AST4007W Computational Methods

Luis A. Balona, Ed Elson, Masimba Paradza and Mayhew Steyn

May 27, 2020

Contents

Home

UCT NASSP AST4007W Computational Methods

Scientific Programming in Python

By Luis A. Balona, Ed Elson, Masimba Paradza and Mayhew Steyn

This online book is intended as reference material for the UCT NASSP Honours course (AST4007W) Computational Methods module. The intended use of this book is as a reference after the lecture pre-reading and slides unless directly referred to.

These notes are by no means the complete picture and you are encouraged to read further in the references provided.

If you find any errors (factual, grammatical or otherwise) feel free to contact Mayhew via email or create an issue on GitHub.

Please note that this book is not yet complete. Material will be added as the semester progresses and edits may be made to existing content.

Introduction

Programming and Python in a Nutshell

In this chapter we will go over a brief overview of programming languages and Python. While important, not having a full grasp of this chapter should not hinder your ability to program using Python.

What is a programming language?

Programming is, in essence, writing a series of instructions for the computer to execute. This is done using a programming language, which can be understood or translated to a form that can be understood by the computer.

At the lowest level the language of computers is called machine code. This language is used to communicate with the computer's CPU through binary or hexadecimal instructions. Machine code is dependent on the computer hardware being used and is not easy to understand as humans. A step up from this is an assembly language, which uses some human language, but is still difficult to understand and dependent on the computer architecture {% cite wikipedia_machine_code wikipedia_assembly %}.

Hardware dependence could be a big problem. Programs written for one computer would not necessarily work on another computer, they would have to be translated (by a human) first. To bridge this difference between hardware specifications high level programming languages were developed.

Most of the programming languages you are likely to use these days are high-level programming languages. Besides being CPU independent, these languages are designed to be readable by humans. At some level these languages will need to be compiled (translated {% cite wikipedia_compiler %}) to machine code {% cite wikipedia_machine_code %}.

What is a Script?

A script is a text file containing source code (instructions for the computer) written in a programming language. Programs can be composed of a single script, or many scripts working together.

Generally scripts for a particular language are given a specific file suffix. Relevant to us, Python scripts end with a `.py`.

Python, a Dynamic Programming Language.

Python is a high level language and thus needs to eventually be compiled down.

Many high level programming languages' source code is compiled to machine code once, and then can be executed in this form. These are called static programming languages (C is an example).

Dynamic programming languages are languages where operations that would normally take place at compile-time (when the code is compiled) can instead be done at run-time (when the program is executed) {% cite mozilla_dynamic %}. Python is a dynamic programming language.

When you run a python script it is compiled to byte code (if it hasn't been already). This byte code is a lower-level, platform independent representation of your source code . {% cite ned_python_interp net-info_compiling_python %}.

Byte code is similar to the CPU specific assembly code, but is instead executed by software called a virtual machine (which simulates a CPU environment). {% cite ned_python_interp %} The Python Virtual Machine is always present in the Python system and is the last step of the Python Interpreter {% cite net-info_compiling_python %}.

References

{% bibliography –cited %}

Python Basics

Python Basics

Python Basics

In this chapter we shall discuss some of the basics of programming in Python, namely variables, operations and using functions.

Variables

Variables

Python receives information by means of variables. A variable is a dedicated piece of computer memory that holds some information. For example,

[2]:
```
a = 5
```

tells python to assign 5 (an integer number) to the variable with the variable name `a`. Note that the `=` here is used for variable assignment, it does not have the same meaning as the mathematical symbol ("assign-variable-to" rather than "is-equal-to").

If we wanted to access the value that our variable `a` holds, we can refer to it by it's name. For example, if we want to print the value to terminal:

[3]:
```
print(a)
```

```
5
```

Data Types

The information stored in memory needs to be interprated if it's to be of any use to us. To achieve this Python (and many other programming languages) uses variable types.

In the example above we used an integer or `int` type. In order to check what type a variable has, we can use the `type()` function:

[4]:
```
print(type(a))
```

```
<class 'int'>
```

The other basic variable types we will be working with are floating point numbers and strings.

Floating point numbers (or `float`) are numbers with decimal parts, for example:

[6]:
```
print(type(5.2))
```

```
<class 'float'>
```

Strings (or `str`), are a collection of unicode characters (letters, numbers, symbols, ect). Basically, the contents of any text file can be seen as a string. Strings are represented using parenthesis:

[7]:
```
print(type('This is a string.'))
```

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

You are not limited to single quotes. For single line strings you can use double quotes as well:

[8]:
```
print('String using single quotes, " does not break the string.')
print("String using double quotes, ' doesn't break the string.")
```

[8]:
```
"String using double qotes, ' doesn't break the string"
```

For strings containing line breaks, you can use `'''` or `"""`:

[2]:
```
print(
'''String with a
line break'''
)

print(
"""Another string with a
line break"""
)
```

```
String with a
line break
Another string with a
line break
```

You could also insert line breaks using a the special character `'\n`

Variable Names

So far we have been using single letters (`a`, `b`, `c`, `d`, …) as variable names, but this approach can be confusing for long segments of codes. Variable names should be as clear and descriptive as possible (describing what they are used for), while still being short enough to type out efficiently.

To this end we should delve into some of the restrictions on the character sequences that make up variable names: * The characters must all be letters, digits, or underscores (_) , and must start with a letter. In particular, punctuation and blanks are not allowed. * There are some words that are reserved for special use in Python. You may not use these words as your own identifiers. This is the full list:

False

class

finally

is

return

None

continue

for

lambda

try

True

def

from

nonlocal

while

and

del

global

not

with

as

if

elif

or

yield

assert

else

import

pass

break

except

in

raise

- Python is case sensitive: The variable names `last`, `LAST`, and `LaSt` are all different.

Now, you may want to use a variable that is more than one word long, for example `price at opening`, but blanks are illegal! One poor option is just leaving out the blanks, like `priceatopening`. Then it may be hard to figure out where words split. Two practical options are: * Underscore separated: putting underscores (which are legal) in place of the blanks, like `price_at_opening`. * Using camel-case: omitting spaces and using all lowercase, except capitalizing all words after the first, like `priceAtOpening`. * Using Pascal-case: similar to camel-case but capitalising the first word, `PriceAtOpening`.

The standard in Python is to use underscore seperations for variable and function names.

Assigning Variables to other Variables

You can assign the value of one variable to another:

[2]:
```
var1 = 3
var2 = var1

print('Variable 1 is', var1)
print('Variable 2 is', var2)
```

```
Variable 1 is 3
Variable 2 is 3
```

When you assign assign a variable using another variable, in most cases it is only the value of the variable that is assigned:

[3]:
```
var1 = 3
var2 = var1

print('Variable 1 is', var1)
print('Variable 2 is', var2)

var1 = 2

print('')
print('Variable 1 is', var1)
print('Variable 2 is', var2)
```

```
Variable 1 is 3
Variable 2 is 3

Variable 1 is 2
Variable 2 is 3
```

Notice how, even though we change the value of `var1`, the value of `var2` remains the same.

Comments

Comments

Comments make it possible to write messages in our scripts that are not to be read by the computer, but fellow humans.

In Python you can write an in-line comment by using the # symbol. Everything after this symbol until the end of the line will be considered a part of the comment and the computer will not read this as code. For example:

[1]:
```
print('Not a comment') # This is a comment print('Part of the comment')
```

```
Not a comment
```

Comments can be useful for explaining what a script/section of a script does or why you've made the choices you have made in a particular line. It is not normally necessary to explain what each line of code does, as it should be easy enough to read the actual code to determine this.

Commenting on a Line of Code

If you want to comment about a particular line of code it is common practice to put the comment at the end of that line of code:

[3]:
```
print('some code') #comment on code
```

```
some code
```

If the comment is too long to fit on the line, you can write the comment on a separate line above the code:

[4]:
```
#Comment line that is too long to fit on the end of the line of code
print('some code')
```

```
some code
```

Commenting Out Portions of Code

Sometimes you may want to comment out code to temporarily remove it from the program without deleting it. It is especially useful when you want to isolate code snippets during debugging or print statements used in debugging during normal runtime. For example:

[1]:
```
var1 = 3
var2 = var1

#print('Variable 1 is', var1)
#print('Variable 2 is', var2)

var1 = 2

#print('')
print('Variable 1 is', var1)
print('Variable 2 is', var2)
```

```
Variable 1 is 2
Variable 2 is 3
```

Type Conversion

Type Conversion

So far we have looked at three variable types: integers, floats and strings; and how to check what type a variable is.

Sometimes we want to convert between different variable types. To do this we can use the `int`, `float` and `str` functions:

[3]:
```
int_var = 1
print(int_var, type(int_var))

float_var = float(int_var)
print(float_var, type(float_var))
```

```
1 <class 'int'>
1.0 <class 'float'>
```

[4]:
```
float_var = 5.7
print(float_var, type(float_var))

int_var = int(float_var)
print(int_var, type(int_var))
```

```
5.7 <class 'float'>
5 <class 'int'>
```

Note that when you convert a float to an integer Python does simply discards the decimal part (if you wish to round-off a float you can use the `round` function).

[6]:
```
str_var = '1.43'
print(str_var, type(str_var))

float_var = float(str_var)
print(float_var, type(float_var))
```

```
1.43 <class 'str'>
1.43 <class 'float'>
```

[8]:
```
str_var = '12'
print(str_var, type(str_var))

int_var = int(str_var)
print(int_var, type(int_var))
```

```
12 <class 'str'>
12 <class 'int'>
```

Note that anything other than a number cannot be converted from a string to a float or int:

[9]:
```
str_var = 'not a number'
print(str_var, type(str_var))

float_var = float(str_var)
print(float_var, type(float_var))
```

```
not a number <class 'str'>


      ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-9-4224f055b9d6> in <module>
         2 print(str_var, type(str_var))
         3
   ----> 4 float_var = float(str_var)
         5 print(float_var, type(float_var))


       ValueError: could not convert string to float: 'not a number'
```

Even strings that contain a number with a decimal part cannot be converted to an integer:

[10]:
```
str_var = '4.563'
print(str_var, type(str_var))

int_var = int(str_var)
print(int_var, type(int_var))
```

```
4.563 <class 'str'>


      ␣
↪---------------------------------------------------------------------------
```

```
       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-10-a3094efdeb44> in <module>
         2 print(str_var, type(str_var))
         3
   ----> 4 int_var = int(str_var)
         5 print(int_var, type(int_var))


       ValueError: invalid literal for int() with base 10: '4.563'
```

Operators

Numerical Operators

Coding, in the simplest sense, is merely the action of assigning values to variables, and then 'doing things' with those variables.

In this section we will look at some of the basic operators used for integers and floats. Other variable types have different operators, which we shall see later.

An obvious starting point is basic arithmetic; addition, subtraction, multiplication and division:

[2]:
```
a = 2.0
b = 3.0
c = 10.0

print('a added to b is', a + b)
print('a multiplied by c is', a*c)
print('c divided by b is', c/b)
print('a subtracted from c', c - a)
```

```
a added to b is 5.0
a multiplied by c is 20.0
c divided by b is 3.3333333333333335
a subtracted from c 8.0
```

These operators can also be used for integers:

[3]:
```
print('2 multiplied by 3 is', 2*3)
```

```
2 multiplied by 3 is 6
```

and between integers and floats:

[4]:
```
print('1.5 multiplied by 2 is', 1.5*2)
```

```
1.5 multiplied by 2 is 3.0
```

If you are using Python version 2.xxx or below, beware of dividing by integers…

The Exponential Operator **

Another useful operator is the exponential operator **. This returns the left number to the power of the right:

[6]:
```
print(2**3)
```

```
8
```

which can be read as 2^3.

Note that this operator also works on floats and float-integer combinations:

[8]:
```
print(4**0.5) #square root of 4
```

```
2.0
```

The Modulo Operator %

The modulo operator returns the remainder of the left number divided by the right:

[9]:
```
print(16%3)
```

```
1
```

In mathematics this would be expressed as

16 mod 3.

This operator can also act on floats and integer-float combinations:

[10]:
```
print(16.3%3)
```

```
1.3000000000000007
```

The Floor Division Operator //

This returns the result of the left number divided by the right, but without the remainder:

[11]:
```
print(16//3)
```

```
5
```

Like the others this works for both integers and floats.

Special Functions and Advanced Mathematics

For more complex mathematics involving logs, trigonometry, etc. we'll rely on the scientific packages SciPy and NumPy. We'll discuss these at a later stage.

Multiple Operations in a Single Expression

Though we have only seen one operation or function used per line, you can combine as many as you'd like:

[14]:
```
print( 2**3 + 4)
```

```
12
```

If you want to group or control the order in which operations are executed use brackets. For example:

[15]:
```
print( (2 + 17//2)**5 )
```

```
100000
```

where the `//` is applied first, then the `+` and lastly the `**`. We shall discuss the order in which operations and function calls are executed later.

Assignment Operators

Compound Assignment Operators

We've already discussed the `=` operator which sets the value of a variable. There are a few more assignment operators which are mostly used for convenience. These are the compound operators : `+=`, `-=`, `*=`, `/=`.

These operators apply their respective operation between the variable being assigned to and the value on the right and assign that value to the variable. In other words:

```
var += 2
```

can be read as

```
var = var + 2
```

and

```
var /= 2
```

can be read as

```
var = var / 2
```

etc..

Introduction to Functions

Introduction to Using Functions

In this section we shall discuss some of the details on how to use functions from the Standard Library. We have already come across a few functions, namely `print()` and `type()`. We shall cover how to define your own functions in a later section.

Python functions essentially take variables or values/objects as arguments, perform a task and them return values/objects. As we have seen before the syntax of a function call is:

```
function_name(argument, argument, ...)
```

Note that some functions return a `None` type when a return value isn't necessary. For example, the `print()` function:

[4]:
```
print_return = print('print out')

print('print function return:', print_return)
```

```
print out
print function return: None
```

If you want to know what a particular function does or how to use it, a quick way to find out is to pull up the docstring. In an IPython environment (such as a Jupyter notebook) this can be done by typing a `?` symbol after the function name and pressing enter. For example:

[5]:
```
print?
```

```
[0;31mDocstring:[0m
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
[0;31mType:[0m      builtin_function_or_method
```

In a default Python shell or script you can print out the docstring using the `help()` function. For example:

[8]:
```
help(print)
```

```
Help on built-in function print in module builtins:

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

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

Alternatively you can refer to the Python documentation.

Now, let's take a look at the `print()` docstring itself. The first line of the docstring shows us the function name; and the name of the function arguments and their default values (if they have). Following this is a description of what the function does. Following this is a list of optional keyword arguments.

Sometimes a docstring will also contain examples on how to use the function, though none are present in this one.

We will see more examples of how optional keyword arguments work (in particular in the Chapter discussing Matplotlib…), for now let's use one of them. Let's change the argument `sep`, which is the string inserted between the values you put into the `print()` function. As we can see, it's default value is a space `' '`. As a first example, let's change the separation to an empty string (no space):

[1]:
```
print('There', 'are', 'no', 'spaces', sep = '')
```

```
Therearenospaces
```

As another example, let's put commas (`','`) in between the values:

[2]:
```
print('There', 'are', 'commas', 'between', 'values', sep = ',')
```

```
There,are,commas,between,values
```

The `input()` Function

Another important function in the Python Standard Library is the `input()` function. This function allows us to collect user inputs from the terminal.

`input()` takes a string as an argument, this gets printed to the terminal and the script is halted until the user has entered a string and pressed enter. This string that the user has entered is returned by the `input()` function; and can, therefore, be used in the rest of the script.

As a first example, consider the following code that asks the user for their name:

[5]:
```
user_name = input('What is your name?')

print('Hello', user_name, ', nice to meet you!')
```

```
What is your name? Mayhew

Hello Mayhew , nice to meet you!
```

Remember that the program will wait for your input before it continues. If you are using a Jupyter Notebook, this means that other cells from the notebook will not run until the code has been fully executed or terminated.

Now, something that is important to note is that the return value from input is a string, even when a number is typed into the terminal:

[7]:
```
user_number = input('Enter a number: ')

print(user_number, type(user_number))
```

```
Enter a number:  12

12 <class 'str'>
```

If you intend for the input to be used as a number, you must remember to convert it to one (an `int` or `float`):

[9]:
```
user_int = int(input('Enter an integer: '))

print('Entered:', user_int, type(user_int))

user_float = float(input('Enter a number: '))

print('Entered:', user_float, type(user_float))
```

```
Enter an integer:  6

Entered: 6 <class 'int'>

Enter a number:  57.5

Entered: 57.5 <class 'float'>
```

Strings

Strings

In this section we shall take a closer look at the string type and some of the operations associated with them. The following section makes heavy reference to online notes by Dr. Andrew N. Harrington, Hands-on Python 3 Tutorial released under the CC BY-NC-SA 4.0 license.

Concatenation +

For strings the + symbol is used to concatenate two strings together. For example:

[3]:
```
print('One string' + ' and another')
```

```
One string and another
```

Duplication *

The duplication * operator takes a string and an integer and repeats the string as many times as the integer value:

[6]:
```
print('hello '*4)
print(2*'bye ')
```

```
hello hello hello hello
bye bye
```

Indexing []

Strings can be seen as a collection of characters. Each of these character has an integer index associated with it, based on it's position in the string. For example, take the string `'computer'`:

character

c

o

m

p

u

t

e

r

index

0

1

2

3

4

5

6

7

You can access individual characters in the string by index using:

`string[index]`

for example:

[1]:
```
computer_string = 'computer'

print('Index 3:', computer_string[3])

print('Index 7:', computer_string[7])
```

```
Index 3: p
Index 7: r
```

If you use an index that is too large for the given string, Python will return an error:

[8]:
```
print('Index 11', computer_string[11])
```

```
␣
↪---------------------------------------------------------------------------

IndexError                                Traceback (most recent call␣
↪last)

<ipython-input-8-abeba3add71f> in <module>()
----> 1 print('Index 11', computer_string[11])


IndexError: string index out of range
```

You can find the number of characters in a string using the `len()` function:

[9]:
```
print('There are', len(computer_string), 'characters in the string')
```

```
There are 8 characters in the string
```

Notice how the length of `computer_string` is one greater than its largest index. This is because Python indexes from `0`.

Thus, if we don't know how long a string is before hand (if a variable holding a string is subject to change for instance) and we want to index the last value of the string, we could use `len() - 1` as the index:

[11]:
```
print('The last character:', computer_string[len(computer_string) - 1])
```

```
The last character: r
```

This method works, but Python gives us a far cleaner way of doing this: using an index of `-1`. This won't work for most other programming languages.

[12]:
```
print('The last character:', computer_string[-1])
```

```
The last character: r
```

In general, negative indices in Python index the strings (and other objects) backwards:

[13]:
```
print('Second last character', computer_string[-2])

print('Third last character', computer_string[-3])
```

```
Second last character e
Third last character t
```

Note that the index `-8` corresponds to the `0` index (`len(computer_string) - 8` is `0`) so anything less than this would be out of bounds.

Slicing

Slicing allows us to extract segments of the string, as apposed to individual characters. The syntax for string slicing is:

`string[start_index:stop_index]`

where the `stop_index` is not included in the slice, rather the slice stops before this index. For example, consider the slice:

[14]:
```
print(computer_string[2:5])
```

```
mpu
```

where the last character is `'u'`, but the character with index `5` is `'t'`.

If we want to take a slice from the beginning of a string we could use `0` as the `start_index`:

[21]:
```
print(computer_string[0:3])
```

```
com
```

Alternatively if we left the `start_index` blank Python will interprate this as starting from the beginning of the string:

[22]:
```
print(computer_string[:3])
```

```
com
```

Similarly if we wanted to take a slice up to and including the last character in the string, we can use:

[25]:
```
print(computer_string[3:len(computer_string)])
```

```
puter
```

or simply leave the `stop_index` blank:

[24]:
```
print(computer_string[3:])
```

```
puter
```

Notice the slice above is not the same as if we used `-1` as the `stop_index`:

[27]:
```
print(computer_string[3:-1])
```

```
pute
```

even though the same rules apply as with indexing, the slice always stops **before** the `stop_index`.

We can use a third index when slicing as a step size:

`string[start_index: stop_index: step_size]`

For example, we can get every second character from a string using a step size of 2:

[20]:
```
print('Starting from 0:', computer_string[0:8:2])
print('Starting from 1:', computer_string[1:8:2])
```

```
Starting from 0: cmue
Starting from 1: optr
```

The step size can be any integer. Note that by default it is set to 1. As another example lets print out every second character from `computer_string` starting from the first:

[4]:
```
print(computer_string[::3])
```

```
cpe
```

The step size need not be positive. If a negative step size is used the string will be sliced backwards. For example if we want to print out the whole of `computer_string` backwards:

[6]:
```
print(computer_string[::-1])
```

```
retupmoc
```

Note, when slicing with a negative step size you must ensure that `start_index` is greater than `stop_index`, otherwise your slice will be empty.

[9]:
```
print('Empty slice:', computer_string[0:6:-1])
print('Not empty slice:', computer_string[6:0:-1])
```

```
Empty slice:
Not empty slice: etupmo
```

Also notice how, in the second slice above, the `0` index character is not present. Even when slicing with a negative step size the `stop_index` is **not** included in the slice.

String Formatting

String Formatting

Concatenating strings can sometimes be cumbersome and hard to automate. If you need to include variables and/or values in your string, you may be better off using string formatting. We will use this technique more extensively later on.

There are a few ways to format strings. We will cover one of the ways introduced in Python 3. That is using the `string.format()` method.

This method treats everything contained in curly braces`{}` in the string as a replacement field, everything in and including the braces are replaced with the arguments of format in the output string.

[1]:
```
print('Hello {}, how are you?'.format('world'))
```

```
Hello world, how are you?
```

As you can see above, the blank curly braces were replaced with the string argument `'world'`.

Note that the method does not change the string itself but returns a new string.

You can make multiple replacements at a time if you have a string with multiple replacement fields:

[2]:
```
print('{}, {}, {}'.format(1, 2, 3))
```

```
1, 2, 3
```

Sometimes you will want more control over how the arguments of format are placed into the string. There is a specific syntax for formatting which you can read in the documentation. We will cover a few examples.

Specify Arguments by Position

If you want to specify the order in which the arguments of format are placed into the string, you can put numbers in the replacement fields to reference the positional arguments:

[2]:
```
print('{0}, {2}, {1}'.format(1, 2, 3))
```

