PHY224 Python Review

Release

Aug 29, 2017

CONTENTS

CHAPTER

ONE

TUTORIAL 1

Python is generally used as a scripting language : Scripts are files that contain programs and functions. Scripts interact with data files and other programs. A script file (here named *script.py*) can be run from the command line using the `python` command.

```
python script.py
```

Expressions and functions can also be called from an *interactive interpreter* launched by the `python` command without listing a script to run.

```
python

#My version prints the following at the start
Python 3.6.0 |Anaconda custom (x86_64)| (default, Dec 23 2016, 13:19:00)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

1.1 Introduction

1.1.1 Interpreter / Shell

You can use the Python interpreter as a calculator

```
In [2]: 3+5

Out[2]: 8

In [3]: 3.141592 / 2

Out[3]: 1.570796

In [4]: (1+1+2+3+5)*5 - 24/3

Out[4]: 52.0

In [5]: 9**2

Out[5]: 81
```

Python is an evolving language and has some quirks. In Python 2 (the older version of Python) the calculation `9/4` produces a different answer to Python 3 (the latest version of Python).

This happens because the '/' operator is treated differently in the two versions, but produces the right answer in both cases if you understand the behavior. In Python 2, when you divide two `integers` (numbers without a decimal point), the answer will also be the nearest `integer`. In Python 3, the divide operator always converts `integers`

into `floating point` (or `real`) numbers. Including the decimal point in your floating point numbers will help protect you from this problem. Unlike in *Matlab*, there is no `./` operator to force a floating point division.

```
In [6]: 9 / 4
```

```
Out[6]: 2.25
```

```
In [7]: 9. / 4
```

```
Out[7]: 2.25
```

You can force an integer division using the // operator

```
In [8]: 9//4
```

```
Out[8]: 2
```

1.1.2 Whitespace

Whitespace is often important in Python. Whitespace characters are characters in the program that often appear empty, like spaces and tabs. When writing Python code everything with the same indentation is treated as the same block of code, to be executed in order.

The number of spaces or tabs at the start of each line is important and needs to be consistent. The number of spaces/tabs you use is not important but 4 spaces for each indentation level is common. Some text editors allow you to show whitespace characters (to maintain a consistent indentation) and convert tabs to spaces (to save typing 4 spaces each time, for example).

1.1.3 Comments

Any text appearing after a # will be treated as a comment and ignored by Python, e.g.

```
# A comment at the start of a line
val = 1. #another comment, val is a value.
#val= -1
```

`val` is set to `1` in the above code. The line changing it to -1 is a comment and is not executed.

1.1.4 Functions

A function is a block of code that can be called by other programs or functions. Functions usually perform some work, and might return a value, save a file, plot a graph, etc.

Functions are called using their name and providing the function values in a comma separated list in parentheses (also known as the arguments to the function), e.g.

```
add(3,2)
```

calls the function `add` with two arguments, 3 and 2. The function name can be meaningless but should describe the behavior of the function if possible. If the answer from `add(3,2)` was not 5 you would be annoyed!

1.1.5 Numbers

Python has built in numerical types like `integers`, `float`, `long`, and `complex`.

- `integers` range from -2 billion to + 2 billion.

- `long` numbers are integers that can be infinitely large.
- `float` numbers are 64-bits long, and can hold numbers from about -2e300 to 2e300, and can be as small as 2e-300. `float`ing point numbers typically hold 14 digits of accuracy.
- `complex` numbers are made from two `float` numbers.

If you mix numerical types in your code, Python will convert values to the `largest` type it needs, usually `integers -> long -> floats -> complex`. You can force a conversion using the functions `float`,`int`,`long`, and `complex`.

```
In [9]: float(1)
```

```
Out[9]: 1.0
```

```
In [10]: int(3.14159)
```

```
Out[10]: 3
```

```
In [11]: complex(3)
```

```
Out[11]: (3+0j)
```

```
In [12]: complex(float(3.14159),int(2.71828))
```

```
Out[12]: (3.14159+2j)
```

You can check the type of a variable with the `type` function

```
In [13]: type(1.0)
```

```
Out[13]: float
```

```
In [14]: type(1)
```

```
Out[14]: int
```

1.1.6 Strings

Strings are defined using single or double quotation marks

```
'a string'
"also a string"
```

Strings can be concatenated with the + operator, and duplicated with the *** operator.

```
In [15]: a='eggs '
         print(a)
```

```
eggs
```

```
In [16]: print(a+a)
```

```
eggs eggs
```

```
In [17]: print(a*4)
```

```
eggs eggs eggs eggs
```

Characters in a string can be accessed using square brackets "[]", e.g.

```
In [18]: b=a*4 # eggs eggs eggs eggs
         print(b[0]) #indexing starts at 0! This is the first 'e'
```

```
e
```

```
In [19]: print(b[14]) #The space between words is also a character
```

```
In [20]: print(b[13]) #s in Python 2
```

```
s
```

1.1.7 Logic and Boolean data types.

Python supports logical values (also known as Boolean data types) with the values *True* and *False*. Every other data type in Python can be interpreted as a Boolean data type, most of them are equivalent to the Boolean *True*. Special Values that are equivalent to *False* include the number 0, an empty string, an empty list/array. In general, you shouldn't rely on these equivalent values and should check explicitly for the values you want to consider.

Python also supports the logical operators *and*, *or*, and *not*, and the comparison operators <,<=,==,!=,>=,>.

```
True and False #False
True or False #True
not True #False
True and not False #True
a < b # True if a is less than b
a <= b # True if a is less than or equal to b
a == b # True if a is equal to b
a != b # True if a is not equal to b
a >= b # True if a is greater than or equal to b
a > b # True if a is greater than b
a and True # True if a is interpreted as True
a and False # False always.
```

and four examples of possibly unexpected resultsv (assuming a is defined)

```
a or True # a
True or a # True
a or False # a
False or a # a
```

1.1.8 Variables

You can save values in `variables`

```
x=3
```

means save the value of 3 in the variable `x`. Whatever value was previously in `x` is lost unless you store it somewhere else. Variables can be reused in other expressions, including saving them to other variables

```
x=3
y=x
z=x * x
x=0 #changes x from 3 to 0, but doesn't affect y or z
y=0 #changes y to 0
```

1.1.9 Operators

The common mathematical operators work as expected in Python

- x + y -> add

- x - y -> subtract
- x * y -> multiply
- x / y -> quotient (divide)
- x // y -> integer quotient (divide)
- -x -> negate x (unary minus)
- x**y -> raise x to the power of y
- x % y -> remainder of x / y

1.1.10 Loading modules, libraries, or packages

Python provides many standard modules (also called libraries, or packages) for you to use, and many others are available for free from package *repositories* like **pypi.python.org**

Loading a module is called *import*ing. You can import an entire module using

```
from math import *
```

or just part of the package

```
from math import sin, cos
```

You can import a module and keep it as a separate *namespace* to protect your functions from being overwritten,

```
import math #use math.sin instead of just sin
```

```
In [21]: import math #import the library
         math.sin(0.5) #Calculate the sin of 0.5

Out[21]: 0.479425538604203

In [22]: import math as mylib #import and 'rename' the library inside your code.
         mylib.sin(0.5)

Out[22]: 0.479425538604203

In [23]: from math import sin,cos #just import sin and cos
         sin(0.5)

Out[23]: 0.479425538604203

In [24]: from math import * #import all functions from the library into your 'namespace'
         sin(0.5)

Out[24]: 0.479425538604203
```

Importing entire modules using `import *` can be covenient, but can also be dangerous. If two packages define the same function or variable the last package to be imported overwrites any earlier imports.

1.2 Exercise 1

What are the results of the following expressions? You should use the Python shell to get your answer.

1. False and True
2. True or False

3. True and False or True
4. 2>0
5. not True
6. not 1
7. 3*5+8
8. 3*(5+8)
9. (3*5)+8
10. What is $sin(90)$? and $sin(1.570596)$? (using the sine function from the math package)
11. What is 98/6?
12. What is the remainder of 98/6?
13. What is 2^2, $2^2 + 2$, 2^{2+2}?
14. Calculate the base 3 logarithm of the number 27. The math package provides the natural log function *log* and the base 10 function *log10*.

1.3 Functions and Objects

1.3.1 Defining functions

Functions in Python can be created using the following syntax

```
def function_empty():
    pass #literally do nothing

def function_taking_parameters(parameter, keyword=default_value):
    some code

def function_returning_a_value(parameter, keyword=default_value):
    some code
    return value
```

The function name can be (almost) any string. There can be zero or more *parameters*, whose position is important, and zero or more *keywords*, whose position is not important. Keywords always have a default value associated with them. If you call the function without a keyword, it will be available inside the function with the default value.

```
In [25]: def func(a,b,c,d=1,e=2,f=3):
             print(a,b,c,d,e,f)

         func(1,2,3) # prints 1,2,3,1,2,3
         func(1,2,3,d=4,e=5,f=6) # prints 1,2,3,4,5,6
         func(1,2,3,e=5) #prints 1,2,3,1,5,3
         func(1,2,3,e=5,d=4) # prints 1,2,3,4,5,3
         # func(1,2) # error, not enough positional parameters
```

```
1 2 3 1 2 3
1 2 3 4 5 6
1 2 3 1 5 3
1 2 3 4 5 3
```

Functions can return no value, a single value, or a group of values

```
def func1():
    return
def func2():
    return 1
def func3():
    return 1,2
```

1.3.2 Containers

Python provides a number of containers - objects designed to hold other variables.

1.3.3 Lists

The *list* container stores an *ordered* group of variables whose ordering is determined by their position in the list.

```
list1 = [0,1,2,3,4,5] # a list of integers
list2 = ['a','b','c','d','e'] # a list of strings
list3 = [0,'b','c',3,4,'f'] # a list of strings and integers.
```

Each element, or entry, in the list can be accessed using square brackets, starting from element 0

```
list1[0] # 0
list2[0] # 'a'
list3[5] # 'f'
```

Accessing negative elements is the same as counting from the end of the list, the last element in the list being `-1` (**not** `-0`).

```
list1[-2] # 4
list1[-5] # 1
```

You can create a list using square brackets `[]` or the `list` function.

```
a=list() # an empty list, this calls the 'list' function
b=[] # a second empty list
```

1.3.4 Dictionaries

A dictionary stores a group of `keys` and `values` in a way that lets you look up the `key` quickly and get the `value` from the dictionary.

```
dict1={'a':1,'b':2,'c':3}
dict2=dict(a=1,b=2,c=3)
```

The values stored in the dictionary are accessed using the square brackets

```
dict1['a'] # 1
dict2['b'] # 2
```

and values can be updated (or new key,value pairs created) by assigning with the = sign

```
dict1['d'] = 4 # a new key 'd' with value 4
dict1['a'] = 3 # changes the value of the 'a' key to 3
```

Python allows many strings and data types to be used as the key, and anything for value.

1.3.5 Tuples

Tuples are like lists, storing a group of values, but once created cannot be changed. They are said to be ***immutable***. Changing the value in a tuple is not allowed.

```
tuple1 = ('a','b',3)
tuple1[0] = 1 # an Error
```

Tuples can be "unpacked" into separate variables by assigning it to a comma separated list of variables

```
a,b,c = tuple1 #the values are a='a', b='b', c=3
```

When 'unpacking' the tuple in this way, the number of elements on the left side of the equality should be the same number as the length of the tuple.

1.3.6 Sets

Sets are like lists, but only store a value once, and support 'Set logic' operations like *intersections* and *unions*. You can define a set using

```
set1 = set([0,1,2,3,4])
set2 = {0,1,2,3,4}
```

The *union* of two sets is the set that contains **all elements that exist in both sets**. The *intersection* includes **only elements that exist in both sets**. The *difference* includes **elements that exist in one set but not the other**.

1.3.7 Objects

Many variables in Python are treated as an `object`, with one or more *values* and *methods*. Values are like variables that are stored inside the object, `number` or `string`. Methods are functions that are allowed to access the values inside an object.

Numbers and strings are also objects in Python, although most of the mechanics are hidden from the user.

The container types listed above are all examples objects. The *values* they hold are the items in the container. The *methods* are functions that can perform operations on the data inside the container.

In general, to call a *method* you use the syntax

```
object.method(parameters) # if the method needs data
object.method() # if no data is required.
```

For the containers above, the following *methods* are commonly used

```
#l is a list object
l=list([0,1,2,3,4])
l.append(5) #add the value 5 to the end of the list
l.insert(3,5) #insert the value 5 into the list at position 3, starting from 0.
len(l) # return the number of elements in the list
l.remove(5) #remove the value 5 from the list, but only once
l.pop(4) #remove the fourth element from the list.
l.extend([3,4]) # extend the list l with the list [3,4]
```

```
#d and d2 are dictionaries
d=dict(a=1,b=2)
d2=dict(c=3)
d.keys() #return a list of all keys in the dictionary
d.values() #return a list of all values in the dictionary
len(d) # return the number of keys (or values) in the dictionary
d.update(d2) #update the dictionary d with the dictionary d2

#sa and sb are sets
sa=set([0,1,2,3])
sb=set([2,3,4,5,2])

sa.union(sb)
sa.difference(sb)
sa.intersection(sb)
```

1.4 Exercise 2

Define the following containers

```
la = [1,2,3,4]
lb = [5,6,7,8]
lc = [9,10,11,12]
va = 13
vb = 14
vc = 15
```

What does the *la* list look like after the following operations (reset the variables after each stage)?

1. extend *la* with *lb*
2. append *lb* onto *la*
3. append *va* onto *la*
4. insert *va* at index 3 of *la*
5. remove the element with value 3 from *la*
6. remove (pop) the element at index 3 from *la*

1.5 Control Structures

1.5.1 Loops: for and while

Looping in Python happens using a *for* loop, or sometimes a *while* loop. *For* loops run the block of code for each element in a sequence (e.g. a list of numbers). A *while* loop runs the block of code for as long as a *condition* is True.

