and from a function may sound restrictive, but it is desirable from the point of view of keeping track of what is going on in a program.

Scripts can call functions defined within the script, and functions themselves can call other functions. However, for this to work, the definition of the function must occur before the call to the function – otherwise, Python doesn't know that the function exists.

2.4 Branching

An alternate approach to handling the quadratic equation problem is to test beforehand for various possibilities that might cause difficulties with the calculation using the *if* statement. Let us rewrite our function as follows:

```
# quadratic2.py -- Quadratic equation solver.
def qsolns(a,b,c):
    temp = b**2 - 4.*a*c
    if a == 0:
        #a = 0 is a special case
        if b != 0:
            return [-float(c) / b]
        return []
    elif temp < 0:
        #solutions are complex
        return []
    elif temp == 0:
        #only one solution
        return [(-b + temp**0.5)/(2.*a)]
    else:
        x1 = (-b + temp**0.5)/(2.*a)
        x2 = (-b - temp**0.5)/(2.*a)
        return [x1, x2]
```

If `a == 0` is true, i. e., if the variable `a` is equal to zero, then the clause of the `if` statement is executed. Alternatively, if this is false, but the `elif` condition is true, the clause of this statement is executed. If neither is true, the clause of the `else` statement is executed. This function illustrates the full range of possibilities of the *if* statement. For each `if` there are zero or more `elif` clauses. The `else` clause considers all other possibilities. Try this function to see what happens!

Comparison operators between numbers are `>`, `>=`, `==`, `!=`, `<=`, and `<`, the meanings of which are obvious except possibly `==` and `!=`; the first yields true if the two numbers are equal and false otherwise, whereas the second yields the reverse. Don't confuse `==` with `=` which tries to assign the right value to the left value, and is an error in this context! Note that if a single number or numerical expression is used in place of the boolean expression,

it is treated as false if it evaluates to zero and true if it is nonzero. Logical combinations of boolean expressions are produced with the *and* and *or* operators, **and** and **or**. Thus, $(4 > 3)$ **or** $(3 < 2)$ is true, while $(4 > 3)$ **and** $(3 < 2)$ is false. The parentheses are not needed in this context but are helpful to avoid ambiguity.

2.5 Looping

2.5.1 the *while* loop

Suppose we want to find the roots of some function $f(x)$. Newton's method is perhaps the simplest (and least reliable!) means of doing this. If x_0 is a first guess at the root, then a Taylor expansion of $f(x)$ about this point can be written to first order as

$$f(x) \approx f(x_0) + \frac{df}{dx}(x - x_0). \quad (2.3)$$

Setting this to zero to find the root of the linear approximation to $f(x)$ results in the equation

$$x = x_0 - \frac{f(x_0)}{(df/dx)}. \quad (2.4)$$

However, this is only an approximation, so this equation has to be recalculated a number of times, replacing x_0 by the new x each time until convergence occurs. Here is a simple Python function which does this:

```
# newton1.py -- A newton's method rootfinder.

def newton(xstart, fn):
    x = xstart
    eps = 0.0001
    xold = x + 10*eps
    delta = 0.1
    loops = 0
    maxloops = 100

    while (abs(x - xold) > eps) and (loops < maxloops):
        loops = loops + 1
        fval = fn(x)
        dfdx = (fn(x + delta) - fn(x))/delta
        xold = x
        x = xold - fval/dfdx
        print "loop =", loops, "-- x =", x, "-- f(x) =", fn(x)
```

This function uses the **while** statement – the clause of the statement is executed while the boolean expression following **while** is true. In this expression **x** is the current value of the variable x whereas **xold** is the previous value, x_0 . As long as the absolute value of the difference between these two is greater than some small number **eps**, this part of

the expression is true. Convergence is reached when x stops changing, which means that $\text{abs}(x - \text{xold})$ becomes really small. The other part simply limits the number of iterations of the loop to 100 in order to guard against a runaway loop – which is all too common with Newton’s method! (Recall that **and** indicates a logical *and*.)

The function definition has two arguments, the initial value or first guess for x and the name of the function for which we wish to find the roots. The latter must be defined separately. Notice that the initial value of `xold` is set far enough away from `x` so that the **while** statement executes at least once.

Let’s try out our new root finder! We define a cubic polynomial function $f(x)$ directly in the interpreter (notice the indentation needed after the **def** statement and the blank line at the end of the function) and then call `newton` with a variety of starting values. (Also notice that the name of the function in its definition need not be the same as the name of the function within `newton`. This is a general property of function arguments.)

```
>>> from newton1 import *
>>> def f(x):
...     return x**3 - 2.*x
...
>>> newton(0.5,f)
loop = 1 -- x = -0.302752293578 -- f(x) = 0.577754629433
loop = 2 -- x = 0.0171829183112 -- f(x) = -0.0343607633197
loop = 3 -- x = -0.000136369523294 -- f(x) = 0.000272739044053
loop = 4 -- x = 6.82459009549e-07 -- f(x) = -1.3649180191e-06
loop = 5 -- x = -3.42951282581e-09 -- f(x) = 6.85902565163e-09
>>> newton(4,f)
loop = 1 -- x = 2.81381063334 -- f(x) = 16.6508096258
loop = 2 -- x = 2.07726860993 -- f(x) = 4.80897005523
loop = 3 -- x = 1.66192579318 -- f(x) = 1.26638303704
loop = 4 -- x = 1.47554415592 -- f(x) = 0.261511511278
loop = 5 -- x = 1.42307768519 -- f(x) = 0.0357905433661
loop = 6 -- x = 1.41514604066 -- f(x) = 0.0037336030139
loop = 7 -- x = 1.4143056045 -- f(x) = 0.000368204435166
loop = 8 -- x = 1.41422258344 -- f(x) = 3.60845992158e-05
>>> newton(-2,f)
loop = 1 -- x = -1.57492029756 -- f(x) = -0.756550674276
loop = 2 -- x = -1.4229611678 -- f(x) = -0.0353157404884
loop = 3 -- x = -1.4133056401 -- f(x) = 0.00362819253248
loop = 4 -- x = -1.41431958126 -- f(x) = -0.000424123216164
loop = 5 -- x = -1.41420132921 -- f(x) = 4.89320029273e-05
loop = 6 -- x = -1.41421497589 -- f(x) = -5.65406673259e-06
>>>
```

By a suitable choice of starting values, we have found all three roots of the polynomial, as indicated by the near-zero values of the functions in each case at the end of the iterations.

2.5.2 The *for* loop

Here is an alternate Newton's method solver which uses Python's `for` loop rather than a `while` loop:

```
# newton2.py -- A newton's method rootfinder.

def newton(xstart, fn):
    x = xstart
    eps = 0.0001
    xold = x + 10*eps
    delta = 0.1
    maxloops = 10

    for loops in range(maxloops):
        fval = fn(x)
        dfdx = (fn(x + delta) - fn(x))/delta
        xold = x
        x = xold - fval/dfdx
        print "loop =", loops, "-- x =", x, "-- f(x) =", fn(x)
        if (abs(x - xold) < eps) and (abs(fval) < eps):
            break
    else:
        print "no solution!"
```

This solver illustrates several new Python features. First, the `for` statement executes the associated clause for each value that the *loop variable* – `loops` in this case – takes on. The variable after `in` is formally known as a *sequence*. A sequence is nothing more than an ordered collection of things; `loops` takes on the value of each element of this sequence in turn as the looping proceeds.

The easiest way to write a sequence is via a `list` – an example would be `[0, 1, 2, 3]`. The elements of the list don't need to be integers; they can be any legal Python variables or constants, including other lists. They don't even have to all be of the same type. Alternate sequences are strings: `'abcdefg'`; or a tuple: `(0, 1, 2, 3)`. A tuple is just like a list, but it cannot be changed in the way that a list can – i. e., it is immutable – think of it as a list constant. A string is just an immutable sequence of characters.

The sequence in the above example is the function *range*, which generates a list of integers. For example,

```
range(3) = [0, 1, 2],
```

```
while
```

```
range(1,3) = [1, 2],
```

```
and
```

```
range(1,6,2) = [1, 3, 5].
```

Thus, our for statement sets `loops` successively to elements of the list

```
range(maxloops) = [0, 1, 2, ..., maxloops - 1]
```

The range function generates an actual list which the for loop steps through. Sometimes it is more efficient (as when the number of loop iterations is large) to use the *xrange* function in place of the range function in a for loop. This yields the same results as the range function, but it creates the sequence over which looping occurs “on the fly”, one element at a time. The *xrange* function can therefore only be used to replace the range function in for statements, not more generally. The xrange function is an example of an *iterator* which may be used in place of a real sequence in a for loop.

The test for convergence of the iteration in the above example is more sophisticated than in the previous case; not only does `x - xold` have to be sufficiently small, the function evaluated at `x` must be close to zero as well. When this occurs, the loop terminates early courtesy of the `break` statement. The `else` clause is only executed if the for loop terminates without executing a break statement. Here we use it to alert the user that convergence was not attained. The `break` and `else` statements can be used analogously with the `while` loop also. The `else` is optional in both cases.

So, how does our new Newton’s method solver work? Below it is applied to two functions, $f(x) = x^2 - 4$ and $f(x) = x^2 + 4$. The former has real roots, which `newton` can find. However the roots of the latter are complex, and hence are not found. (Newton’s method can be made to work for complex analytic functions as well. Try invoking `newton` with an initial `x = (1 + 3j)!`)

```
>>> from newton2 import *
>>> def func(x):
...     return x**2 - 4.
...
>>> newton(3,func)
loop = 0 -- x = 2.18032786885 -- f(x) = 0.753829615695
loop = 1 -- x = 2.01133262241 -- f(x) = 0.045458917965
loop = 2 -- x = 2.0003060376 -- f(x) = 0.00122424405357
loop = 3 -- x = 2.00000748606 -- f(x) = 2.99442871095e-05
loop = 4 -- x = 2.0000001826 -- f(x) = 7.30399140281e-07
>>>
>>> def func2(x):
...     return x**2 + 4.
...
>>> newton(3,func2)
loop = 0 -- x = 0.868852459016 -- f(x) = 4.75490459554
loop = 1 -- x = -1.7185621737 -- f(x) = 6.95345594488
loop = 2 -- x = 0.365104846463 -- f(x) = 4.13330154891
loop = 3 -- x = -4.61351872796 -- f(x) = 25.2845550532
loop = 4 -- x = -1.84322714371 -- f(x) = 7.39748630332
```

```
loop = 5 -- x = 0.21939117245 -- f(x) = 4.04813248655
loop = 6 -- x = -7.29409275083 -- f(x) = 57.2037890577
loop = 7 -- x = -3.34578679829 -- f(x) = 15.1942892996
loop = 8 -- x = -1.0406787574 -- f(x) = 5.0830122761
loop = 9 -- x = 1.52474027382 -- f(x) = 6.3248329026
no solution!
>>>
```

2.5.3 Leaving loops early

Two statements change execution flow in Python loops, the *break* and *continue* statements:

```
>>> for i in range(5):
...     if i == 3:
...         break
...     print i
...
0
1
2
>>> for i in range(5):
...     if i == 3:
...         continue
...     print i
...
0
1
2
4
>>>
```

Note that the `break` statement throws the program out of the innermost loop when it is encountered, as illustrated in the previous sub-section. The `continue` statement interrupts the current iteration of the loop and goes on to the next. The `pass` statement does nothing at all! The purpose of this last statement is solely as a placeholder.

2.6 Fun with sequences and dictionaries

Many of Python's normal applications involve operations on text. Such text operations are typically not used much in numerical programming, so we will only touch lightly on this subject. The online documentation for Python is particularly complete and extensive for use of strings, lists, tuples, and dictionaries in text operations, so further information is readily available. Nevertheless, sometimes these language elements come in handy in scientific work, so we will spend some time on them.

First some definitions:

- String: A string is just a sequence of characters surrounded by single or double quotes (no, it doesn't matter!): `x = 'abcde'`. Strings cannot be changed – they are *immutable*.
- List: A list is a sequence of items of any kind separated by commas and enclosed in square brackets: `x = [1, 2.0, 'three']`. The elements of lists can be changed – lists are *mutable*.
- Tuple: A tuple is like a list enclosed in parentheses rather than square brackets. The only difference is that tuples are immutable.
- Dictionaries: A dictionary is a collection key-value combinations. The keys are immutable objects, generally strings or numbers. The values can be anything. Dictionaries are delimited by curly braces:
`x = {'dog': 'animal', 'cat' : 2.1, 'rose' : (1+2j)}`.
 A dictionary is not a sequence, so its element pairs cannot be accessed by the type of indexing used with lists, tuples, and strings. Dictionaries have their own methods of construction, modification, and access.

2.6.1 Indexing and slicing of sequences

Indexing, which like the C language, starts at zero, allows one to extract individual elements or subsets of sequences. Demonstrating this on strings is easier than explaining it:

```
>>> x = 'abcdefg'
>>> x[0]
'a'
>>> x[1]
'b'
>>> x[-1]
'g'
>>> x[-2]
'f'
>>> x[0:2]
'ab'
>>> x[:]
'abcdefg'
>>> x[0:]
'abcdefg'
>>> x[:-1]
'abcdef'
>>> x[0:6:2]
'ace'
>>>
```

The colon-based notation returns a *slice* of the initial sequence, or a sequence of elements starting with that indexed by the number before the colon and ending with the that indexed by the number after the colon *minus one*. The negative index notation is a clever way of

indexing back from the end of the sequence. The optional third index is the *stride* – for instance, the 2 tells Python to jump 2 elements per step in marching through the sequence rather than the default 1.