```
1, 3, 2
```

Note that this also allows you to repeat elements:

[3]:
```
print('{0}, {2}, {1}, {2}'.format(1, 2, 3))
```

```
1, 3, 2, 3
```

Specify Arguments by Name

You can also specify arguments by name, the arguments must then be presented as keyword arguments:

[35]:
```
print('You can find the point at position ({x}, {y}).'.format(x = 2, y = 6))␣
↪#Arguments with names 'x' and 'y'
```

```
You can find the point at position (2, 6).
```

Specifying Numerical Types and Precision

To put it simply, when formatting numerical arguments the format specifier (to be placed in the replacement field) is of the structure: `[argument_reference]:[width][.precision][type]`

Where - `argument_reference` is the position of or name of the argument. - `width` specifies the minimum width that a replacement will take (look to the docs for alignment options) - For floats `precision` can be seen as the number of decimal places. - `type` specifies what type you want to display the number as. Multiple types exist for both integers and floats, but the most commonly used types are `d` for decimal integer and `f` for fixed point number (which you can use for floats)

Each of these parts of the format specifier are optional.

As a first example, lets display an integer:

[27]:
```
print('{:d}'.format(5))
```

```
5
```

Now, lets see how the width affects the output:

[28]:
```
print('{:d}'.format(5)) #minimum width of 0
print('{:1d}'.format(5)) #minimum width of 1
print('{:2d}'.format(5)) #minimum width of 2
print('{:3d}'.format(5)) #minimum width of 3
```

```
5
5
 5
  5
```

As you can see the first 2 outputs are the same. That is because the output is of length 1.

If you want to display a float to 2 decimal places, specify precision:

[29]:
```
print('{:.2f}'.format(1.232435455))
```

```
1.23
```

If you want to specify the position of the argument, include a reference to the argument position:

[32]:
```
print('{1:.3f}'.format(1.232435455, 5.35362)) #argument position of 1
```

```
5.354
```

Alternative Syntax

Instead of using the `.format()` method on a string, you could alternatively use an f-string for formatting (f prefixed before a string literal).

[2]:
```
subject = 'World'
time = 'today'

f'Hello {subject}! How are you doing {time}?'
```

[2]:
```
'Hello World! How are you doing today?'
```

This simplifies things substantially, but has less range of applicability than `.format()`.

Data Structures

Data Structures

Data Structures

In this chapter we will present a brief overview of Python's standard data structures, namely tuples, lists and dictionaries. For a more broad overview you can refer to the documentation.

Tuple

Tuple

Just as strings are a sequence of characters, tuples are a sequence of objects. This makes their use far more general.

You can set a tuple by separating the objects using commas. For example:

[1]:
```
t = 1, 2, 3, 'a', 'b', 'c'

print(t)
```

```
(1, 2, 3, 'a', 'b', 'c')
```

This is called tuple packing.

You can also put brackets around the objects, which is useful if you need to instance a tuple and use it in the same line (for example as a function argument):

[2]:
```
print(('a', 1, 'b', 2, 'c', 3))
```

```
('a', 1, 'b', 2, 'c', 3)
```

Like strings, tuples can be indexed and sliced:

[3]:
```
print('Index 3:', t[3])
print('Slice from index 3:', t[3:])
```

```
Index 3: a
Slice from index 3: ('a', 'b', 'c')
```

Tuples are also immutable (like strings):

[5]:
```
t[2] = 5
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)
```

```
    <ipython-input-5-5255d5d095a8> in <module>
----> 1 t[2] = 5


TypeError: 'tuple' object does not support item assignment
```

You can unpack a tuple into multiple variables, just like you can pack multiple values into a tuple:

[7]:
```
t = (1, 2, 3)
print('t is ', t)

x, y, z = t
print('x is', x)
print('y is', y)
print('z is', z)
```

```
t is  (1, 2, 3)
x is 1
y is 2
z is 3
```

Lists

Lists

Lists are used to store a collection of objects but are more flexible than tuples. You can create lists using the `list` function with another iterable object or square brackets `[]`:

[2]:
```
list1 = list((1, 2, 3))
print('list1', list1)

list2 = [4, 8, 9]
print('list2', list2)
```

```
list1 [1, 2, 3]
list2 [4, 8, 9]
```

You can access elements of the list by indexing and slicing it:

[6]:
```
letters = ['a', 'b', 'c', 'd', 'e']
print('Letters:', letters)
print('First character:', letters[0])
print('Second character:', letters[1])
print('Last character:', letters[-1])
print('Every second character:', letters[::2])
```

```
Letters: ['a', 'b', 'c', 'd', 'e']
First character: a
Second character: b
Last character: e
Every second character: ['a', 'c', 'e']
```

Unlike tuples you can alter the elements of a list after instancing it:

[5]:
```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters)

print('Changing the third character')

letters[2] = 'z'
print(letters)
```

```
['a', 'b', 'c', 'd', 'e']
Changing the third character
['a', 'b', 'z', 'd', 'e']
```

You can also assign new values to slices:

[7]:
```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters)

print('Changing the first three characters')
letters[:3] = ['x', 'y', 'z']
print(letters)
```

```
['a', 'b', 'c', 'd', 'e']
Changing the first three characters
['x', 'y', 'z', 'd', 'e']
```

Concatenating Lists

The + operator acts on lists in a similar way to strings, concatenating the two lists:

[8]:
```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

print(list1 + list2)
```

```
[1, 2, 3, 'a', 'b', 'c']
```

`list.append()`

You can add elements to the end of the list using the `.append()` method:

[15]:
```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters)

print('Appending an additional letter')

letters.append('f')
print(letters)
```

```
['a', 'b', 'c', 'd', 'e']
Appending an additional letter
['a', 'b', 'c', 'd', 'e', 'f']
```

`list.insert()`

If you want to insert an element into a specific place in the list you can use the `.insert()` method. This takes the index and the object you want to add as the arguments:

[16]:
```
numbers = [1, 2, 4, 5, 6]
print(numbers)

print('Inserting number 3 at index 2')

numbers.insert(2, 3)
print(numbers)
```

```
[1, 2, 4, 5, 6]
Inserting number 3 at index 2
[1, 2, 3, 4, 5, 6]
```

lists.remove()

If you want to remove the first instance of an element of a list with a specific value you can use the `.remove()` method:

[23]:
```
numbers = [1, 2, 1, 3, 4]
print(numbers)

print('Removing first 1 from numbers')

numbers.remove(1)
print(numbers)
```

```
[1, 2, 1, 3, 4]
Removing first 1 from numbers
[2, 1, 3, 4]
```

list.pop()

If you want to retrieve and remove an element at a particular index you can use the `.remove()` method, which takes the index of the element you want to retrieve as the argument:

[22]:
```
numbers = [1, 2, 3, 4, 5]
print(numbers)

print('Retrieving number at index 2:', numbers.pop(2))

print(numbers)
```

```
[1, 2, 3, 4, 5]
Retrieving number at index 2: 3
[1, 2, 4, 5]
```

Dictionaries

Dictionaries

So far we have only looked at sequence data structures, where elements are referred to by their position in the sequence. In dictionaries, however, the objects stored are referred to by a key. Keys must be an immutable type, for example a string, number or tuple containing only immutable types.

You can create a dictionary using the `dict` function; and assign values using the subscript notation:

```
dictionary[key] = value
```

[2]:
```
d = dict()

d[1] = 'a'
d['lst'] = [1, 2, 3]

print(d)
```

```
{1: 'a', 'lst': [1, 2, 3]}
```

You can also access dictionary values using the subscript notation:

[4]:
```
print(d[1])
```

```
a
```

An alternative way to initialize a dictionary with key-value pairs is:

```
{key1 : value1, key2 : value2}
```

much like it appears in the print output:

[6]:
```
d = {1 : 'a',  'lst' : [1, 2, 3]}

print(d)
```

```
{1: 'a', 'lst': [1, 2, 3]}
```

Note that using a key that doesn't exist in the dictionary will give you a `KeyError`:

[5]:
```
print(d[2])
```

```
␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call␣
↪last)

<ipython-input-5-c8f93a31d4a2> in <module>
----> 1 print(d[2])


KeyError: 2
```

Listing the Keys Which Exist

Often you will want a list of the keys which a dictionary has. For this you can use the `dict.keys()` method:

[7]:
```
print(d.keys())
```

```
dict_keys([1, 'lst'])
```

One use for this is to check if a dictionary has the key you're looking for if you want to avoid an error:

[8]:
```
if 2 in d.keys():
    print(d[2])
else:
    print(2, 'not a key in d')
```

```
2 not a key in d
```

If Statements

If Statements

Control Flow: If Statements

Control flow dictates the order in which your lines of code are executed. In order to make complex programs, we need to make more than a list of statements to be executed sequentially. Control flow gives us tools to execute blocks of code conditionally (with if statements) and repeatedly (with loops).

In this chapter we shall cover if statements, but first we need to discuss Boolean data types and the logical operators which act on them.

Booleans

Booleans (`bool`)

The boolean data type (or bools) hold one of two values: `True` or `False`.

[2]:
```
bool_var1 = True

print('Var 1', bool_var1, type(bool_var1))

bool_var2 = False

print('Var 2', bool_var2, type(bool_var2))
```

```
Var 1 True <class 'bool'>
Var 2 False <class 'bool'>
```

Comparison Operators

Comparison Operators

Comparison operators operate on two variables and return a boolean result.

Less-than < and Greater-than >

These operators act in the same way as the mathematical objects you are familiar with. If `a` is less than `b`, then `a < b` will return `True` and `a > b` will return `False`. For example:

[3]:
```
print('3 > 2 is', 3 > 2)
print('2.54 < 1 is', 2.54 < 1)
print('1 < 1 is', 1 < 1)
```

```
3 > 2 is True
2.54 < 1 is False
1 < 1 is False
```

Note that these operators act on both integers and floats interchangeably.

Less-than-equal-to <= and Greater-than-equal-to >=

As there names suggest, the `<=` operator is related to the $\leq$ assertion in mathematics. Similarly `>=` is related to $\geq$.

[4]:
```
print('3.3 < 3.4 is', 3.3 <= 3.4)
print('2 <= 2 is', 2 <= 2)
print('2 >= 3.4 is', 2 >= 3.4)
```

```
3.3 < 3.4 is True
2 <= 2 is True
2 >= 3.4 is False
```

Equals-to ==

The `==` operator is used to check equality. When used on numbers, this is similar to the mathematical $=$.

[7]:
```
print('3 == 2 is', 3==2)
print('5.3 == 5.3 is', 5.3 == 5.3)
print('6 == 6.0 is', 6 == 6.0)
```

```
3 == 2 is False
5.3 == 5.3 is True
6 == 6.0 is True
```

The `==` operator is used more generally to compare non-numerical values. For example, it can be used to compare two strings:

[8]:
```
print("'apple' == 'apple' is", 'apple' == 'apple')
print("'banana' == 'apple' is", 'banana' == 'apple')
print('''"banana" == 'banana' is''', "banana" == 'banana')
```

```
'apple' == 'apple' is True
'banana' == 'apple' is False
"banana" == 'banana' is True
```

Not-equal-to `!=`

This operator returns `True` if the two objects being compared aren't equivalent (if `==` would return `False`). For example:

[10]:
```
print('3 != 2 is', 3 != 2)
print('7.3 != 7.3 is', 7.3 != 7.3)
print("'apple' != 'banana' is", 'apple' != 'banana')
```

```
3 != 2 is True
7.3 != 7.3 is False
'apple' != 'banana' is True
```

Logical Operators

Logical Operators

Logical operators act on booleans and return booleans. The logical operators are `and`, `or` and `not`.

Logical `and`

This acts on 2 booleans. It returns `True` if both booleans are `True` and `False` otherwise. For example:

[1]:
```
print('True and True is', True and True)
print('True and False is', True and False)
print('False and True is', False and True)
print('False and False is', False and False)
```

```
True and True is True
True and False is False
False and True is False
False and False is False
```

Logical `or`

This operator acts on 2 booleans. It returns `True` if at least one of the booleans is `True` and `False` if both booleans are `False`. For example:

[1]:
```
print('True or True is', True or True)
print('True or False is', True or False)
print('False or True is', False or True)
print('False or False is', False or False)
```

```
True or True is True
True or False is True
False or True is True
False or False is False
```

Logical `not`

This operator acts on a single boolean. It returns the opposite of the boolean:

[2]:
```
print('not True is', not True)
print('not False is', not False)
```

```
not True is False
not False is True
```

Combining Logical Operations

Although logical operations only act on up to 2 booleans at a time, just like arithmetic operators they can be combined in a single statement. For example:

[2]:
```
print('True and False or True is ', True and False or True)
print('True or True and False is', True or True and False)
print(not True or True and False)
```

```
True and False or True is  True
True or True and False is True
False
```

Although it isn't important for the cases above, if you need to ensure a specific order for the operations you can use brackets to group them.

If Statements

If Statement

The `if` statement is used to execute a block of code if a condition is true. The syntax of an if statement is:

```
if condition:
    block of code
```

where `condition` must be/evaluate to a boolean value. If `condition` evaluates to `True` then control moves to the block of code indented after the `:` and it is executed. If `condition` evaluates to `False`, then the block of code is skipped and control moves on to the code after the `if` statement.

Worked Example

Let's consider the case where we want to check if one variable is greater than the other:

[2]:
```
a = 3
b = 2

if a > b:
    print(a, 'is greater than', b)
```

```
3 is greater than 2
```

If we ran the code above but with

```
a = 2
b = 2
```

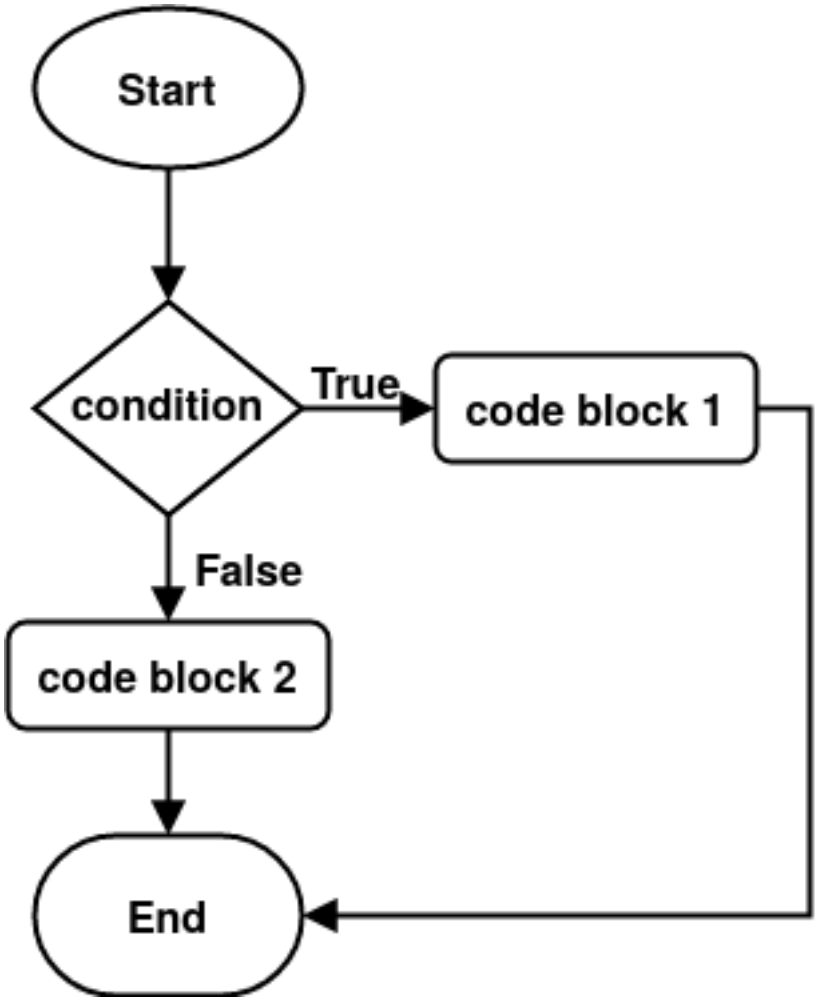
then we would see nothing printed out.

Else

What if you wanted to execute a code block if a statement is true; and another if it's false. The `else` part of an if statement can be used for this:

```
if condition:
    code block 1
else:
    code block 2
```

If `condition` evaluates to `True` then `code block 1` will be executed. If, on the other hand, `condition` evaluates to `False`, `code block 2` will be executed.



Worked Example

Let's take our first example and add an `else` part to it:

[3]:
```
a = 3
b = 2

if a > b:
    print(a, 'is greater than', b)
else:
    print(a, 'is less than or equal to', b)
```

```
3 is greater than 2
```

[4]:
```
a = 1
b = 2
```

```
if a > b:
    print(a, 'is greater than', b)
else:
    print(a, 'is less than or equal to', b)
```

```
1 is less than or equal to 2
```

Elif

Now, what if you have more than 2 mutually/partially exclusive conditions? This is a job for `elif` statements:

```
if condition_1:
    code block 1
elif condition_2:
    code block 2
elif condition_3:
    code block 3
```

Here computer first checks `condition_1`. If `condition_1` evaluates `True` then `code block 1` is executed and control moves leaves the if statement. If `condition_1` evaluates as `False` then the computer will check `condition_2`, if this evaluates as `True` then `code block 2` will be executed and control will leave the if statement. If both `condition_1` and `condition_2` are both `False`, then the computer will check `condition_3`, if this evaluates to `True` `code block 3` will be executed and control will leave the if statement.

See the flow chart in the **`else` and `elif`** section if this explanation was confusing.

An alternative to using `elif` statements is nested `if`/`else` statements:

```
if condition_1:
    code block 1
else:
    if condition_2:
        code block 2
    else:
        if condition_3:
            code block 3
```

This is quite messy. A general rule of thumb for Python is to avoid nesting where possible. There are certain scenarios where nested `if` statements are desired, however.

Never use multiple `if` statements to do the job of `elif` statements:

```
if condition_1:
    code block 1
if (not condition_1) and condition_2:
    code block 2
if (not condition_1) and (not condition_2) and condition_3:
    code block 3
```

this is not only annoying to write, it is computationally expensive. If one of these conditions is true, then the others aren't, but the computer will still check each condition in turn.

For example, lets write a script that checks if a variable is a multiple of 2, 3, or 5.

[4]:
```
var = 4

if var % 2 == 0:
    print('Variable is a multiple of 2')
elif var % 3 == 0:
    print('Variable is a multiple of 3')
elif var % 5 == 0:
    print('Variable is a multiple of 5')
```

```
Variable is a multiple of 2
```

[6]:
```
var = 21

if var % 2 == 0:
    print('Variable is a multiple of 2')
elif var % 3 == 0:
    print('Variable is a multiple of 3')
elif var % 5 == 0:
    print('Variable is a multiple of 5')
```

```
Variable is a multiple of 3
```

[7]:
```
var = 25

if var % 2 == 0:
    print('Variable is a multiple of 2')
elif var % 3 == 0:
    print('Variable is a multiple of 3')
elif var % 5 == 0:
    print('Variable is a multiple of 5')
```

```
Variable is a multiple of 5
```

[8]:
```
var = 6

if var % 2 == 0:
    print('Variable is a multiple of 2')
elif var % 3 == 0:
    print('Variable is a multiple of 3')
elif var % 5 == 0:
    print('Variable is a multiple of 5')
```

```
Variable is a multiple of 2
```

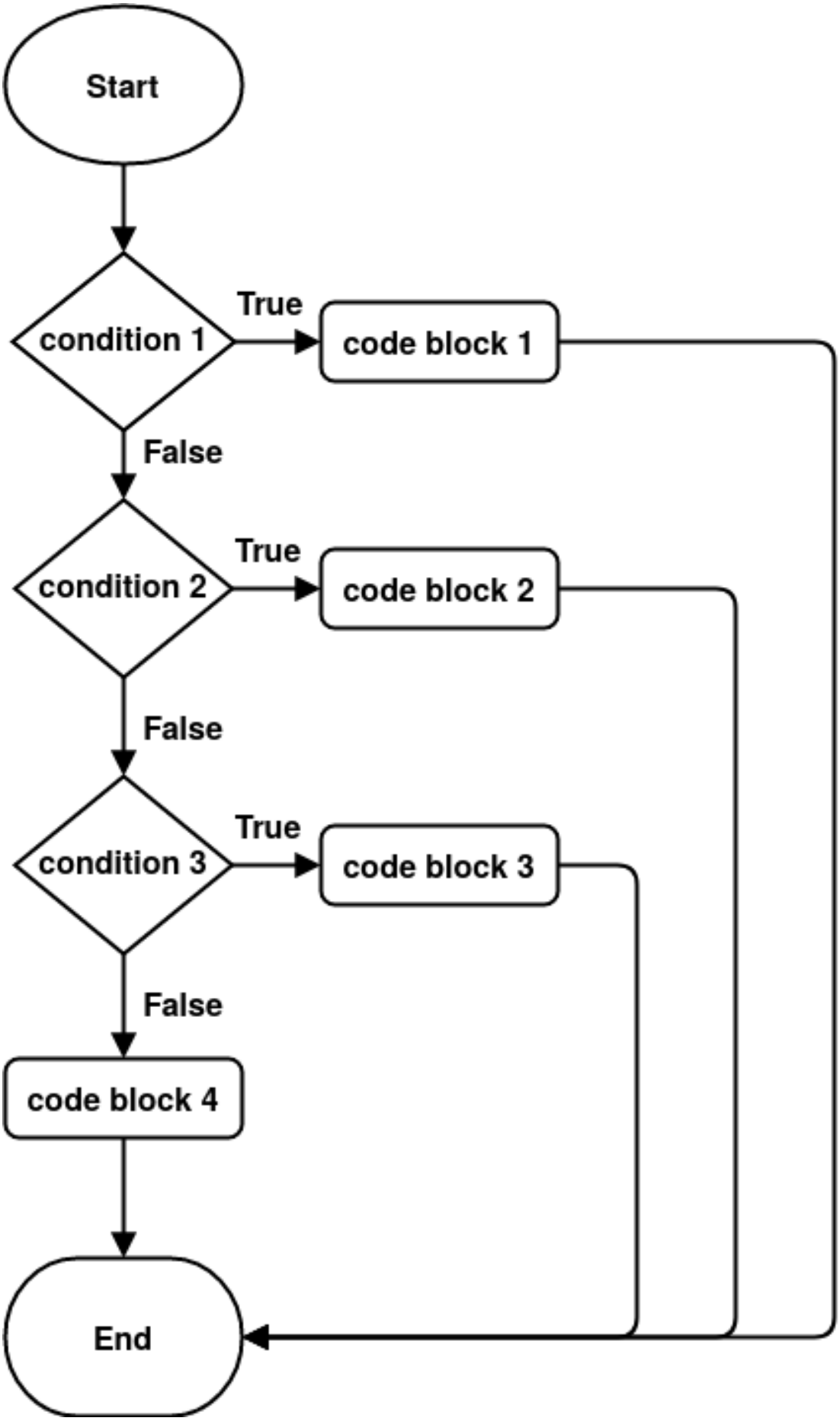
Note that even though 6 is a multiple of both 2 and 3, because 2 appears above 3 in the `if` statement that's the message we see.

[9]:
```
var = 7

if var % 2 == 0:
    print('Variable is a multiple of 2')
elif var % 3 == 0:
    print('Variable is a multiple of 3')
elif var % 5 == 0:
    print('Variable is a multiple of 5')
```

Note that 7 is not a multiple 2, 3 or 5 and thus all of the `if` and `elif` conditions are false.

Else and Elif

If you include an `else` part of an `if` statement with `elif` parts, the code block in the `else` statement only executes if all of the conditions in the `if` and `elif` statements that precede it are false.


Start
condition 1
True
code block 1
False
condition 2
True
code block 2
False
condition 3
True
code block 3
False
code block 4
End

As an example of this let's go back to our original example:

[1]:
```
a = 3
b = 2

if a > b:
    print(a, 'is greater than', b)
elif a < b:
    print(a, 'is less than', b)
else:
    print(a, 'is equal to', b)
```