```
for element in sequence:
    body_of_code

while condition:
    body_of_code
```

The simplest *for* loop is to run the code N times. You can use the `range(N)` function to generate a list of integers from zero to N:

```
for counter in range(N):
    body_of_code

#equivalently
counter=0
while counter < N:
    body_of_code
    counter+=1
```

For loops should be used when you have a list of items to iterate through (even if they are a list of numbers constructed just for the loop). *While* loops should be used when you don't know how many times the code will run

```
In [26]: #a better use for the while loop
         from numpy.random import randint # choose a random integer
         sumdice=0
         counter=0
         while sumdice!=12:
             #roll a pair of dice until they're faces add to 12 (two sixes!)
             #randint is quirky, and needs the maximum value to be 1 higher
             dice = [randint(1,7),randint(1,7)]
             sumdice=dice[0]+dice[1]
             counter+=1
         print(counter) #should be 1/36th of the rolls, but it's random and can finish at any time
```

```
19
```

1.5.2 Conditional code: if, elif and else.

Blocks of code can be made optional using the *if* statement. The code is executed **only if** the logical expression after the *if* statement evaluates to True.

```
#single block of code
if test1:
    block_of_code

#code with an alternate block
if test1:
    block_of_code
else:
    alternate_block_of_code

#many alternate blocks. only 1 will run
if test1:
    block_of_code
elif test2:
    alternate_block_of_code
elif test3:
    second_alternate
else:
    final_alternate_block_of_code
```

The 'logical expressions' above (e.g. test1) can be composed of any logical expression, combined with *or* and *and*, negated with *not*, may include function calls, or simply be equal to *False* (then the code will never run, of course).

For loops and if statements can be nested. However, if the nesting becomes too deep, consider the readability of the code, and perhaps move the inner code to a new function.

```
#version 1
for loop1 in list1:
    for loop2 in list 2:
        if test1:
            result = code1
        elif test2:
            for loop3 in list3:
                result = code2
        else:
            for loop3 in list4:
                result = code3

#version 2
for loop1 in list1:
   for loop2 in list2:
        result = function_call(loop1, loop2)
```

For loops can be used on a sequence of elements that aren't numbers, either by counting the number of elements and generating a list of numbers with *range*, or by using the list of elements directly. For example

```
fruit_list = ['apple','banana','carrot','durian','eggplant','fig']

for fruit in fruit_list:
    print fruit

#or

for fruit_counter in range(len(fruit_list)):
    print fruit_ist[fruit_counter]
```

You can combine the two loops with the *enumerate* that provides the index and value inside the loop

```
for index,value in enumerate(fruit_list):
    print value, fruit_list[index] #the same string is printed.
```

1.6 Exercise 3

The python function `range(N)` will generate and return a list of numbers from 0 to N (** not including N **) 1. Write a for loop that loops from 0 to 19 and prints each number. 2. Write a for loop that loops from 0 to 19, print each number > 8 3. Write a for loop that loops from 0 to 19, print each even number > 8. 4. Write a while loop that counts from 10 down to 1, printing each number. 5. Write a for loop that loops from 6 to 100 and prints the number *only* if it doesn't divide by 2, 3, or 5 exactly. (that is, with no remainder after division). Your output should start with 7,11,13 and end with 91,97.

1.7 Numpy and Scipy

Python lists are inefficient for mathematical operations, both in writing code and running the code. For efficient calculations you should you a numerical Python library like `numpy`. `numpy` provides a special type of array that can be treated just like a number in calculations (something you can't do with `list`) and provides a large number of specialized functions for working with these arrays.

A related library called *scipy* (an abbreviation of Scientific Python) contains a large number of functions for common and not-so-common scientific equations and algorithms. For example, Fourier transforms (*scipy.fftpack*), numerical integration (*scipy.integrate*), and random number generators (*scipy.random*).

One caveat with *numpy* is that it contains functions that are also available in the *math* module **with the same name**. The *numpy* functions usually accept normal numbers and return the right answer. The *math* library functions will not take the special *numpy* arrays.

```
from numpy import sin
from math import sin
sin(0.5) #This calls the math version,
         #which does not take arrays or
         #lists as a parameter/argument.
```

1.7.1 Arrays

Numpy provides the *array* object, and many functions that work on the array as a whole (e.g. *sum, product, length, max, min*) or each element of the array in isolation (e.g. *sin,cos,power,exp*).

```
from numpy import *
a = zeros(5) #make an array of five zeros.
b = arange(5) # make an array of five numbers from 0 to 4
c = ones(5) #make an array of five ones with integer type
d = zeros(5,int) # An array fives zeros with integer type
e = zeros(5,float) # A floating point array filled with zeros
f = zeros((2,2)) # make an array of 4 zeros, in two dimensions
```

The *arange* function, and its alternatives *linspace*, and *logspace* generate a series of numbers when given the correct arguments. *arange* takes the arguments for the start of the array, end of the array, and step size of the array. *linspace* takes the start of the array, end of the array, and *number of elements* in the array. *logspace* does the same in logarithmic space.

```
arange(5) # 0,1,2,3,4
arange(0,5) # 0,1,2,3,4
arange(3,5) # 3,4
arange(0,5,2) # 0,2,4
linspace(0,5,6) # 0,1,2,3,4,5
linspace(0,5,5) # 0,1.25,2.5,3.75,5.
linspace(0,5,1) # 0
linspace(0,5,2) # 0,5
logspace(-2,2,5) # 0.01,0.1,1,10,100
```

Notice that the last value returned from *arange* is not the argument used in the function call. It's usually 1 less. On the other hand, *linspace* does include the upper value.

The *range* function built into Python provides a similar function for lists to the `arange` function in `numpy`.

Accessing the array elements is similar to accessing a list element

```
b[4] # element 4 of array b
f[0,1] # element in row 0, column 1
```

You can also access sections of the array using *slicing* notation. In this notation you can specify the *start* element, the *stop* element, and the *stepping*. **Remember that the *stop* element is not included in the slice**.

```
b[1:5:1] # element 1,2,3,4 from array b
b[1:5:2] # 1,3
```

```
b[1:5]    # 1,2,3,4
b[:5]     # 0,1,2,3,4
b[1:]     # 1,2,3,4
b[:5:2]   # 0,2,4
b[1::2]   # 1,3
b[::2]    # 0,2,4
```

The double colon (::) notation is shorthand for the `slice` function, which takes the argument `slice(start, stop,step)`, and works the same way as the `arange` function.

```
b[slice(1,5,2)] # 1,3
```

Arrays can be used as arguments to functions, like *sin*, and treated as variables in equations

```
sin(b)
b*2
b**3 + 2
```

1.7.2 Modules and Libraries

In addition to using libraries written by others you can reuse your own code as a module/library by saving the code in a file and *import*ing that file. For example, if you wrote a function to calculate the first N Fibonacci numbers (the list of numbers defined as $F_n = F_{n-1} + F_{n-2}$,starting with the numbers $F_0 = 1$ and $F_1 = 1$).

```
def fibN(N):
    if N==0:
        s=[]
    elif N==1:
        s=[1]
    elif N>=2:
        s=[1,1]
        for i in range(2,N):
            s.append(s[-1]+s[-2])
    return s
```

you could save this function in a file called 'fib.py'. Then in a new program you can simply write

```
import fib
print(fib.fibN(8))

>[1, 1, 2, 3, 5, 8, 13, 21]
```

This only works if Python knows where to find your code. By default Python will search its library, and the current directory. You can add a directory to the path stored in *sys.path*. You should change the path before you try to import the new package