One slight difference between indexing of lists and strings is that a simple string index (i. e., not a slice) returns a string consisting of a single character, whereas a simple index of a list returns the corresponding element of the list, not a list consisting of a single element. However, a slice referencing a single list element returns a list consisting of that single element:

```
>>> y = [0, 1, 2, 3]
>>> y[0:2]
[0, 1]
>>> y[0]
0
>>> y[0:1]
[0]
>>>
```

Tuples work just like lists except that the notation for a tuple consisting of just one element is (for example) (5,), not (5). The latter is just an integer enclosed in parentheses.

2.6.2 String methods

Python is an object oriented language, so data elements contain not only data, but useful *methods* which perform common methods of data manipulation (see the next section). For instance:

```
>>> x = 'I found a dog.'
>>> x.find('dog')
10
>>> x.find('frog')
-1
>>> x.split(' ')
['I', 'found', 'a', 'dog.']
>>> x.partition('found')
('I ', 'found', ' a dog.')
>>> x.replace('dog','cat')
'I found a cat.'
>>> x = 'I+found+a+dog'
>>> x.split('+')
['I', 'found', 'a', 'dog']
>>>
```

There are many more methods for strings; these are just some of the most useful.

2.6.3 List methods

Python lists have some methods specific to them. Some of the most useful are illustrated below.

```

>>> x = [1, 2, 3, 4, 5]
>>> x.append(6)
>>> x
[1, 2, 3, 4, 5, 6]
>>> x.insert(2,2.5)
>>> x
[1, 2, 2.5, 3, 4, 5, 6]
>>> x.remove(2.5)
>>> x
[1, 2, 3, 4, 5, 6]
>>> y = x.pop()
>>> y
6
>>> x
[1, 2, 3, 4, 5]
>>> x.reverse()
>>> x
[5, 4, 3, 2, 1]
>>> x.sort()
>>> x
[1, 2, 3, 4, 5]
>>>

```

2.6.4 Sequence math

The addition + and multiplication * operators have straightforward meanings with sequences:

```

>>> 'I found ' + 'a dog.'
'I found a dog.'
>>> 'dog '*3
'dog dog dog '
>>> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
>>> 2*[1, 2, 3]
[1, 2, 3, 1, 2, 3]
>>>

```

The % operator is more complex, but quite useful for incorporating numbers and other strings into strings in a specified format:

```

>>> a = 'I found %d dogs.' % (3)
>>> a
'I found 3 dogs.'
>>> 'The %s weighed %f grams.' % ('mouse', 43.5)
'The mouse weighed 43.500000 grams.'
>>> 'The %s weighed %e grams.' % ('mouse', 43.5)

```

```
'The mouse weighed 4.350000e+01 grams.'
>>> 'The %s weighed %.1f grams.' % ('mouse', 43.5)
'The mouse weighed 43.5 grams.'
>>> 'The speed of light is %.2e m/s.' % (3e8)
'The speed of light is 3.00e+08 m/s.'
>>>
```

The string contains C-like format statements, each preceded by a percent sign `%`. A percent sign `%` also separates this string from a tuple, the elements of which can be strings or numbers, but must match the corresponding types in the initial string. The first example above reminds us that the results of a string operation can be assigned to a variable, which can then be printed (implicitly or explicitly).

Finally, it is easy to find the length of sequences with the function `len()`:

```
>>> z = 'abcde'
>>> len(z)
5
>>> x = [0, 2, 4, 6]
>>> len(x)
4
>>>
```

Likewise, it is easy to determine whether a particular element exists in a sequence using the `in` operator:

```
>>> x = [1,2,3,4,5,6]
>>> 1 in x
True
>>> 1.5 in x
False
>>> 'a' in 'abc'
True
>>> 'x' in 'abc'
False
>>>
```

Logical expressions involving the `in` operator can be used in `if` and `while` statements to control the flow of the program.

2.6.5 Dictionaries

The *dictionary* is a Python type consisting of a collection of *key:value* pairs. Its main use is to construct a kind of data base. For instance, the dictionary called `phones` below contains peoples' names (the keys) associated with their phone numbers (the values). Dictionaries are displayed with surrounding curly braces and the key:value pairs are separated by commas.

```
>>> phones = {'george': '835-4427', 'sandy': '838-1192'}
>>> phones['george']
'835-4427'
>>> phones['harry'] = '892-9553'
>>> phones['harry']
'892-9553'
>>> phones
{'george': '835-4427', 'sandy': '838-1192', 'harry': '892-9553'}
>>> phones.keys()
['george', 'sandy', 'harry']
>>> del phones['sandy']
>>> phones
{'george': '835-4427', 'harry': '892-9553'}
>>> 'harry' in phones
True
>>>
```

As shown, indexing a dictionary by a key returns the value as long as the specified key exists in the dictionary. New entries in the dictionary may be made by assigning the value of the new key:value pair to the dictionary indexed by the new key. The `del` command removes the indexed key, as shown above. The `key` method for dictionaries illustrated above returns a list of the keys in the dictionary. Finally, the `in` operator can be used to see if a dictionary has a particular key.

Keys in dictionaries must be immutable, and are most commonly strings. Keys cannot appear more than once. The value can be any python data type, and different keys can have values of different types associated with them.

Dictionaries are not sequences, as the key:element combinations are not guaranteed to occur in any particular order. Thus, dictionaries cannot be used directly for iteration in `for` loops. However, it is possible to loop over the elements of a dictionary, returning each key and value successively, using the dictionary's `iteritems` method, which effectively converts the dictionary into a list of the dictionary's entries. Using the `phones` dictionary defined above:

```
>>> for k, v in phones.iteritems():
...     print k, v
...
george 835-4427
harry 892-9553
>>>
```

2.6.6 Assignments and aliases

Assignments involving lists and dictionaries can have some possibly unexpected results. Consider the following:

```
>>> a = ["a", "b", "c"]
```

```
>>> b = a
>>> c = a[:]
>>> a
['a', 'b', 'c']
>>> b
['a', 'b', 'c']
>>> c
['a', 'b', 'c']
>>> a.append("d")
>>> a
['a', 'b', 'c', 'd']
>>> b
['a', 'b', 'c', 'd']
>>> c
['a', 'b', 'c']
>>>
```

The assignment `b = a` just makes the variable `b` an alias of the variable `a`, so that when `a` is changed by the `append` method, `b` changes correspondingly. However, the assignment `c = a[:]` constructs a new list `c` by extracting the contents of list `a` element by element. Thus, when `a` changes, `c` does not. Any assignment other than a simple “`a = b`” causes this data transfer to happen, creating a new object instead of an alias to the old object, even if the net effect is to make the new object identical to the old object, as occurs with `c = a[:]`.

2.7 Object oriented programming in Python

You may never write an object oriented program in Python as a scientific programmer. Nevertheless, it is important to have a basic understanding of how object oriented programming works in this language, since the construct is used so frequently.

2.7.1 Defining classes

The basic notion in object oriented programming is the *class*. Think of a class as a blueprint of a data structure and *methods* for operating on this structure. For instance, consider the following module named `workstuff.py` which contains the class definition `worker_pay`:

```
class worker_pay:
    version = "1.2.4"

    def __init__(self):
        self.data = {}

    def add_worker(self, name, pay):
        self.data[name] = pay
```

```

def list_worker(self,name):
    return self.data[name]

def list_all(self):
    return self.data.keys()

```

The defs inside the class definition define the class methods, which are really just functions.

2.7.2 Creating and working with class instances

Consider now the sequence of statements shown below:

```

>>> import workstuff
>>> x = workstuff.worker_pay()
>>> x.add_worker("george", 3000)
>>> x.add_worker("frank", 2500)
>>> x.list_all()
['frank', 'george']
>>> x.list_worker("george")
3000
>>> a = x.list_worker("george")
>>> a
3000
>>> x.data
{'frank': 2500, 'george': 3000}
>>>

```

The first statement simply imports the module containing the class definition. The next statement `x = workstuff.worker_pay()` creates an *instance* `x` of the class `worker_pay`. This instance contains information which is manipulated by the four methods defined in the class. The first method, `__init__(self)` is a specially named method which is invoked automatically when an instance of the class is created. In this case it assigns an empty dictionary to the variable `data` contained in the instance, though in fact it could do any number of things. The other methods respectively add data (worker names and their salaries) to the dictionary, return particular worker's salaries, and return the names of all workers. The returned values can be assigned to other variables.

Notice that the first argument in the definitions of the methods, `self`, is omitted in the method invocations. `Self` in the class definition lets Python know that the method for the object `x` should be applied to the class instance `x` itself. There is nothing magic about the name "self"; it is just convention to use this terminology and it could be any other name defined as the first argument of a method and used consistently in the method body to indicate the invoked object.

The variable `data`, which is defined in the invocation of the instance `x`, can be accessed from the outside via the statement `x.data`. Normally it is considered bad form to take advantage of this in object oriented programming; all access to the data in an object should normally be indirect via methods. Python depends on the "honor system" to enforce this programming code of conduct!

One other type of information in a class may be accessed; variables defined inside the class definition but outside of any method. Note the assignment statement `version = '1.2.4'` in our class. The variable is called an *attribute* of the class. Its value may be obtained using notation similar to that of a method invocation, but without the (possibly empty) argument list, as illustrated below:

```
>>> x = workstuff.worker_pay()
>>> x.version
'1.2.4'
>>> workstuff.worker_pay.version
'1.2.4'
>>>
```

Attributes are used represent constant values specified in the class, in this case the version of the class. The attribute may be obtained either from the instance or from the class definition itself, as the second example illustrates above.

Multiple instances of our class can be invoked:

```
>>> y = workstuff.worker_pay()
>>> y.add_worker("mary", 5000)
>>> x.list_all()
['frank', 'george']
>>> y.list_all()
['mary']
>>>
```

The instance `y` shares methods with `x`, but the data are completely independent of each other. Thus, one could use this single class to keep track of employees and salaries in different departments by invoking multiple instances.

2.7.3 Class inheritance

Classes in Python can *inherit* methods from other classes. For instance, a class named `matrix` may be a two-dimensional subclass of a more general class called `array`. Assuming `array` has been previously defined, `matrix` would inherit all the methods defined in `array`. The sub-class `matrix` could then define additional methods specific to this class, e. g., matrix multiplication or computation of eigenvalues, in addition to taking advantage of methods defined in the super-class. Methods in the super-class could even be replaced by method definitions in the sub-class with the same name if appropriate. The way inheritance is implemented is to define the sub-class with a reference to the super-class as follows:

```
>>> def subclass(superclass):
>>>     ...
```

The super-class may itself be a sub-class of a super-super-class, etc.

Nothing in object-oriented programming cannot be done with, say, functions. However, it does have the advantage of keeping structured data and the methods used to operate on the data together. You can choose whether to use it or not in Python.

2.8 More input and output

We learned a bit about the `readline()` and `print` statements earlier. We now expand a bit on this.

2.8.1 General file handling

In order to read or write from a file, we first need to open it. When we are finished using it, we need to close the file, especially if we have written to it. To open a file for writing, we use the `open` command as follow:

```
>>> f = open("testfile.txt", "w")
>>> f.write("This is a test string\n")
>>> f.close()
>>>
```

The first argument of the `open` command is the name of the file to which we are writing and the second string, `'w'` tells Python to create a new, empty file with the specified file name, which is ready to be written to. The `open` command returns a *file object* which has several methods associated with it. The method `write()` writes the variable enclosed in the parentheses to the file in question. We can invoke the `write` methods as many times as we want, before closing the file with the `close()` method. The file name string can actually contain a full path such as, for instance, `''/home/raymond/testfile.txt''`.

Note that the special character represented by the `\n` is a “newline” character, actually an ASCII linefeed. This character is used to terminate lines in Unix and Linux systems. Windows and Macintosh systems each use different characters for this, which is one of the reasons why interchanging text between these systems and Unix/Linux can be so annoying.

Let’s see how we did; we attempt to read the file that we wrote:

```
>>> g = open("testfile.txt", "r")
>>> x = g.read()
>>> x
'This is a test string\n'
>>> g.close()
>>>
```

Success! The file we wrote was where we intended to put it (the current working directory) and had the desired content, a fact that can be verified independently with a text editor. In this case we opened the existing file for read-only access with the `'r'` argument in the `open` statement. Unlike the `readline()` method we discussed earlier, which reads a single line of text, the `read()` with no argument reads the entire file, transferring it to the variable `x` in the above script. What does a numerical argument do? Try the above code with `x = g.read(7)` and find out!

The above `read()` and `write()` methods can actually read and write data in arbitrary binary form, not just text strings. However, beware – on Macs and Windows machines, some extra fiddling has to be done to be sure binary data are not corrupted.

2.8.2 Standard input, output, and error

The `sys` module pre-defines three file objects which are always open and need not be closed, `stdin`, `stdout`, and `stderr`. These correspond to the standard Unix/Linux input-output streams of the same name; `stdin` reads typed input from the keyboard and both `stdout` and `stderr` write output to the terminal. The latter is by custom reserved for error messages which often need to be kept separate from the main `stdout` data stream. These data streams can be redirected to access files in the usual Unix/Linux fashion.

Here are some examples of the use of these data streams:

```
>>> import sys
>>> sys.stderr.write("ugh!")
ugh!>>>
>>> sys.stdout.write("gerbils\n")
gerbils
>>> x = sys.stdin.readline()
This is a line to standard input.
>>> x
'This is a line to standard input.\n'
>>>
```

Note the role of the “\n” in these examples.