```
3 is greater than 2
```

[2]:
```
a = 1
b = 2

if a > b:
    print(a, 'is greater than', b)
elif a < b:
    print(a, 'is less than', b)
else:
    print(a, 'is equal to', b)
```

```
1 is less than 2
```

[3]:
```
a = 2
b = 2

if a > b:
    print(a, 'is greater than', b)
elif a < b:
    print(a, 'is less than', b)
else:
    print(a, 'is equal to', b)
```

```
2 is equal to 2
```

Loops

Loops

Control Flow: Loops

As mentioned in the previous chapter, control flow is the order in which a program executes.

In this chapter we will discuss loops which are used to repeat code blocks (allowing variation each time).

In Python there are two types of loops: the `for` loop and the `while` loop.

For Loops

For Loops

For loops can be used to repeat a block of code by iterating through specified values.

The structure of a for loop is as follows:

```
for i in iterator:
    code block to be repeated
```

where `i` is the iteration variable and the iterator represents a sequence of values that `i` will be set to each time the loop is repeated. Here `i` can be treated like a variable and can take on any allowed variable name.

The code block to be repeated must be indented after the `:`. The loop repeats until `i` has run through all of the values in the iterator, or until the loop is broken.

Be careful not to alter the iterator inside the code block being repeated.

Worked Example

As a simple example, let's say we wanted to print out the first n integer squares starting with 0.

This can be achieved manually for small n, for example $n = 5$:

[3]:
```
print('0 squared is', 0**2)
print('1 squared is', 1**2)
print('2 squared is', 2**2)
print('3 squared is', 3**2)
print('4 squared is', 4**2)
print('5 squared is', 5**2)
```

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

This quickly becomes tedious and produces messy code. To achieve the same goal using a `for` loop we can do the following:

[5]:
```
for i in range(6):
    print(i, 'squared is', i**2)
```

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

Here we have made use of the `range` function to create our iterator.

The `range()` Function

The `range()` function takes integer arguments and produces a series of integers. As we shall see, the `range()` function's arguments are very similar to slicing.

In the example above we used it with one argument,

`range(stop)`

Here `range()` will produce a series of integers starting at zero and ending just before the `stop` value.

If we were to use range with 2 arguments:

`range(start, stop)`

`range()` will produce a series of integers starting with the `start` value and ending with the `stop` value.

For example:

[1]:
```
for i in range (2, 5):
    print(i)
```

```
2
3
4
```

Lastly if we were to use `range()` with 3 arguments:

`range(start, stop, step)`

`range()` returns a series starting with the `start` value and with step sizes of `step` in between each value until it reaches the value before `stop`.

For example:

[2]:
```
for i in range(2, 10, 3):
    print(i)
```

```
2
5
8
```

The default value for `step` is 1. If you want the series to descend, you can use a negative step size:

[5]:
```
for i in range(11, 3, -2):
    print(i)
```

```
11
9
7
5
```

Looping Through Sequences

Sequences such as tuples, lists and strings can also be used as iterators. For example:

[6]:
```
for char in 'This string':
    print(char)
```

```
T
h
i
s

s
t
r
i
n
g
```

and

[7]:
```
for item in [1, 2, 3, 'a', 'b', 'c']:
    print(item)
```

```
1
2
3
a
b
c
```

`enumerate()` To Iterate Through Sequence and Index

Sometimes you want to loop through a sequence but also want to keep track of the index. This can be achieved by using range:

[3]:
```
string = 'string'

for i in range(len(string)):
    print(i, string[i])
```

```
0 s
1 t
2 r
3 i
4 n
5 g
```

but there is a far more convenient way using the `enumerate()` function:

[1]:
```
for i,char in enumerate('string'):
    print(i, char)
```

```
0 s
1 t
2 r
3 i
4 n
5 g
```

`zip()` To Iterate Through More Than One Sequence Simultaneously

If you wanted to loop through more than one sequence at a time you could iterate through the index:

[5]:
```
list_a = [1,2,3]
list_b = ['a', 'b', 'c']

for i in range(len(list_a)):
    print(list_a[i], list_b[i])
```

```
1 a
2 b
3 c
```

but there is a cleaner way using the `zip` function:

[7]:
```
for a,b in zip([1,2,3], ['a', 'b', 'c']):
    print(a, b)
```

```
1 a
2 b
3 c
```

Note that the loop will only iterate as much as the shortest sequence.

Looping Through Dictionaries

To loop through the key-value pairs of a dictionary you can use the `dict.items()` method:

[1]:
```
d = {'a' : 54, 'b' : 754, 'c' : 42}

for k,v in d.items():
    print(k, v)
```

```
a 54
b 754
c 42
```

List Comprehension

List Comprehension

There will be many times you will want to automate the creation of a list. You can use loops for this but can become impractical. A nice way to generate lists is using **list comprehension**:

[2]:
```
#Generating a list of integers in ascending order
numbers = [i for i in range(6)]
print(numbers)
```

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

You can treat the `for` inside the list just like a `for` loop, including looping through collections:

[4]:
```
string = 'abcdefg'

#Generating a list of characters from a string
char_list = [char for char in string]
print(char_list)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

Only use list comprehension if you are interested in the list itself. Do not use it in place of a `for` loop.

You can also embed list comprehension:

[5]:
```
print([[i + j for j in range(3)] for i in range(4) ])
```

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

While Loops

While Loops

For loops are useful if you know what you want to iterate over, but what if you wanted to keep looping until a certain condition is met? `while` loops are the tool for this job.

The syntax for a `while` loop is:

```
while condition:
    block of code to be repeated
```

where `condition` is/evaluates to a boolean value. The loop will keep repeating, executing the block of code indented after the `:` as long as `condition` evaluates to `True`. When `condition` evaluates to `False` the loop will no longer be repeated and control will progress to the code after the loop. Note that if `condition` starts as `False`, the code inside the loop will never be executed.

Worked Example

Let's consider the following problem where we can make use of a `while` loop. Consider the recursive series:

$$T_n = T_{n-1}^{3/4} \quad (1)$$
$$T_0 = 100 \quad (2)$$

We want to know when this series drops below 2 (what is the first value of n for which $T_n < 2$). One solution is:

[3]:
```
T = 100 #T_0 term

n = 0

while T >= 2:
    T = T**(3/4.) #T_{n+1} term
    n += 1

print('T_n is less than 2 for n =', n)
```

```
T_n is less than 2 for n = 7
```

Notice how the condition is `T >= 2` and not `T < 2`. That is because the loop continues **while** the condition is true and we want the loop to stop when `T < 2` is `True` (and the converse `T >= 2` is `False`).

Avoiding Infinite Recursion

Something to be careful of when using `while` loops is a loop that doesn't stop looping. If `condition` never evaluates to `False`, or if you never break out of the loop in another way, control will never leave the loop. Sometimes it is useful to use a maximum number of loop iterations to avoid this:

```
counter = 0

while condition and counter < max_count:
    block of code
```

where `max_count` is the chosen maximum number of recursions (normally chosen as a very large number).

Replacing For Loops

`while` loops can be used to replace for `loops`, for example:

[4]:
```
## For loop
print('for loop')

for i in range(5):
    print(i)

## While loop
print('')
print('while loop')

i = 0

while i < 5:
    print(i)
    i+=1
```

```
for loop
0
1
2
3
4

while loop
0
1
2
```

```
3
4
```

As you can see the `while` loop is a bit less convenient than the `for` loop in this case. The `while` loop becomes even less convenient when looping through a collection:

[5]:
```
string = 'a string'

## For loop
print('for loop')

for char in string:
    print(char)

## While loop
print('')
print('while loop')

index = 0

while index < len(string):
    print(string[index])
    index += 1
```

```
for loop
a

s
t
r
i
n
g

while loop
a

s
t
r
i
n
g
```

Breaking Out of Loops

Breaking Out of Loops

Sometimes you want to exit a loop before it's finished, or skip the remainder of a loop and move to the next iteration. To do this you can use the `break` and `continue` statements respectively.

`break`

As a first example, consider:

[1]:
```
for i in range(10):
    print(i)

    if i == 5:
        break
```

```
0
1
2
3
4
5
```

where you can see that the loop terminated before it was finished iterating through `range(10)`. The `break` may be inside the `if` statement, but it's the loop that it affects.

The `break` statement exits the first loop that it's nested in. For example, if we had multiple nested loops:

[5]:
```
for i in range(3):
    print('Loop1', i)
    for j in range(3):
        print('    Loop2', j)

        if j == 1:
            break
```

```
Loop1 0
    Loop2 0
    Loop2 1
Loop1 1
```

```
    Loop2 0
    Loop2 1
Loop1 2
    Loop2 0
    Loop2 1
```

We can see that the outer loop (Loop1) iterated through all of `range(3)`, while Loop2 terminates before it can reach the last iteration.

`continue`

If you want to end the current loop iteration, but you don't want to break out of the loop, you can use the `continue` statement.

[9]:
```
for i in range(10):
    if i == 5:
        continue
    print(i)
```

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

As you can see in the example above, `5` is not printed.

Else Statement and Loops

Else Statement and Loops

You can use an else statement after a `for` or `while` loop. The code in this `else` statement is executed if the loop completed without being terminated.

[3]:
```
for i in range(3):
    print(i)
else:
    print('Loop completed')
```

```
0
1
2
Loop completed
```

The only time the `else` part will not be executed is if you `break` out of a loop:

[4]:
```
for i in range(5):
    print(i)

    if i == 3:
        break
else:
    print('Loop completed')
```

```
0
1
2
3
```

Worked Example

A common use for this structure is if you're searching for an object. Consider this example where we are trying to find a `'fish'` in a list:

[5]:
```
animals = ['zebra', 'cow', 'crow', 'eel']

for animal in animals:
    if animal == 'fish':
```

```
        print('We caught a fish!')
        break
else:
    print('We did not catch a fish.')
```

```
We did not catch a fish.
```

[6]:
```
animals = ['human', 'bear', 'fish', 'squid', 'crab']

for animal in animals:
    if animal == 'fish':
        print('We caught a fish!')
        break
else:
    print('We did not catch a fish.')
```

```
We caught a fish!
```

Of course, finding a particular object in a list is quicker and simpler using:

[7]:
```
animals = animals = ['human', 'bear', 'fish', 'squid', 'crab']

if 'fish' in animals:
    print('We caught a fish!')
else:
    print('We did not catch a fish.')
```

```
We caught a fish!
```

but for more complex procedures this may not be an option.

Functions

Functions

Defining Functions

In this chapter we cover how to define custom functions.

Functions are defined using the keyword `def`.

The basic syntax for creating a function is:

```
def function_name(arguments):
    Code block
    return return_value
```

where - everything indented after the `:` is part of the function body - `arguments` can be multiple arguments with names to refer to in the function body - the `return` statement exits the function and returns the `return_value`

The function above can be called in the usual way: `function_name(argument_values)`

Worked Example

As a first example, let's create a function that takes a single argument and doubles it's value

[4]:
```
def double(value):
    return 2*value
```

Again, we can call this argument by name and enter a value or variable as an argument:

[5]:
```
double(1)
```

[5]: 2

[6]:
```
double(5.5)
```

[6]: 11.0

[7]:
```
double('a')
```

[7]: 'aa'

Return Statement

`return` Statement

`return None`

Some functions return nothing (for example the `print()` function). To achieve this you can either return `None`, leave the return value blank after `return`, or put no `return` statement at all.

[11]:
```
def none1():
    return

def none2():
    return None

def none3():
    x = 2 #Needs code to work
```

[13]:
```
type(none1())
```

[13]: NoneType

[14]:
```
type(none2())
```

[14]: NoneType

[15]:
```
type(none3())
```

[15]: NoneType

`return` Breaks Out of the Function

It was stated above that the `return` statement breaks out of the function. This means that anything that comes directly after a `return` inside the function body will not execute. Consider the following example to illustrate this:

[16]:
```
def message():
    print('This code will execute')
    return
    print('This code will not execute')
```

[17]:
```
message()
```

```
This code will execute
```

It can be useful to use this feature of `return` to break out of a loop, or even to ignore the `else` or `elif` parts of an `if` statement.

For example, consider the function that checks if it's argument is even or odd:

[1]:
```
def is_even(value):
    if value%2 == 0:
        return True
    else:
        return False
```

[2]:
```
is_even(3)
```

[2]:
```
False
```

[3]:
```
is_even(6)
```

[3]:
```
True
```

The else part of the function is unnecessary:

[1]:
```
def is_even(value):
    if value%2 == 0:
        return True
    return False
```

[18]:
```
is_even(3)
```

[18]:
```
False
```

Arguments

Function Arguments

You can include as many arguments as you want in your function definition. Inside the function, these arguments can be treated as variables.

[21]:
```
def arg3(arg1, arg2, arg3):
    print(arg1)
    print(arg2)
    return arg3
```

[22]:
```
arg3(1, 2, 3)
```

```
1
2
```

[22]: 3

You may use variables or statements (anything that resolves to a value or object) as arguments:

[24]:
```
var1 = 45

arg3(var1, 3*4, 7)
```

```
45
12
```

[24]: 7

Positional Arguments

The arguments defined above are called **positional arguments**. In order to set them correctly, you need to parse them in the order they are defined in the function. You must also provide a value for each argument:

[23]:
```
arg3('a', 'b')
```

```
␣
↪---------------------------------------------------------------------------
```

```
       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-23-964cf24ab02e> in <module>
    ----> 1 arg3('a', 'b')


       TypeError: arg3() missing 1 required positional argument: 'arg3'
```

Keyword Arguments

If you want to set optional arguments with a default value, you can use keyword arguments. The syntax is:

```
def function_name(keyword_arg = default_value):
```

For example:

[1]:
```
def hello(name = 'World', time = 'today'):
    return f'Hello {name}! How are you {time}?'
```

(If you are unfamiliar with f-strings `f''`, see the section String Formatting)

This function can be called with no arguments, in which case the default values will be used:

[3]:
```
hello()
```

[3]:
```
'Hello World! How are you today?'
```

We can also parse the arguments like positional arguments:

[4]:
```
hello('reader', 'feeling')
```

[4]:
```
'Hello reader! How are you feeling?'
```

[5]:
```
hello('reader')
```

[5]:
```
'Hello reader! How are you today?'
```

Keyword arguments can be referred to by name, and out of order:

[6]:
```
hello(time = 'this morning')
```

[6]:
```
'Hello World! How are you this morning?'
```

Combining Positional and Keyword Arguments

If you define a function with both positional and keyword arguments, the positional arguments must appear **before** the keyword arguments.

For example:

[4]:
```
def hello_hello(num, name = 'World', time = 'today', weather = 'good'):

    return f"Hello {num*'hello'} {name}! How are you {time}?"
```

Here the positional argument `num` must be provided

[3]:
```
hello_hello()
```

```
␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-3-260a0f5fbf0a> in <module>
   ----> 1 hello_hello()


       TypeError: hello_hello() missing 1 required positional argument: 'num'
```

As before, the keyword arguments can be provided like positional arguments or by name:

[5]:
```
hello_hello(1, 'there', 'doing')
```

[5]:
```
'Hello hello there! How are you doing?'
```

[6]:
```
hello_hello(0, time = 'awake this early', name = 'sleepy head')
```

[6]:
```
'Hello  sleepy head! How are you awake this early?'
```

Local Variables

Local Variables

Variables defined in the main body of a script are called **global** variables. These variables are accessible inside of functions:

[4]:
```
x = 5

def get_x():
    return x
```

[5]:
```
get_x()
```

[5]: 5

The arguments parsed into and the variables defined inside the function are **local variables**. They only exist in a particular instance of a function.

In other words, these variables are not accessable from outside the function. For example:

[1]:
```
def make_var():
    func_var = 4
    return func_var
```

[2]:
```
make_var()
```

[2]: 4

[3]:
```
func_var
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-3-96e608aeebf3> in <module>
----> 1 func_var
```

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

If we were to define `func_var` as a global variable, `make_var` will instance a local variable instead of reassigning the global variable:

[6]:
```
func_var = 6

print('Before function', func_var)
print('Function return', make_var())
print('After function', func_var)
```

```
Before function 6
Function return 4
After function 6
```

Note that when referencing a variable, Python will check the local namespace **before** the global namespace (i.e. local variables are given preference).

As stated above, function arguments can also be treated as local variables.

[7]:
```
def arg_var(x):
    return x
```

[8]:
```
x = 5

arg_var(2)
```

[8]:
```
2
```

Recursive Functions

Recursive Functions

Recursive functions are functions that make calls to themselves.

They can be used in place of loops. Though in Python they don't necessarily provide a more efficient solution, there are many problems for which a recursive function is the most elegant and convenient solution.

Worked Example: Factorial

One of the most famous implementations of a recursive function is to implement the factorial:

$$0! = 1$$
$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 2 \times 1$$

This is achieved by using the recurrence relation:

$$n! = n \times (n-1)!$$

The recursive function which solves this is:

[10]:
```
def factorial(n):
    if not type(n) is int:
        print('n must be an integer')
        return
    if n <0:
        print('n must be greater than or equal to 0')
        return

    if n == 0:
        return 1

    return n*factorial(n-1)
```

Note, an important aspect of this function is the return value of 1 for `n == 0`. This is called the base class, without it the function would never finish it's recursion.

Putting this function into action:

[11]:
```
factorial(-1)
```

```
n must be greater than or equal to 0
```

[12]:
```
factorial(0.5)
```

```
n must be an integer
```

[4]:
```
factorial(0)
```

[4]: 1

[5]:
```
factorial(1)
```

[5]: 1

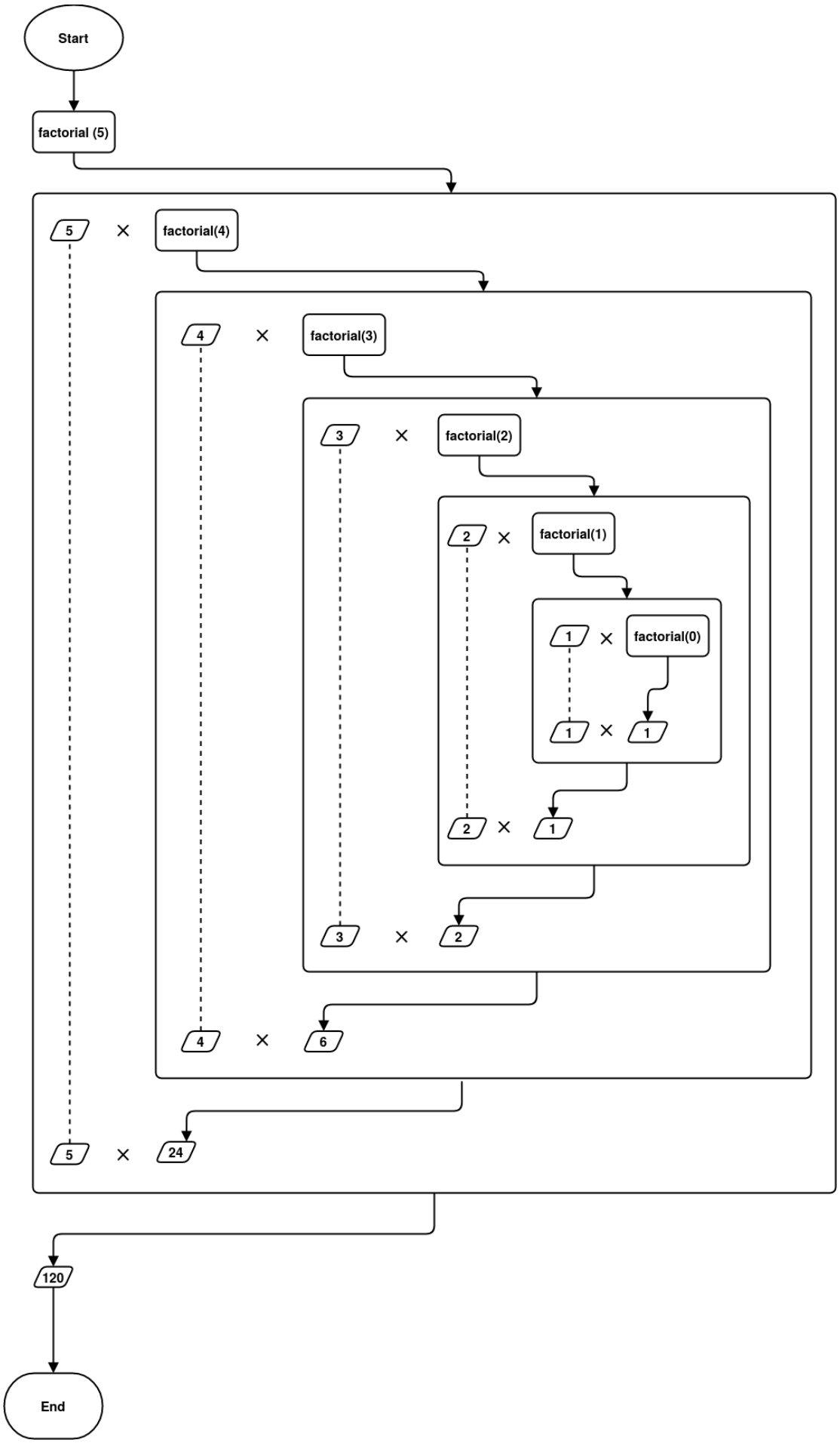
[8]:
```
factorial(5)
```

[8]: 120

[9]:
```
factorial(10)
```

[9]: 3628800

The inner workings of this `factorial()` function are fairly subtle. The (informal) flow diagram below illustrates the function call for `factorial(5)`:


Start
factorial (5)
5
×
factorial(4)
4
×
factorial(3)
3
×
factorial(2)
2
×
factorial(1)
1
×
factorial(0)
1
×
1
2
×
1
3
×
2
4
×
6
5
×
24
120
End

The Base Class

As mentioned earlier, a recursive function must have at least one base class. The base class is a return state that **doesn't** make another recursive function call.