```
import sys
sys.path.append('the_full_path_to_my_directory')
import fib
```

1.8 Exercise 4

Use the Numpy package where you can. Save all the code you generate to a text file.

1. Generate an array of 30 floating point values from 0 to 29 inclusive. Calculate the mean value. (There may be a function to calculate the `mean` of the array, if not there is a function to calculate the `sum` and `len` or `size` of the array).
2. Take the array generated in question 1, subtract the mean value and square the result. Calculate the mean value of this new array.
3. Using functions from the *numpy* package, calculate the `mean`,`min`,`max`,`std` (standard deviation), and `var` (variance) of the array in question 1. Which one corresponds to the your answer to question 2.
4. Generate 12 values from 0 to 90 degrees using the `arange` function. Calculate the `sin` of each value and print the results. What angle units was the sine function expecting?
5. Repeat question 4 with the `linspace` function from the numpy library.
6. Write a function that takes an argument 'x' and returns the value calculated from the quadratic equation 'y=2x^2 +3x +1'.
7. Call the function you wrote in question 6 with the value 2.0, and the *numpy* array generated in question 4 (i.e. 12 values from 0-90 degrees)
8. Write a function that takes a single number and calculates its factorial. The factorial is sometimes written as '!' and N! is the product of integers from 1 to N inclusive, so 3! is 123. Make sure that the code produces the correct answer for N=0 (`0!=1`). Calculate the value of 9!

1.9 Plotting with Pylab

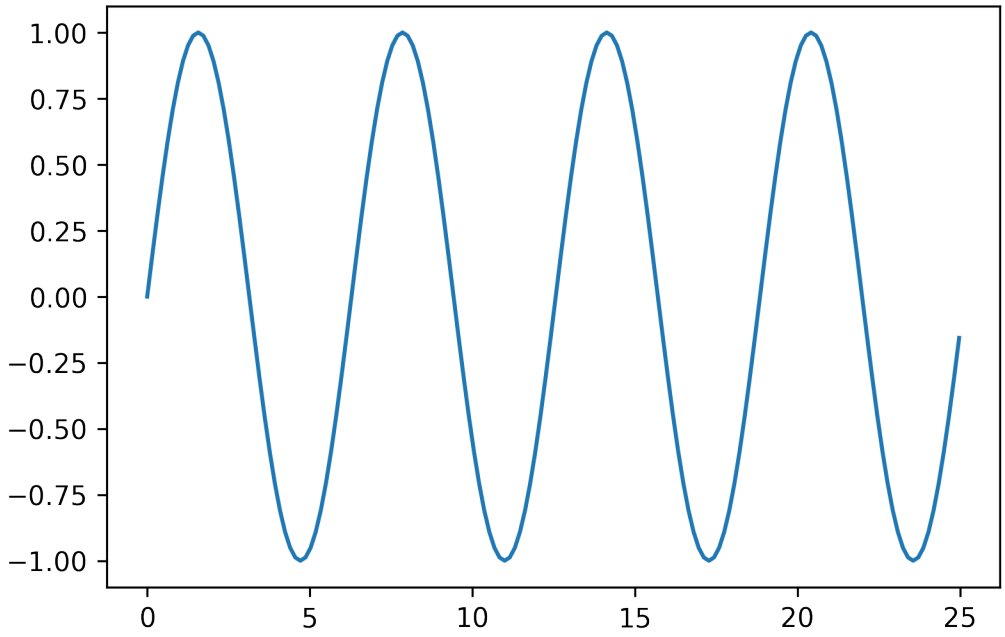
The most common way of generating line plots,maps, or contour plots from data is using a package called *matplotlib*. This package exposes a lot of control over the style and structure of the plots you can generate. Much of the control can be hidden using a simpler interface called *pylab* .

pylab includes packages from *matplotlib*, *numpy*, and *scipy* to provide a basic scientific toolkit. You can import pylab like any other library:

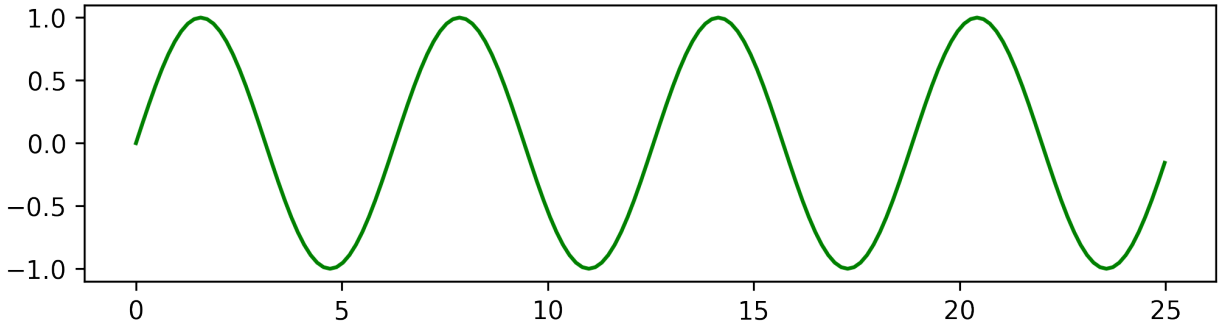
```
import pylab
```

Then you can access the plotting functions it provides

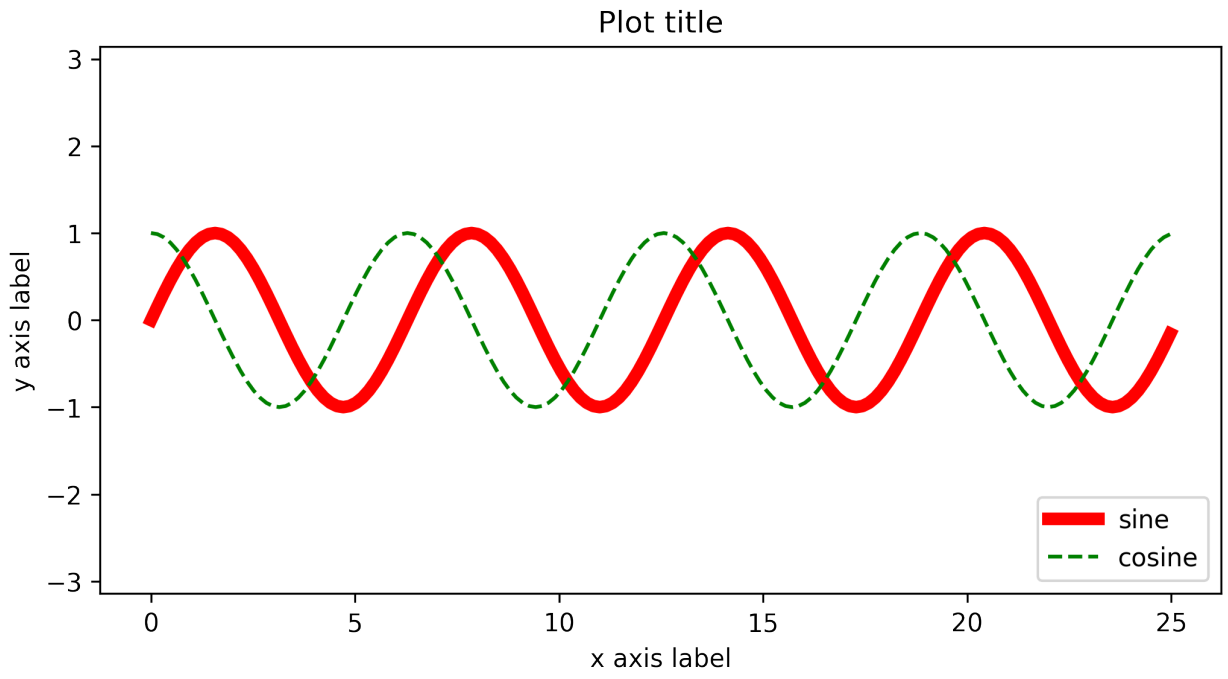
```
In [27]: import pylab
         #If don't specify the figure size,
         #Python will use its default values.
         x=pylab.arange(0,8*pylab.pi,0.05*pylab.pi)
         y=pylab.sin(x)
         pylab.plot(x,y); # A Sine wave
```




```
In [28]: pylab.figure(figsize=(8,2)) #different figsize
         pylab.plot(x,y,color='green'); #A green sine wave
```




```
In [29]: pylab.figure(figsize=(8,4))
         # a thick line
         pylab.plot(x,pylab.sin(x),color='red',label='sine', linewidth=5)
         # a dashed line
         pylab.plot(x,pylab.cos(x),color='green',label='cosine',linestyle='--')
         #print a figure legend using the line labels
         pylab.legend(loc='lower right')
         # add a label to the x axis
         pylab.xlabel("x axis label")
         pylab.ylabel('y axis label')
         #extend the y axis limits to +- 3.14159
         pylab.ylim(-pylab.pi,pylab.pi)
         pylab.title("Plot title");
```



1.9.1 Reading data with Pylab

Pylab provides the function *loadtxt* to read data from a text file. and *savetxt* to save data to a text file. You can optionally specify a delimiter used to separate the data (e.g. commas, tabs, spaces, semi-colons) instead of the default method which considers any whitespace:

```
loadtxt("data.txt", delimiter=',') #look for commas to separate fields.
```

tell pylab to skip the file header with the *skiprows* keyword

```
loadtxt("data.txt", skiprows=5) #ignore the first five rows
```

use only certain columns with *usecols*

```
loadtxt("data.txt", usecols=(0,3)) #use only columns 0 and 3
```

split a file with many columns into individual variables

```
loadtxt("data.txt", unpack=True)
```

loadtxt does not provide a way of ignoring the **last** lines of a file.

Saving files works in a similar way, using the *savetxt* function. *savetxt* takes the filename and the data array as parameters and optional keywords like 'delimiter'.

```
savetxt("data.txt",data)
```

```
In [30]: from numpy import arange,savetxt,loadtxt
         #Generate a single column of data
         data = arange(10.)
         print ("data={0}".format(data))
         savetxt("output_files/mydata.txt",data,delimiter=',') #save the data
```

```
data_loaded = loadtxt("output_files/mydata.txt",delimiter=',') #load the data
print ("data_loaded={0}".format(data_loaded))

#generate 30 complex numbers and arrange in 3 columns of 10 rows each
data2 = arange(30,dtype=complex).reshape(10,3)
print ("data2={0}".format(data2[3,:])) #only print the row at index 3 (the 4th row!)
savetxt("output_files/mydata2.txt",data2,delimiter=',') #save the data

#load the data, you need to tell loadtxt that the data is complex.
data_loaded2 = loadtxt("output_files/mydata2.txt",delimiter=',',dtype=complex)
print ("data_loaded2={0}".format(data_loaded2[3,:]))
```

```
data=[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
data_loaded=[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
data2=[  9.+0.j  10.+0.j  11.+0.j]
data_loaded2=[  9.+0.j  10.+0.j  11.+0.j]
```

1.9.2 Saving figures with Pylab

Figures can saved from pylab with the *savefig* command. Figures can be saved different formats such as 'pdf' (vector) files and 'png' (image) files. Some pylab packages support other formats (like postscript 'eps' for journal publication and 'svg' for web-browser use). The first argument to savefig should be the file name, including the format

```
savefig("plot.pdf") #save a pdf image in plot.pdf
```

Some formats support extra options, such as resolution ('dpi') for images, bounding boxes for postscript formats, and transparency to make using the figure on a different background color (or overlayed on another image) possible.

```
from pylab import arange, plot, savefig
plot(arange(10))
savefig("plot.pdf") #save a pdf
savefig("plot.png") #save a png
savefig("plot300.png",dpi=300,transparent=True) # high resolution
                                                # transparent background
```

1.10 Exercise 5

Save each figure in a different file.

1. Plot the $sin^{-1}(x)$ function for x values from -1 to +1.
2. Plot the functions $sin(x)$ and $sin^2(x)$ from -5 to 5. Make sure the lines can be identified and are labelled.
3. Plot the functions $\frac{e^x - e^{-x}}{2}$ and the "hyperbolic sine function" (sinh) from -5 to 5. Make sure both lines are visible.
4. Write a function to calculate the sinc function (i.e. $\left(\frac{sin(x)}{x}\right)^2$) . Make sure you calculate the correct answer for x=0. Plot the sinc function from -10 to 10.
5. Plot the cumulative sum (use `cumsum`) of the sinc function from -50 to 50. What is the sum? How many points did you plot? The integral of the sinc function over this range is about 3.12169. What is the maximum value in your plot? Why is your sum different to the integral?

1.11 Data Analysis with Python

The goal of many lab experiments is to determine the relationship between two quantities. One quantity x can be controlled and is called the **independent variable**, the second quantity y is measured and is called the **dependent variable** (y depends on x).

The relationship between x and y can often be found by fitting a function f that takes the independent variable as an argument $f = f(x)$. Then the value of x can be changed throughout an experiment to provide *N* different values, and the result of calling the function $f(x)$ for each value of x can be compared against the measured value y. If the function f is a good model of the real experiment, the comparison will be good enough (we'll define "good" later).

The procedure for fitting data in this way is common enough that Python provides the function *curve_fit* that takes the data collected and the function being tested and returns quantitative results about the goodness of the fit function. *curve_fit* is used in its simplest form as

```
from scipy.optimize import curve_fit
popt, pcov = curve_fit(func, xdata, ydata)
```

where the input arguments are

- func = the function being tested as a fit function. The format is special and discussed next
- xdata = the independent variables in a list or numpy array
- ydata = the dependent variables in a list or numpy array

and the output is

- popt = a list of parameters that, when given to func, produce the fit to the data with smallest errors
- pcov = the variance of each parameter and the co-variance between each pair of parameters

The function being tested can be any function written in Python providing it is in the form

```
def func(xdata, p0,p1,p2):
   code_to_model_the_experiment
   return value
```

where the p0,p1,p2 are the parameters being fit. You can also define the function as

```
def func(xdata, *p):
     code
     return value
```

where the '*' tells Python to accept any number of arguments and store them as a list in p, so `p0` becomes `p[0]` . This can be useful when you are testing different functions and don't want to rewrite the function call each time. If you use this method, you must tell *curve_fit* how many parameters are expected by providing an initial set of values for the parameters in the `p0` keyword, calling curve_fit as

```
popt, pcov = curve_fit(func, xdata, ydata,p0=(arg1,arg2,arg3))
```

1.11.1 The linear regression method.

With data collected in for x and y, a linear regression uses a function $f(x)$ that is linear in the unknown parameters, for example

$$f(x) = a + bx$$

is linear in a and b (and x),

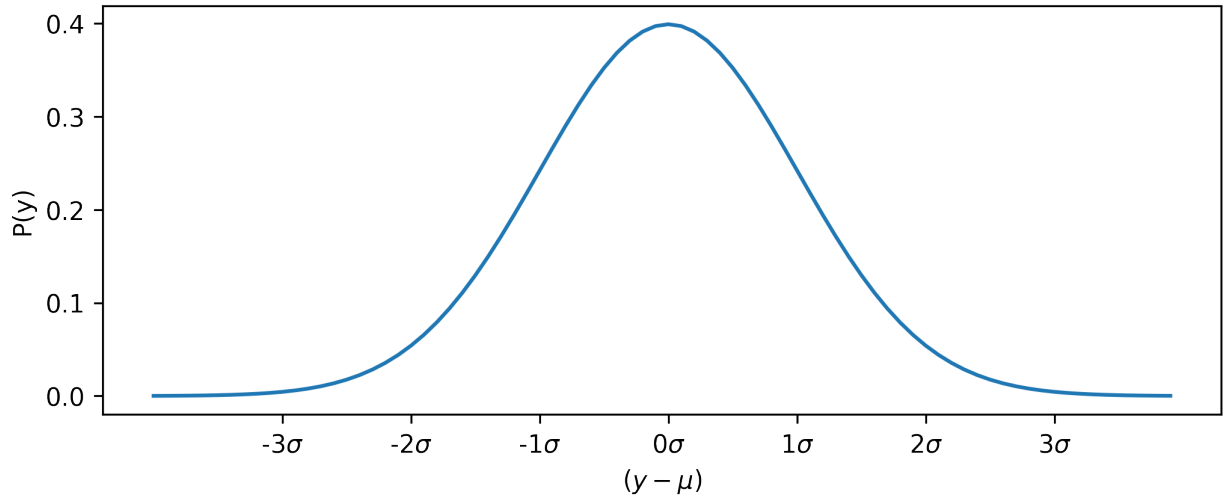
$$f(x) = a + bx + cx^2$$

is linear in a,b, and c (but not x)

Linear regression calculates the unknown parameters that creates the function f that is closest to the measured value y assuming that errors in the measured values have errors from measurement that obey **Gauss' distribution**. For an observation with the true value μ, an observation y has the following probability of occurring:

$$P(y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