2.9 Further information

The Python home page <http://www.python.org/> is an excellent source of documentation for standard Python and Python libraries. Look particularly at the Tutorial <http://docs.python.org/tutorial/> and the Library Reference <http://docs.python.org/library/>. Note that this documentation refers to the 2.X versions of Python, 2.5 in particular. The 3.X versions are still somewhat experimental and not all modules discussed here have been ported to 3.X yet.

Chapter 3

Numerical calculations with NumPy

NumPy (numerical python) is a module which was created allow efficient numerical calculations on multi-dimensional arrays of numbers from within Python. It is derived from the merger of two earlier modules named *Numeric* and *Numarray*. The actual work is done by calls to routines written in the Fortran and C languages.

NumPy defines a new data type called the *ndarray* or n-dimensional array. (We refer to them here simply as *arrays*. (Another Python module called *array* defines one-dimensional arrays, so don't confuse the arrays of NumPy with this other type.) It also defines functions of *ndarrays* (*ufuncs* or universal functions) which operate on each element of the array. Replacements for the standard functions of the *math* module exist. (Math module functions cannot be used directly on *ndarrays* because they only accept scalar, not array arguments.) The elements of *ndarrays* can be (among other things) integers, floats, and complex numbers of various sizes. When the NumPy package is loaded, *ndarrays* become as much a part of the Python language as standard Python data types such as lists and dictionaries.

To load NumPy, import the NumPy module:

```
>>> from numpy import *
>>>
```

This allows NumPy functions to be used without qualifying them with the prefix *numpy*. Alternatively, if NumPy names might conflict with names from other modules such as the *math* module, then the alternate import method

```
>>> import numpy
>>>
```

may be used. In the first case the NumPy version of the sine function would just be *sin*, whereas in the second case it would be *numpy.sin*. For brevity we will use the first form in these notes.

3.1 Creating arrays

3.1.1 The *array* function

There are many ways to create arrays in NumPy. The simplest is to use the *array* function to make a direct definition:

```

>>> from numpy import *
>>> a = array([[1., 2., 3.], [4., 5., 6.]])
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a.shape
(2, 3)
>>>

```

The syntax of the argument of the `array` function looks like nested lists of numbers with the level of nesting being equal to the dimensionality of the array – 2 in the above case. The attribute `shape` returns a tuple which gives the size of the array along each dimension axis. Consistent with Python indexing, the numbering of successive axes starts at 0, so the size along the zero axis is 2 and the size along the 1 axis is 3.

3.1.2 Creating one-dimensional sequences

The `arange` function creates a one-dimensional array consisting of a sequence of numbers:

```

>>> b = arange(0,11)
>>> b
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> c = arange(0.,2.,0.4)
>>> c
array([ 0. ,  0.4,  0.8,  1.2,  1.6])
>>>

```

The first two arguments of `arange` are the start and stop values. Note that the sequence stops *before* the stop value, as with the standard Python `range` function. The third argument is the step value in the sequence with a default value of 1.

Floating point arithmetic causes a potential pitfall with non-integer values of the step parameter. Look at the following example:

```

>>> arange(0.,2.1,0.3)
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8,  2.1])

```

One would expect the last value in this array to be 1.8, but it is 2.1. However, with the miniscule inaccuracies in floating point arithmetic, the final value is actually a tiny bit less than 2.1, an inaccuracy that is not represented in the printed output. Thus, the rule that the sequence stops before the stop value is reached is technically obeyed, though the result is probably not what was expected. Be forewarned and don't set your stop value to an element of the sequence if you are using floating point numbers! Make it one sequence element beyond the last desired element minus a small number, say, half the step size.

An unambiguous alternative to `arange` in such situations is the function `linspace(start, stop, number)`. this function returns an array with number evenly spaced elements with the specified starting and stopping values:

```
>>> a = linspace(0.,3.5,8)
>>> a
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5])
>>>
```

3.1.3 The *zeros* and *ones* functions

A couple of other methods for generating arrays are provided by the *zeros* and *ones* functions. Observe:

```
>>> zeros((2,4))
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> ones((3,3,3))
array([[[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]],
       [[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]],
       [[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
>>>
```

The tuple in the argument list of these functions defines the shape of the array. The arrays are filled with the values indicated by the function names.

3.1.4 Two-dimensional arrays with *meshgrid*

One often deals with fields in an N -dimensional vector space as approximated by arrays defined over an N -dimensional rectangular grid. With each grid dimension one defines a one-dimensional array containing “position” values along the corresponding axis. It is useful to define N -dimensional arrays over the vector space, one for each dimension, containing these position values. Such *meshgrid* arrays can be used as inputs to construct fields which are functions of position.

For the special case of two dimensions, NumPy provides a convenient function for generating meshgrid arrays, called, appropriately, *meshgrid*. Meshgrid returns the two desired meshgrid arrays in a tuple:

```
>>> x = arange(0.,5.1)
>>> y = arange(0.,3.1)
>>> (X,Y) = meshgrid(x,y)
>>> X
```

```

array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.]])
>>> Y
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.,  3.]])
>>> a = X*X + Y*Y
>>> a
array([[ 0.,  1.,  4.,  9., 16., 25.],
       [ 1.,  2.,  5., 10., 17., 26.],
       [ 4.,  5.,  8., 13., 20., 29.],
       [ 9., 10., 13., 18., 25., 34.]])
>>> b = x*x + y*y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape
>>>

```

The one-dimensional arrays `x` and `y` contain the position values along the x and y axes, while the two-dimensional arrays `X` and `Y` (which are distinct from `x` and `y` because Python variable names are case-sensitive) define these position values at each point in the vector space. Note how the array `a` can be defined in terms of `X` and `Y`, but the corresponding attempt to define `b` in terms of `x` and `y` fails.

Notice that variations in the first argument of `meshgrid` correspond to variations in the second index of `X`, `Y`, and `a`. This curious inversion probably reflects the influence of Matlab on the world of numerical computing. However, this convention is used consistently in the Matplotlib plotting module discussed in the next section, so there is no getting away from it. It does have the virtue that the first argument of `meshgrid` (the x axis by convention) is the dimension that corresponds to the horizontal axis in Matplotlib. However, it may not always represent the best way to organize dimensions in numerical computing.

3.1.5 Array data types

Some of the arrays created above contain integers and some contain floating point values. NumPy arrays can be made up of a variety of different numerical types, though all elements of a given array must be of the same type. The default float type in Python contains 64 bits (like a C-language *double*) and the default integer type generally contains 32 or 64 bits, depending on the architecture of the underlying computer. The type of the elements in an array can be defined upon array creation using the above functions by including the positional parameter *dtype*:

```

>>> a = array([1, 2], dtype = float64)
>>> a

```

```

array([ 1.,  2.])
>>> a = ones((2,3), dtype = complex128)
>>> a
array([[ 1.+0.j,  1.+0.j,  1.+0.j],
       [ 1.+0.j,  1.+0.j,  1.+0.j]])
>>> a = array([1.3, 2.7], dtype = int32)
>>> a
array([1, 2])
>>>

```

The data types most useful for numerical work are int8, int16, int32, int64, float32, float64, complex64, complex128. The number indicates the number of bits, with, for instance, complex64 consisting of two float32s.

It is easy to determine the the data type in NumPy arrays, as this type is an attribute of of these objects:

```

>>> a = array([[2.1, 3.5, 7.0], [10.4, 12.9, 5.1]])
>>> a
array([[ 2.1,  3.5,  7. ],
       [10.4, 12.9,  5.1]])
>>> a.dtype
dtype('float64')
>>> b = arange(0,5)
>>> b
array([0, 1, 2, 3, 4])
>>> b.dtype
dtype('int32')
>>> c = array([0 + 1j, 2 + 4j])
>>> c.dtype
dtype('complex128')
>>>

```

Generally the data type of array elements is obvious when the arrays are created by the simple methods illustrated in this section. However, if you are confronted with a “mystery array”, the dtype attribute allows you obtain its type. (Notice, by the way, that even though integers are used for the real and imaginary parts of the complex data elements in *c* above, the type still comes out “complex”, which is composed of two floats. In other words, an automatic conversion from int to float takes place.)

It is possible to convert arrays from one type to another using the *astype* method:

```

>>> from numpy import *
>>> a = arange(0,5)
>>> a.dtype
dtype('int32')
>>> b = a.astype('float32')
>>> b

```

```

array([ 0.,  1.,  2.,  3.,  4.], dtype=float32)
>>> c = a.astype('float64')
>>> c
array([ 0.,  1.,  2.,  3.,  4.])
>>>

```

In many cases data are imported in the form of an array from some outside source. In this case one need not create the array “by hand”.

3.2 Array indexing and looping

3.2.1 Basic indexing

Individual elements and sets of elements are extracted from an array by indexing. NumPy adopts and extends the indexing methods used in standard Python for strings and lists.

```

>>> a = array([[2.1, 3.5, 7.0], [10.4, 12.9, 5.1]])
>>> a
array([[ 2.1,  3.5,  7. ],
       [10.4, 12.9,  5.1]])
>>> a[0,0]
2.1000000000000001
>>> a[0,1]
3.5
>>> a[1,0]
10.4
>>> a[0,-1]
7.0
>>>

```

The above examples show how to extract single elements as in standard Python. The extension is that since NumPy arrays can be multi-dimensional, a list of N indices (really, a tuple) is needed for an N -dimensional array. As in standard Python, indexing starts at 0 and negative indices index backwards from the end of the array, starting with -1.

Array elements may be assigned to scalars and vice versa. Starting with the array `a` defined above:

```

>>> b = a[1,2]
>>> b
5.0999999999999996
>>> a[1,2] = 137.25
>>> a
array([[ 2.1 ,  3.5 ,  7. ],
       [10.4 , 12.9 , 137.25]])
>>>

```

(Notice above another example of the difficulty of representing decimal numbers in binary floating point form!)

3.2.2 Slicing

The slicing methods used in Python strings and lists also work for NumPy arrays:

```
>>> a = array([[2.1, 3.5, 7.0], [10.4, 12.9, 5.1]])
>>> a
array([[ 2.1,  3.5,  7. ],
       [10.4, 12.9,  5.1]])
>>> b = a[:,1:2]
>>> b
array([[ 3.5],
       [12.9]])
>>>
```

The index “:” returns all elements along the corresponding axis. Thus, `a[:,1:2]` creates an array consisting of the second column (column 1 with zero-based indexing) and all rows of array `a`.

Note that if the number of colons is less than the number of dimensions, then all elements of the remaining dimensions are included:

```
>>> a = zeros((2,2))
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> a[:]
array([[ 0.,  0.],
       [ 0.,  0.]])
>>>
```

Thus, as illustrated above, `a[:]` indicates the entire array regardless of the number of dimensions.

Also note that slicing may be used on an array on the left side of the equals sign to facilitate assignment to part of that array:

```
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> b = array([1.,2.])
>>> b
array([ 1.,  2.])
>>> a[0,:] = b
>>> a
array([[ 1.,  2.],
       [ 0.,  0.]])
>>>
```

Here are some other useful tricks in slicing:

```

>>> a = arange(0,10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0:10:2]
array([0, 2, 4, 6, 8])
>>> a[-1:0:-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1])
>>> a[-1::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
>>>

```

The first example illustrates the use of a non-default step of 2, while the last two show how to reverse the order of the elements in an array. The second of these demonstrates that by omitting the stop value, all elements up to the beginning of the array (or the end if the start value is positive) are included in the new view.

3.2.3 Views and copies

Here is a surprise:

```

>>> a = array([[2.1, 3.5, 7.0], [10.4, 12.9, 5.1]])
>>> b = a[:,1:2]
>>> b
array([[ 3.5],
       [12.9]])
>>> a[0,1] = 14.5
>>> b
array([[ 14.5],
       [12.9]])
>>>

```

As with lists, the assignment of one array to another, e. g., `c = a`, doesn't create a new array, just an alias, `c`, to the original array `a`. However, NumPy carries this farther. When an array is sliced, i. e., when a subset of it is defined as illustrated above, a new array is not created. Instead, a new *view* of the original array is created. Thus, when the original array `a` is changed, the new view, `b` is also changed. This is analogous to the aliases which occur for lists and dictionaries.

Note that aliasing does not happen with the simple assignment of an array element to a scalar. Contemplate the meaning of the following:

```

>>> a = array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> b = a[0:1,1:2]
>>> b

```

```

array([[2]])
>>> c = a[0,1]
>>> c
2
>>> a[0,1] = 42
>>> b
array([[42]])
>>> c
2
>>>

```

If you really need to make a new array out of a slice of the original array, use of the “+” prefix causes this to happen:

```

>>> a = arange(0,10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = +a[1:3]
>>> b
array([1, 2])
>>> a[1] = 42
>>> a
array([ 0, 42,  2,  3,  4,  5,  6,  7,  8,  9])
>>> b
array([1, 2])

```

This is because the plus sign in the expression on right side of the equals sign technically signals computation, which might change element values, though of course it doesn’t actually do so in this case. Sometimes the “literal-mindedness” of computers is an advantage!

3.2.4 Reshaping arrays

NumPy arrays have a *reshape* method which allows the number of dimensions and the size of each dimension to be changed as long as the total number of array elements remains the same:

```

>>> a = array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.reshape(4)
array([1, 2, 3, 4])
>>> a.reshape(4,1)
array([[1],
       [2],
       [3],

```

```

    [4]))
>>> a.reshape(1,4)
array([[1, 2, 3, 4]])
>>> a.reshape(2,2,1)
array([[1],
       [2]],

      [[3],
       [4]]])
>>>

```

Reshape has been used three different ways in the above examples. Starting with a two-dimensional array with 2×2 elements, we have (1) converted this into a one-dimensional array of 4 elements, (2) a two-dimensional array of 4×1 elements, (3) a two-dimensional array of 1×4 elements, and (4) a three-dimensional array of $2 \times 2 \times 1$ elements.