It's also important to make sure that the recursion eventually reaches the base class when designing your function.

File I/O

File I/O

File I/O

So far we have focused on using terminal outputs (using `print()`) and inputs (using `input()`). File I/O, or rather file input/output, is simply reading inputs from and writing outputs to files stored on disc.

In this chapter we will cover the standard library approach. In practice, many modules/packages (such as NumPy) contain their own methods for reading and writing files that are more practical to use in certain contexts.

File I/O

File I/O

`open()` Function

The `open()` function is the Standard Library option for reading and writing both text and binary files. It returns a file object, the exact type depending on the type of file you read.

The file object has methods for reading from and writing to the file.

- The file object is iterable
- Signature `python  open(filename, mode = 'r')`

File Modes

The different modes for `open()` are (taken from the docstring):

Character	Meaning
'r'	open for reading (**default**)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (**default**)
'+'	open a disk file for updating (reading and writing)
'U'	universal newline mode (deprecated)

Modes can be combined. For example, the default mode is `'rt'`, or "read text-file". If you wanted to open a binary file in write mode, you could use `'wb'`. See the docstring for more of an explanation.

Open Files in a `with` Statement

A good practice when opening files with `open()` is to use a with statement:

```
with open('new_file.txt', 'w') as f:
    #file object can be used here
#file object is closed
```

Here the variable `f` refers to the file object that `open()` returns. When control leaves the `with` statement, the file is closed (see the section at the bottom of this page on how to do this manually). Outside the `with` the file object will still exist, but you won't be able to read from or write to it.

Writing to Files

You can write to files using the `'w'` mode in `open()`. This creates a new file if the file specified doesn't already exist, or **over-writes** the file if it already does (replacing the content). To write to the file use the `.write()` method on the file object:

[1]:
```
with open('new_file.txt', 'w') as f:
    f.write('A line of text in the file.')
```

In the example above we have written to a file called `'new_file.txt'`. This file will be located in the same directory as your script/notebook. The contents of the file is a single line:

```
A line of text in the file
```

The `.rite()` method writes strings to the file. Unlike `print`, `write` **only** takes strings. Also, if you want to write to a new line you need to use the new line special character `'\n'`:

[2]:
```
with open('new_file.txt', 'w') as f:
    f.write('First line of the file.\n')
    f.write('Second line of the file. ')
    f.write('This is still the second line of the file.')
```

The contents of **new_file.txt** now reads as follows:

```
First line of the file.
Second line of the file. This is still the second line of the file.
```

Alternatively you could write the contents as a multi-lined string literal:

[3]:
```
with open('new_file.txt', 'w') as f:
    f.write('''First line.
Second line.
Third line.''')
```

As you can see the string literal above does not have indentations. This is because those indentations would be a part of the string literal itself. Multi-lined string literals can, thus make it difficult to read indented code blocks. The contents of **new_file.txt** now reads:

```
First line.
Second line.
Third line.
```

Reading Files

You can read a function using the `'r'` mode of `open()`. The file object returned has a few options for reading the content.

The `.read()` Method

If you want to read the entire contents of the file into a single string, you can use the `.read()` method of the file object:

[4]:
```
with open('new_file.txt', 'r') as f:
    data = f.read()

print(data)
```

```
First line.
Second line.
Third line.
```

Note that the file object keeps track of where you have read to in the file. When you have reached the end of the file (as is the case after using `.read()`) you cannot read more content.

[5]:
```
with open('new_file.txt', 'r') as f:
    data1 = f.read()
    data2 = f.read() #A second reading

print('First reading:')
print(data1)
print('')
print('Second reading')
print(data2)
```

```
First reading:
First line.
Second line.
Third line.

Second reading
```

The `.readline()` Method

If you want to read the next line of the file object, you can use the `.readline()` method:

[10]:
```
with open('new_file.txt', 'r') as f:
    print('1', f.readline())
    print('2', f.readline())
```

```
1 First line.

2 Second line.
```

The `.readlines()` Method

If you want to create a list of the lines in the file, you can use the `.readlines()` method:

[11]:
```
with open('new_file.txt', 'r') as f:
    lines = f.readlines()
```

```
print(lines)
```

```
['First line.\n', 'Second line.\n', 'Third line.']
```

Iterating Through File Objects

The file object returned by `open()` is iterable. Each iteration call returns a line of the file. We can use this in a `for` loop:

[6]:
```
with open('new_file.txt', 'r') as f:
    for line in f:
        print(line)
```

```
First line.

Second line.

Third line.
```

Let's print the corresponding line numbers of each line to further illustrate what is happening:

[8]:
```
with open('new_file.txt', 'r') as f:
    for i,line in enumerate(f):
        print(i+1, line)
```

```
1 First line.

2 Second line.

3 Third line.
```

Opening Files Without `with`

If, for some reason, you don't want to make use of the `with` statement when opening your files, make sure to close your file objects when you are done with them:

```
f = open('new_file.txt', 'w')

#file object used

f.close()
```

Data Files

Structured Data Files

In this section we focus on reading from and writing to files with a row-column format, such as is found in comma-separated (csv) and tab-separated (tsv) data files.

Although `numpy.loadtxt()` is suitable for this task, it is valuable to be able to write your own code solution.

Writing a Data File

Let us generate some data and write it in a csv format (comma-separated values). In general what you use as the separator (delimiter) for your data is up to you, but if we use a .csv file extension it's best to stick to the standard.

[2]:
```
import numpy as np

#Generating data
x = np.linspace(0, 2*np.pi)
y = np.sin(x)
z = np.cos(x)

#Writing the data to file in csv format
with open('data1.csv', 'w') as f:
    f.write('x,sin(x),cos(x)\n') #Header

    for xx, yy, zz in zip(x, y, z):
        f.write(f'{xx},{yy},{zz}\n')
```

If you are not familiar with the string formatting used (`f'{xx},{yy},{zz}\n'`) see the page on String Formatting. Note that it is in this line (and also in the header) that we have separated the values with commas.

Note that the file extension **.csv** acts more as a hint for other software. There is no physical difference between a file we write with this extension or any other extension (including no extension). As long as the file mode is set to text (`'t'`), we are writing plain text files.

The output of our data file **data1.csv** looks like:

```
x,sin(x),cos(x)
0.0,0.0,1.0
0.1282282715750936,0.127877161684506,0.9917900138232462
```

```
0.2564565431501872,0.25365458390950735,0.9672948630390295
0.38468481472528077,0.3752670048793741,0.9269167573460217
0.5129130863003744,0.49071755200393785,0.8713187041233894
0.6411413578754679,0.5981105304912159,0.8014136218679567
0.7693696294505615,0.6956825506034864,0.7183493500977276
0.8975979010256552,0.7818314824680298,0.6234898018587336
1.0258261726007487,0.8551427630053461,0.5183925683105252
```

Or in a more presentable format:

```
<IPython.core.display.HTML object>
```

Reading a Data File

Now, let's read the data file we wrote. If we want to store each column in a separate list or array, it will be best to iterate through the lines of the file.

We will need to divide the values from each line using the separator. To do this, we will use the `.split()` string method:

[7]:
```
'a b c d'.split()
```

[7]:
```
['a', 'b', 'c', 'd']
```

As you can see this splits the string into a list of strings. By default it uses a space as the dividing character, given a string argument it will use that as the delimiter instead:

[8]:
```
'a,b,c,d'.split(',')
```

[8]:
```
['a', 'b', 'c', 'd']
```

We must keep in mind that the file we are reading has a header we want to read before any of the data.

Something else to keep in mind is that the file contains text (or rather the content is a string). If we want to store the data as numbers, we need to convert them first.

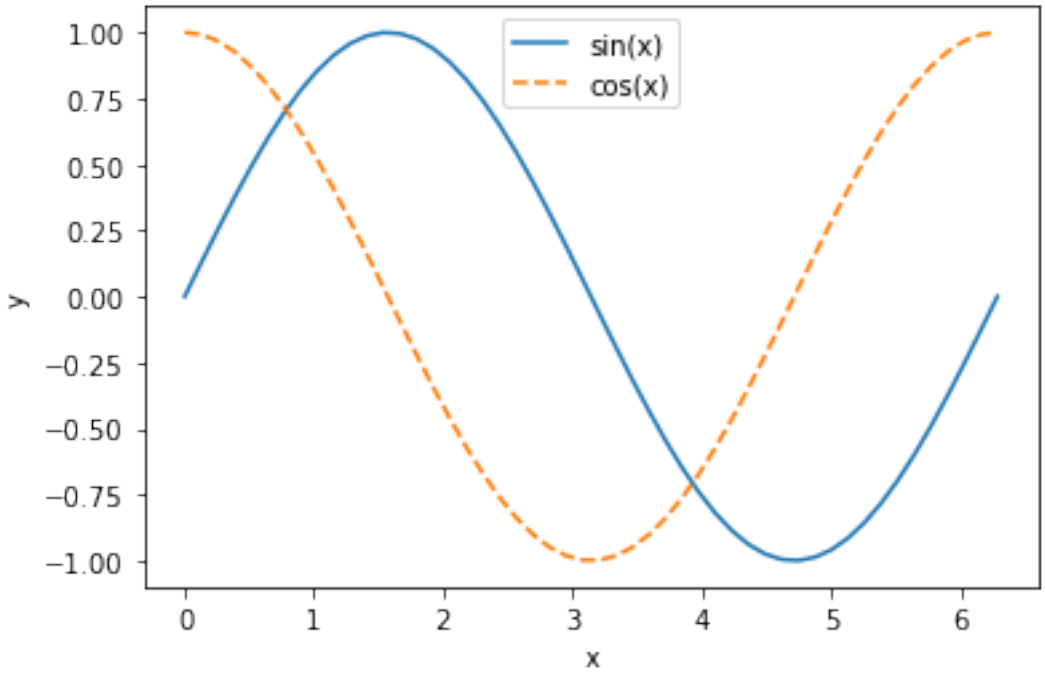
[9]:
```
#Lists to hold the data
x = []
y = []
z = []

with open('data1.csv', 'r') as f:
    header = f.readline() #read header

    for line in f:
        line = line.strip() #This clears trailing whitespace (e.g. \n)

        #Makes a list from the string using ',' as the seperator
```

```
        line = line.split(',')

        x.append(float(line[0]))
        y.append(float(line[1]))
        z.append(float(line[2]))

#If you need to convert x, y, z to arrays:
x = np.array(x)
y = np.array(y)
z = np.array(z)
```

Note that we start with lists and convert to an array later (if an array is needed). The reason for doing this is that we don't necessarily know how many lines the file has before we begin, and appending to lists is more easy and efficient than concatenating arrays.

As a sanity check, let's plot the data we have just read:

[12]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot(x, y, label = 'sin(x)')
ax.plot(x, z, '--',  label = 'cos(x)')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend(loc = 9)

plt.show()
```



Writing and Reading a Tab Separated File

If you are comfortable with the sections above, you may skip this one. If you'd like to see another example of a data file with a different delimiter, we will write a data file using tab separation instead of commas (tsv).

[1]:
```
import numpy as np

#Generating data
x = np.linspace(0, 2, 100)
y = np.sqrt(x)
z = x*x

#Writing a data file with space seperations
with open('data2.tsv', 'w') as f:
    f.write('x\ty\tz\n') #Header

    for xx, yy, zz in zip(x, y, z):
        f.write(f'{xx}\t{yy}\t{zz}\n')
```

Here we use the special character `'\t'` which stands for tabs.

Again, the use of the **.tsv** file extension is a convention, it does not alter the nature of the file itself.

The contents of the data file we have generated looks like this:

```
x       y       z
```

```
0.0     0.0     0.0
0.020202020202020204    0.1421338109037403      0.0004081216202428324
0.04040404040404041     0.20100756305184242     0.0016324864809713297
0.06060606060606061     0.24618298195866548     0.003673094582185491 7
0.08080808080808081     0.2842676218074806      0.006529945923885319
0.10101010101010102     0.31782086308186414     0.010203040506070812
0.12121212121212122     0.3481553119113957      0.014692378328741967
0.14141414141414144     0.3760507165451775      0.019997959391898794
0.16161616161616163     0.40201512610368484     0.026119783695541274
```

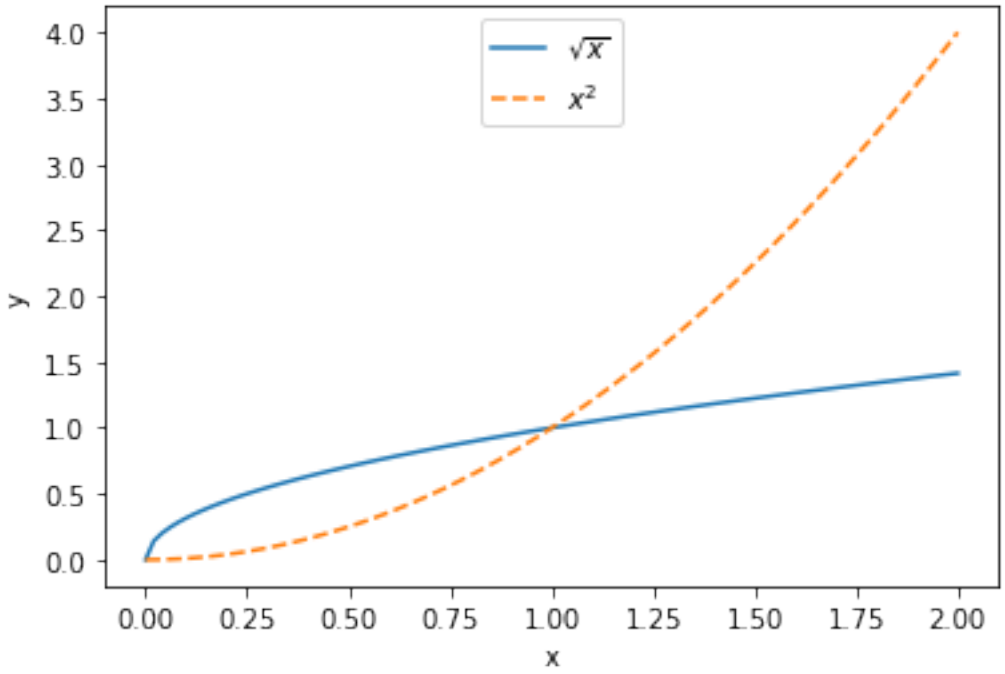
Now, let's read the data keeping in mind that the values are now separated with tabs.

[9]:
```
#Lists to hold the data
x = []
y = []
z = []

with open('data2.tsv', 'r') as f:
    header = f.readline() #read header

    for line in f:
        line = line.strip() #This clears trailing whitespace (e.g. \n)

        #Makes a list from the string using '\t' as the seperator
        line = line.split('\t')

        x.append(float(line[0]))
        y.append(float(line[1]))
        z.append(float(line[2]))

#If you need to convert x, y, z to arrays:
x = np.array(x)
y = np.array(y)
z = np.array(z)
```

Plotting this data:

[29]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot(x, y, label = r'$\sqrt{x}$')
ax.plot(x, z, '--',  label = r'$x^2$')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend(loc = 9)
```

```
plt.show()
```



Reading Data in as a Single Array

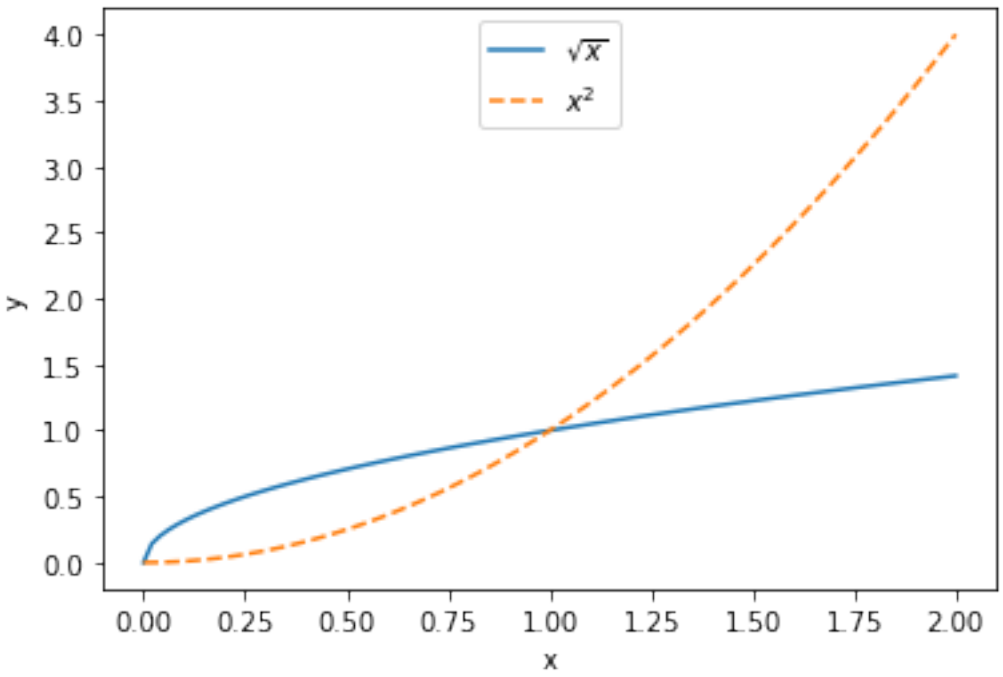
Sometimes you want to read the data in as a single 2D array (for example if you have a large data file or if the number of columns in your data file aren't fixed). Let's read the file **data2.tsv** in this manner:

[3]:
```
#Lists to hold the data
data = []

with open('data2.tsv', 'r') as f:
    header = f.readline() #read header

    for line in f:
        line = line.strip() #This clears trailing whitespace (e.g. \n)
        line = line.split('\t') #Makes a list

        #Converting data to floats
        for i,col in enumerate(line):
            line[i] = np.float(col)

        data.append(line)
```

```
#Converting data to array
data = np.array(data)
```

Note that this gives us a similar output to NumPy's `numpy.loadtxt()`. Plotting the data (use slices to extract the columns):

[5]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot(data[:,0], data[:,1], label = r'$\sqrt{x}$')
ax.plot(data[:,0], data[:,2], '--',  label = r'$x^2$')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend(loc = 9)

plt.show()
```



Benchmarking

Benchmarking

Benchmarking

In the computation industry, benchmarking is an umbrella term for comparing the performance of different hardware, software solutions, program segments, algorithms, etc..

For hardware and software solutions, benchmarking is often used to draw comparisons with competing solutions. In developing a program or software solution benchmarking can also be used to make decisions, either to decide between two different solutions to the same problem, or to find which parts of the program need optimization (there's no use spending time to optimize a part of the program that has no impact on performance).

There are many aspects of performance that one can monitor. For example, when benchmarking software or code, one could monitor the memory use, execution time, etc.

In this course we will be focusing on simple benchmarks of our Python code. In particular we will be looking at the time taken for code segments to execute.

Measuring Time in Python

Timing With `time.perf_counter`

We will use the `time` module to perform our benchmarks. More specialized modules exist, but for this course we will keep benchmarking simple.

The `time` module gives access to the computers system clocks, as well as functions for converting time formats.

The documentation for `time` can be found here.

The function that is of particular importance to us is `perf_counter()` :

[2]:
```
from time import perf_counter
```

It uses the most precise system wide clock available to it to return a time in seconds. The starting point of the timer is arbitrary and system dependent, so only time differences are of use to us. When timing a block of code make sure to take the time directly before and after the block.

Worked Example

Let's say we want to compare the time it takes to perform the sum

$$\sum_{n=1}^{1000} \frac{(-1)^n}{n}$$

using a for loop to using NumPy array functions.

[5]:
```
#Timing the use of a loop
start_time = perf_counter()

s = 0
for n in range(1, 1001):
    s += (-1)**n/n

loop_time = perf_counter() - start_time

print('Using a loop:')
print('The value of the sum is:', s)
print('The time taken to compute the sum is:', loop_time)
```

```
Using a loop:
The value of the sum is: -0.6926474305598223
The time taken to compute the sum is: 0.0018592540000099689
```

[4]:
```python
#Timing the use of numpy functions
import numpy as np

start_time = perf_counter()

n_arr = np.arange(1, 1001)
s = np.sum( (-1)**n_arr/n_arr )

np_time = perf_counter() - start_time

print('Using NumPy:')
print('The value of the sum is:', s)
print('The time taken to compute the sum is:', np_time)
```