```
In [31]: #Plotting the Gaussian function in Python
         from pylab import arange,sqrt, exp, plot, xticks, xlabel, ylabel, pi,figure
         y=arange(-4,4,0.1)
         sigma=1.0
         mu=0.0
         p=1./sqrt(2*pi*sigma**2) * exp(-(y-mu)**2/(2*sigma**2))
         figure(figsize=(8,3))
         plot(y,p)
         s="$\sigma$"
         xticks([-3,-2,-1,0,1,2,3],['-3'+s,'-2'+s,'-1'+s,'0'+s,'1'+s,'2'+s,'3'+s])
         xlabel("$(y-\mu)$")
         ylabel("P(y)");
```



The linear regression method starts by calculating the probability of obtaining **one** dependent value y_i

$$P_{a,b}(y_i) = \frac{1}{\sqrt{2\pi\sigma_y^2}} e^{-\frac{(y_i - f(x_i))^2}{2\sigma_y^2}}$$

The combined probability of all of the measurements occuring if the fitting function $a + bx$ is a good approximation is then the product of each single probability $P_{a,b}(y_i)$

$$P_{a,b}(y_1, y_2 \ldots, y_N) = P_{a,b}(y_1) P_{a,b}(y_2) \ldots P_{a,b}(y_N)$$

This value is proportial to the product of the exponent in each Gaussian term (and a normalizing constant that doesn't depend on the measured data). The **product of the exponents** of each probability is the same as the **exponent of the sum of the exponent terms**, i.e.

$$e^{v_1} e^{v_2} e^{v_3} = e^{v_1 + v_2 + v_3}$$

For the Gaussian probabilities, the sum of the exponents is called χ^2.

$$\chi^2 = \sum \frac{(y_i - f(x_i))^2}{\sigma_y^2}$$

This is the only part of the problem that depends on the data, and the highest value of Probability (i.e. the best function $f(x)$) is found by *minimizing* the value of χ^2 (minimizing because of the minus sign).

The function $f(x)$ that minimizes the value of χ^2 is called the *least-squares fit* to the data, or *least-squared-error fit*, because the expression being minimized is $(y - f(x))^2$ is the square of the error between the measured value y and the predicted value $f(x)$.

The value of χ^2 depends on the number of points N in the dataset. To make interpretation easier we can define a reduced χ_r^2 as

$$\chi_r^2 = \frac{\chi^2}{\mathrm{dof}}$$

where dof is the number of degrees of freedom, calculated as the difference between the number of data points N and the number of parameters being fit m ($f(x) = a + bx$ has two parameters, $m = 2$).

$$\mathrm{dof} = N - m$$

A fitting function that is a good approximation of the data has a $\chi_r^2 \approx 1$. Much higher and the data is 'underfit' (large errors still exist), much lower and the data is 'overfit' (small residuals suggest that the noise is being fit).

1.11.2 Non-linear Regression

If our fitting function is non-linear, the derivation above starts becomes harder. However, if the non-linear function can be converted to a linear function (and our assumptions about Gaussian errors are valid for the linearly transform function) the linear regression method can still be used.

For example, the function $y = ae^{bx}$ is non-linear in the parameters, but taking the logarithm creates $\ln y = \ln a + bx$. The linear regression method can be used to find the parameters $\ln a$ and b, using x as the independent data **but** $\ln y$ as the dependent data.

Using *curve_fit*, it's possible to fit a non-linear fit function using the same Python function. That means your function can be defined as $y = ae^{bx}$ and the data can be given to *curve_fit* unchanged.

1.11.3 Variance, Covariance, and Correlations

Variance measures the dispersion of values in one variable x, written as σ_x^2 and defined as

$$\sigma_x^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

Covariance describes the variance of two variables, written as σ_{xy}^2

$$\sigma_{xy}^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})$$

The sample coefficient for linear correlation (usually labelled r) measures the extent to which the dataset x, y have a linear relationship. Defined as

$$r = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

A covariance matrix contains the covariance of each pair of variables. For two variables

$$\begin{pmatrix} \sigma_{xx}^2 & \sigma_{xy}^2 \\ \sigma_{yx}^2 & \sigma_{yy}^2 \end{pmatrix}$$

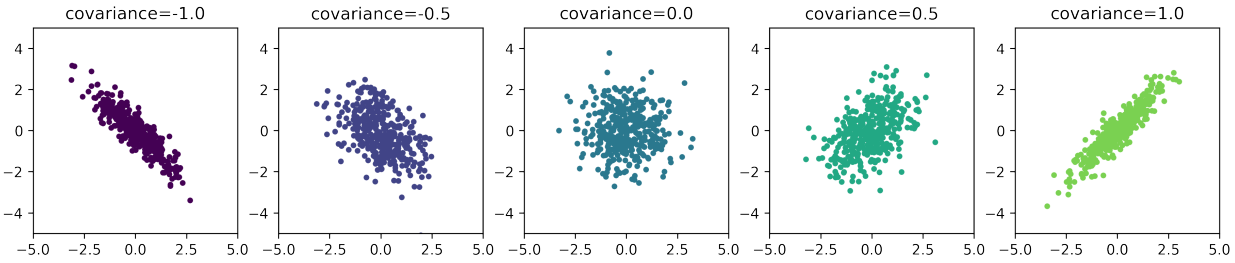
where σ_{xx}^2 is the variance of x.

Using *curve_fit*, the covariance matrix is returned as the second variable *pcov*, it is a 2D variable defined following the equation above. Variances are provided in the diagonal elements, covariances in the off-diagonal elements. The square-root of the diagonal elements are the standard deviations of the parameters. These numbers are usually used as the 'error' in the parameter fit to show the possible range of values for the parameter. For example a parameter with mean 4 and variance 9 can be written as 4 ± 3, suggesting a large range of likely values. The same parameter with variance 1 would be written as 4 ± 1, suggesting a much smaller range of likely values.

The off-diagonal elements describe how one variable changes as the other changes (accounting for different magnitudes if any), so a covariance of 0 means the two variables *change* independently, a positive indicates that a positive change in one variable coincides (or *correlates*) with a positive change in the other.

We can generate examples of this covariance matrix using *numpy*. In the following example, the variance of x and y are kept constant (near 1.0) while the covariances are changed from -1 to +1.

```
In [32]: from numpy.random import multivariate_normal, normal
         import pylab
         num_samples = 400
         cov = [-1.0,-0.5,0.0,0.5,1.0]
         #setup the figure
         pylab.figure(figsize=(15,10))
         mean=pylab.array([0.0,0.0])
         for i,covariance in enumerate(cov):
             #make a subplot in the figure for each covariance value
             #force the aspect ratio to 1 so that
             #circles look like circles and 1:1 gradients are at 45 degrees
             pylab.subplot(1,5,i+1,aspect=1)
             covmatrix = pylab.array([
                     [  1.1, covariance,],
                     [ covariance,  1.1,],
                 ])
             #generate random data with a known mean and covariance matrix.
             y = multivariate_normal(mean,covmatrix, size=num_samples)
             #plot each distribution, fix the x and y axis to the same size
             pylab.plot(y[:,0],y[:,1],'.',color=pylab.cm.viridis(i/5.))
             pylab.ylim(-5,5)
             pylab.xlim(-5,5)
             pylab.title("covariance={0}".format(covariance))
```

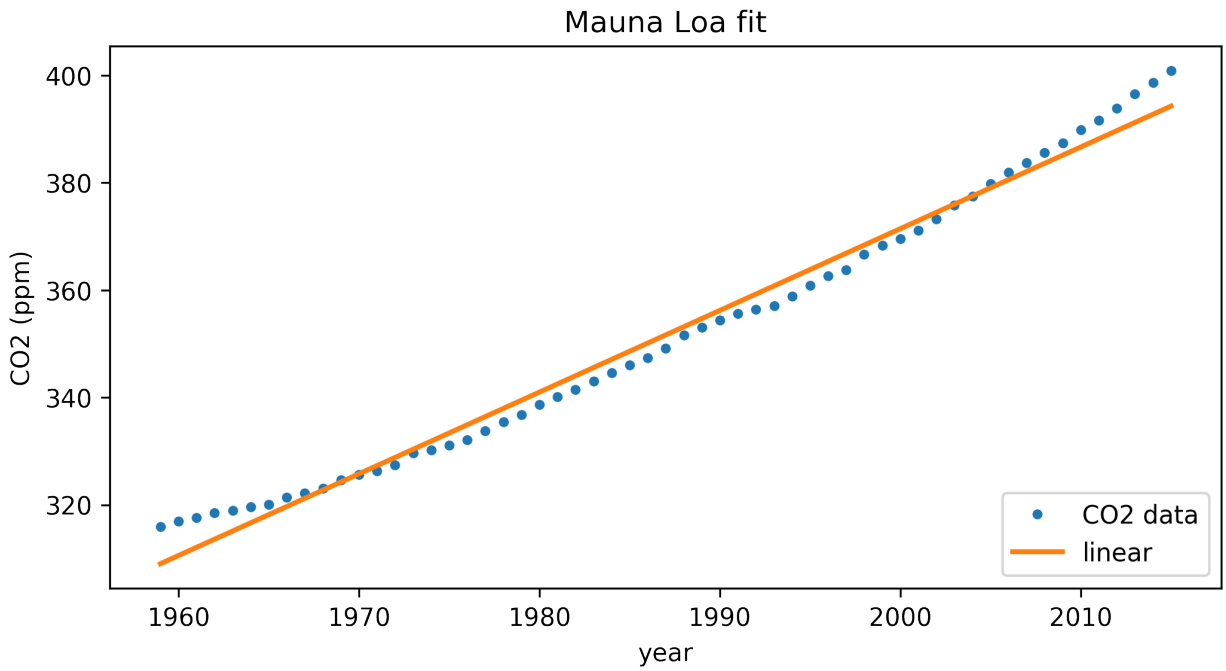


1.11.4 Worked example 1 - Mauna Loa CO2

Scientists have been measuring the amount of carbon dioxide in the atmosphere from a station on top of the Hawai'ian mountain Mauna Loa since the 1950's. Using a dataset that includes the annual mean value of co2 fit a line to this data

and calculate the growth rate in the amount atmospheric of carbon dioxide in the atmosphere.

```
In [33]: ### Worked example 1 - Mauna Loa data
         import pylab # import pylab
         source_file = "input_data/co2_annmean_mlo.txt" #the source file.
         comment_char="#" #The file includes a description of
                          #the dataset at the top the file.
                          #Each line is commented with a #

         year,co2 = pylab.loadtxt(source_file,
                                  usecols=(0,1),
                                  unpack=True,
                                  comments=comment_char)

         #fit a straight line
         def fitfunc(x,*p):
             return p[0] + p[1]*x

         #fit a quadratic line
         def fitquad(x,p1,p2,p3):
             return p1 + p2*x + p3*x**2

         #import the fitting routine
         from scipy.optimize import curve_fit

         #call the fitting routine, remember to give initial values for the parameters
         popt, pcov = curve_fit(fitfunc,year,co2,p0=[1,1],epsfcn=1.0,maxfev=200)

         #setup the figure
         pylab.figure(figsize=(8,4))

         #plot the data
         pylab.plot(year,co2,'.',label="CO2 data")

         #calculate the line fit by calling the fit function directly
         fitted_data = fitfunc(year,*popt)

         #plot the fitting line
         pylab.plot(year,fitted_data,
                    linewidth=2,
                    label="linear".format(*popt))

         #decorate the plot
         pylab.xlabel("year")
         pylab.ylabel("CO2 (ppm)")
         pylab.title("Mauna Loa fit")
         pylab.legend(loc='lower right');
         pylab.savefig("output_files/co2fit.pdf")
         print("f(year) = a + b*year".format(*popt))
         print("a = {0:9.3f} +- {1:6.3f} ppm".format(popt[0],sqrt(pcov[0,0])))
         print("b = {0:9.3f} +- {1:6.3f} ppm/year".format(popt[1],sqrt(pcov[1,1])))