As with slicing, reshaping doesn't normally create a new array, just a new view of the original array:

```

>>> a = array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> b = a.reshape(4)
>>> a[0,1] = 137
>>> a[1,0] = 42
>>> b
array([ 1, 137, 42, 4])
>>>

```

Reshaping brings up a sticky question. When reshaping `[[1, 2], [3, 4]]` into `[1, 2, 3, 4]`, why doesn't it come out `[1, 3, 2, 4]`? The answer is that by default NumPy iterates the last index in a multi-dimensional array most rapidly and the first index least rapidly while assigning elements to successive memory locations. Thus, the order in which the elements appear is `a[0,0]`, `a[0,1]`, `a[1,0]`, `a[1,1]` in the above example. As we discovered above, this ordering is revealed in the conversion of a multi-dimensional array to a one-dimensional array, and is the same as that used by the C language. The Fortran language uses the opposite convention; the first index is iterated most rapidly. It is possible to make NumPy behave like Fortran in this regard, but special options and methods must be invoked.

3.2.5 Array looping

NumPy arrays are Python sequences, which means that *for* loops can be used to iterate over them. For example:

```

>>> a = arange(12).reshape(4,3)
>>> a

```

```

array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> for x in a:
...   x
...
array([0, 1, 2])
array([3, 4, 5])
array([6, 7, 8])
array([ 9, 10, 11])
>>>

```

This example reveals that a two-dimensional NumPy array is actually an array of arrays, so iterating over `a` doesn't yield the scalar array elements in sequence. Instead this loop accesses in sequence the subarrays from which the array `a` is constructed. As a result, `len(a) = 4` rather than 12 and `a[0] = array([0, 1, 2])`!

To get the scalar elements, two loops are required, one over the array of subarrays and the other over the subarrays themselves:

```

>>> for x in a:
...   for y in x:
...     y
...
0
1
2
3
4
5
6
7
8
9
10
11
>>>

```

The generalization to arrays of more than two dimensions is obvious.

To access individual array elements of a multi-dimensional array in a single loop, use the *ravel* method, which is equivalent to reshaping the array into one-dimensional form:

```

>>> for y in a.ravel():
...   y
...
0

```

```
1
2
3
4
5
6
7
8
9
10
11
>>>
```

Generally speaking, iterating over the elements of a NumPy array in Python should be avoided where possible, as it is computationally inefficient due to the interpreted nature of the Python language. As we shall see, there are many NumPy array methods and functions which reduce the necessity for such explicit iteration.

3.3 Array expressions and assignments

In this section we begin to learn how to use NumPy arrays. Array expressions and assignment statements work much like standard scalar math:

```
>>> a = array([[1,2],[3,4]])
>>> b = array([[5,6],[7,8]])
>>> c = a + b
>>> c
array([[ 6,  8],
       [10, 12]])
>>> d = a*b
>>> d
array([[ 5, 12],
       [21, 32]])
>>> e = a*b + c
>>> e
array([[11, 20],
       [31, 44]])
>>> f = c + a*b
>>> f
array([[11, 20],
       [31, 44]])
>>>
```

As the above example shows, array math operates on an element by element basis, i. e., unlike matrix math. The usual algebraic precedence rules apply, e. g., multiplication is done before addition in otherwise ambiguous situations.

3.3.1 Broadcasting

Because array math occurs element by element, arrays normally have to be the same shape for math operations between them to succeed:

```
>>> a = array([[1,2],[3,4]])
>>> e = array([10, 20, 30])
>>> a + e
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape
>>>
```

The shape of array `a` is (2,2), while the shape of array `e` is (3,), and they therefore cannot be added together.

The exception occurs when one of the arrays is the same shape as the other except that the size of one or more of the dimensions is reduced to 1, or else absent completely:

```
>>> a = array([[1,2],[3,4]])
>>> a.shape
(2, 2)
>>> b = array([[5,10]])
>>> b.shape
(1, 2)
>>> a + b
array([[ 6, 12],
       [ 8, 14]])
>>>
```

In this case array `b`, which has one row and two columns, is (conceptually) extended along axis 0, repeating values as necessary, to create an array with two rows `array([[5,10],[5,10]])` and then added to `a`. This extension process is called *broadcasting*.

Consider the alternative case where an array `c` has two rows and one column:

```
>>> c = array([[5],[10]])
>>> c.shape
(2, 1)
>>> a + c
array([[ 6,  7],
       [13, 14]])
>>>
```

In this case array `c` is extended along axis 1, to create an array with two rows `array([[5,5],[10,10]])`, which is then added to `a`. Notice that the result here is different than that in the previous case, since the broadcast axis is different.

A case of double broadcasting is illustrated below, where arrays `b` and `c` are as shown above:

```
>>> b + c
array([[10, 15],
       [15, 20]])
>>>
```

The number of dimensions of the two arrays need not be the same. For example:

```
>>> d = array([5,10])
>>> d.shape
(2,)
>>> a + d
array([[ 6, 12],
       [ 8, 14]])
>>>
```

In this case a new axis of size 1 is first added onto the beginning of the initial shape (2,), resulting in the shape (1, 2), which is the same as that of array `b`. Notice that this can be done unambiguously, since the number of array elements does not change. Then broadcasting occurs along the new axis, as is done when computing `a + b`. However, cases like this leave one wondering whether the result is what one intended. Just to be safe, it is probably better to reshape array `d` explicitly the way one wants it before adding it to `a`, i. e.,

```
>>> dd = d.reshape(1,2)
>>> a + dd
array([[ 6, 12],
       [ 8, 14]])
>>>
```

or

```
>>> dd = d.reshape(2,1)
>>> a + dd
array([[ 6,  7],
       [13, 14]])
>>>
```

Notice that combining a scalar with an array, for instance by addition or multiplication, is a special case of broadcasting in which one of the components (the scalar) has fewer dimensions than the other. In this case there is no ambiguity in the broadcasting as all dimensions are filled with the scalar value and there is no reason to avoid this type of construction.

3.4 Array methods

Since NumPy arrays are Python objects, they have methods associated with them. Here we describe some of the most useful methods. Some methods operate on the array in place and don't return anything. Others return some object, generally a modified form of the original array. Which does which is indicated.

3.4.1 General methods

- **tolist()**: This method converts a NumPy array to an ordinary Python list:

```
>>> a = arange(0.,9.5)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> a.tolist()
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
>>> b = a.reshape(2,5)
>>> b
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> b.tolist()
[[0.0, 1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0, 9.0]]
>>>
```

This script also reminds us about the *reshape* method, which was discussed in the last section.

- **tofile(file=, sep=",", format="")**: This writes an array to a file. *File* is either a string representing a file name, in which case this method opens this file and writes the array to it, or it is the file object for a previously opened file. If the optional keyword parameters *sep* and *format* are absent, the array is written in native binary format:

```
>>> a = array([[1., 2., 3.], [9., 8., 7.]])
>>> a.tofile("junk.bin")
```

No meta-information about the array is written – just the array element values, as can be inferred by the size of *junk.bin*; 48 bytes. The parameter *sep* defines an ascii string (for instance, a space or a tab) to separate the array elements. The elements are converted to ascii string format for writing. If the *format* command is used, then the elements are written in the specified format, e. g., “%f”, “%g”, or “%d”. For example, using the same array:

```
    a.tofile(sys.stderr, sep = " ")
1.0 2.0 3.0 9.0 8.0 7.0>>>
```

The Python prompt appears after the output because no newline is appended to the file.

- **astype()**: This method converts the type of elements to the type listed in the argument as discussed previously.
- **byteswap()**: This method swaps bytes as previously discussed. If the argument is *False* (the default), bytes are swapped in place. If *True*, the method returns a new array with the swapped bytes.

- **copy()**: This returns a copy of the array.
- **fill(scalar)**: This fills all of the elements of the array with the scalar value and is faster than element-by-element assignment:

```
>>> a = array([1,4,7])
>>> a
array([1, 4, 7])
>>> a.fill(42)
>>> a
array([42, 42, 42])
>>>
```

- **reshape(newshape)**: As noted previously, reshape changes the number and size of dimensions of an array, returning a new array (actually a new view – additional data memory isn't allocated). The new shape is given by the newshape tuple. The total size of the array cannot change.
- **transpose(tuple of axes), T**: This method transposes arrays of two and higher dimensions:

```
>>> a = array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>>
```

For arrays of greater than two dimensions a tuple of axes can be used to show how the existing axes should be ordered in the returned array. `a.T` is shorthand for `a.transpose()` with no arguments.

- **sort(axis=-1)**: This sorts the elements along the defined dimension, with the default being the last (-1). Dimension numbering starts with 0:

```
>>> a = array([3, 7, 4, 8, 2, 15])
>>> a
array([ 3,  7,  4,  8,  2, 15])
>>> a.sort()
>>> a
array([ 2,  3,  4,  7,  8, 15])
>>>
```

3.4.2 Mathematical methods

- **max(axis=None)**: For no arguments, this returns the scalar value which is the largest element in the entire array. If an axis is defined for an N-dimensional array, the maximum values along that axis are returned as an array with N - 1 dimensions:

```
>>> a = array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.max()
4
>>> a.max(0)
array([3, 4])
>>> a.max(1)
array([2, 4])
>>>
```

- **min()**: Works analogously to `max()`.
- **clip(min=, max=)**: Returns an array in which elements larger than the value assigned to `max` are set to that value and those smaller than `min` are set to `min`.
- **conj()**: Returns an array with all (complex) elements conjugated.
- **trace()**: Returns the trace of a two-dimensional array. With optional arguments, works for higher-dimensioned arrays as well, but this is probably not too useful.

```
>>> a
array([[1, 2],
       [3, 4]])
>>> a.trace()
5
>>>
```

- **sum(axis = None)**: With no argument, returns the sum of all the elements in the array. With an axis argument, sums along the specified axis, returning an array of N - 1 dimensions for an N-dimensional initial array:

```
>>> a
array([[1, 2],
       [3, 4]])
>>> a.sum()
10
>>> a.sum(axis = 0)
array([4, 6])
```

```
>>> a.sum(axis = 1)
array([3, 7])
>>>
```

- **mean(axis = None)**: Like `sum()` except computes the mean.
- **var(axis = None)**: Like `sum()` except computes the variance.
- **std(axis = None)**: Like `sum()` except computes the standard deviation.
- **prod(axis = None)**: Like `sum()` except computes the product.
- **cumsum()**: With no argument, reshape the array to a single dimension and perform a cumulative sum along that dimension. With an optional axis, perform cumulative sums along that axis.

```
>>> a
array([[1, 2],
       [3, 4]])
>>> a.cumsum()
array([ 1,  3,  6, 10])
>>> a.cumsum(axis = 0)
array([[1, 2],
       [4, 6]])
>>> a.cumsum(axis = 1)
array([[1, 3],
       [3, 7]])
>>>
```

- **cumprod()**: Like `cumsum()` but computes the cumulative product.

3.5 Array functions

Some of the most useful functions we haven't discussed previously are now described. In certain cases obscure options are omitted in this documentation for simplicity. Also, many NumPy functions are omitted because they reproduce the functionality of NumPy methods discussed above.

- **(num, bins) = histogram(x, bins=None, range=None)**: This function returns an array in `num` (a histogram) which contains the number of values of the array `x` sorted into bins defined by the optional keyword arguments `bins` and `range`. The argument `range` is a tuple containing the minimum and maximum values of `x` that are accepted – values outside this range are discarded. Normally the argument `bin` contains a scalar giving the number of bins, though it can also be a tuple giving the desired boundaries of the bins. The return argument `bin` is an array containing the bounding values of the bins.

```

>>> a = log2(arange(1.,50.,0.1))
>>> (num, bins) = histogram(b, bins = 10, range = (0,6))
>>> bins
array([ 0. ,  0.6,  1.2,  1.8,  2.4,  3. ,  3.6,  4.2,  4.8,  5.4,  6. ])
>>> num
array([ 6,  7, 12, 18, 27, 42, 62, 95, 144, 77])
>>>

```

- **(num, xedges, yedges) = histogram2d(x, y, bins=10, range=)**: This function returns a two-dimensional histogram (num) and the edges of the bins in the x and y directions resulting from the data in the arrays x and y. By default 10 bins are created in each dimension and the range spans the data values. This may be changed by assigning values to bins and range, e. g., bins=(12, 8) and range=((-1, 11), (-2, 12)). (Either tuples or lists suffice here.)

```

>>> x = linspace(0,10,100)
>>> y = x*x/10
>>> (num, xedges, yedges) = histogram2d(x,y)
>>> num
array([[ 10.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 10.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 10.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [  2.,  8.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [  0.,  5.,  5.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [  0.,  0.,  5.,  5.,  0.,  0.,  0.,  0.,  0.,  0.],
       [  0.,  0.,  0.,  3.,  7.,  0.,  0.,  0.,  0.,  0.],
       [  0.,  0.,  0.,  0.,  1.,  6.,  3.,  0.,  0.,  0.],
       [  0.,  0.,  0.,  0.,  0.,  0.,  3.,  6.,  1.,  0.],
       [  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  4.,  6.]])
>>> xedges
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
>>> yedges
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
>>>

```

- **c = correlation(x, y, mode='valid')**: This routine returns the one-dimensional array c containing the cross correlation between the one-dimensional arrays x and y. This is defined

$$c[n + n_0] = \sum_i x[i]y[n + i] \quad (3.1)$$

where the value of n_0 and the range of the summation depend on the mode selected, which may be *valid*, *same*, or *full*. The input arrays need not be the same size. Suppose that the size of array x is X and the size of array Y is y. The array x and the shifted array y will not completely overlap in general. Sums at points beyond the limits of either array are eliminated. The option *valid* (the default) limits n to the case in which

the smaller array completely overlaps the larger array after being shifted by n points. The size of c in this case is $|X - Y| + 1$. Notice that for input arrays of equal size, this means that c has a size of 1. For the *same* option, shifts are allowed up to the point at which the size N of array c is the larger of X and Y . For the *full* option, shifts are allowed which result in minimal (i. e., one point) overlap, so $N = |X + Y| - 1$. The value of $n0$ if x and y have the same length is $n0 = N/2$ for all options, including the effects of truncation in integer division. This means that for an odd length $n0$ refers to the element at the center of the output array, e. g., $n0 = 5/2 = 2$. For an even length, it refers to the element just right of center, e. g., $n0 = 4/2 = 2$. If the input arrays are of differing length, the proper value of $n0$ is in general hard to predict. However, if the arrays are both of odd length, the output is always of odd length and $n0$ refers to the center element in all cases. No complex conjugation is done on either input array if the arrays are complex.