```
Using NumPy:
The value of the sum is: -0.6926474305598204
The time taken to compute the sum is: 0.001659153999753471
```

Note that the times for both the looping method and NumPy method are very similar. Every time the code is run the values also fluctuate wildly. This makes it difficult to compare the performance of these solutions.

As a solution to this problem, we can take many runs of the code block in question and quote the total time taken to execute all of them. This will limit the effects of the fluctuation on our measurement and it will also make it less likely for us to run into floating point errors (the times of individual code runs can be very small).

Something to keep in mind when running a benchmark is to limit the number of background processes you have running on your computer, in particular those who's resource requirements fluctuate.

Don't time what you don't intend to measure, only time the code you are interested in benchmarking. If you use a loop to repeat measurements (i.e. it isn't an essential part of the code you're testing) don't include the overhead from the loop in your timing. If you are using print function calls for debugging purposes, you should also exclude those from the timing (in the example above the print calls were performed **after** the time was taken).

Uncertainty

Measurement and Uncertainty

This section is a summary of some key points that are relevant to this course taken from the UCT Physics laboratory manual {% cite phy-gum %}.

Measurements

When taking a physical measurement of a quantity (the **measurand**), the value you acquire is only a best approximation of the "true value" of the quantity (if that even exists for the given measurement).

Many factors, from the precision of your apparatus to random or systematic noise, introduce some form of **uncertainty** to your measurand. That is a quantifiable measure of the interval in which the "true value" for your measurand may lie with knowledge of the probability of it lying in that interval.

Sources of Uncertainty in Computing

Seeing as we are working with computers, we don't have measurement apparatus to worry about. There are, however, other sources to consider.

Precision of Data Values

Floating point numbers can only approximately represent a real numbers and have a finite precision due to having finite memory available.

Although we generally won't have to worry about the precision of our floats in this course. Be careful when preforming numerical operations (especially division) that intermediate values don't become too small.

Background Processes

If you are using a personal computer with an operating system there are always going to be background processes being preformed. These processes (however light they may be) are competing with your programs for resources.

This is especially relevant when you are running benchmarks for your code (time it takes to complete execution, memory use, etc.).

This form of uncertainty can be quite noisy and thus treated as **Type A** uncertainty (see the section below).

Error or Approximations in Numerical Solutions

Every numerical solution or simulation contains some approximation. This leads to uncertainty in the results you've acquired. How you handle this uncertainty depends on the particular method in question.

Type A Uncertainty

Sometimes you encounter fluctuation in your measurements of a single measurand. If this dispersion is unavoidable, the recommended approach is to take a data set of repeated measurements.

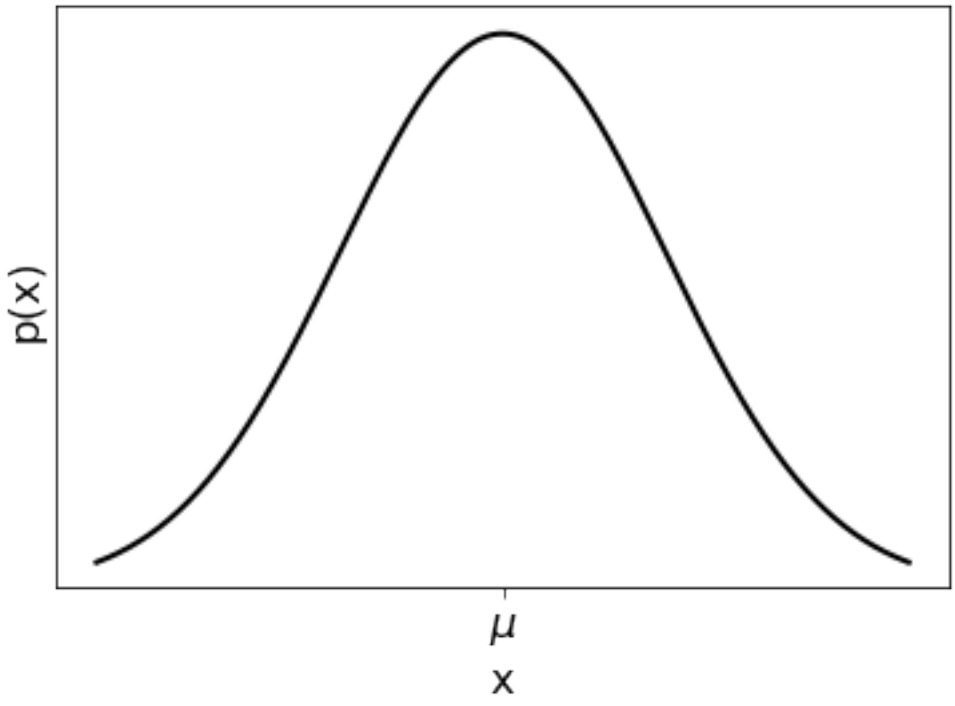
In many cases, given enough data points, the data will appear to follow a Gaussian probability distribution function (plot the data in a histogram if you're not sure).

Here we will denote the data set as x_i where $i = 1, 2, 3, \ldots, N$ (there are N data points).

The Gaussian probability distribution function is:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2 \right)$$

where exp is the natural exponential function, x is a possible value of a data point, μ is the first moment of the distribution and σ^2 is the second moment of the distribution.



As you can see illustrated in the figure above, the Gaussian peaks around $x = \mu$. We could say the most likely value of x lies in an interval around μ (continuous probability distribution functions represent probability densities).

For Gaussian distributed data, such as x_i, we can approximate μ using the arithmetic mean, $\bar{x}$:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

This value will be taken as our **best approximation** of the value of the measurand.

σ^2 can be approximated by the variance of the data:

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

The variance gives us a measure of the spread of the data (it is essentially the average square distance of each data point from the mean). Note the division by $N-1$ and not N.

Of more immediate use to us is the square root of the variance, or the **experimental standard deviation** $s(d)$:

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

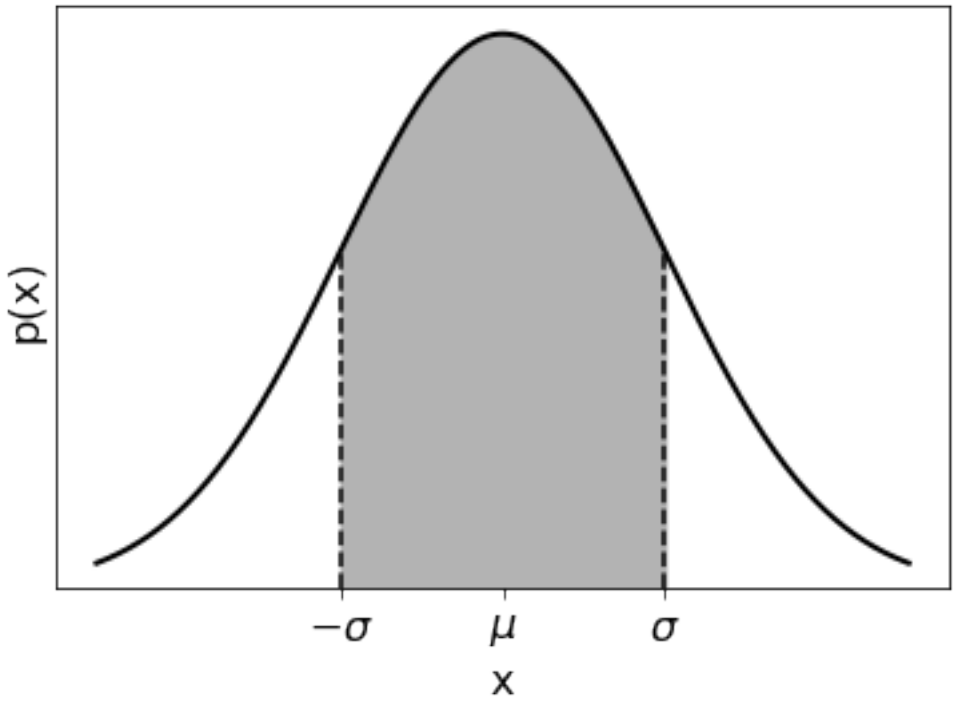
But how do we use $s(x)$, a measure of the spread of our data x_i, to determine the uncertainty of our best approximation of the value of the measurand $\bar{x}$?

Coverage Probability

Now we need to consider the uncertainty of this best approximation. The uncertainty will represent an interval around our best approximation of the value of the measurand in which we have some confidence that the "true value" lies.

We can quantify this interval by looking back at our pdf. If we take the area under $p(x)$ for a given interval that tells us the probability of finding x in that interval.

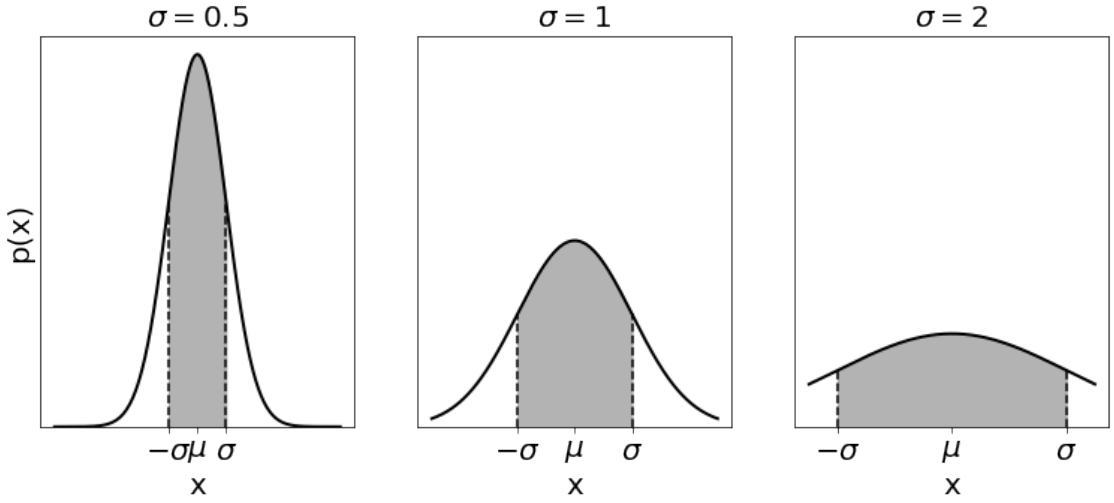
For example, taking an interval of $[\mu-\sigma, \mu+\sigma]$ (or $\mu \pm \sigma$) gives us a probability of 68% (or rather the value of the area is 0.68). That is the probability of finding any x value inside that interval.



This interval is often used for quoting the coverage probability of a measured variable x. Sometimes an interval of $\mu \pm 2\sigma$ is used, which gives us a coverage probability of 95% (though normally the narrower interval of 1σ is preferred).

Note that, for a Gaussian distribution, 100% coverage requires an infinitely large interval and is thus meaningless.

Remember that σ characterizes the width of the Gaussian, so this interval is also an indication of the spread of the measured data (illustrated below, each region has an area of 0.68 and the scale is maintained):



So how do we turn an interval with a coverage probability of 68% into an uncertainty?

Firstly, for our data set of measured x_i values, the value of our measurand is approximated as the arithmetic mean of the data, $\bar{x}$. We are, therefore, not interested in the spread of the x_i values, but rather the spread in possible values of $\bar{x}$.

If we want to quote the interval with a coverage probability of 68% (the standard uncertainty) for $\bar{x}$, we will need to calculate $s(\bar{x})$ (an approximation of σ for the distribution of possible $\bar{x}$ values). This can be calculated using the relationship:

$$s(\bar{x}) = \frac{s(x)}{\sqrt{N}}$$

This result is derived by taking samples of the data set and calculating the variance of the mean values of the samples. The derivation is not included in these notes.

In Summary

For a data set x_i where $i = 1, 2, 3, \ldots, N$ of measured values, if the data is Gaussian distributed, the **best approximation** for the value of the **measurand** is the **arithmetic mean**:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N}$$

The **standard uncertainty** for the mean value is given by the **experimental standard deviation of the mean**:

$$u(\bar{x}) = s(\bar{x}) = \frac{s(x)}{\sqrt{N}}$$

which gives us a 68% probability coverage.

$s(x)$ is the **experimental standard deviation** of the data:

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

(Note: when using `numpy.std()` to calculate $s(x)$, make sure to set the keyword argument `ddof = 1`, as the divisor is given by `N - ddof`. See the documentation for more.)

Quoting a Measurement With It's Uncertainty

You have a measurand Y (that is a quantity you wish to measure). You've found that the best approximation for the value of that measurand is y and you've determined an uncertainty of that value $u(y)$ with a coverage probability of P%.

When quoting the results of the measurement, make sure all of this data is present and clearly stated.

One way to quote your data is:
$Y = y \pm u(y)$ units ($P\%$ coverage probability)

or in a body of text, you can use something along the lines of:
"The best approximation of the measurand Y was found to be $y \pm u(y)$ units ($P\%$ coverage probability)"

Don't forget to quote your units!

Significant Figures

Your approximation of the value of the measurand will likely carry a lengthy decimal part (potentially up to floating point precision). You select the number of decimal places to quote by looking at your uncertainty.

You should generally quote your uncertainty within **two** significant figures and round your measurand value to match that. Significant figures are the first non-zero figures in a value.

For example, the measurement (with coverage probability omitted):

$L = 14.567354536267\ldots \pm 0.00346735838\ldots\text{cm}$

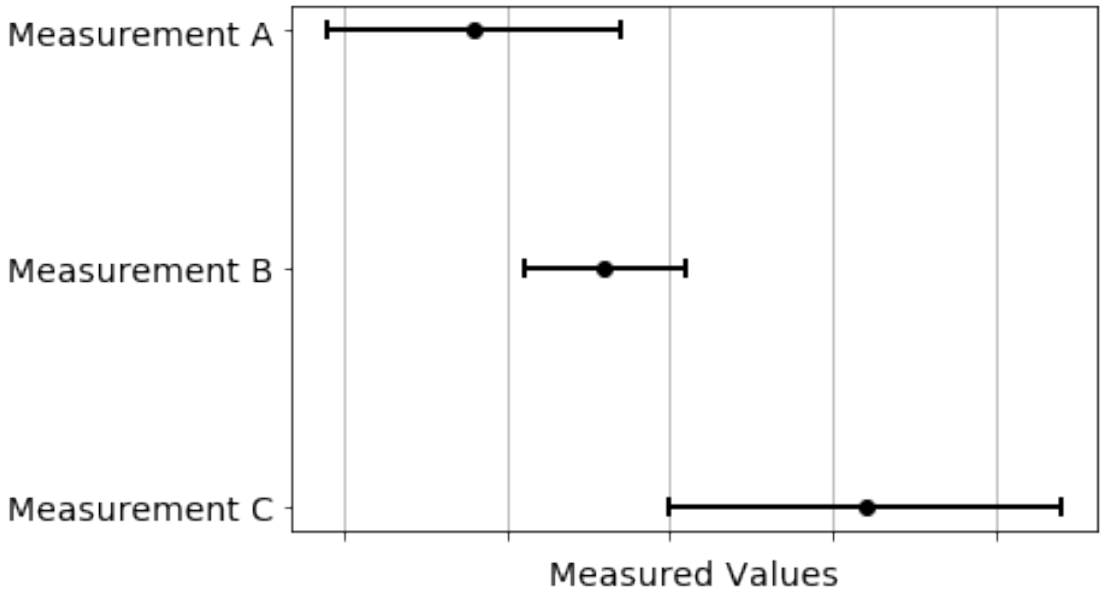
should be rounded to:

$L = 14.5674 \pm 0.0035\text{cm}$

Comparing Different Measurements

Now that we have a way of quoting measurements, let's discuss how to compare measurements.

Consider the figure below, where measurements A, B and C are compared. The best approximation of the value of the measurand is represented by the dots and the uncertainty intervals are represented by the capped lines.



In this case we can say that the results of measurements A and B **agree within their stated experimental uncertainties**, as do B and C, however B and C **do not agree** with each other.

References

{% bibliography –cited %}

NumPy

NumPy

Numpy

The NumPy package provides us with arrays and matrices (efficient data structures), sepcial functions, random number generators, and more.

The documentation for the SciPy, NumPy and many other scientific packages can be found here: https://www.scipy.org.

Importing NumPy

The standard way to import NumPy is using the alternative name `np`:

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

Arrays

Arrays

Arrays are one of NumPy's most important objects.

An array is a sequence of homogeneous data (each element must be the same data type). NumPy arrays use NumPy specific data types which are listed here.

Though we shall see that arrays can be indexed and sliced similarly to strings, tuples and lists, they behave differently under operations.

Arrays can have any number of dimensions. In this section we will only consider the 1 dimensional case.

Creating Arrays

Arrays can be created using the `np.array()` function with a list, tuple or another array as the argument:

[3]:
```
#Array of integers
np.array([1, 2, 3, 4])
```

[3]:
```
array([1, 2, 3, 4])
```

[4]:
```
#Array pf strings
np.array(('a', 'b', 'c'))
```

[4]:
```
array(['a', 'b', 'c'], dtype='<U1')
```

Remember that arrays are homogeneous:

[7]:
```
#Trying to create an array with different types
np.array([1, 2.3, 'x'])
```

[7]:
```
array(['1', '2.3', 'x'], dtype='<U32')
```

Indexing and Slicing

As said before, arrays can be indexed and sliced similarly to lists and strings

[15]:
```
letters = np.array(['a', 'b', 'c', 'd', 'e'])
print('Letters:', letters)
print('First character:', letters[0])
print('Second character:', letters[1])
print('Last character:', letters[-1])
print('Every second character:', letters[::2])
```

```
Letters: ['a' 'b' 'c' 'd' 'e']
First character: a
Second character: b
Last character: e
Every second character: ['a' 'c' 'e']
```

Mutable But Tricky To Resize

Similarly to lists, arrays are mutable (you can change the array after initializing it). For example, you can change an element of an array:

[9]:
```
arr = np.array((1, 2, 3 ,4))
print('Array:', arr)

print('')
print('Changing element 2')
print('')

arr[2] = 7
print('Array:', arr)
```

```
Array: [1 2 3 4]

Changing element 2

Array: [1 2 7 4]
```

However, unlike lists, it's not easy or efficient to alter the size of an array. It is still possible to resize (with `np.resize()`) and to concatenate (with `np.concatenate()`) arrays, but they don't have certain handy functions for lists like `.append()` and `.insert()`.

In general you should only create an array once you know how big it needs to be. If you need to add elements to an array, consider starting with a list and converting that to an array when you need the array properties.

Iterating Through Arrays

Like strings, tuples and lists, arrays are iterable:

[16]:
```
arr = np.array([1, 2, 3, 4])

for a in arr:
```

```
    print(a)
```

```
1
2
3
4
```

Vectorized Operations

One of the most useful properties of NumPy arrays is their vectorized operations. That is arithmetic operations between an array and array, and an array and scalar are performed element by element.

For example consider the scalar operations:

[10]:
```
2*np.array([1, 2, 3, 4])
```

[10]:
```
array([2, 4, 6, 8])
```

[12]:
```
np.array([1, 4, 5]) + 1
```

[12]:
```
array([2, 5, 6])
```

Array on array operations are also performed element by element:

[13]:
```
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([2, 4, 6, 8])

print(arr2, '-', arr1, 'is', arr2 - arr1)
print(arr2, '/', arr1, 'is', arr2/arr1)
```

```
[2 4 6 8] - [1 2 3 4] is [1 2 3 4]
[2 4 6 8] / [1 2 3 4] is [2. 2. 2. 2.]
```

These vectorized operations are far more efficient than iterating through the arrays and operating on each element individually, i.e.

[14]:
```
#More efficient:
print(arr1, '+', arr2, 'is', arr1 + arr2)
```

```
[1 2 3 4] + [2 4 6 8] is [ 3  6  9 12]
```

[16]:
```
#Less efficient
arr3 = np.array(4*[0])

for  i in range(4):
    arr3[i] = arr1[i] + arr2[i]

print(arr1, '+', arr2, 'is', arr3)
```

```
[1 2 3 4] + [2 4 6 8] is [ 3  6  9 12]
```

Creating Structured Arrays

Often we would like to create a large array with a particular structure. We could create these arrays from lists using list comprehension, but NumPy provides some useful built in functions to use instead.

`np.arange()`

This function is analogous to the `range()` function. It produces a series of values where you can specify the starting value, stopping value and the step size.

The syntax is:

`np.arange(start, stop, step)`

Similar to the `range()` function, you can use 1, 2 or 3 arguments:

[19]:
```
#1 argument: stop
np.arange(5)
```

[19]:
```
array([0, 1, 2, 3, 4])
```

[21]:
```
#2 arguments: start, stop
np.arange(1, 5)
```

[21]:
```
array([1, 2, 3, 4])
```

[22]:
```
#3 arguments: start, stop, step
np.arange(1, 10 ,2)
```

[22]:
```
array([1, 3, 5, 7, 9])
```

Unlike the `range()` function. `np.arange()` also allows for floating point values:

[24]:
```
np.arange(2.3, 3, 0.1)
```

[24]:
```
array([2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3. ])
```

`np.linspace()`

This function creates a series of evenly spaced values between a stopping and starting value. The number of items in the array can also be specified.

The syntax:

`np.linspace(start, stop, number)`

If `number` is not specified an array of length 50 is created.

[7]:
```
np.linspace(0, 1, 10)
```

[7]:
```
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

np.zeroes()

This function creates a uniform array of zeros. It takes the shape of the array you want to generate as an argument.

`np.zeros(shape)`

For a one dimensional array **shape** is just the size of the array:

[11]:
```
np.zeros(5)
```

[11]:
```
array([0., 0., 0., 0., 0.])
```

np.zeros() can be useful if you wish to create an array with a particular size, but will only be filling in the values later.

np.ones()

np.ones() is similar to **np.zeros()**, except it generates a uniform array of ones.

[13]:
```
np.ones(7)
```

[13]:
```
array([1., 1., 1., 1., 1., 1., 1.])
```

Note that, if you want a uniform array of a different value, you can either add that value to an array of zeros or multiply that value with an array of ones.

Array Methods and Attributes

Array Methods and Attributes

In this section we will look at some methods and attributes that arrays have. This is not a complete list, but rather highlighting things you may find useful.