         #import scipy
         #help(scipy.optimize.leastsq)

f(year) = a + b*year
a = -2672.020 +- 48.829 ppm
b =     1.522 +-  0.025 ppm/year
```



1.12 Exercise 6

1. Mauna Loa data

   (a) The value of a is known as the intercept because it's the value of f(x) when x=0 (i.e. f(0)). It does not make sense for this value to be referenced to year 0 because negative CO2 concentration is impossible and the record starts in 1958. Change the independent variable or the fitting function so that the year 1960 is removed from the value of *year* before it's used in the fitting routine. This will change the meaning of a to be the value at f(1960) instead of f(0). What is the new values for the fitting parameters `a` and `b`? One way of doing this is to subtract 1960 from the data as soon as it's read into the program, but there may be a better way.

   (b) The straight line fit is not very good. Calculate the reduced χ^2 value for this data (the variance of the data is 0.12^2). Add a quadratic (x^2) term to the fit function and recalculate the fit. What is the new reduced χ^2? Is the quadratic fit better than the straight line fit?

   (c) The slope of the line in this example is the rate of change (or 'growth rate') of carbon dioxide in the atmosphere. What is the growth rate in 2015 using the straight line fit? and the quadrative fit from question b?

CHAPTER

TWO

PYTHON EXAMPLES

2.1 Print a list as a comma separated string

```
In [11]: # Print out a list by creating a string of elements joined with a comma

         datalist = [4,2,1,5,3,0,1,0,2,4,1,0,0,0,2,5,8,9,9,9,8,7,4]

         def list_to_string(alist):
             #join the elements together with a comma
             strlist=[]
             for val in alist:
                 strlist.append(str(val)) #make sure we are dealing with strings
             return ", ".join(strlist)

         print("My list is\n {0}".format(list_to_string(datalist)))
```

```
My list is
 4, 2, 1, 5, 3, 0, 1, 0, 2, 4, 1, 0, 0, 0, 2, 5, 8, 9, 9, 9, 8, 7, 4
```

2.2 Remove the last element in a list

```
In [12]: #Remove the last element in a list
         #by returning a new list
         datalist = [4,2,1,5,3,0,1,0,2,4,1,0,0,0,2,5,8,9,9,9,8,7,4]

         def remove_last_element(alist):
             #remove the last element in the list
             all_but_last = alist[:-1]
             return all_but_last

         newlist = remove_last_element(datalist)
         print("My list without the last element is\n  {0}".format(list_to_string(newlist)))
         print("My original list is still\n  {0}".format(list_to_string(datalist)))
```

```
My list without the last element is
  4, 2, 1, 5, 3, 0, 1, 0, 2, 4, 1, 0, 0, 0, 2, 5, 8, 9, 9, 9, 8, 7
My original list is still
  4, 2, 1, 5, 3, 0, 1, 0, 2, 4, 1, 0, 0, 0, 2, 5, 8, 9, 9, 9, 8, 7, 4
```

2.3 Save all elements matching value

```
In [13]: #Save all elements matching value
         datalist = [4,2,1,5,3,0,1,0,2,4,1,0,0,0,2,5,8,9,9,9,8,7,4]

         def save_all_elements_matching_value(alist,value):
             #save all of the elements that match the value into a new list
             newlist=list()
             for item in alist:
                 if item==value:
                     newlist.append(item)
             return newlist

         save_zeros = save_all_elements_matching_value(datalist,0)
         nlist_str = list_to_string(save_zeros)
         print("Saving the items in the array that match the value is\n {0}".format(nlist_str))
```

```
Saving the items in the array that match the value is
 0, 0, 0, 0, 0
```

2.4 Save all elements with a particular length

```
In [14]: fruits = ["Apple","Banana","Currant","Damson","Elderberry","Fig",
                   "Gooseberry","Huckleberry","Jabuticaba","Kiwifruit",
                   "Lemon","Mango","Nectarine","Orange","Papaya","Quince",
                   "Raspberry","Strawberry",
                   "Tomato","Ugli fruit", "Yuzu"]

         def save_all_string_with_length(alist,length):
             #save all of the elements that match the value into a new list
             newlist=list()
             for item in alist:
                 if len(item)==length:
                     newlist.append(item)
             return newlist

         l=6
         savelist = save_all_string_with_length(fruits,l)
         nlist_str = list_to_string(savelist)
         print("Fruits with names of length {0} include\n {1}".format(l,nlist_str))
```

```
Fruits with names of length 6 include
 Banana, Damson, Orange, Papaya, Quince, Tomato
```

2.5 Remove matching elements from a list, *destructively*.

```
In [15]: ### Remove matching elements from a list, *destructively*.
         #The original list is altered
         datalist = [4,2,1,5,3,0,1,0,2,4,1,0,0,0,2,5,8,9,9,9,8,7,4]

         def remove_matching_elements_inplace(alist, value):
             #This time, don't create a new list to remove elements
             #if we loop upwards through the list we affect the elements
             #above us we haven't looked at yet. If we loop downwards we
             #affect elements we have looked at already.
```

```
            #Be careful with the range here,
            #We start at len(alist)-1 . The index of the last element
            #We want to end at 0, but range always stops one before the value we ask for
            #so we ask for stop="-1"
            #We're counting backwards, so the "step" is -1.
            for i in range(len(alist)-1,-1,-1):
                if alist[i] == value:
                    alist.pop(i)

        remove_matching_elements_inplace(datalist,0)
        nlist_str = list_to_string(datalist)
        print("after removing zeros my original list is\n {0}".format(nlist_str))
```

```
after removing zeros my original list is
 4, 2, 1, 5, 3, 1, 2, 4, 1, 2, 5, 8, 9, 9, 9, 8, 7, 4
```

2.6 Remove matching elements, with a while loop

```
In [16]: #Using a while loop
         datalist = [4,2,1,5,3,0,1,0,2,4,1,0,0,0,2,5,8,9,9,9,8,7,4]

         def remove_matching_elements_inplace3(alist, value):
             #count up, not down
             #recheck the size of the list each time
             #and be careful with the counter
             i=0
             while i<len(alist):
                 if alist[i] == value:
                     alist.pop(i)
                     #if we find a value, pop it from the list,
                     #but do not increase the counter.
                     #we removed an item from the list, so
                     #every item in the list above i moved
                     #down one item
                 else:
                     i+=1 #found nothing, increment counter

         remove_matching_elements_inplace3(datalist,0)
         nlist_str = list_to_string(datalist)
         print("After removing zeros my original list is\n {0}".format(nlist_str))
```

```
After removing zeros my original list is
 4, 2, 1, 5, 3, 1, 2, 4, 1, 2, 5, 8, 9, 9, 9, 8, 7, 4
```

2.7 Another way of removing values

```
In [17]: datalist = [4,2,1,5,3,0,1,0,2,4,1,0,0,0,2,5,8,9,9,9,8,7,4]
         def ex9_remove(alist, value):
             for i in range(alist.count(value)):
                 #the remove method looks for the first occurrence of value and removes it.
                 alist.remove(value)
```

```
        ex9_remove(datalist,0)
        nlist_str = list_to_string(datalist)
        print("after removing zeros my original list is\n {0}".format(nlist_str))
```

```
after removing zeros my original list is
 4, 2, 1, 5, 3, 1, 2, 4, 1, 2, 5, 8, 9, 9, 9, 8, 7, 4
```

2.8 Count each element in the list

```
In [18]: datalist = [4,2,1,5,3,0,1,0,2,4,1,0,0,0,2,5,8,9,9,9,8,7,4]

         def count_each_element(alist):
             #count the occurrence of each item in the list
             #find the unique elements. 'sets' is a
             #quick way of doing this because it
             #only holds one of each item

             unique=list(set(alist))
             count=list()
             for i in range(len(unique)):
                 #count the number of entries with the count function,
                 #passing it the value of the i'th element in unique
                 c=alist.count(unique[i])
                 count.append( c )
             return unique, count #we need to return the values in
                                  #unique and the count in count

         def plural_s(c):
             #help to get the print statement correct with plural numbers
             if c != 1 :
                 return "s"
             else:
                 return "" #an empty string

         values,count = count_each_element(datalist)
         for i in range(len(values)):
             print("{0} appears {1} time{2}".format(values[i],count[i],plural_s(count[i])))
```

```
0 appears 5 times
1 appears 3 times
2 appears 3 times
3 appears 1 time
4 appears 3 times
5 appears 2 times
7 appears 1 time
8 appears 2 times
9 appears 3 times
```

2.9 Count each element in the list. using a dictionary

```
In [19]: datalist = [4,2,1,5,3,0,1,0,2,4,1,0,0,0,2,5,8,9,9,9,8,7,4]

         def count_each_element(alist):
             #count the occurrence of each item in the list
             count=dict()
```

```
            for v in alist:
                #count the number of entries with the count function,
                #passing it the value of the i'th element in unique
                if v in count.keys():
                    count[v]+=1
                else:
                    count[v]=1
            return count #the dictionary hold the number and count

        def plural_s(c):
            #help to get the print statement correct with plural numbers
            if c != 1 :
                return "s"
            else:
                return "" #an empty string

        count = count_each_element(datalist)
        for k,v in count.items():
            print("{0} appears {1} time{2}".format(k,v,plural_s(count[i])))
```

```
4 appears 3 times
2 appears 3 times
1 appears 3 times
5 appears 2 times
3 appears 1 times
0 appears 5 times
8 appears 2 times
9 appears 3 times
7 appears 1 times
```

2.10 Sorting a list

```
In [20]: datalist = [4,7,2,6,0,1]

         sort_list = sorted(datalist)
         print("My original list is\n {0}".format(list_to_string(datalist)))
         print("After sorting the list it is\n {0}".format(list_to_string(sort_list)))
         print("and my original list is still\n {0}".format(list_to_string(datalist)))
```

```
My original list is
 4, 7, 2, 6, 0, 1
After sorting the list it is
 0, 1, 2, 4, 6, 7
and my original list is still
 4, 7, 2, 6, 0, 1
```

2.11 In place sorting

```
In [21]: datalist = [4,7,2,6,0,1]
         print("My original list is\n {0}".format(list_to_string(datalist)))

         datalist.sort()
         print("After sorting in-place we've")
```

```
        print("changed our original list to\n {0}".format(list_to_string(datalist)))
```

```
My original list is
 4, 7, 2, 6, 0, 1
After sorting in-place we've
changed our original list to
 0, 1, 2, 4, 6, 7
```

2.12 Return the largest value of two numbers

```
In [22]: def return_the_largest(a,b):
             #return whichever value is larger
             if a>b:
                 return a
             else:
                 return b

         a,b=6,3
         print("The largest value of {0} and {1} is {2}".format(a,b,return_the_largest(a,b)))
         a,b=3,-3
         print("The largest value of {0} and {1} is {2}".format(a,b,return_the_largest(a,b)))
```

```
The largest value of 6 and 3 is 6
The largest value of 3 and -3 is 3
```

2.13 Loop through values from 0 to a, comparing against b

```
In [23]: def compare_range_against_b(a,b):
             #loop from 0 to a, comparing the size of the counter against b
             c=0
             for c in range(a):
                 if c>b:
                     print("{0} is greater than {1}".format(c,b))
                 elif c<b:
                     print("{0} is less than {1}".format(c,b))
                 else:
                     print("{0} is equal to {1}".format(c,b))

         a=5
         b=2
         compare_range_against_b(a,b)
```

```
0 is less than 2
1 is less than 2
2 is equal to 2
3 is greater than 2
4 is greater than 2
```

2.14 Two dimensional loop

```
In [25]: #count from 0,0 to a,b
         def two_dimensional_loop(a,b):
```

```
            #loop from 0-a, repeat b times
            for ib in range(b):
                #The inner loop goes fastest, and we want to loop 0-a fastest
                for ia in range(a):
                    print("Loop {0},{1}".format(ia,ib))

        a,b=3,2
        two_dimensional_loop(a,b)
```

```
Loop 0,0
Loop 1,0
Loop 2,0
Loop 0,1
Loop 1,1
Loop 2,1
```

2.15 Two dimensional loop

```
In [26]: #where the inner loop depends on the outer loop
         def triangle_loop(a):
             #loop from 0-a, then 1-a, then 2-a, then 3-a, until it's just a
             for ib in range(a):
                 line=""
                 for ia in range(ib,a):
                     line=line+"({0},{1}) ".format(ib,ia)
                 print(line)
         a,b=6,3
         triangle_loop(a)
```

```
(0,0) (0,1) (0,2) (0,3) (0,4) (0,5)
(1,1) (1,2) (1,3) (1,4) (1,5)
(2,2) (2,3) (2,4) (2,5)
(3,3) (3,4) (3,5)
(4,4) (4,5)
(5,5)
```

2.16 Time

```
In [27]: import time
         print("The time is {0}".format(time.time())) #This is the time, but in
         #milliseconds since 1970 Jan 1

         #using the time package to time your code
         #start the timer
         start_time = time.time()