- **`c = convolve(x, y, mode='full')`**: The convolution function is identical to the correlation function except that the calculation returns

$$c[n + n0] = \sum_i x[i]y[n - i] \quad (3.2)$$

and the default mode is *full* rather than *valid*. Some simple examples:

```
>>> convolve([1, 0, 0], [0, 0, 1], mode='full')
array([0, 0, 1, 0, 0])
>>> correlate([1, 0, 0], [0, 0, 1], mode='full')
array([1, 0, 0, 0, 0])
>>> correlate([0, 0, 1], [1, 0, 0], mode='full')
array([0, 0, 0, 0, 1])
>>>
```

3.6 Matrices

3.7 Masked arrays

Sometimes regions of data on a grid are missing. For instance meteorological radar measurements lack data where there are no clouds. NumPy and Matplotlib provide ways of dealing with such situations. The tool used is NumPy's masked array.

To create a masked array from a normal array, use the *ma.array* function:

```
>>> a
array([[ 0.,  1.,  4.,  9.],
       [ 1.,  2.,  5., 10.],
       [ 4.,  5.,  8., 13.],
       [ 9., 10., 13., 18.]])
>>> m = a > 10.
>>> amasked = ma.array(a,mask=m)
```

```

>>> amasked
masked_array(data =
  [[0.0 1.0 4.0 9.0]
  [1.0 2.0 5.0 10.0]
  [4.0 5.0 8.0 --]
  [9.0 10.0 -- --]],
             mask =
  [[False False False False]
  [False False False False]
  [False False False  True]
  [False False  True  True]],
             fill_value = 1e+20)

```

```
>>>
```

We desire to mask the array `a` (created by means not shown) for element values exceeding 10. We first create a boolean *mask* with the command `m = a > 10`. We then use this mask as the value of the keyword argument `mask`. Printing out the resulting masked array `amasked` reveals that this object consists of the data array itself with the masked values blanked out, the boolean array with `True` values occurring where the original array elements exceeded 10, and the *fill value*, which is the actual value assigned to masked elements of the data array. If a different fill value is desired, this may be assigned via a keyword parameter in the `ma.array` call, e. g., `fill_value=1e30`.

Computations are not done on masked values in NumPy calculations. Furthermore, various other packages such as the Matplotlib graphics package (discussed in the next chapter) recognize masked regions and act appropriately.

3.8 Endianness

One ugly, but potentially important issue can arise with the transfer of data between computers. Different computer architectures store the bytes in multi-byte numbers in different orders. Intel-compatible machines store the least significant byte first, and are called little-endian. Various other architectures, most notably the Sparc architecture from Sun Microsystems, store the most significant byte first, and are called big-endian. The only time this architectural detail is of importance is when data are moved from one machine type to another. In order to make sense out of data coming from a machine of the opposite type, we must reverse the byte order in the data. This is done with the `byteswap` method. To show what happens when bytes are in the wrong order, let's do a byteswap on an array:

```

>>> a = arange(0,5)
>>> a
array([0, 1, 2, 3, 4])
>>> a.byteswap()
array([      0, 16777216, 33554432, 50331648, 67108864])
>>>

```

It is possible to detect the endianness of the machine on which you are working using a routine from the `sys` module:

```
>>> import sys
>>> sys.byteorder
'little'
>>>
```

3.9 Further information

The NumPy home page provides some useful information on NumPy: <http://numpy.scipy.org/>. The User Guide is rather incomplete at this point, but the NumPy Reference is quite useful. There is also a separate Tutorial, and the source code may be downloaded. Travis Oliphant's Guide to NumPy <http://www.tramy.us/> is a complete reference work, over 300 pages of PDF. It is the definitive work, but daunting for a beginner. The Numerical Python page <http://numpy.sourceforge.net/numdoc/HTML/numdoc.htm> is out of date, but may still be useful in certain respects.

Version 1.13 of NumPy is documented here. Changes (for the better) were made to the histogram function between 1.11 and 1.13.

Chapter 4

Graphics with Matplotlib

Matplotlib is a large and sophisticated graphics package for Python written in object oriented style. However, a layer built on top of this basic structure called *pyplot* accesses the underlying package using function calls. We describe a simple but useful subset of pyplot here.

4.1 Line plots

4.1.1 Simple plots

Let's type the following into Python:

```
>>> from numpy import *
>>> import matplotlib.pyplot as plt
>>> x = arange(0.,10.,0.1)
>>> y = sin(x)
>>> l1 = plt.plot(x,y)
>>> plt.show()
>>>
```

The first two lines import respectively the numpy and matplotlib pyplot modules. The second import is of a form we haven't seen before. If we had typed `import matplotlib.pyplot`, all pyplot functions would need to be prefixed by the cumbersome `matplotlib.pyplot`. If on the other hand, we had used `from matplotlib.pyplot import *`, pyplot functions could have been used without any prefix. However, this would have left open the possibility that pyplot functions might clash with functions from other modules. The form used allows pyplot functions to be prefixed with the relatively short word `plt`. The next two lines define the arrays `x` and `y` which we want to plot against each other. The command `plt.plot(x,y)` performs this action, but we don't see the result displayed on the screen until we type `plt.show()`. The result is shown in figure 4.1.

Calls to most pyplot functions return stuff that can sometimes be used in subsequent functions. That is why we assign these return values to variables, e. g., as in `l1 = plt.plot(x,y)` in the above script, even though we don't use these values here. An advantage of doing this is that the output is not cluttered up with obscure return values.

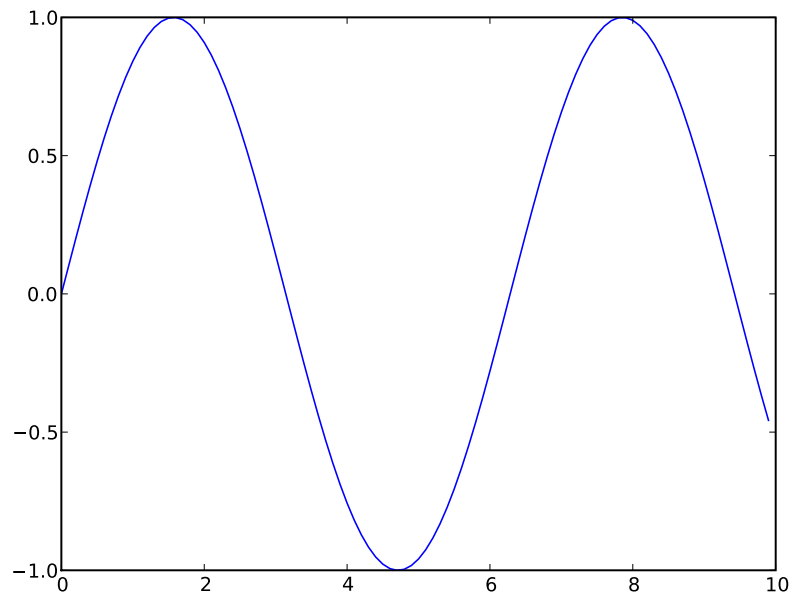


Figure 4.1: A simple plot of a sine function.

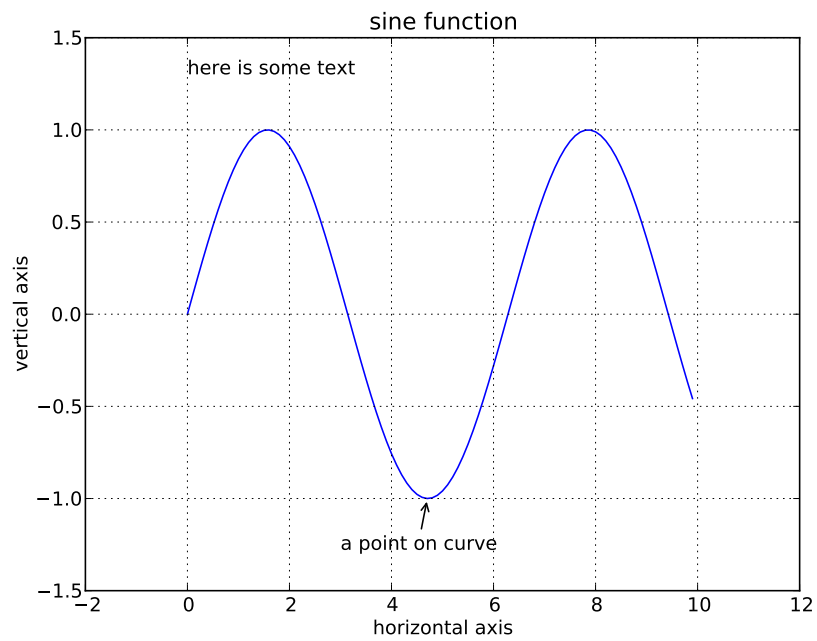


Figure 4.2: Adding annotations to the plot.

Text such as a title, labels, and annotations can be added to the plot between the `plt.plot` and the `plt.show` commands. We can also change the axes if we don't like the default choice and add grid lines to the plot:

```
>>> l1 = plt.plot(x,y)
>>> xl = plt.xlabel('horizontal axis')
>>> yl = plt.ylabel('vertical axis')
>>> ttl = plt.title('sine function')
>>> ax = plt.axis([-2, 12, -1.5, 1.5])
>>> grd = plt.grid(True)
>>> txt = plt.text(0,1.3,'here is some text')
>>> ann = plt.annotate('a point on curve',xy=(4.7,-1),xytext=(3,-1.3),
... arrowprops=dict(arrowstyle='->'))
>>> plt.show()
>>>
```

The resulting plot is shown in figure 4.2. The argument to the `axis` command is a list containing the lower and upper values of the x axis followed by the same for the y axis. The `text` command allows one to put arbitrary text anywhere in the plot. The `annotate` command connects text with plot points by an arrow. There are many arrow properties; we show only the simplest case here. The operation of the x and y label commands is clear.

4.1.2 Multiple lines and points

More complicated plots with multiple lines and line types can be made:

```
>>> x = arange(0.,10,0.1)
>>> a = cos(x)
>>> b = sin(x)
>>> c = exp(x/10)
>>> d = exp(-x/10)
>>> la = plt.plot(x,a,'b-',label='cosine')
>>> lb = plt.plot(x,b,'r--',label='sine')
>>> lc = plt.plot(x,c,'gx',label='exp(+x)')
>>> ld = plt.plot(x,d,'y-', linewidth = 5,label='exp(-x)')
>>> ll = plt.legend(loc='upper left')
>>> lx = plt.xlabel('xaxis')
>>> ly = plt.ylabel('yaxis')
>>> plt.show()
>>>
```

The third argument of the plot command is an optional string, the first character of which is a color; one of "b", "g", "r", "c", "m", "y", "k", and "w" – mostly obvious, except "k" indicates black. The second and following characters indicate the line type. Types "-", "--", "-.", and ":" indicate solid, dashed, dash-dotted, and dotted lines respectively. Other types indicate markers: ".", "x", "+", "o", and "*" are fairly obvious. There are numerous other

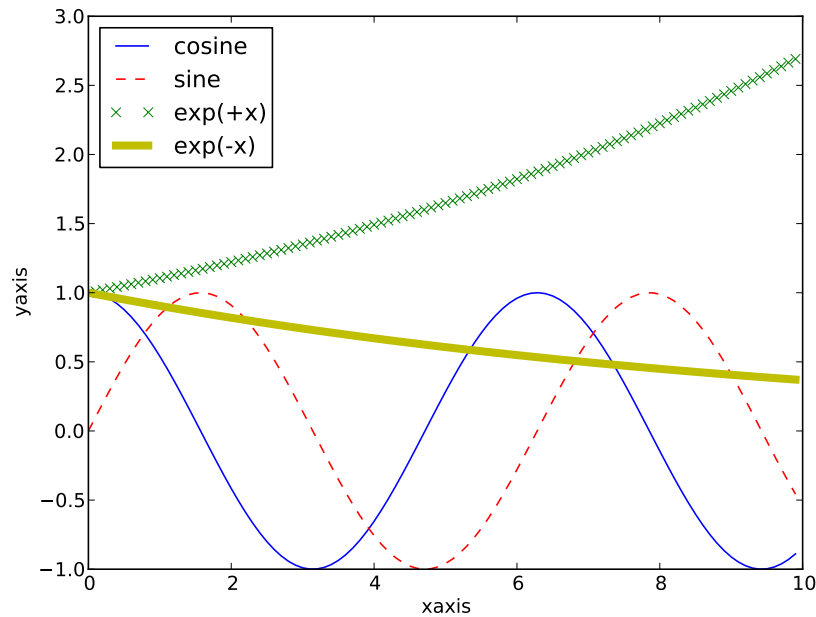


Figure 4.3: Plot showing different line types and colors as well as a legend.

choices as well. In addition to the positional arguments, there is a large list of keyword arguments to the plot command. Perhaps the most useful is the *linewidth*, which is given in points.