Let's start off by creating a fairly large array, for example a collection of human height measurements:

[2]:
```
heights = np.array(
      [ 2.13159377,  1.8864508 ,  1.63504183,  1.71173878,  1.78826872,
        1.60621813,  1.74630706,  2.11123384,  1.54212979,  1.39184441,
        1.7919224 ,  1.80299245,  1.73770464,  1.95233673,  1.47179093,
        1.70506609,  1.41194434,  2.05643464,  1.8262583 ,  1.47764985,
        1.61362183,  1.65600316,  1.42078883,  1.78059602,  1.80600655,
        1.91634004,  1.82746488,  1.82688072,  1.82053352,  1.84882458,
        1.80672297,  1.4646136 ,  1.71033286,  1.83272236,  1.97074545,
        1.96265325,  1.39817665,  1.55933323,  1.59111903,  1.53108805,
        1.33635392,  1.74971951,  1.56885338,  1.6614742 ,  1.70868504,
        1.58476337,  1.69233894,  1.73520641,  1.71248418,  1.75484377])
```

To get the number of elements in an array, we can use the `size` attribute:

[12]:
```
print('The size of the heights array:', heights.size)
```

```
The size of the heights array: 50
```

For 1 dimensional arrays this is gives us the same value as using `len()`, but for multidimensional arrays, `len()` will not return the total number of elements.

Minimum and Maximum Values

You can use the `min()` and `max()` methods to get the minimum and maximum values of an array respectively.

[13]:
```
print('Minimum height:', heights.min())
print('Maximum height', heights.max())
```

```
Minimum height: 1.33635392
Maximum height 2.13159377
```

Again, this gives you similar results to the functions in the Standard Library, but is the only option for arrays of higher dimensions.

Statistical Functions

NumPy provides us with some basic statistical functions out of the box. For example the `mean()` (arithmetic mean or average) and `std()` (standard deviation).

[9]:
```
print('Average height: ', heights.mean())
print('Standard deviation of heights: ', heights.std())
```

```
Average height:  1.712684356
Standard deviation of heights:  0.18476698650385862
```

[8]:
```
print('Average height:', np.mean(heights))
print('Standard deviation of heights:', np.std(heights))
print('Maximum height:', np.max(heights))
print('Mimimum height:', np.min(heights))
```

```
Average height: 1.712684356
Standard deviation of heights: 0.18476698650385862
Maximum height: 2.13159377
Mimimum height: 1.33635392
```

2D Arrays and Matrices

[2]:
```
import numpy as np
```

2D Arrays and Matrices

NumPy arrays can have any number of dimensions, but in this course we will only go up to 2. 2D arrays are quite common if you are working with images or running certain simulations of 3D systems.

You can created 2D arrays from a nested sequence using the `np.array()` function:

[3]:
```
print(
np.array(
    [[1, 22, 45, 6, 3, 2],
      [34, 2, 56, 2, 7, 2],
      [2, 35, 64, 11, 1, 5]]
))
```

```
[[ 1 22 45  6  3  2]
 [34  2 56  2  7  2]
 [ 2 35 64 11  1  5]]
```

When you are doing this, make sure that your dimensions are correct, otherwise you will end up with an array of sequences:

[4]:
```
print(np.array([[1, 2, 3], [4, 5]]))
```

```
[list([1, 2, 3]) list([4, 5])]
```

Shape and Size

Now would be a good time to talk about the distinction between the `shape` and `size` attributes of an array.

[5]:
```
arr = np.array(
    [[0, 1],
     [0, 1],
     [0, 1]] )
```

The `size` of the array is a count of how many elements the array contains.

[6]:
```
arr.size
```

[6]:
```
6
```

The `shape` of an array is a tuple which tells you the length of each axis:

[7]:
```
arr.shape
```

[7]:
```
(3, 2)
```

Note that **axis 0** (the first value in the tuple) corresponds to the "rows" and **axis 1** (the second value in the tuple) corresponds to the "columns" of the 2D array (this makes more sense when thinking about matrices).

Generating 2D Arrays

You can also generate 2D arrays quickly by using the `np.ones()` and `np.zeros()` functions by specifying the shape the array instead of the size:

[8]:
```
np.ones( (3, 6) )
```

[8]:
```
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

[9]:
```
np.zeros( (5, 2) )
```

[9]:
```
array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]])
```

Remember that the shape is a **tuple**. It is a common mistake to enter each axis as a separate argument.

Indexing and Slicing

To index a multidimensional array you specify the index you want for each axis:

```
array[axis0_index, axis1_index, axis2_index, ... ]
```

Note the use of commas to separate each axis. For example, let's index the 2D array:

[10]:
```
arr = np.array(
    [[1,  2,  3,  4],
     [5,  6,  7,  8],
     [9, 10, 11, 12]]
)
```

[11]:
```
arr[1, 2]
```

[11]:
```
7
```

[12]:
```
arr[2, -1]
```

[12]:
```
12
```

You can slice multidimensional arrays by separating the slice along each axis by commas:

[13]:
```
arr[:, 1:3]
```

[13]:
```
array([[ 2,  3],
       [ 6,  7],
       [10, 11]])
```

You can extract individual rows and columns by slicing along one axis and indexing the other. For example:

[14]:
```
#Slicing the first row
arr[0, :]
```

[14]:
```
array([1, 2, 3, 4])
```

[18]:
```
#Slicing the third column
arr[:, 2]
```

[18]:
```
array([ 3,  7, 11])
```

[19]:
```
#Slicing the last column
arr[:, -1]
```

[19]:
```
array([ 4,  8, 12])
```

Transpose

You can use the `.T` attribute of an array (or matrix) to get the transpose (swap the rows and columns):

[20]:
```
arr.T
```

[20]:
```
array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]])
```

Matrices

NumPy's matrices are similar to 2D arrays, except for some matrix specific attributes, methods and operations.

You can create a matrix by using the `np.matrix()` function with a sequence argument:

[21]:
```
np.matrix(
    [[1, 2],
     [3, 4],
     [5, 6]]
)
```

[21]:
```
matrix([[1, 2],
        [3, 4],
        [5, 6]])
```

To generate large, structured matrices, you can use some of the array generating functions:

[22]:
```
np.matrix(np.ones( (2, 3) ))
```

[22]:
```
matrix([[1., 1., 1.],
        [1., 1., 1.]])
```

Slicing and Indexing

Slicing and indexing matrices is the same as for 2D arrays.

Matrix Operations

Consider the 2 by 3 matrix `mat1` and the 3 by 2 matrix `mat2`.

[23]:
```
mat1 = np.matrix(np.ones( (2, 3) ))
mat2 = np.matrix([[1, 2],
                  [3, 4],
                  [5, 6]])
```

Addition, **subtraction** and **division** between matrices are the same as for arrays (vectorized):

[24]:
```
mat1.T + mat2
```

[24]:
```
matrix([[2., 3.],
        [4., 5.],
        [6., 7.]])
```

[25]:
```
mat1.T/mat2
```

[25]:
```
matrix([[1.        , 0.5       ],
        [0.33333333, 0.25      ],
        [0.2       , 0.16666667]])
```

(`mat1` has been transposed to ensure the matrices shape's match)

Multiplication is matrix multiplication:

[26]:
```
mat1*mat2
```

[26]:
```
matrix([[ 9., 12.],
        [ 9., 12.]])
```

[27]:
```
mat2*mat1
```

[27]:
```
matrix([[ 3.,  3.,  3.],
        [ 7.,  7.,  7.],
        [11., 11., 11.]])
```

Inverse

You can use the `.I` attribute to get the multiplicative inverse of a matrix.

[28]:
```
mat = np.matrix(
    [[1, 0, 1],
     [0, 1, 0],
     [1, 0, 2]]
)
```

[29]:
```
mat.I
```

[29]:
```
matrix([[ 2.,  0., -1.],
        [ 0.,  1.,  0.],
        [-1.,  0.,  1.]])
```

[30]:
```
mat*mat.I
```

[30]:
```
matrix([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

[31]:
```
mat.I*mat
```

[31]:
```
matrix([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

Random Sampling

NumPy Random Module

The `numpy.random` module provides us with random number generators (RNG). You can find the documentation here. As there name suggests, random number generators produce random numbers. In this section we highlight a few essential functions from the module:

`np.random.random()`

This function produces random floating point numbers from a uniform probability distribution function (PDF) on the interval $[0, 1)$ (1 is excluded). If no arguments are provided a single number is generated:

[2]:
```
np.random.random()
```

[2]:
```
0.12055240311517734
```

If the length or shape is specified, `random()` returns an array of random numbers:

[4]:
```
np.random.random(5)
```

[4]:
```
array([0.08250099, 0.6587371 , 0.53175303, 0.67709712, 0.1558734 ])
```

[5]:
```
np.random.random((2, 3))
```

[5]:
```
array([[0.59302342, 0.84327141, 0.05504497],
       [0.98913576, 0.63069964, 0.73478334]])
```

If you want to produce uniformly distributed random numbers R on the interval $[a, b)$, you can use random numbers r from the interval $[0, 1)$ by scaling and shifting them:

$$R = a + r * (b - a)$$

For example, to generate uniform random numbers on the interval $[18, 30)$:

[6]:
```
np.random.random(4)*(30 -18) + 18
```

[6]:
```
array([28.90915681, 26.98028702, 29.02897959, 20.18055287])
```

To read more about `numpy.random.random()`, see the documentation.

`np.random.randint()`

This function produces random integers sampled from a uniform probability distribution on a **specified** interval.

The interval is defined by the first 2 arguments of `randint()`, the end of the interval (second number) is not included in the interval:

[10]:
```
#Random numbers from 1 up to 10
np.random.randint(1, 10)
```

[10]:
```
9
```

Again, you can specify a size or shape of the output array:

[11]:
```
np.random.randint(1, 10, 3)
```

[11]:
```
array([7, 9, 7])
```

[9]:
```
np.random.randint(1, 10, (2, 4))
```

[9]:
```
array([[5, 8, 7, 1],
       [9, 4, 9, 7]])
```

Random Numbers From Other Distributions

`numpy.random` provides us with many more RNG functions that sample from many of the most popular PDFs. You can see the full list in the documentation.

For example, the `np.random.norm()` function produces random numbers sampled from the normal (Gaussian) distribution. Parameters like the mean and standard deviation (or first 2 moments) can be specified.

All of these functions can generate array outputs

Array Conditional Statements and numpy.where()

Array Conditional Statements and `numpy.where()`

Comparison and Bitwise Operations on Arrays

We can apply comparison operators to arrays:

[51]:
```
a1 = np.array([1,  2, 3, 4, 5])

a2 = np.array([2, 1, 5, 6, 4])

a1 < a2
```

[51]:
```
array([ True, False,  True,  True, False])
```

As you can see this gives us an array of booleans, each element representing the outcome of comparing the corresponding element of `a1` to `a2`.

What if we wanted to combine the boolean arrays with a logical operator? For example, if we want an array of booleans for the condition `a1` is less-than `a2` and greater than `2`. Unfortunately the boolean comparison operators we used in the If Statements won't work, for example using `and`:

[53]:
```
a1 < a2 and a1 > 2
```

```
␣
↪---------------------------------------------------------------------------

ValueError                                Traceback (most recent call␣
↪last)

<ipython-input-53-74cb81b02f02> in <module>
----> 1 a1 < a2 and a1 > 2


ValueError: The truth value of an array with more than one element is␣
↪ambiguous. Use a.any() or a.all()
```

In order to combine boolean arrays (without a loop) we need to use **bitwise** operators.

Bitwise operators treat numbers as a string of bits and act on them bit by bit. In the case of a boolean array, the operator acts on it element by element. The bitwise operators we are interested are:

Operator

Name

Analogous boolean operator

&

Bitwise and

and

|

Bitwise or

or

~

Bitwise complement

not

(See https://wiki.python.org/moin/BitwiseOperators for a more comprehensive list and explanation of bitwise operations.)

Returning to our original example:

[55]:
```
(a1 < a2) & (a1 > 2)
```

[55]:
```
array([False, False,  True,  True, False])
```

Note that the comparisons must be grouped in brackets for this to work:

[56]:
```
a1 < a2 & a1 > 2
```

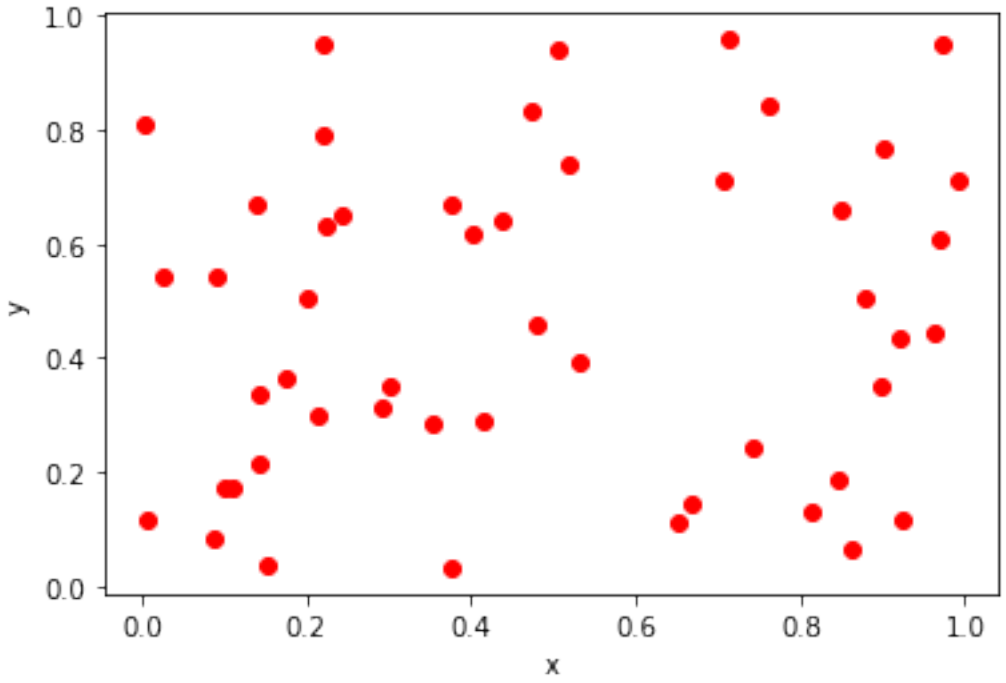
```
␣
↪---------------------------------------------------------------------------

ValueError                                Traceback (most recent call␣
↪last)

<ipython-input-56-c3606bc24b97> in <module>
----> 1 a1 < a2 & a1 > 2


ValueError: The truth value of an array with more than one element is␣
↪ambiguous. Use a.any() or a.all()
```

Example - Random Points in a Region

We can use `np.where()` to check which points in an array lie inside or outside of region.

First let's generate an array of 50 random points in 2D space:

[46]:
```
points = np.random.random((2, 50))

plt.plot(points[0, :], points[1, :], 'ro')

plt.xlabel('x')
plt.ylabel('y')

plt.show()
```



Note that axis 0 of `points` is used to represent the x and y values, and axis 1 represents points. i.e. for the points $(x_0, y_0), (x_1, y_1), \ldots, (x_{49}, y_{49})$, `points` is:

x0

x1

x2

x3

x4

...

x48

x49

y0

y1

y2

y3

y4

...

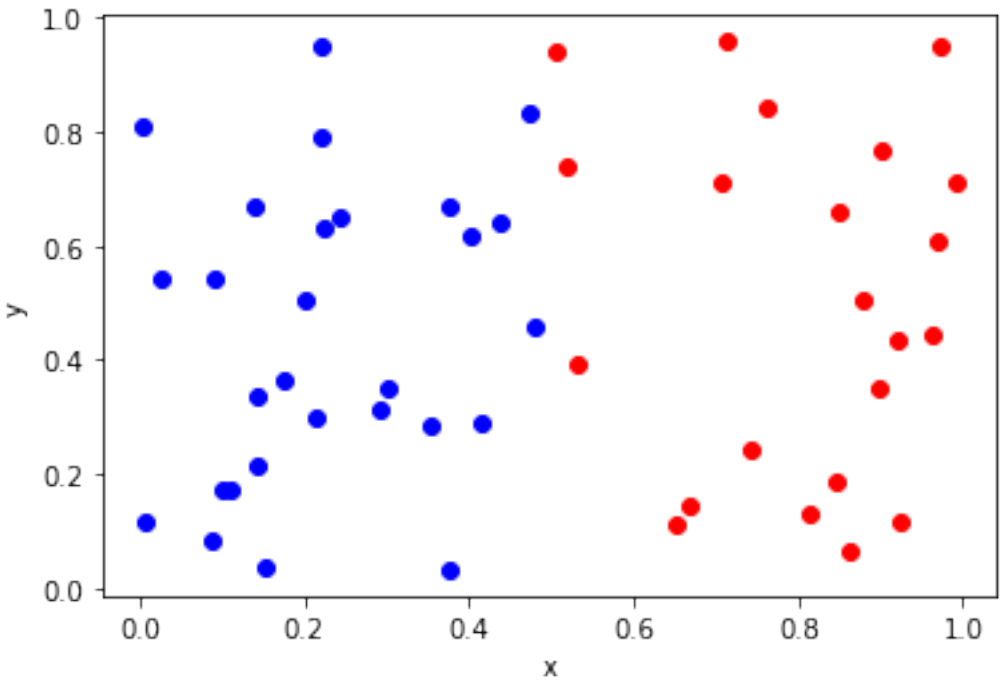
y48

y49

Now, let's plot the points which lie to the left of 0.5 as blue and the others as red:

[47]:
```
is_left = points[0, :] < 0.5 #True where left of 0.5

plt.plot(points[0, is_left], points[1, is_left], 'bo')
plt.plot(points[0, ~ (is_left)], points[1, ~ is_left], 'ro')

plt.xlabel('x')
plt.ylabel('y')
```

[47]:
```
Text(0, 0.5, 'y')
```

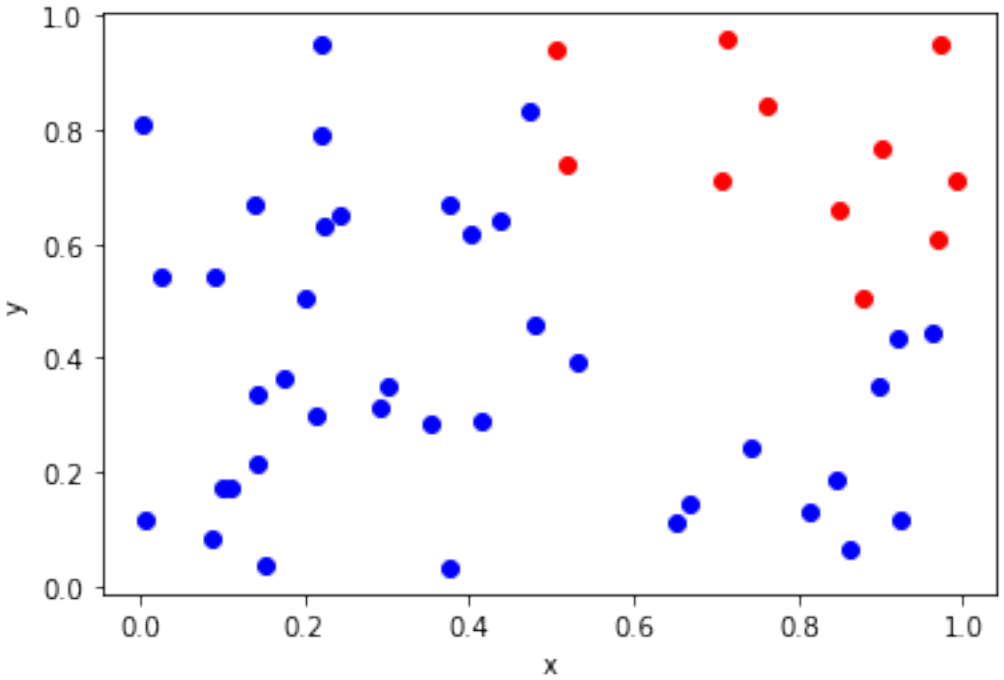


Note that, in the example above, we have used an array of booleans to **slice the elements of the array which are true**. We have also use the **bitwise compliment** to get the complement of our comparison result, there is no need to recalculate it.

Now, lets plot the points right of 0.5 and above 0.5 (the top left square) as red and the rest as blue (remember the **bitwise and**):

[48]:
```
#True if in top left square
is_top_left = (points[0, :] > 0.5) & (points[1, :] > 0.5)

plt.plot(points[0, is_top_left], points[1, is_top_left], 'ro')
plt.plot(points[0, ~ (is_top_left)], points[1, ~ is_top_left], 'bo')

plt.xlabel('x')
plt.ylabel('y')
```

[48]:
```
Text(0, 0.5, 'y')
```



numpy.where()

Documentation

```
numpy.where(condition[,x, y])
```

Returns elements chosen from `x` or `y` depending on the condition. If no `x` or `y` arguments are provided it returns and array of indices.

[17]:
```
arr = np.arange(10, 20)

arr_where = np.where(arr > 15)

print('arr1:', arr)
print('Indices where arr1 is greater than 15:', arr_where)
print('The sub-array of arr1 that is greater than 15:', arr[arr1_where])
```

```
arr1: [10 11 12 13 14 15 16 17 18 19]
Indices where arr1 is greater than 15: (array([6, 7, 8, 9]),)
The sub-array of arr1 that is greater than 15: [16 17 18 19]
```

If both `x` and `y` is specified, the elements of the returned array come from `x` if `condition` is true, or from `y` if `condition` is false.

[22]:
```
x = np.linspace(1, 5, 5)
#y = np.linspace(-5, -1, 5)
y = -x

condition = [True, False, True, True, False]

print('x:', x)
print('y:', y)
print('Condition:', condition)
print('x where True, y where False:', np.where(condition, x, y))
```

```
x: [1. 2. 3. 4. 5.]
y: [-1. -2. -3. -4. -5.]
Condition: [True, False, True, True, False]
x where True, y where False: [ 1. -2.  3.  4. -5.]
```

Example - Piecewise defined functions

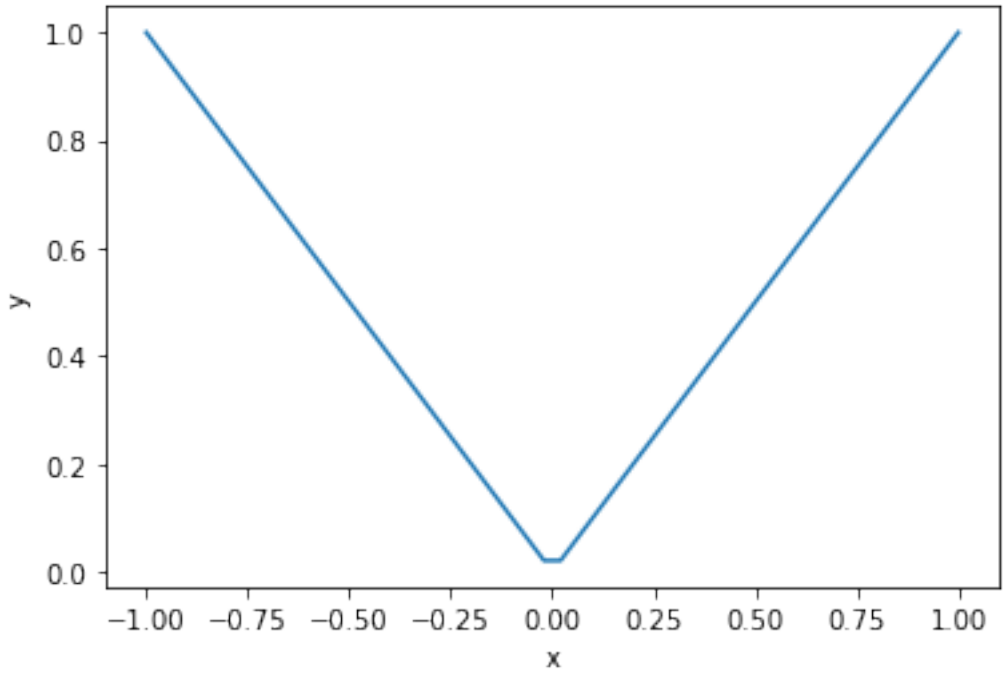
One use for `np.where()` is to define a piecewise defined function that works on arrays.

As a first example, let's use `np.where()` to plot the absolute value function (you should really use `np.abs()` for this):

$$y = \begin{cases} -x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

[24]:
```
x = np.linspace(-1, 1)

y = np.where(x >= 0, x, -x)
```

```
plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Note that, in the plot above, the line does not reach zero, but flattens out to a value above it. This is because the array `x` does not contain the value 0, but values around it.