         #do something slow
         Nrand=1000000
         s=0
         import random
         for i in range(Nrand):
             s=s+random.random()

         #stop the timer
         end_time = time.time()
```

```
        print("It took {0} seconds ".format(end_time-start_time) +\
              "to add {0} random numbers".format(Nrand))
        print("The mean value of the random "\
              "numbers is {0}. It should be 0.5".format(s/Nrand))
```

```
The time is 1503950968.4647412
It took 0.12888503074645996 seconds to add 1000000 random numbers
The mean value of the random numbers is 0.49956732630161493. It should be 0.5
```

2.17 Math package

```
In [29]: from math import sin,cos,pi
         print("Sine(0) is {0}".format(sin(0)))
         print("Cosine(0) is {0}".format(cos(0)))

         print("Sine(pi) is {0}".format(sin(pi)))
         print("\tThis should be zero, but the computer uses an approximation of 'pi'")
         print("\tthat is wrong after the sixteenth digit")
         print("Cosine(pi) is {0}".format(cos(pi)))
         print("Cosine(pi/2) is {0}".format(cos(pi/2)))
         print("\tThe same rounding error as above")
```

```
Sine(0) is 0.0
Cosine(0) is 1.0
Sine(pi) is 1.2246467991473532e-16
        This should be zero, but the computer uses an approximation of 'pi'
        that is wrong after the sixteenth digit
Cosine(pi) is -1.0
Cosine(pi/2) is 6.123233995736766e-17
        The same rounding error as above
```

2.18 Importing Numpy

```
In [34]: import numpy
         import numpy as np

         print (numpy.cos(1.0))
         print(np.cos(1.0))
```

```
0.540302305868
0.540302305868
```

2.19 Generating arrays with numpy

```
In [35]: import numpy as np

         #generate 10 numbers from 0 to 10
         a1 = np.arange(10)
         print("a1 is {0}".format(a1))
```

```
a1 is [0 1 2 3 4 5 6 7 8 9]
```

```
In [36]: #Numbers from 0 to 150 spaced by 10
         a2 = np.arange(0,150,10)
         print("a2 is {0}".format(a2))
```

```
a2 is [  0  10  20  30  40  50  60  70  80  90 100 110 120 130 140]
```

```
In [37]:
         #10 Numbers from 0 to 150
         a3 = np.arange(0,150,150/10.)
         print("\na3 is {0}".format(a3))
```

```
a3 is [   0.   15.   30.   45.   60.   75.   90.  105.  120.  135.]
```

```
In [38]:
         #A 2d array from 0 to 25
         a4 = np.arange(25.)
         a4 = a4.reshape((5,5))
         print("\na4 is \n{0}".format(a4))
```

```
a4 is
[[  0.   1.   2.   3.   4.]
 [  5.   6.   7.   8.   9.]
 [ 10.  11.  12.  13.  14.]
 [ 15.  16.  17.  18.  19.]
 [ 20.  21.  22.  23.  24.]]
```

```
In [39]: #transpose a4
         a4transposed = a4.T
         print("\na4transposed is \n{0}".format(a4transposed))
```

```
a4transposed is
[[  0.   5.  10.  15.  20.]
 [  1.   6.  11.  16.  21.]
 [  2.   7.  12.  17.  22.]
 [  3.   8.  13.  18.  23.]
 [  4.   9.  14.  19.  24.]]
```

```
In [40]: #multiply two arrays
         a5 = np.arange(3)
         a6 = np.arange(3)+2
         print("\na5 is {0}, a6 is {1}".format( a5,a6 ))
```

```
a5 is [0 1 2], a6 is [2 3 4]
```

```
In [41]: #multiply the two arrays
         print("\na5*a6 is {0}".format( a5*a6 ))
```

```
a5*a6 is [0 3 8]
```

```
In [42]: #calculate the vector dot product
         print("\n(dot product) a5.a6 is {0}".format( np.dot(a5,a6) ))
```

```
(dot product) a5.a6 is 11
```

```
In [43]: #calculate the cross product
         print("\n(cross product) a5 x a6 is {0}".format( np.cross(a5,a6) ))
```

```
(cross product) a5 x a6 is [-2  4 -2]
```

```
In [44]: #calculate the sum of the elements
         print("\nsum(a5) is {0}".format( np.sum(a5) ))
```

```
sum(a5) is 3
```

```
In [45]: #calculate the mean of the elements
         print("\nmean(a5) is {0}".format( np.mean(a5) ))
```

```
mean(a5) is 1.0
```

```
In [46]: #generate a 3x3 identity matrix
         i3 = np.eye(3)
         print("\n3x3 identity matrix is \n{0}".format(i3))
```

```
3x3 identity matrix is
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

2.20 Accessing arrays

```
In [48]: import numpy as np

         oned = np.arange(5)

         print("\n oned is {0}".format( oned ))
         print("\n oned[0] is {0}".format( oned[0] ))
         print("\n oned[::-1] is oned reversed, and is {0}".format( oned[::-1] ))
```

```
 oned is [0 1 2 3 4]

 oned[0] is 0

 oned[::-1] is oned reversed, and is [4 3 2 1 0]
```

```
In [49]: #Two dimensional array
         import numpy as np

         twod = np.arange(15).reshape((5,3))

         print("\n twod is \n{0}".format( twod ))
         print("\n twod[0] is {0}, the whole first row!".format( twod[0] )) #The entire row!
         print("\n twod[0,1] is {0}".format( twod[0,1] )) #The first row, second column
```

```
 twod is
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]

 twod[0] is [0 1 2], the whole first row!
```

```
 twod[0,1] is 1
```

```
In [50]: print("\n twod[:,1] is {0}, the whole second column!".format( twod[:,1] ))
```

```
 twod[:,1] is [ 1  4  7 10 13], the whole second column!
```

```
In [55]: print("\n twod[::2,1] is {0}, every second element".format( twod[::2,1] ))
         print(" from the second column")
```

```
 twod[::2,1] is [ 1  7 13], every second element
 from the second column
```

```
In [54]: print("\n twod[::2,::2] is every second element")
         print(" from the every second column\n{0}".format( twod[::2,::2] ))
```

```
 twod[::2,::2] is every second element
 from the every second column
[[ 0  2]
 [ 6  8]
 [12 14]]
```

2.21 Using numpy arrays in a calculation

```
In [57]: import numpy as np
         a=np.arange(4)
         b=np.arange(4)
         c=np.arange(10,14,1)

         print("a={0}, b={1}, c={2}".format(a,b,c))

         #a*a + b*b
         d=a**2 + b**2
         print("a^2 + b^2 is {0}".format( d ))

         k=2.*a+5.
         print("2a+5 is {0}".format(k))
```

```
a=[0 1 2 3], b=[0 1 2 3], c=[10 11 12 13]
a^2 + b^2 is [ 0  2  8 18]
2a+5 is [  5.   7.   9.  11.]
```

2.22 Using numpy arrays in function calls

```
In [58]: import numpy as np
         a=np.arange(3, dtype=float)
         b=np.arange(3, dtype=float)
         c=np.arange(10,13,1, dtype=float)

         #Adding some exponentials
         f = (np.exp(-c) + np.exp(c))/2.
         print("(exp(-c) + exp(c))/2. is {0}".format(f))

         #Hyperbolic cosine
         g=np.cosh(c)
```

```
        print("cosh(c) is {0}".format(g))

        print("mean(c) is {0}".format( np.mean(c) ))
        print("std(c) is {0}".format( np.std(c) ))
        print("median(c) is {0}".format( np.median(c) ))

        def myproduct(arr):
            p=1.
            for value in arr:
                p=p*value
            return p

        print("myproduct(c) is {0}".format(myproduct(c)))
        print("numpy's product(c) is {0}".format(np.product(c)))
```

```
(exp(-c) + exp(c))/2. is [ 11013.2329201   29937.07086595  81377.39571257]
cosh(c) is [ 11013.2329201   29937.07086595  81377.39571257]
mean(c) is 11.0
std(c) is 0.816496580927726
median(c) is 11.0
myproduct(c) is 1320.0
numpy's product(c) is 1320.0
```

2.23 Saving numpy arrays to a file

```
In [61]: import numpy as np

        data = np.arange(15)

        #save to a file with no decoration
        np.savetxt("output_files/simple_file.txt",data)

        #save using commas to separate data
        np.savetxt("output_files/comma_delimited.txt",data,delimiter=',')

        #save with a header
        np.savetxt("output_files/with_header.txt",data,header='Example , Aug 16, 2016')

        #save with a footer
        np.savetxt("output_files/with_footer.txt",data,
                   footer='University of Toronto, PHY 224', comments='~')

        data2d = np.arange(15.).reshape(5,3)

        #save 2d
        np.savetxt("output_files/twod.txt",data2d,header="Aug 16, 2016",
                   footer='University of Toronto, PHY 224')
        print("The 2d saved in is \n{0}".format(data2d))
```

```
The 2d saved in is
[[  0.   1.   2.]
 [  3.   4.   5.]
 [  6.   7.   8.]
 [  9.  10.  11.]
 [ 12.  13.  14.]]
```

2.24 Loading numpy arrays from a file

```
In [62]: import numpy as np

         data_simple = np.loadtxt("output_files/simple_file.txt")
         data_comma = np.loadtxt("output_files/simple_file.txt",delimiter=',')

         #Python knows to skip comment lines,
         #marked with the comment maker that is usually a #
         data_header = np.loadtxt("output_files/with_header.txt")

         #but it doesn't know if you change the comment marker, so use the comments keyword
         data_footer = np.loadtxt("output_files/with_footer.txt",comments='~')
         data2d = np.loadtxt("output_files/twod.txt")

         print("The 2d data read in is \n{0}".format(data2d))
```

```
The 2d data read in is
[[  0.   1.   2.]
 [  3.   4.   5.]
 [  6.   7.   8.]
 [  9.  10.  11.]
 [ 12.  13.  14.]]
```

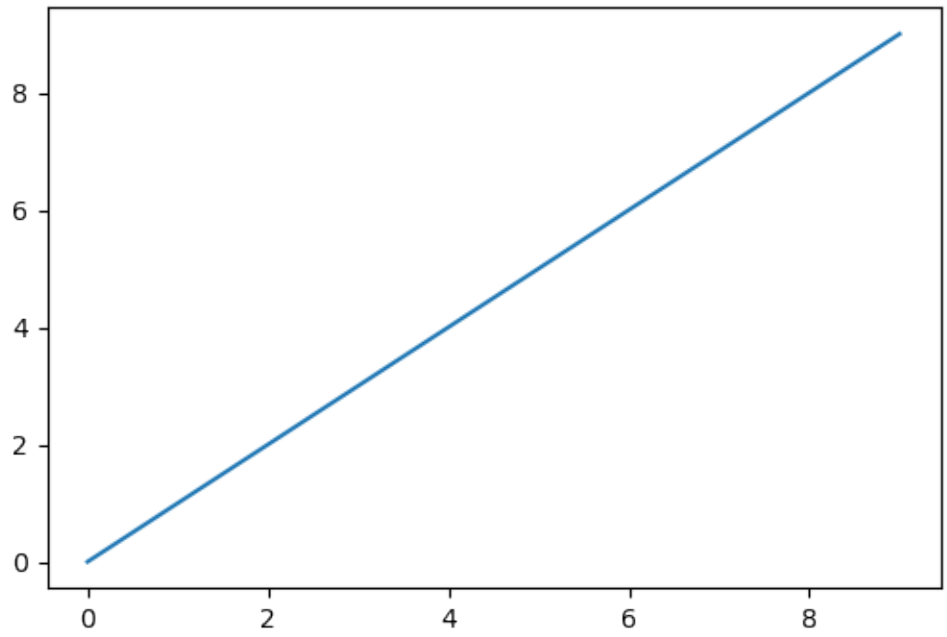
2.25 Loading numpy arrays in 2 dimensions

```
In [65]: import numpy as np
         #generate some data
         data2d = np.arange(50).reshape(10,5)
         #save the data
         np.savetxt("output_files/largedata.txt",data2d)

         #read all of the data
         indata= np.loadtxt("output_files/largedata.txt")
         #read one column only
         onedata = np.loadtxt("output_files/largedata.txt",usecols=(1,))
         #read column 1 and 3 only (that's the second and fourth columns!)
         onethreedata = np.loadtxt("output_files/largedata.txt",usecols=(1,3))

         #skip three rows, then read columns 2 and 4,
         #and unpack the data into two variables
         coltwo, colfour = np.loadtxt("output_files/largedata.txt",
                                      usecols=(2,4),skiprows=3,unpack=True)
         print("data2d shape is {0}".format(data2d.shape))
         print("indata shape is {0}".format(indata.shape))
         print("onedata shape is {0}".format(onedata.shape))
         print("onethreedata shape is {0}".format(onethreedata.shape))
         print("'coltwo's shape is {0} and has values {1}".format(coltwo.shape,coltwo))
         print("'colfour's shape is {0} and has values {1}".format(colfour.shape,colfour))
```