The *label* argument in the plot command is used later by the *legend* command, which draws a legend in the specified location. Location choices are strings of the form 'upper left', 'lower center', 'right', etc. Alternatively, the *loc* argument may be omitted and Matplotlib puts the legend where it sees fit.

The resulting plot is shown in figure 4.3.

4.2 Log plots and histograms

A variety of other one-dimensional plotting modes are available in Matplotlib. The following script illustrates logarithmic plots and histograms:

```
#!/usr/bin/python
#
# variety.py -- Make a variety of plots in a single figure

from numpy import *
import matplotlib.pyplot as plt

fg = plt.figure(figsize=(10,8))
adj = plt.subplots_adjust(hspace=0.4,wspace=0.4)
```

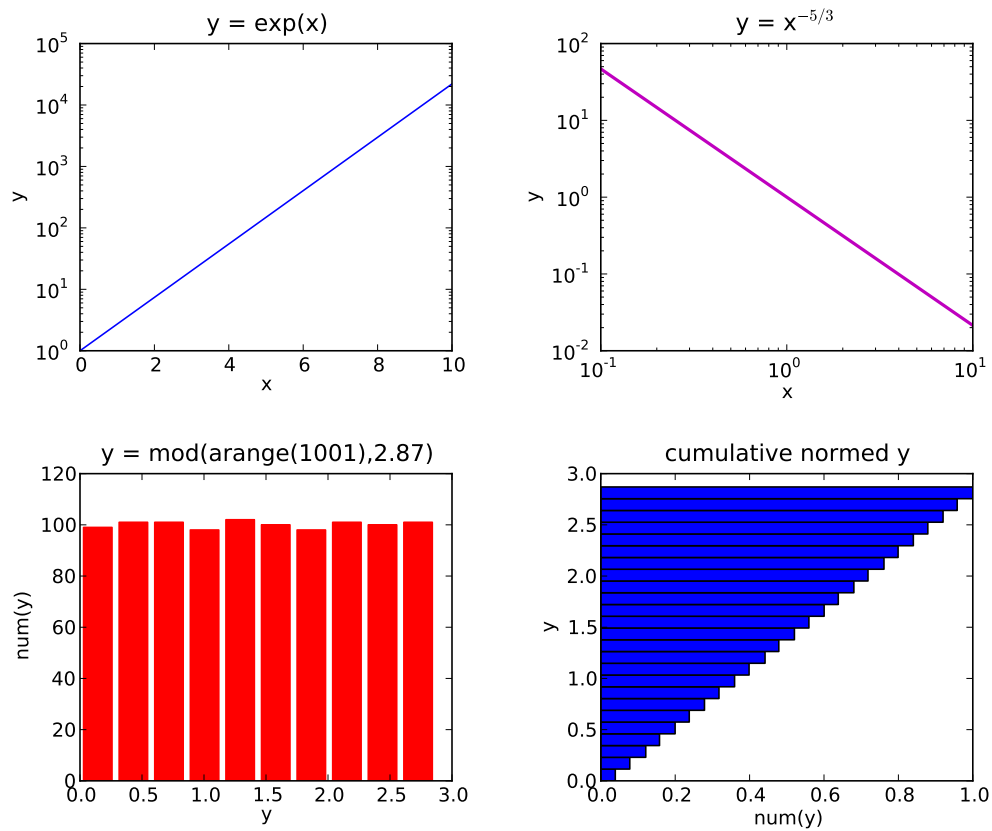


Figure 4.4: Semi-log and log-log plots as well as two types of histogram plots.

```

sp = plt.subplot(2,2,1)
x = linspace(0,10,101)
y = exp(x)
l1 = plt.semilogy(x,y,color='m',linewidth=2)
lx = plt.xlabel("x")
ly = plt.ylabel("y")
tl = plt.title("y = exp(x)")

sp = plt.subplot(2,2,2)
y = x**-1.67
l1 = plt.loglog(x,y)
lx = plt.xlabel("x")
ly = plt.ylabel("y")
tl = plt.title("y = x-5/3")

sp = plt.subplot(2,2,3)
x = arange(1001)
y = mod(x,2.87)
l1 = plt.hist(y,color='r',rwidth = 0.8)
lx = plt.xlabel("y")
ly = plt.ylabel("num(y)")
tl = plt.title("y = mod(arange(1001),2.87)")

sp = plt.subplot(2,2,4)
l1 = plt.hist(y, bins=25, normed=True, cumulative=True, orientation='horizontal')
lx = plt.xlabel("num(y)")
ly = plt.ylabel("y")
tl = plt.title("cumulative normed y")

plt.show()

```

The semi-log and log-log plots are largely self-explanatory. The plot command *semilogx* places the log scale along the x axis. The histogram plots are similarly simple. The *rwidth* option specifies the width of the histogram bars relative to the bin size, with a default of 1. The *cumulative* option plots a cumulative distribution and *normed* causes the integral of the distribution to be unity in the non-cumulative case. In the cumulative case the last histogram bar has a length of unity.

4.3 Two-dimensional plots

We now learn how to make two-dimensional plots of various kinds. These plots are created from two-dimensional NumPy arrays. In these arrays the second dimension (the column index) corresponds to the horizontal axis of the plot while the first dimension (the row

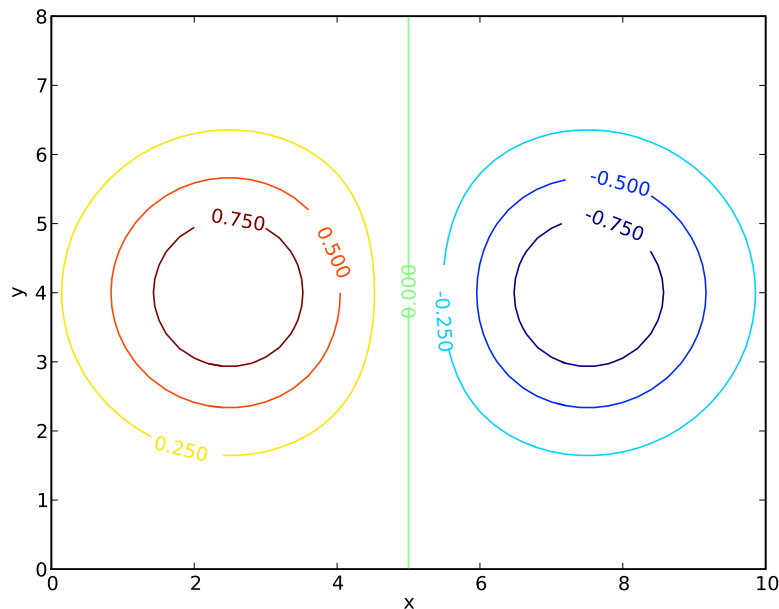


Figure 4.5: A simple contour plot with labels.

index) corresponds to the vertical axis. Thus, arrays are plotted the same way they are printed, except that plots go from the bottom up while printing goes from the top down.

4.3.1 Contour plots

Here is the simplest way to make a contour plot:

```
>>> x = linspace(0,10,51)
>>> y = linspace(0,8,41)
>>> (X,Y) = meshgrid(x,y)
>>> a = exp(-((X-2.5)**2 + (Y-4)**2)/4) - exp(-((X-7.5)**2 + (Y-4)**2)/4)
>>> c = plt.contour(x,y,a)
>>> l = plt.clabel(c)
>>> lx = plt.xlabel("x")
>>> ly = plt.ylabel("y")
>>> plt.show()
>>>
```

The result is shown in figure 4.5. The first four lines in the above script create the two-dimensional array `a`, which is then contoured `contour(x,y,a)`. The `x` and `y` are optional in this command, but if they are omitted, the axes default to the range `[0, 1]`.

The contour command doesn't produce contour labels on its own; this action is provided by `clabel(c)`. Note that the value returned by the contour function is needed by the `clabel` function to generate the proper labels.

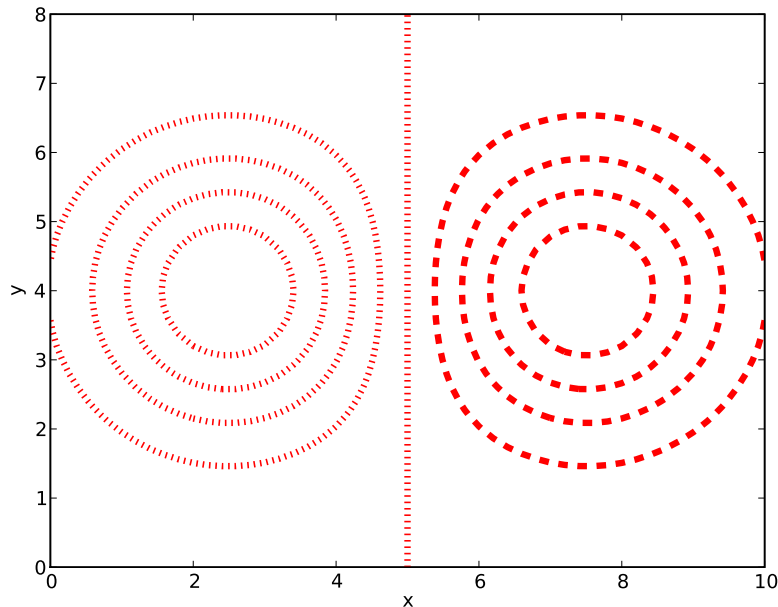


Figure 4.6: An unlabelled contour plot in red with thick dotted lines and contour interval 0.2.

A sequence (tuple or list) of values can be used to define the levels contoured. Furthermore, various keyword arguments can be used to modify the presentation. The keyword *colors* can be used to define the color of the contours (using the color values indicated previously) and the keyword *linewidths* can be used to specify the width of lines. The style of lines can be set by the keyword *linestyles*, taking the possible values *solid*, *dashed*, *dashdot*, and *dotted*:

```
>>> c = plt.contour(x,y,a,linspace(-1,1,11),colors='r',linewidths=4,
... linestyles='dotted')
>>> lx = plt.xlabel("x")
>>> ly = plt.ylabel("y")
>>> plt.show()
>>>
```

The result is shown in figure 4.6. Notice that regardless of the specified line style, Matplotlib insists on making negative contours dashed as long as a single color is specified! A solution is to fool Matplotlib into thinking multiple colors are being requested, by, for instance, specifying `colors=('r','r')` in the call to `contour`. This mechanism can also be used to (truly) specify different contour colors as well as different line widths and styles for different contours. The specified values are applied in sequence from negative to positive contours, repeating as necessary.

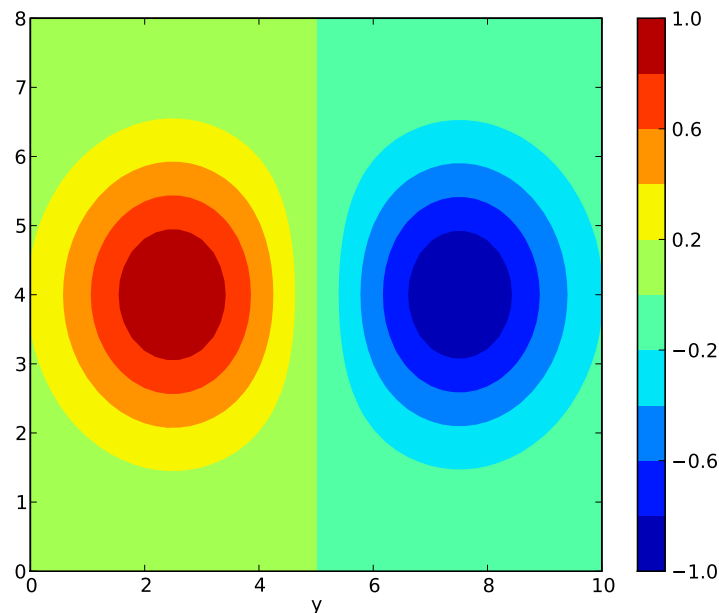


Figure 4.7: A contour plot with filled contours and a color bar.

4.3.2 Filled contours

An alternate type of contour plot fills the gaps between contours:

```
>>> c = plt.contourf(x,y,a,linspace(-1,1,11))
>>> b = plt.colorbar(c, orientation='vertical')
>>> lx = plt.xlabel("x")
>>> ly = plt.ylabel("y")
>>> plt.show()
>>>
```

The result is shown in figure 4.7. The `colorbar()` command shown produces the labeled color bar on the right side of the plot. If `orientation='horizontal'`, the color bar appears below the plot.