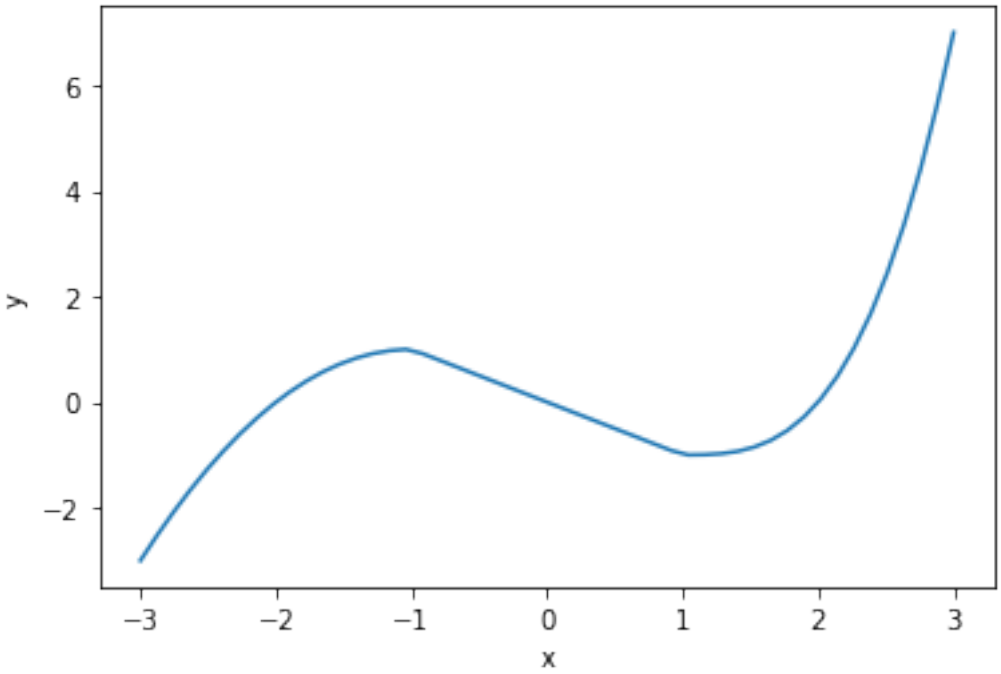
Now, consider the piecewise function:

$$f(x) = \begin{cases} -(x+1)^2 + 1 & \text{if } x < -1 \\ -x & \text{if } -1 \leq x \geq 1 \\ (x-1)^3 - 1 & \text{if } x > 1 \end{cases}$$

where there are three regions. To handle this we can use 2 `np.where()` calls:

[34]:
```
x = np.linspace(-3, 3)

#Left condition
y = np.where(x < -1, -(x+1)**2 + 1, -x)

#Right condition
y = np.where(x > 1, (x - 1)**3 - 1, y)

plt.plot(x, y)
```

```
plt.xlabel('x')
plt.ylabel('y')

plt.show()
```



Matplotlib

Matplotlib

Matplotlib

In this chapter we shall take a quick look at plotting with Matplotlib's Pyplot module. Matplotlib offers many plotting functions and plots have many features that can be tweaked. For these reasons we will only be scratching the surface of using Matplotlib. A good resource for finding out what is possible is the Matplotlib Thumbnail Gallery which features many example plots along with their source code. The matplotlib documentation can be found here.

Simple Plots With Pyplot

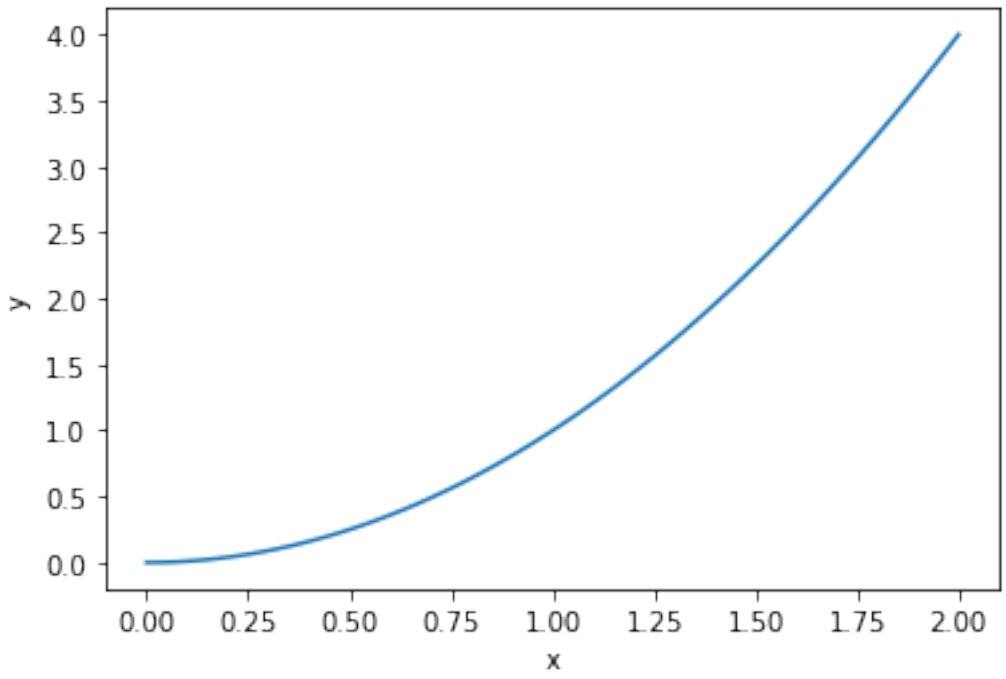
Simple Plots with Pyplot

The Pyplot module of Matplotlib acts as an interface to the Matplotlib package. This gives us access to a library of 2-dimensional plotting functions. The standard way of importing Pyplot is:

[1]:
```
import matplotlib.pyplot as plt
```

As a first example, let's plot the line $y = x^2$:

[6]:
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2, 100)
y = x*x

plt.plot(x, y) #Plots a line

plt.xlabel('x') #x-axis label
plt.ylabel('y')

plt.show() #Visualizes the plot
```



where:

- `plt.plot()` plots a straight line, which is one of the many types of plots available in the module (see the Thumbnail Gallery for more).
- The `plt.xlabel()` and 'plt.ylabel()' functions set the labels for the x and y-axis of the plot to the given arguments respectively.
- `plt.show()` shows the current figure (discussed in the following section). In the regular Python environment this function will pause the code and bring up a window containing the plot. Elements of the plot can be edited in this window and this plot can be saved. Closing the window resumes the script.

  In Jupyter Notebook, running `plt.show()` will display the plot in the cell output and will not pause the script.

Figures

A Matplotlib figure contains plot elements, for example a set of (or multiple sets of) axis, a title etc. Figures can be created using

```
fig = plt.figure()
```

When using `plt.plot()` Matplotlib will automatically add the plot to the last figure that was defined. Refer to the Subplots page for accessing the figure axis directly.

If you want to specify the dimensions of the plot, you can create a figure with the first positional or keyword argument:

`fig = plt.figure(figsize = (width, height) )`

where `figsize` (a 2-tuple of width and height) is in inches.

For more information on the figure class see the documentation.

Saving Figures

You can save figures using the

`plt.savefig(filename)`

function, where `filename` is the filename of the image to be saved. If a file extension is specified, the image will be saved using that type, the default type is a PNG.

This will save the current figure, if you want to save a particular figure then you can use `fig.savefig()`.

If you're not specifying the figure, make sure to save **before** you call `plt.show()` as this will clear the figure.

Line Color

You can specify the line color for the plot using either a positional (single letter) argument:

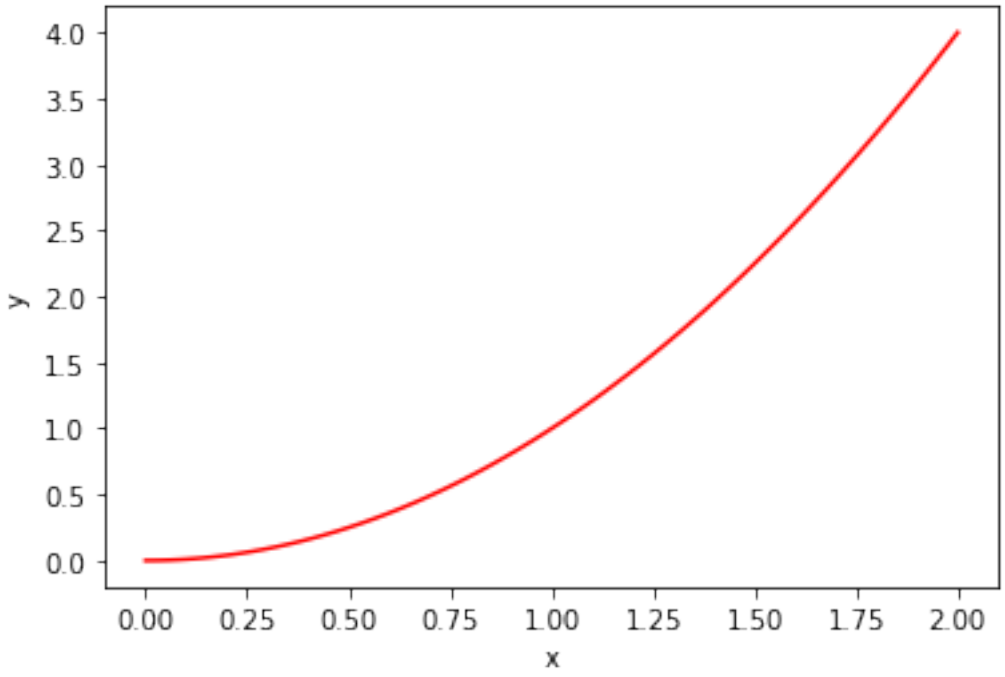
`plt.plot(x, y, 'r')`

or using a keyword argument:

`plt.plot(x, y, color = 'red')`

where the examples above both produce red lines, for example:

[7]:
```
x = np.linspace(0, 2, 100)
y = x*x

plt.plot(x, y, 'r')

plt.xlabel('x')
plt.ylabel('y')

plt.show()
```



The list of colors, as found in the Matplotlib documentation, is:

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

Shades of gray can be given as a string representation of a float between 0 and 1, for example:

```
color = '0.75'
```

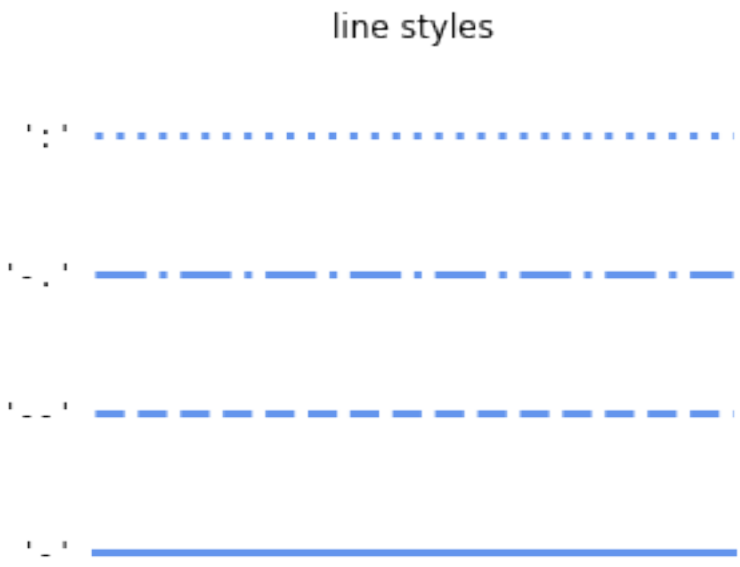
Line Style

Similar to the color of the plot, you can also set the line style, either as a positional argument:

or as a keyword argument:

Note that both the color and line style can be combined when set using the positional argument.

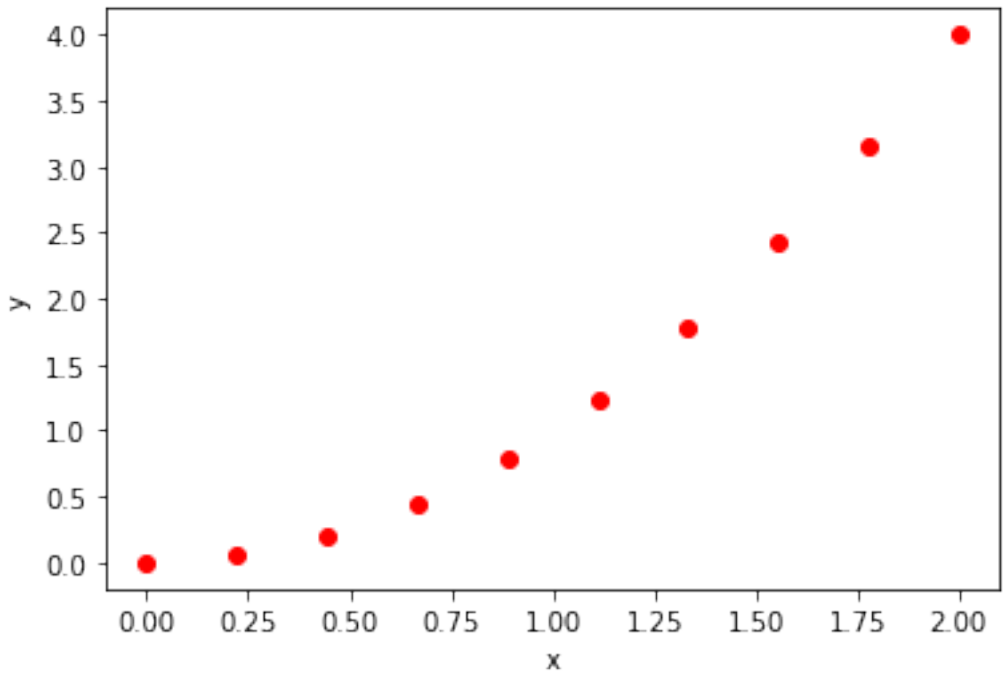
The reference for the lines given below is taken from the documentation:



Marker

In addition to line style and color, you can specify a marker. The markers are placed at each data point. The possible markers are listed in the documentation, as an example let's plot the data points as circles (`'o'` in the positional argument, or `marker = 'o'` as a keyword argument):

[9]:
```
x = np.linspace(0, 2, 10)
y = x*x

plt.plot(x, y, 'ro')

plt.xlabel('x')
plt.ylabel('y')

plt.show()
```

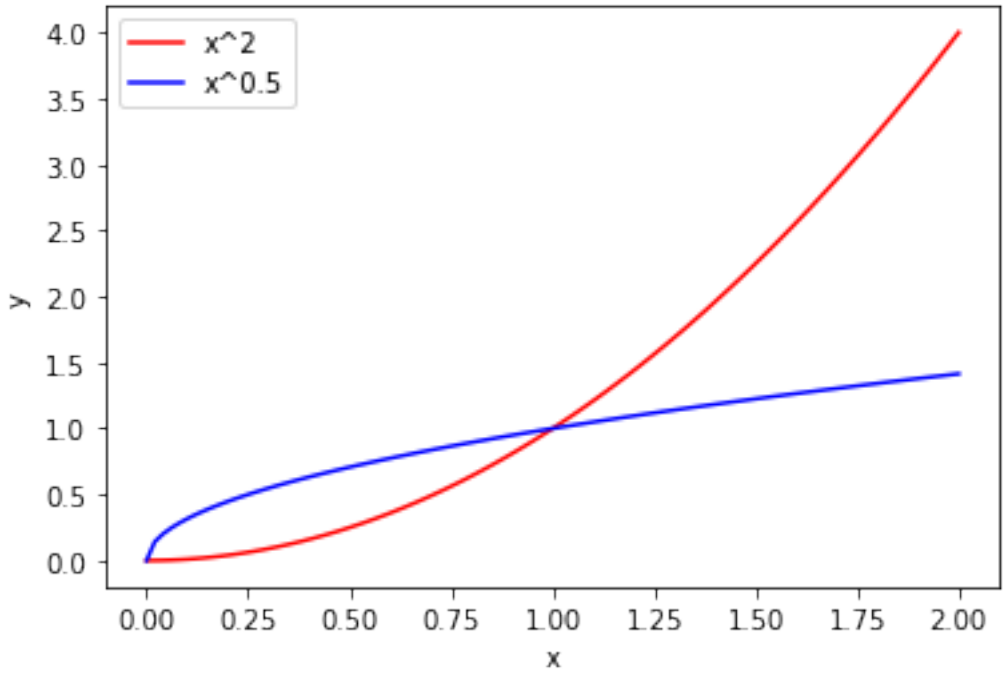


As you can see the line style is set to `'None'` by default if a marker is specified without a line style.

Legends

You can add a legend to your figure by labeling the plots with the keyword argument `label` and calling the `plt.legend()` function:

[10]:
```
x = np.linspace(0, 2, 100)

plt.plot(x, x*x, 'r', label = 'x^2')

plt.plot(x, np.sqrt(x), 'b', label = 'x^0.5')

plt.xlabel('x')
plt.ylabel('y')

plt.legend()

plt.show()
```


x^2
x^0.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
y
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
x

Subplots

Subplots

You can create subplots in two different ways:

`fig.add_subplot()`

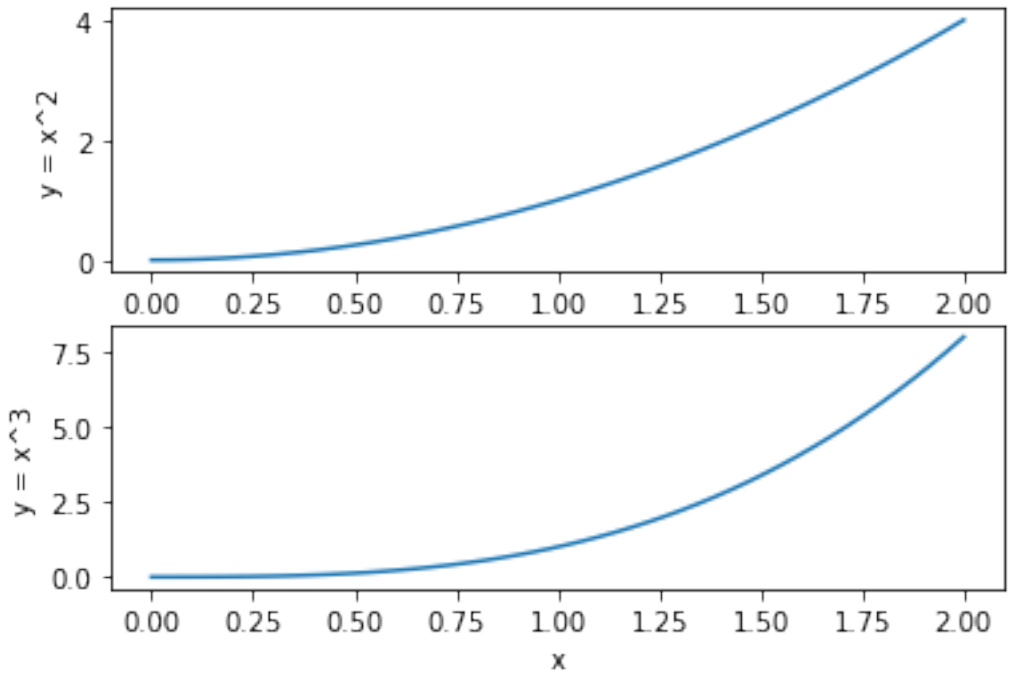
One way to add subplots is by creating a figure and calling the `fig.add_subplot()` method to add an axis to it with (one of) the call signature:

`fig.add_subplot(nrows, ncols, index)`

where `nrows` and `ncols` are the total number of rows and columns of axis and `index` is the position on the grid of axis.

Consider the plot with two rows and a single column:

[6]:
```
x = np.linspace(0, 2)


fig = plt.figure()

#Top axis
ax0 = fig.add_subplot(2, 1, 1)
ax0.plot(x , x**2)
ax0.set_xlabel('x') #Note `set_xlabel` instead of `xlabel`
ax0.set_ylabel('y = x^2')