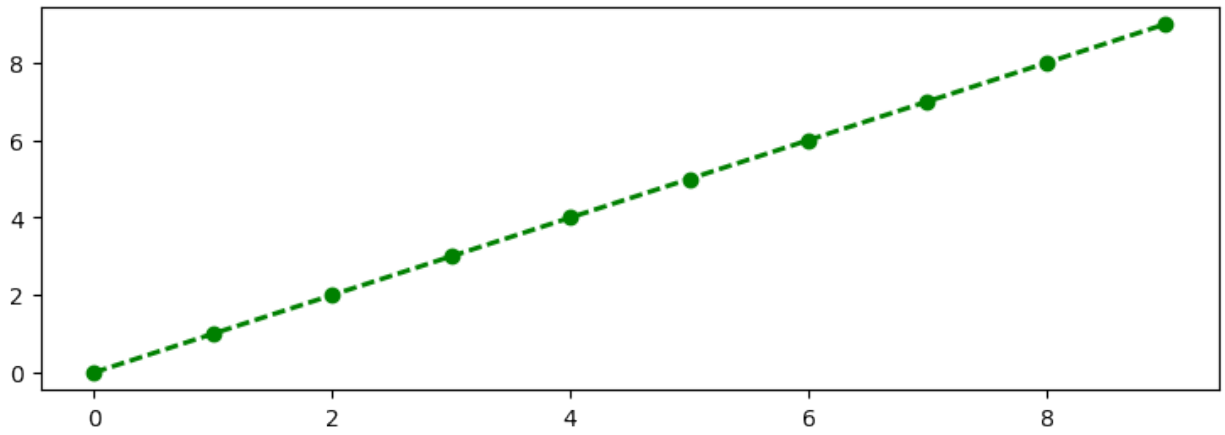
```
data2d shape is (10, 5)
indata shape is (10, 5)
onedata shape is (10,)
onethreedata shape is (10, 2)
'coltwo's shape is (7,) and has values [ 17.  22.  27.  32.  37.  42.  47.]
'colfour's shape is (7,) and has values [ 19.  24.  29.  34.  39.  44.  49.]
```

2.26 Importing Pylab and plotting data

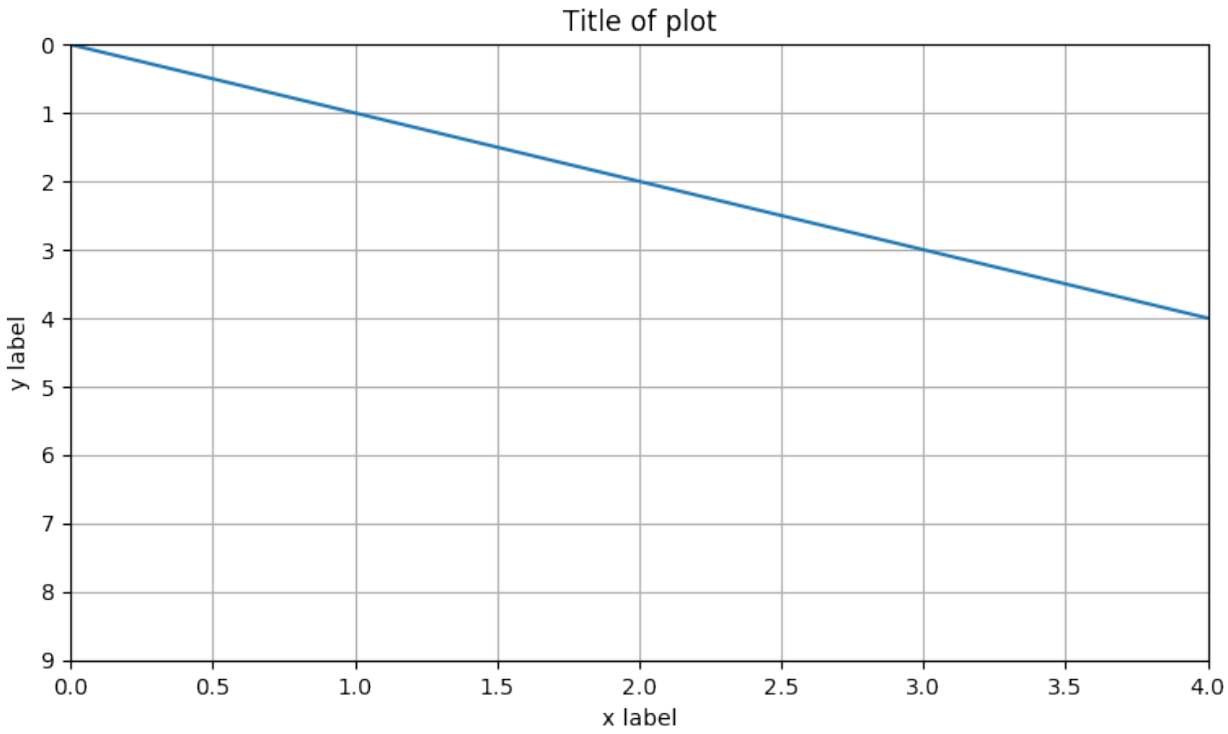
```
In [67]: import pylab as pl
         x=pl.arange(10)
         y=x

         #simple plot
         pl.plot(x,y);
```




```
In [68]: #Customizing the figure
         import pylab as pl
         x=pl.arange(10)
         y=x

         #change the figure size, plot, and save
         pl.figure(figsize=(9,3)) #9 inches wide, 3 inches tall
         pl.plot(x,y,color='green',linewidth=2,linestyle='--',marker='o')
         pl.savefig("output_files/figure1.pdf")
```



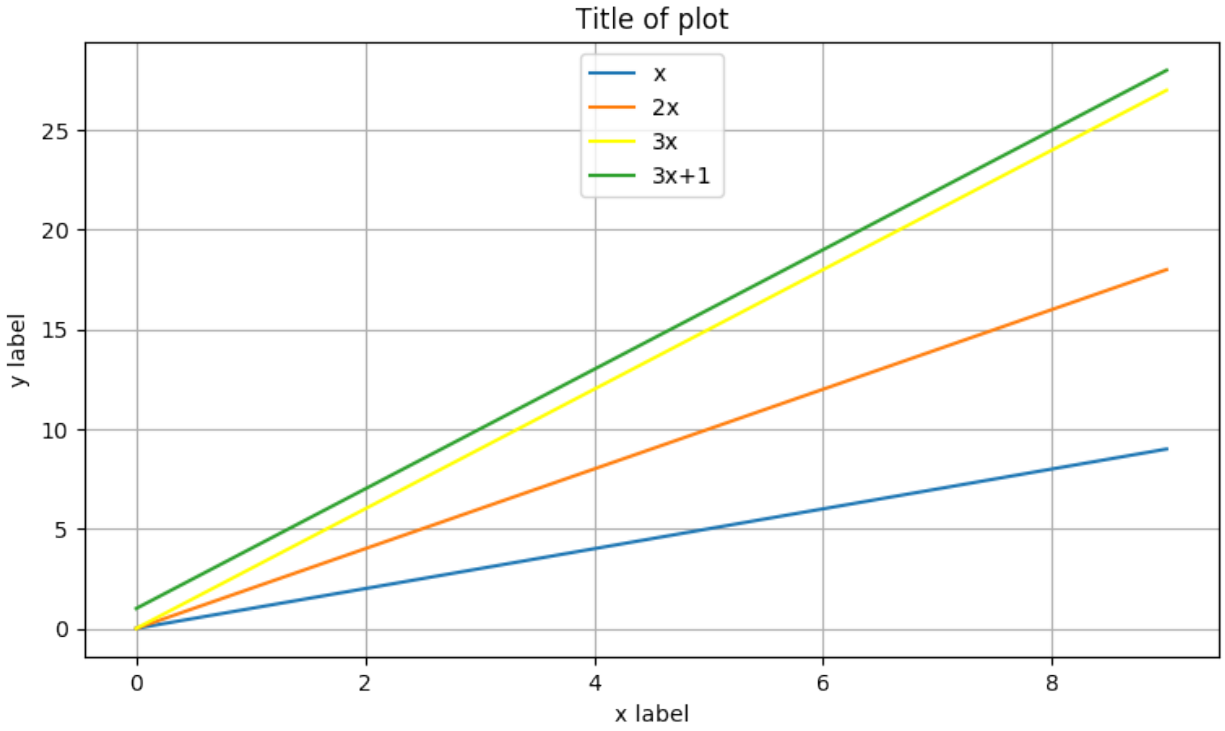
```
In [69]: import pylab as pl
         x=pl.arange(10)
         y=x

         #change the figure size, plot, and save
         pl.figure(figsize=(9,5)) #9 inches wide, 5 inches tall
         pl.plot(x,y)
         #label the plot
         pl.title("Title of plot")
         pl.xlabel("x label")
         pl.ylabel("y label")
         #constain axis limits
         pl.xlim(0,4) #left,right
         pl.ylim(9,0) #reversed the axis because "left" is larger than "right"

         #add a grid
         pl.grid()

         #save
         pl.savefig("output_files/figure2.pdf")
```



2.27 plotting multiple lines

```
In [70]: import pylab as pl
         x=pl.arange(10)
         y=x

         #change the figure size, plot, and save
         pl.figure(figsize=(9,5)) #9 inches wide, 5 inches tall
         pl.plot(x,x,label='x')
```

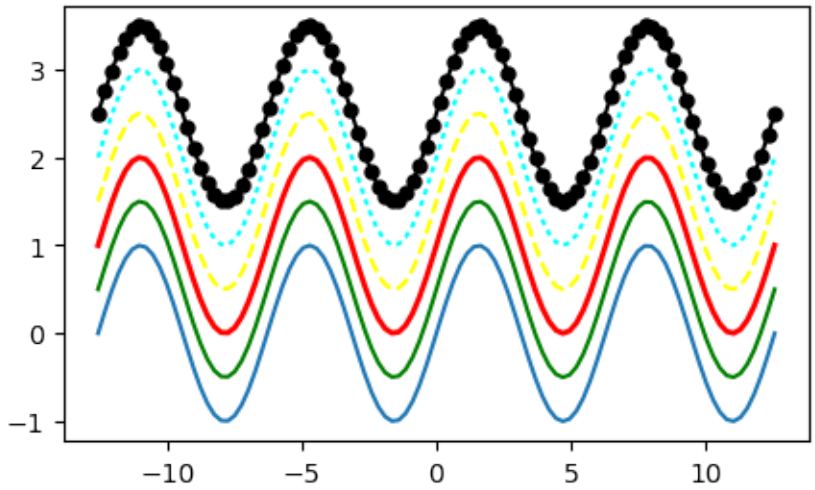
```
pl.plot(x,2*x,label='2x')
pl.plot(x,3*x,label='3x',color='yellow') #choose a color for this line
pl.plot(x,3*x+1,label='3x+1')
#label the plot
pl.title("Title of plot")
pl.xlabel("x label")
pl.ylabel("y label")

pl.legend(loc='upper center')
#'loc' is location, can be any combination of
# upper middle lower : in the vertical
# left center right : in the horizontal
#add a grid
pl.grid()

#save
pl.savefig("output_files/figure3.pdf")
```



2.28 Plotting with options

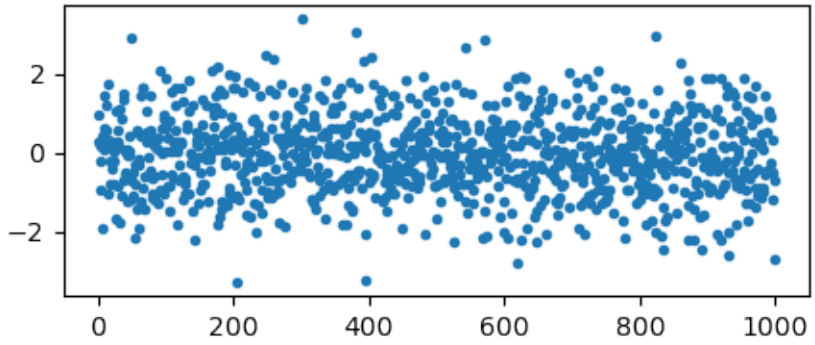
```
In [71]: import pylab as pl
         x=pl.linspace(-4*pl.pi,4*pl.pi,100)

         pl.figure(figsize=(5,3))
         pl.plot(x,pl.sin(x))
         pl.plot(x,0.5+pl.sin(x),color='green') #green
         pl.plot(x,1.0+pl.sin(x),color='red',linewidth=2) #thicker line
         pl.plot(x,1.5+pl.sin(x),color='yellow',linestyle='--') #dashed
         pl.plot(x,2.0+pl.sin(x),color='cyan',linestyle=':') #dotted
         pl.plot(x,2.5+pl.sin(x),color='black',marker='o',markersize=5)
```

```
        #solid line and o's with size 5
        pl.savefig("output_files/figure4.pdf")
```