4.3.3 Color mesh plots

The function `pcolor` produces a plot similar to that produced by `contourf` except that the color distribution is continuous rather than discrete. For example:

```
>>> ac = 0.25*(a[:-1,:-1] + a[:-1,1:] + a[1:,:-1] + a[1:,1:])
>>> c = plt.pcolor(x,y,ac)
>>> d = plt.colorbar(c,orientation='horizontal')
>>> q = plt.winter()
>>> e = plt.contour(x,y,a,linspace(-1,1,11),colors='k')
```

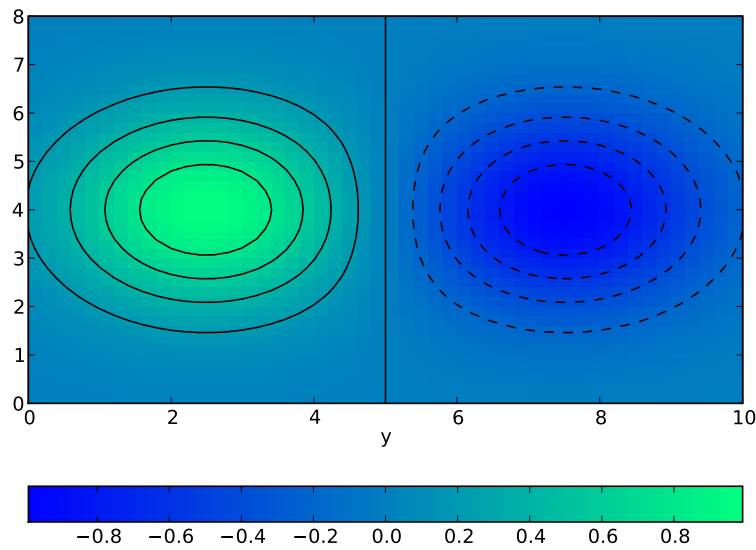


Figure 4.8: A pcolor plot with a color bar and black contours. The *winter* color map is used.

```
>>> lx = plt.xlabel("x")
>>> ly = plt.ylabel("y")
>>> plt.show()
>>>
```

The first statement of this script computes grid-centered values of the array `a`. This is strictly necessary to align the plot properly with the axes, since `pcolor` centers its pixels on grid boxes whereas `contour` and `contourf` assume that `a` is defined on grid edges. However, for very small small grid boxes the half-grid displacement may be insignificant.

4.3.4 Vector plots

Vector plots can also be made in Matplotlib. Here is a script producing a vector plot and a key showing the scale of the vectors:

```
>>> x = linspace(0,10,11)
>>> y = linspace(0,15,16)
>>> (X,Y) = meshgrid(x,y)
>>> u = 5*X
>>> v = 5*Y
>>> q = plt.quiver(X,Y,u,v,angles='xy',scale=1000,color='r')
>>> p = plt.quiverkey(q,1,16.5,50,"50 m/s",coordinates='data',color='r')
>>> xl = plt.xlabel("x (km)")
>>> yl = plt.ylabel("y (km)")
```

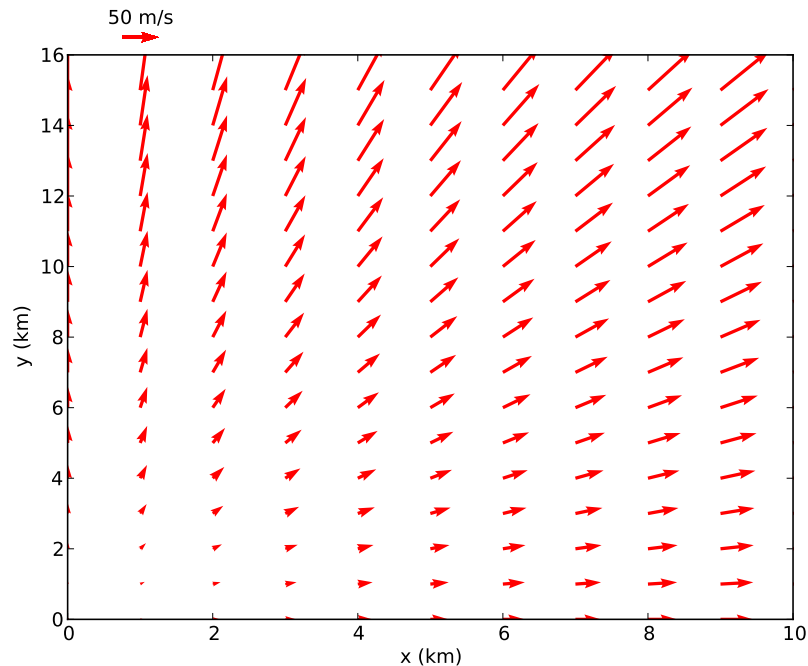


Figure 4.9: A vector plot with a legend.

```
>>> plt.show()
>>>
```

The *quiver* command produces vector plots from two-dimensional arrays (*u* and *v* in this case) containing the vector component values. The grid on which the vectors are plotted is defined by the first two arguments of *quiver* – the two-dimensional arrays *X* and *Y* in this case. However, *quiver* will accept the original one-dimensional axis vectors *x* and *y* as well. The color of the vectors is specified in the usual fashion with the color keyword.

The *quiver* arguments *angles='xy'* and *scale=1000* are very important. Setting the *angles* keyword to *'xy'* means that the vector components are scaled according to the physical axis units rather than geometrical units on the page. The actual scaling factor which multiplicatively converts vector component units to physical axis units is *width/scale* where *width* is the width of the plot in physical units and *scale* is the number specified by the *scale* keyword argument of *quiver*. In our example *width* = 10 km and *scale* = 1000, so vectors are plotted to a scale of 0.01 km/(m/ s). The angle keyword argument is not available in versions of Matplotlib earlier than 0.99, which is a serious omission.

The *quiverkey* command produces a legend consisting of a single vector and a label indicating how long it is in physical units. In order, the positional arguments are (1) the variable returned by the *quiver* command, (2) and (3) the *x* and *y* coordinates of the legend, (4) the length of the arrow in physical units, and (5) the legend label. The keyword argument *coordinates* tells Matplotlib which units define the legend location; the most useful are *'axes'*, in which *x* and *y* range from 0 to 1 across the plot and *'data'* in which physical axis coordinates are used. As in the above example (which is shown in figure 4.9), the legend may be located

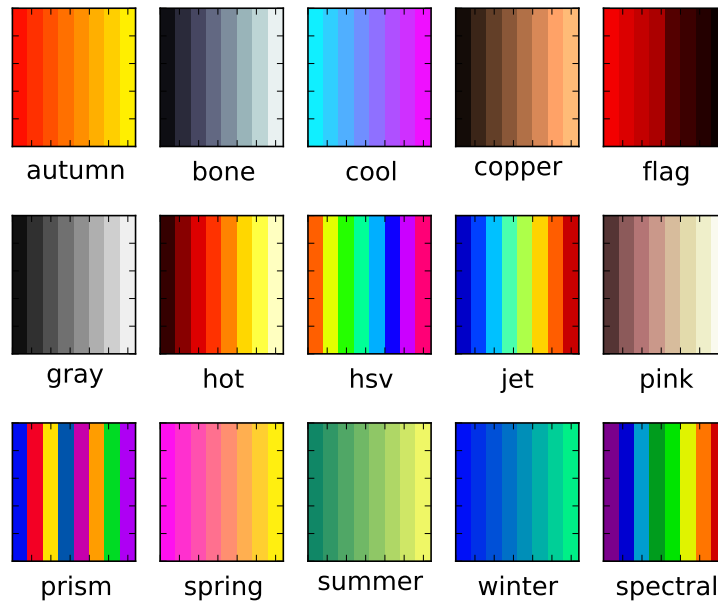


Figure 4.10: The pre-defined colormaps of Matplotlib.

outside of the actual plot.

4.3.5 Masked arrays and Matplotlib

Matplotlib honors the NumPy conventions for masked arrays, in that masked regions of two-dimensional plots are omitted.

The use of masked arrays with vector plots and filled contour plots is a bit buggy at this point. For vectors, it is best to eliminate masked arrays in favor of arrays which give vectors zero length in masked regions. Hopefully this situation will improve in subsequent version of Matplotlib.

4.4 Finer control of color and text

Figure 4.8 illustrates the use of an alternate color map, in this case, *winter*, generated by the `winter()` function. Colormaps provided by Matplotlib include *autumn*, *bone*, *cool*, *copper*, *flag*, *gray*, *hot*, *hsv*, *jet*, *pink*, *prism*, *spring*, *summer*, *winter*, and *spectral* and each has its associated function call. Figure 4.10 illustrates each of these colormaps. The default is *jet*.

An alternate way to set a color is by specifying its additive RGB value in a hex string. The hexadecimal digits are (1, 2, 3, ..., 9, a, b, c, d, f), so pure red is `'#ff0000'`, pure green is `'#00ff00'`, and pure blue is `'#0000ff'`, while black is `'#000000'` and white is `'#ffffff'`. With this scheme each primary color has 256 possible values, which is precisely the number that an eight bit unsigned integer can provide. So, for instance, bright yellow, which is a

mixture of red and green, is specified by `color='#ffff00'`, and a darker red and a lighter blue would be obtained respectively from `color='#aa0000'` and `color='aaaaff'`.

Various global options can be set directly in the `rcParams` dictionary of Matplotlib, which contains default settings for many variables. Set any desired entries before invoking any other plotting commands. Since `rcParams` is a dictionary, the dictionary keys can be printed with the command `plt.rcParams.keys()` and the current value of a particular key (such as `font.size`) can be listed with `plt.rcParams['font.size']`.

The key `font.size` sets the default font size in plots. The default size of 12 pts is generally too small for presentation plots. Increasing this to 16 pts may be done with the command `plt.rcParams['font.size'] = 16`.

Increasing the font size can cause axis labels to overlap – Matplotlib is not (yet) smart enough to compensate for this. Axis numerical labeling can be changed using the `xticks` and `yticks` commands. (Recall that the `xlabel` and `ylabel` commands take care of the axis name labels.) For example, the command

```
>>> tx = plt.xticks(linspace(5,25,5))
```

labels the x axis with the numerical values 5, 10, 15, 20, 25. In general, the argument of the `xticks` command is a list or array with the desired numerical labels, starting at the left end of the axis and ending at the right end. The `yticks` command works similarly for the y axis.

A similar problem can arise with colorbar labels, especially if the color bar is horizontally oriented. The solution here is a keyword argument to the colorbar command named `ticks`. So, for example, one could use

```
>>> plt.colorbar(c,orientation='horizontal',ticks=linspace(-1,1,5))
```

to define the spacing of the labels on the color bar.

4.5 Combination plots

4.5.1 Multiple plots

Technically, before issuing any plotting commands, the `figure` command must be issued. However, pyplot takes care of this in simple cases with the default creation of a figure with dimensions 8 by 6 inches (when printed on paper). If different figure dimensions are desired, the `figure` command needs to be called explicitly with the `figsize` keyword argument.

Figures with multiple plots can be created using the `subplot` command. This command has three arguments, the number of rows of plots in the figure, the number of columns, and the particular plot being created next:

```
>>> x = arange(0.,10.1,0.2)
>>> a = sin(x)
>>> b = cos(x)
>>> fig1 = plt.figure(figsize = (8,8))
>>> plt.subplots_adjust(hspace=0.4)
>>> p1 = plt.subplot(2,1,1)
```

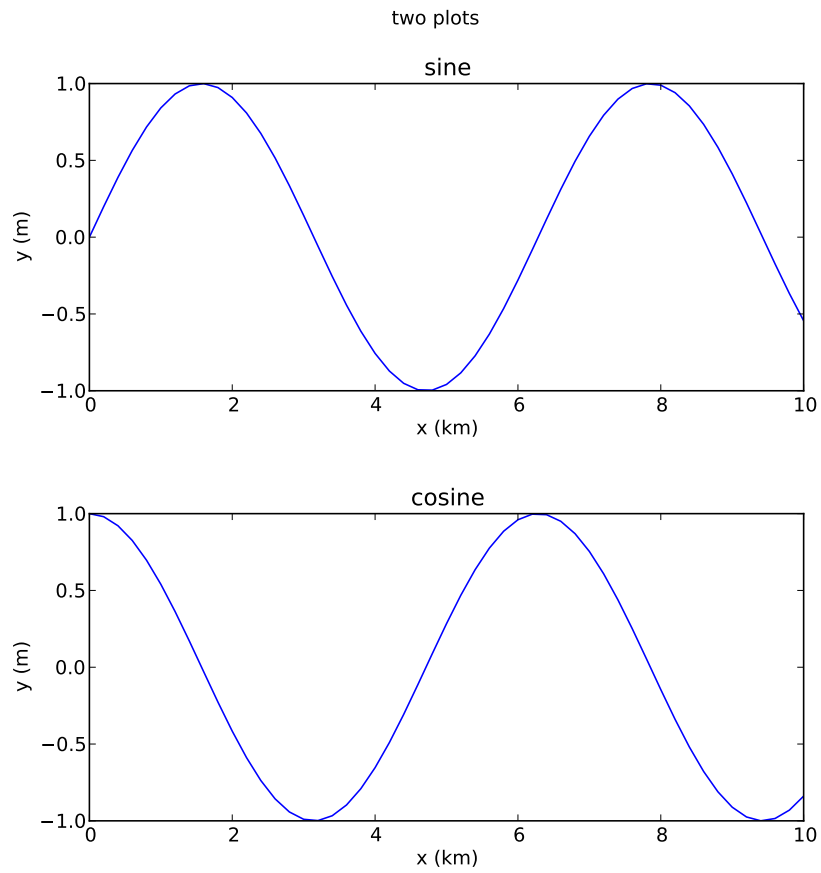


Figure 4.11: Figure with two different plots.

```
>>> l1 = plt.plot(x,a)
>>> lx = plt.xlabel("x (km)")
>>> ly = plt.ylabel("y (m)")
>>> ttl = plt.title("sine")
>>> p2 = plt.subplot(2,1,2)
>>> l2 = plt.plot(x,b)
>>> lx = plt.xlabel("x (km)")
>>> ly = plt.ylabel("y (m)")
>>> ttl = plt.title("cosine")
>>> sttl = plt.suptitle("two plots")
>>> plt.show()
>>>
```

The results are shown in figure 4.11. Notice that an “ubertitle” can be created using the *suptitle* command. Unfortunately, the font size of the ubertitle is less than the font size of the subplot titles. However, the font size of both title and ubertitle can be adjusted with the keyword parameter *fontsize*, e. g., `fontsize=16`.

The `plt.subplots_adjust(hspace=0.4)` command increases the height spacing of subplots from its smaller default value. This keeps the x axis label in the upper plot from overlapping with the title of the lower plot. Another optional keyword parameter for the `subplots_adjust` command is *hspace*, which solves similar problems for horizontal spacing. The use of the `subplots_adjust` command is also illustrated in figure 4.4.

Clearly Matplotlib still has some rough edges when it comes to font size and plot spacing, but at least the tools to fix these problems are available!

4.5.2 Overlaying plots

Figure 4.7 demonstrates that line plots (e. g., plots produced by *plot*, *contour*, *quiver*, etc.) can be overlaid on a filled contour or a pcolor plot. In addition, line plots may be overlaid on each other. We can exert fine control over the order in which these are plotted using the *zorder* keyword option in these plotting commands. For instance including `zorder=0` in one plot and `zorder=5` in another makes the second plot appear on top of the first. Text always appears on top of all graphics.

4.6 Making plot output files

The call to *show()* in the above examples pops up a window with your plot. Various icons are displayed at the bottom of the window which allow you to manipulate the plot in various ways such as zooming in, changing the aspect ratio, etc. The right-most icon pops up a window which allows you to specify an output file for the plot. Various image formats are available as well as postscript, encapsulated postscript, and PDF.

4.7 Further information

The Matplotlib page at Sourceforge <http://matplotlib.sourceforge.net/> is the primary reference for Matplotlib. This documents all plotting commands and has a link to a User Guide. The latest source code can also be downloaded here.

The version documented here is 0.99.1. As noted, a few serious omissions make versions earlier than 0.98 less useful for certain purposes.

Chapter 5

Accessing standard file formats

In many fields of science standard data formats are used. Here we describe packages which implement a few of these standards, thus allowing Python users ready access to such data.

5.1 NetCDF3

NetCDF is a format developed by the Unidata project of the University Corporation for Atmospheric Research. Maintenance and further development has been taken over by the National Center for Atmospheric Research (NCAR). NetCDF is used in a wide variety of fields outside of the atmospheric sciences due to its simplicity, robustness, and ability to provide information about the contents of a data file within the file itself, so called *metadata*. NetCDF3 maintains the general structure and philosophy of NetCDF present from its creation. NetCDF4 constitutes a major departure from the initial philosophy, and is described separately below.

NetCDF stores data in the form of multidimensional arrays. Each array is called a *variable*. Associated with each variable is information about its dimensionality, size along each axis, the data type of the array elements, and *attributes* of the variable, including such things as units and missing values which indicate when data do not exist for particular elements. Information is also stored separately about the *dimensions* of variables existing in the file. Not all variables have to have the same number of dimensions – variables can even be zero-dimensional, or *scalars*. Dimensions are fixed in length and cannot be changed in an existing NetCDF file with the exception of one, called the *unlimited* dimension. This is provided so that a file can be expanded along that dimension. Typically this dimension is used to provide, for example, records of variables at successive times.

In addition to the above, the NetCDF file may contain *global attributes* which provide overall information about the data in the file. A prime example is the *history* attribute, which contains a record of what has been done to the file. Attributes can take the form of numbers or strings, or tuples of these data types. NetCDF imposes no preconditions on the type or number of attributes, though useful conventions have developed in particular scientific fields.

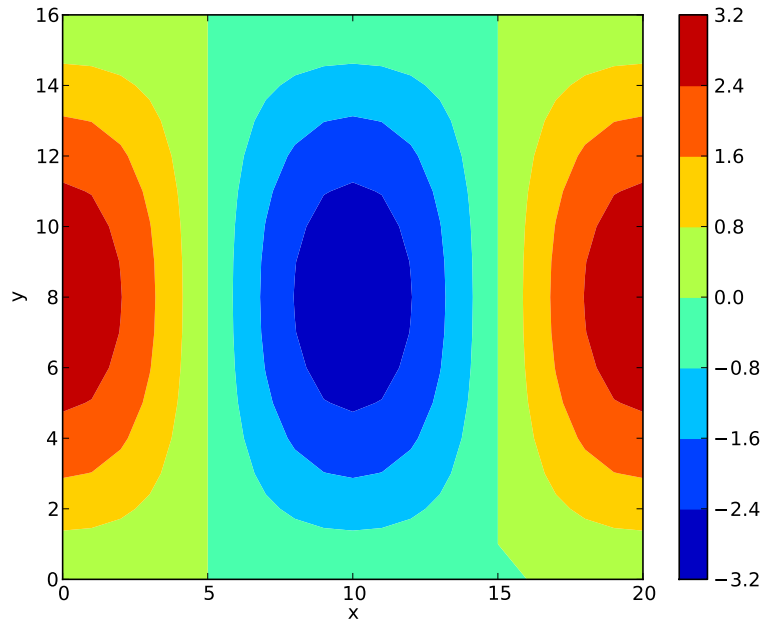


Figure 5.1: Plot of array created for first NetCDF example.

5.1.1 Creating NetCDF files

As our first example, here is a Python script which creates an example of a two-dimensional array, which is plotted and then written to a NetCDF file:

```
#!/usr/bin/python

# Create a sample netcdf file.
from numpy import *
import matplotlib.pyplot as plt

# Use the netCDF3 package of Jeffrey Whitaker. This package is
# similar to the Scientific Python netcdf package, but works with
# NumPy instead of the older (and generally obsolete) Numeric package.
from netCDF3 import *

# Alternatively, use Roberto De Almeida's Pupynere NetCDF package,
# which is written totally in Python. This form of import changes
# the class name to be consistent with that of Whitaker's package.
#from pupynere import NetCDFFile as Dataset

# Create some interesting fields.
x = linspace(0,20,21)
```

```

y = linspace(0,16,17)

# A mis-feature of meshgrid is that it assumes that the most rapidly
# varying index in accessing sequential positions in memory is the
# first one. However, consistent with the C-array-type default of
# NumPy, the last index should vary most rapidly. The netCDF package
# follows the default NumPy convention, so we reverse x and y here to
# limit the spread of this contagion.
(Y,X) = meshgrid(y,x)
a = 3*cos(6.28318*X/20)*sin(3.14159*Y/16)

# Plot our new function to see what it looks like. Notice that we
# have to take the transpose of a to make it plot correctly, in
# agreement with the above comment. This wouldn't need to be done
# if we had not reversed the order of x and y in the call to meshgrid.
# However, it would have then caused problems later in the creation
# of the NetCDF file.
c = plt.contourf(x,y,a.transpose())
ax = plt.axis([0,20,0,16])
b = plt.colorbar(c,orientation='vertical')
xl = plt.xlabel("x")
yl = plt.ylabel("y")
plt.show()

# Write them to a netcdf file.
f = Dataset('create_example_nc.nc','w')
setattr(f,'history','sample nc file from Python')
dx = f.createDimension('x',21)
dy = f.createDimension('y',17)
xvar = f.createVariable('x','f4',('x',))
yvar = f.createVariable('y','f4',('y',))
avar = f.createVariable('a','f4',('x','y'))
svar = f.createVariable('s','i4',())

# The indexing on the left sides of the assignments below is needed to
# tell Python that the netcdf variables should be assigned the values
# of the corresponding arrays, not just redefined out of existence!
xvar[:] = x[:]
yvar[:] = y[:]
avar[:,:] = a[:]

# A different technique is used to assign a value to a scalar variable,
# since the array slicing technique doesn't work in that case.
svar.assignValue(42)

```

```

# A convention used in many applications of NetCDF is that a missing
# value be specified, indicating the existence of missing data in grid
# cells this occurs.
setattr(xvar,'missing_value',1.e30)
setattr(yvar,'missing_value',1.e30)
setattr(avar,'missing_value',1.e30)
setattr(svar,'missing_value',99999)
f.close()

```

Here is a dump of the header of the resulting NetCDF file using the utility *ncdump*, which is included with NCAR's NetCDF package:

```

$ ncdump -h create_example_nc.nc
netcdf create_example_nc {
dimensions:
x = 21 ;
y = 17 ;
variables:
float x(x) ;
x:missing_value = 1.e+30 ;
float y(y) ;
y:missing_value = 1.e+30 ;
float a(x, y) ;
a:missing_value = 1.e+30 ;
int s ;
s:missing_value = 99999 ;

// global attributes:
:history = "sample nc file from Python" ;
}

```

The plot of the variable "a" is shown in figure 5.1.

The above Python script is largely self-explanatory and covers most of what one needs to do to write a new NetCDF file from scratch. If one desires to make one of the dimensions the unlimited dimension, assign it a length of 0 or None in the `f.createDimension` variable. If one of the variables is a scalar, the `assignValue` method must be used instead of the direct assignment.

5.1.2 Reading NetCDF files

Reading a NetCDF file takes some different techniques. After importing NumPy and netCDF3 in the usual way, we open an NetCDF file for reading:

```

>>> from numpy import *
>>> from netCDF3 import *
>>> f = Dataset('create_example_nc.nc')
>>>

```

The names of the global attributes are extracted using the `ncattrs` method applied to the file instance. These names are attributes of the instance and can be extracted using standard Python syntax:

```
>>> f.ncattrs()
['history']
>>> f.history
'sample nc file from Python'
>>>
```

The dimensions of a NetCDF file are represented in a dictionary and the length of each dimension can be extracted with the `len()` function:

```
>>> f.dimensions.keys()
['y', 'x']
>>> len(f.dimensions['y'])
17
>>>
```

The variables of a NetCDF file are also represented in a dictionary:

```
>>> f.variables.keys()
['y', 'x', 's', 'a']
>>>
```

We can extract the variables using standard dictionary notation:

```
>>> xvar = f.variables['x']
>>> yvar = f.variables['y']
>>> avar = f.variables['a']
>>> svar = f.variables['s']
>>>
```

Various important aspects of the variables (including the attributes) can be extracted:

```
>>> avar.ncattrs()
['missing_value']
>>> avar.missing_value
1e+30
>>> avar.dimensions
('x', 'y')
>>> avar.size
357
>>> avar.shape
(21, 17)
>>>
```

The shape attribute can be used to define NumPy variables to receive the actual data:

```

>>> x = zeros(xvar.shape)
>>> y = zeros(yvar.shape)
>>> a = zeros(avar.shape)
>>> a.shape
(21, 17)
>>>

```

The data for arrays can then be transferred by simple assignment with slicing. Scalar variable data can be transferred by the `getValue()` method:

```

>>> x[:] = xvar[:]
>>> y[:] = yvar[:]
>>> a[:] = avar[:]
>>> svar = f.variables['s']
>>> s = svar.getValue()
>>> s
array([42])
>>>

```

Plotting shows that we end up with the same array `a` with which we started.

The NetCDF variable instances such as `xvar`, `yvar`, and `avar` are technically not NumPy arrays, but they have the indexing properties of this data type. This means that transfer of only part of the data from the variable instance to a smaller NumPy array can be done using normal NumPy slicing:

```

>>> b = zeros(yvar.shape)
>>> b.shape
(17,)
>>> b[:] = avar[4,:]
>>> b
array([[ 0.00000000e+00,  1.80859119e-01,  3.54767919e-01,
         5.15043259e-01,  6.55525744e-01,  7.70816803e-01,
         8.56485903e-01,  9.09240723e-01,  9.27053988e-01,
         9.09241199e-01,  8.56486797e-01,  7.70818174e-01,
         6.55527472e-01,  5.15045285e-01,  3.54770213e-01,
         1.80861533e-01,  2.46002105e-06])
>>>

```

In the above example, the NumPy array `b` is created with the shape of the `y` variable and a slice in the `y` direction is transferred to `b`.

5.2 NetCDF4

NetCDF version 4 is still considered to be somewhat experimental, and so we will delay investigating its capabilities.

5.3 Candis

Not yet invented.

5.4 Further information

Information on NetCDF may be obtained from the National Center for Atmospheric Research at <http://www.unidata.ucar.edu/software/netcdf/>. NCAR itself has produced a module allowing Python to access NetCDF files, called PyNIO; <http://www.pyngl.ucar.edu/Nio.shtml>. This module is well documented and allows the user to access a number of other formats, including GRIB1, GRIB2, and HDF 4. However, this module is tightly connected to a wide array of other data analysis tools produced by NCAR, which makes it rather difficult to install from source code.

The first module to allow Python access to NetCDF files is a part of the Scientific Python package <http://www.scipy.org/>. The difficulty from our point of view is that this module works with the Python array package *Numeric*, a predecessor to NumPy. Unfortunately, there are enough incompatibilities between the two array packages to make use of this NetCDF module problematic.

Jeffrey Whitaker of the National Oceanic and Atmospheric Administration's Physical Sciences Division of the Earth System Research Laboratory (<http://www.cdc.noaa.gov/>) has developed Python packages for both NetCDF3 and NetCDF4; <http://code.google.com/p/netcdf4-python/>. These packages build and install with little fuss and are at least minimally documented.

Roberto De Almeida (<http://dealmeida.net/projects/>), a Brazilian oceanographer, has created a package for NetCDF version 3 which is written totally in Python using the Python memory map module. This package also seems to work well. It requires that the Python `setuptools` module.