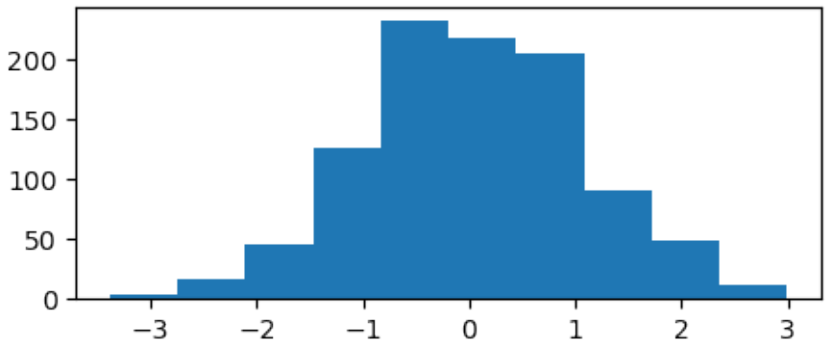


2.29 Plotting histograms

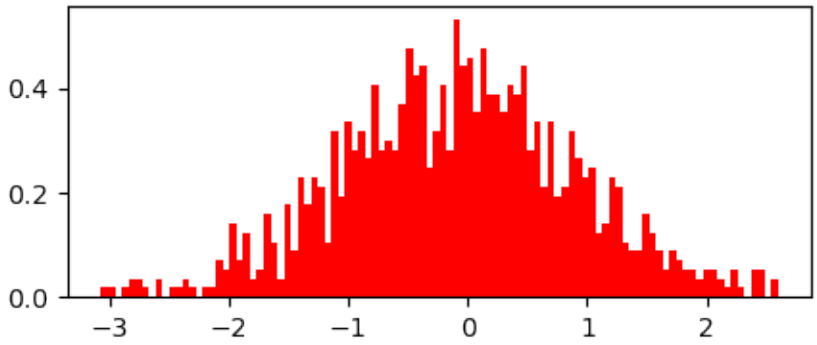
```
In [72]: import pylab as pl
         #generate some random data
         data = pl.randn(1000) #1,000 points distributed with Gauss' distribution

         pl.figure(figsize=(5,2))
         pl.plot(data,marker='.',linestyle='none')
         pl.savefig("output_files/figure5a.pdf")
```



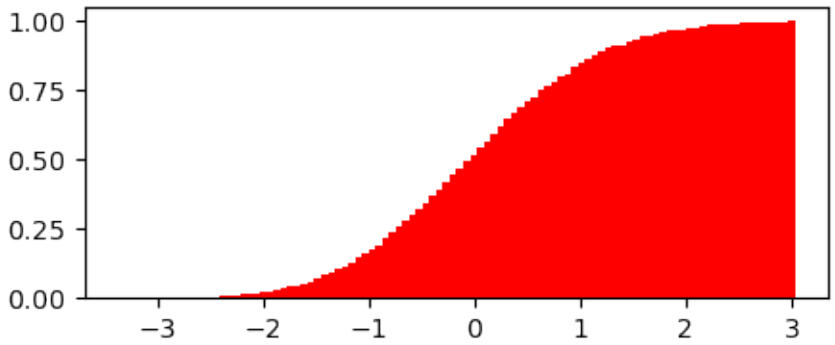

```
In [73]: data = pl.randn(1000) #1,000 points distributed with Gauss' distribution
         pl.figure(figsize=(5,2))
         #hist is used to plot the histogram. The default plot is normally okay
         pl.hist(data)
         #The default y axis is the number of elements in each bin.
         pl.savefig("output_files/figure5b.pdf")
```




```
In [74]: data = pl.randn(1000) #1,000 points distributed with Gauss' distribution
         pl.figure(figsize=(5,2))
         #hist is used to plot the histogram. It can be controlled with options
         pl.hist(data, bins=100,color='red',normed=True)
         #with normed=True, the integral of the values
         #in the histogram is 1.0 (it is a normalized probability distribution)

         pl.savefig("output_files/figure5c.pdf")
```

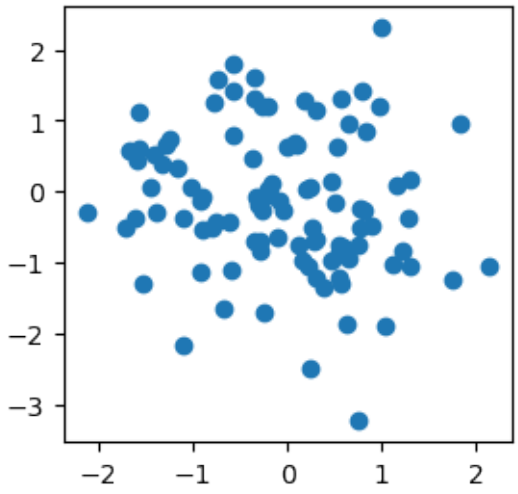



```
In [75]: data = pl.randn(1000) #1,000 points distributed with Gauss' distribution
         #hist can also be used to calculate the cumulative density function (CDF)
         pl.figure(figsize=(5,2))
         #hist is used to plot the histogram. It can be controlled with options
         pl.hist(data, bins=100,color='red',normed=True,cumulative=True)
         #with normed=True, the integral of the values
         #in the histogram is 1.0 (it is a normalized probability distribution)
         pl.savefig("output_files/figure5d.pdf")
```

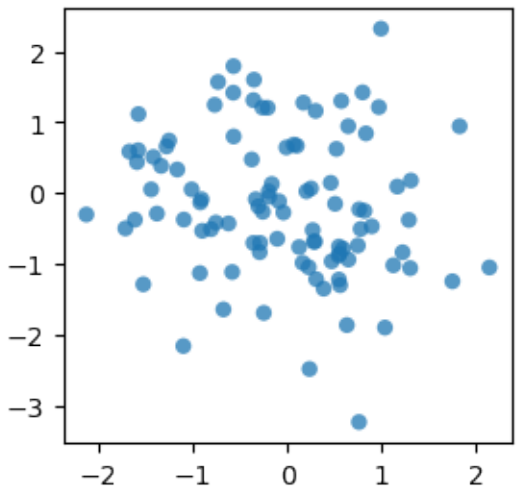


2.30 Scatter plots

```
In [76]: import pylab as pl
         #generate some random data
         xdata = pl.randn(100) #1,000 points distributed with Gauss' distribution
         ydata = pl.randn(100) #1,000 points distributed with Gauss' distribution
         cdata = pl.randn(100)

         pl.figure(figsize=(3,3))
         #The default scatter plot, blue circles with black edges.
         pl.scatter(xdata,ydata)
         pl.savefig("output_files/figure6a.pdf")
```

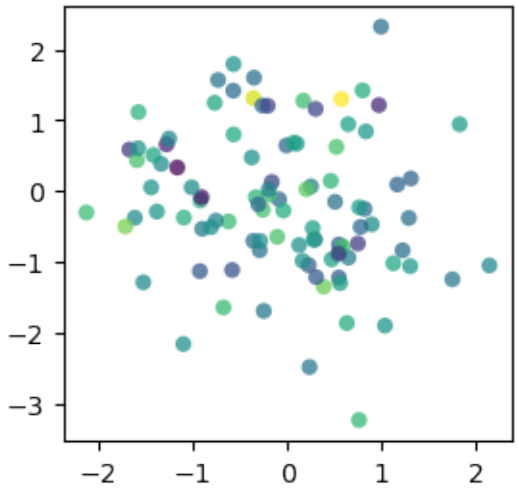



```
In [77]: pl.figure(figsize=(3,3))
         #remove the edge color to make it a little cleaner
         #make each point slightly transparent to reduce 'crowding'
         pl.scatter(xdata,ydata,edgecolor='none',alpha=0.75)
         pl.savefig("output_files/figure6b.pdf")
```

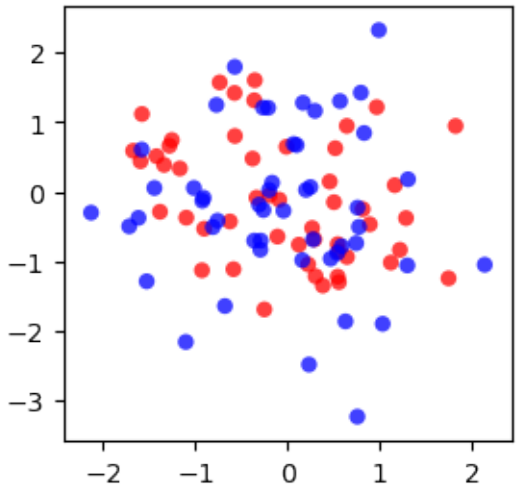



```
In [78]: pl.figure(figsize=(3,3))
         #Encode a third dimension in the color of each point
```

```
        pl.scatter(xdata,ydata,edgecolor='none',alpha=0.75, c=cdata)
        #c (*not* color) is the keyword for the point color
        pl.savefig("output_files/figure6c.pdf")
```

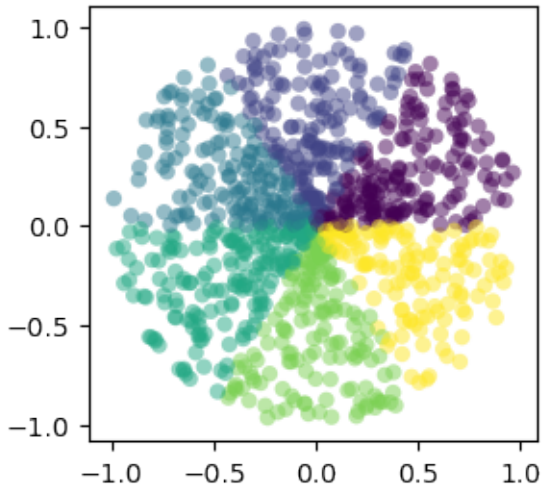


```
In [79]: pl.figure(figsize=(3,3))
         #plot half the data in red
         pl.scatter(xdata[:50],ydata[:50],edgecolor='none',alpha=0.75,c='red')
         #and half the data in blue
         pl.scatter(xdata[50:],ydata[50:],edgecolor='none',alpha=0.75,c='blue')
         #c (*not* color) is the keyword for the point color
         pl.savefig("output_files/figure6d.pdf")
```



```
In [84]: #one last figure
         import pylab as pl
         pl.figure(figsize=(3,3))
         r=pl.rand(1000)*1.
         t=pl.rand(1000)*2*pl.pi

         #convert the r and t parameters into a circle of data
         x=r*pl.cos(t)
         y=r*pl.sin(t)
         c=pl.floor(t/(pl.pi/3.)) # color it based on the t parameter in quarters
```

```
        pl.scatter(x,y,c=c,edgecolor='none',alpha=0.5)
        pl.savefig("output_files/figure6e.pdf")
```



2.31 Saving, Loading, and Plotting

```
In [85]: import pylab as pl

         #generate some data
         xdata=pl.arange(0,10,0.01)

         #the 'perfect' data is e^(-x/2)*sin(pi*x)
         yperfect=pl.exp(-xdata/2.)*pl.sin(pl.pi*xdata)

         #the 'noisy' data has Gaussian noise added to the signal
         ydata=yperfect+pl.randn(1000)/25.

         #save in a text file
         pl.savetxt('output_files/exdata.txt',(xdata, ydata, yperfect),
                    delimiter=',',header='Example data')

         #next up, loading and plotting.
```

```
In [86]: #clear the data to prove we're not cheating!
         xdata,ydata,yperfect = 0., 0,0 # clear data

         #load the data into new variables
         x,y,yp = pl.loadtxt('output_files/exdata.txt',delimiter=',')

         pl.figure(figsize=(5,4))

         #plot the noisy data first
         pl.plot(x,y,linestyle='none',marker='.',alpha=0.5,label='noisy data')
         #then plot the function
         pl.plot(x,yp,linewidth=2,label='idealized function')
         #label, grid, legend
         pl.title("Damped Oscillation")
         pl.xlabel("Time")
         pl.ylabel("Amplitude")
```

```
pl.legend(loc='upper right')
pl.legend
pl.savefig("output_files/figure7.pdf")
```