#Bottom axis
ax1 = fig.add_subplot(2, 1, 2)
ax1.plot(x, x*x*x)
ax1.set_xlabel('x')
ax1.set_ylabel('y = x^3')

plt.show()
```



Refer to the documentation for additional options.

`plt.subplots()`

An alternative way to create subplots is to use the `plt.subplots()` function which returns the figure object and a tuple of axis. The call signature is:

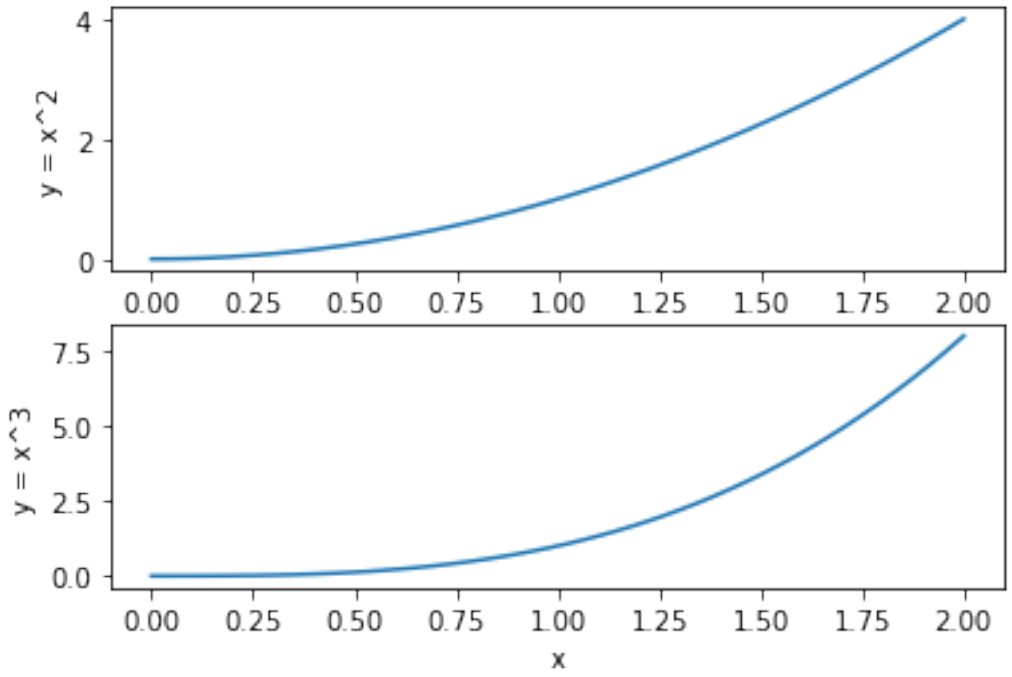
```
plt.subplots(nrows = 1, ncols = 1)
```

where `nrows` and `ncols` are the number of rows an columns as before.

Let's recreate the previous plot using this function:

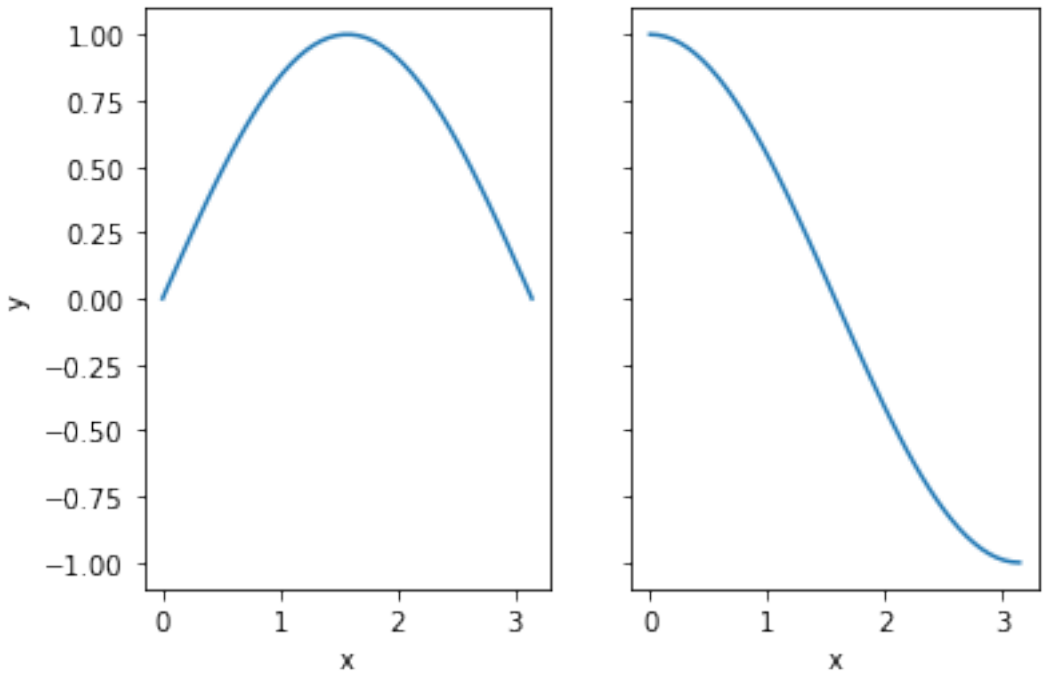
[7]:
```
x = np.linspace(0, 2)

fig, ax = plt.subplots(2, 1)

#Top axis
ax[0].plot(x , x**2)
ax[0].set_xlabel('x') #Note `set_xlabel` instead of `xlabel`
ax[0].set_ylabel('y = x^2')

#Bottom axis
ax[1].plot(x, x*x*x)
ax[1].set_xlabel('x')
ax[1].set_ylabel('y = x^3')
```

```
plt.show()
```



A couple of additional keyword arguments are `sharex` and `sharey`. These take boolean values. If true the subplots will share the relevant axis's ticks. For example:

[11]:
```
x = np.linspace(0, np.pi)

fig, ax = plt.subplots(1, 2, sharey = True)

ax[0].plot(x, np.sin(x))
ax[0].set_xlabel('x')

ax[1].plot(x, np.cos(x))
ax[1].set_xlabel('x')

ax[0].set_ylabel('y') #You can set this for the other axis

plt.show()
```

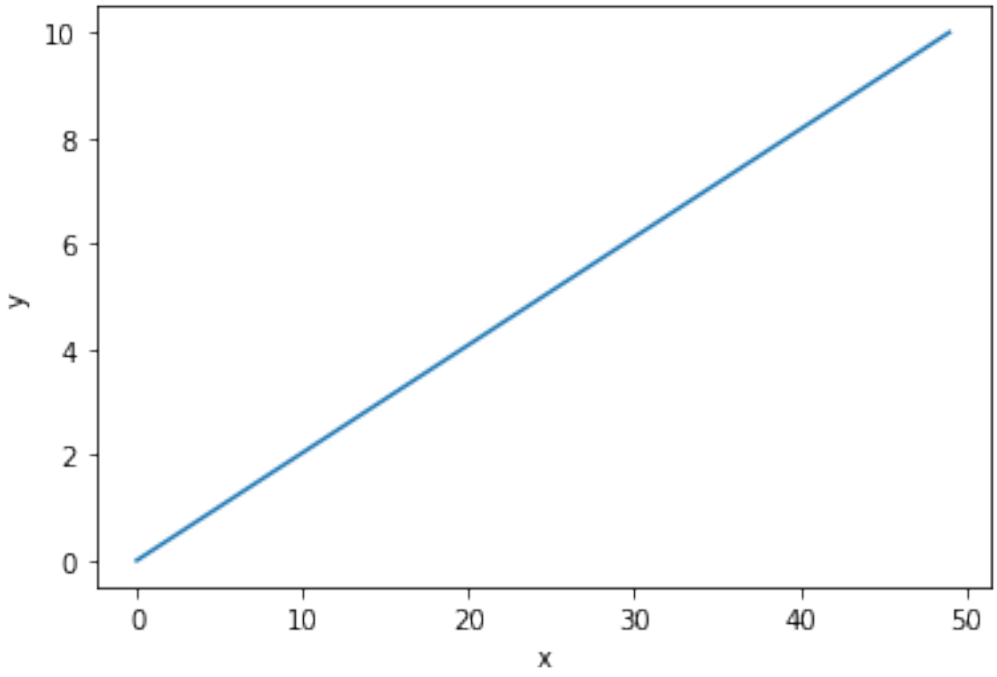


Refer to the documentation for additional options.

Using Subplots For General Plots

The subplot functions above are also used in general practice to create single axis plots, due to the ability to create a reference to the axis, which grants further customization. Simply:

[12]:
```
fig = plt.figure()
ax = fig.add_subplot()

ax.plot(np.linspace(0, 10))
ax.set_xlabel('x')
ax.set_ylabel('y')

plt.show()
```


10
8
6
4
2
0
y
0
10
20
30
40
50
x

Regression and SciPy

Regression and SciPy

Linear Regression Algorithms

Regression is used to fit models to data In this chapter we cover how to implement two linear regression algorithms. We then use SciPy to perform the similar algorithms, but for non-linear cases as well.

- Finding the relationship between a dependent variable and multiple independent variables.
- Uses
  - Curve fitting
  - Machine learning

Linear Least Squares Minimization

Linear Least Squares Minimization

The Problem

We propose a linear functional relation between 2 measurable variables, x and y:

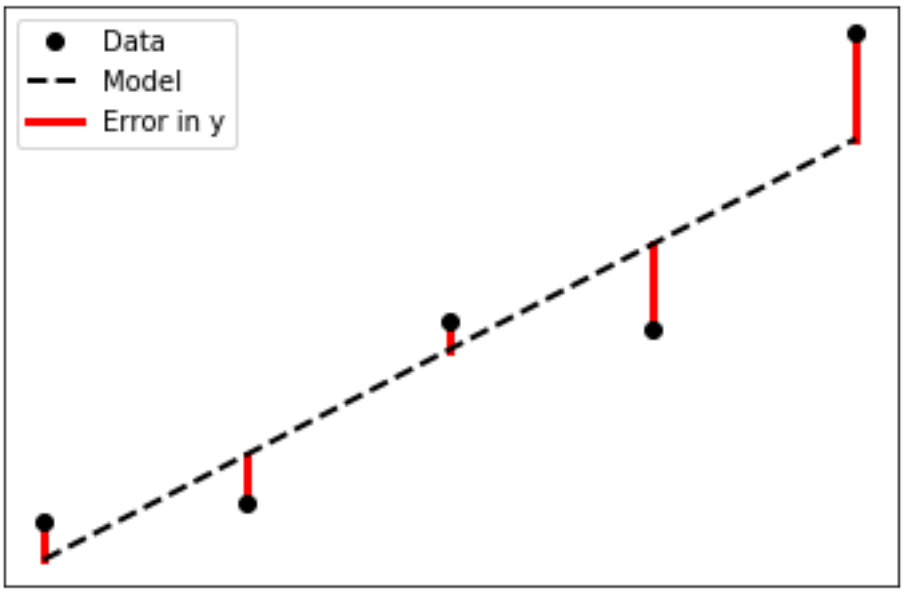
$$y = a_0 + a_1 x$$

where a_0 and a_1 are **unknown** constants. We wish to find these constants.

The Solution

To find these unknown coefficients in practice we measure many x, y pairs (assuming the measurements display some sort of dispersion). We now have a set of measured (x_i, y_i) pairs for $i = 1, 2, 3, \ldots, N$.

If we assume that the x_i are free of error, we can introduce error terms ϵ_i to the y_i data to make up for the dispersion of the data (i.e. that it doesn't follow the linear relation exactly).



With this error term, the relation between our data points can be represented as:

$y_i + \epsilon_i = a_0 + a_1 x_i$

Note that, at this point the error terms we have introduced are unknown to us. They represent the difference between the measured y_i values and the expected values if we plugged x_i into our relation (for which we have yet to determine a_0 and a_1). The error terms can be seen as a means to an end and will soon be done away with.

Now, we need some sort of metric to tell us how much error we have. We can use the sum of the errors squared for this:

$$S = \sum_{i=1}^{N} \epsilon_i^2$$

We use the squares of the error as it is the magnitude of the errors we are concerned about, and with the errors ranging between positive and negative values will end up canceling each other out (these are illustrated as points above and below the lines in the figure above).

We can use the relation between our data points to replace the ϵ_i^2:

$$S = \sum_{i=1}^{N} (a_0 + a_1 x_i - y_i)^2$$

Now, we want our choice of a_0 and a_1 to give us the least amount of error possible, or rather to give us the minimum value of S. To achieve this we minimize S with respect to a_0:

$$\begin{aligned} \frac{\partial S}{\partial a_0} &= 2 \sum_{i=1}^{n} (a_0 + a_1 x_i - y_i) = 0 \\ &= \\ a_0 + a_1 \langle x \rangle &= \langle y \rangle \end{aligned}$$

and a_1:

$$\begin{aligned} \frac{\partial S}{\partial a_1} &= 2 \sum_{i=1}^{n} (a_0 + a_1 x_i - y_i) x_i = 0 \\ &= \\ a_0 \langle x \rangle + a_1 \langle x^2 \rangle &= \langle xy \rangle \end{aligned}$$

To solve this system of equations we could use a matrix equation and let the computer determine the solution to that numerically, but with only two equations and unknowns, an analytic solution is easy enough to find:

$$a_1 = \frac{\langle xy \rangle - \langle x \rangle \langle y \rangle}{\langle x^2 \rangle - \langle x \rangle^2}$$
$$a_0 = \langle y \rangle - a_1 \langle x \rangle$$

Variance of y

If we assume that the y_i data points are distributed around the "true" y values for the given x_i by a Gaussian distribution with constant variance, we can calculate that variance as:

$$\begin{aligned}\sigma^2 &= \frac{1}{N}\sum_{i=1}^{N} \epsilon_i^2 \\ &= \frac{1}{N}\sum_{i=1}^{N} (a_0 + a_1 x_i - y_i)^2\end{aligned}$$

Worked Example - Cepheid Variables

For this worked example we will use data from Cepheid variables. These are pulsating stars with their luminosity (or magnitude M) related to the period (P) of their pulsations:

$$M = a_0 + a_1 \log P$$

Note that the relation above is can be made more accurate by including the color or temperature of the star, which we shall use later in the chapter.

As this relation is consistent across all specimens, these stars can be used as a standard candle for measuring distances, all that is needed are measurements from stars with known distances from Earth to determine a_0 and a_1.

The standard is to measure Cepheids in the Large Magellanic Cloud, whose distance is known. A few of these measurements can be found in the data file 'cepheid_data.csv' provided on Vula (Resources/Exercises/Data/Exercise10/) or on GitHub. The data file contains measurements of:

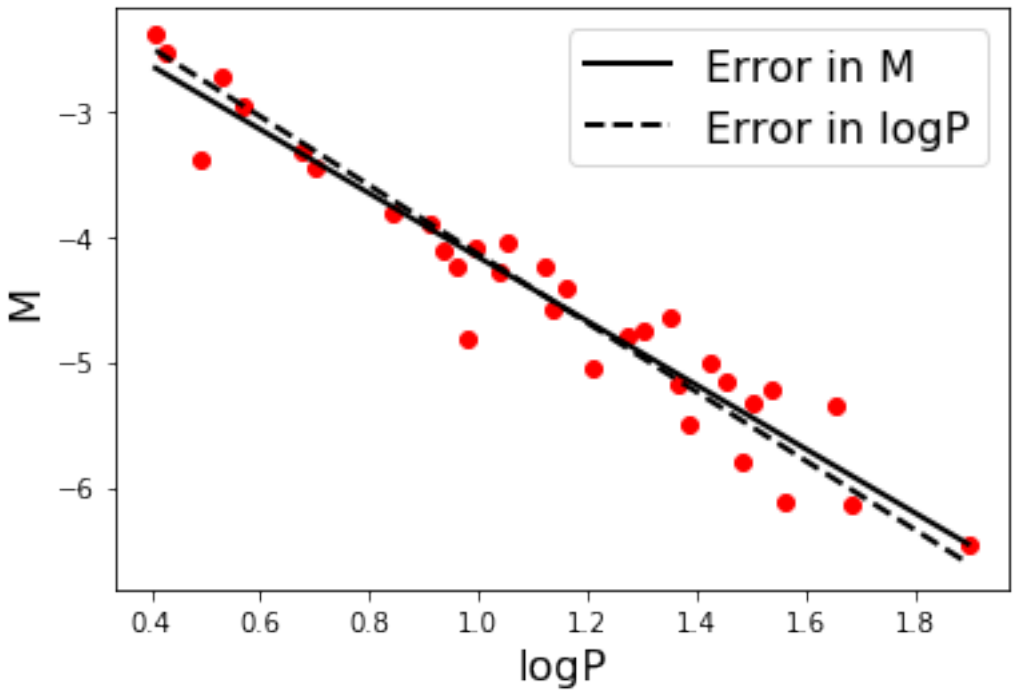
- $\log P$
- M
- $B - V$ (color, not using yet)

Solution

[13]:
```
def least_square(x, y):
    mean_x = np.mean(x)
    mean_y = np.mean(y)
    expect_xy = np.mean(x*y)
    expect_xx = np.mean(x*x)
```

```
    a1 = (expect_xy - mean_x*mean_y)/(expect_xx - mean_x*mean_x)

    return [mean_y - a1*mean_x, a1]

def get_sigma(a0, a1, x, y):
    return np.sqrt(np.mean((a0 + a1*x - y)**2))
```

[15]:
```
fontsize = 16
linewidth = 2


data = np.loadtxt('./data/cepheid_data.csv', delimiter = ',', skiprows = 1)

a0 , a1 = least_square(data[:,0], data[:,1]) # error in M
b0, b1 = least_square(data[:,1], data[:,0]) #error in logP

x = np.linspace(data[:,0].min(), data[:,0].max(), 2)

y_M = a0 + a1*x
y_P = -b0/b1 + x/b1

fig_ceph, ax = plt.subplots()

ax.plot(data[:,0], data[:,1], 'ro')
ax.plot(x, y_M, 'k', label = 'Error in M', lw = linewidth)
ax.plot(x, y_P, 'k--', label = 'Error in logP', lw = linewidth)

ax.set_xlabel('logP', fontsize = fontsize)
ax.set_ylabel('M', fontsize = fontsize)

ax.legend(fontsize = fontsize)

plt.show()
```


Error in M
Error in logP
M
logP
−3
−4
−5
−6
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8

Linear Chi Squared Minimization

Chi Squared Minimization

For the least squares minimization we assumed that one of the variables (y) contained error that accounted for the deviation of the data from the model we want to fit it to.

This error was not quantified by the measurement, furthermore we gave each error term equal importance in the total error to be minimized.

What if we had a measurement for the uncertainty of each of our y measurements? Let's characterize these uncertainties using the standard deviation of each y_i measurement: σ_i.

We now want to weight the contribution that each error value ϵ_i gives to the total error by the uncertainties σ_i. Ideally we want the model to fit within the uncertainties of the data points (or at least the fraction of the data points given by the confidence of the uncertainty). This means that we want to prioritize minimizing the error given by points with low uncertainty, or conversely we want to suppress the points with high uncertainty. To solve this we will minimize the χ^2 value of the data:

$$\chi^2 = \sum_{i=1}^{n} \left(\frac{\epsilon_i}{\sigma_i} \right)$$

where each error value is weighted by dividing it by the uncertainty. Note that if all of the σ_i where constant, we'd be dealing with least squares (the multiplicative factor will drop out in the minimization)

With 2 Variables

Returning to our scenario with two variables x and y, modeled by the functional relation:

$$y = a_0 + a_1 x$$

with a data set of measured x_i and y_i variables, with σ_i as the uncertainty of the y_i values for $i = 1, \ldots, N$, χ^2 can now be written as:

$$\chi^2 = \sum_{i=1}^{n} \left(\frac{\epsilon_i}{\sigma_i} \right)$$
$$= \sum_{i=1}^{n} \left(\frac{a_0 + a_1 x_i - y_i}{\sigma_i} \right)^2$$

Minimizing χ^2 with respect to a_0 and a_1, will yield:

$$a_0 = \left(\sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2} - \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2} \right) / D$$
$$a_1 = \left(\sum_{i=1}^{N} \frac{1}{\sigma_i^2} \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2} - \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} \right) / D$$

where

$$D = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \sum_{i=1}^{N} \frac{x^2}{\sigma_i^2} - \left(\sum_{i=1}^{N} \frac{x}{\sigma_i^2} \right)^2$$

Multivariate Linear Least Squares Minimization

Multivariate Linear Least Squares Minimization

In Linear Least Squares Minimization, we considered the linear functional relation between two measurable variables, x and y:

$$y = a_0 + a_1 x$$

where a_0 and a_1 are unknown conditions to be determined.

On this page we will look at the more generic case, where we solve the problem for an arbitrary number of variables and constants.

Three Variables

Let's start by solving this problem for three measurable variables: y, x_1 and x_2, in the linear functional relation:

$$y = a_0 + a_1 x_1 + a_2 x_2$$

where a_0, a_1 and a_2 are unknown coefficients.

Consider a data set of measured (x_{1i}, x_{2i}, y_i) pairs for $i = 1, 2, 3, \ldots, N$. If we attribute the dispersion of this data from the functional relation to error in the y_i terms, ϵ_i, then we can relate the data points with:

$$\begin{aligned} y_i + \epsilon_i &= a_0 + a_1 x_{1i} + a_2 x_{2i} \\ \therefore \epsilon_i &= a_0 + a_1 x_{1i} + a_2 x_{2i} - y_i \end{aligned}$$

The sum of errors squared is given by:

$$
\begin{aligned}
S &= \sum_{i=1}^{N} \epsilon_i^2 \\
&= \sum_{i=1}^{N} (a_0 + a_1 x_{1i} + a_2 x_{2i} - y_i)
\end{aligned}
$$

We want to minimize S with respect to each of the constants, a_0, a_1 and a_2:

$$
\frac{\partial S}{\partial a_0} = 2 \sum_{i=0}^{n} (a_0 + a_1 x_{1i} + a_2 x_{21} - y_i) = 0
$$

,

$$
\frac{\partial S}{\partial a_1} = 2 \sum_{i=0}^{n} (a_0 + a_1 x_{1i} + a_2 x_{21} - y_i) x_{1i} = 0
$$

and

$$
\frac{\partial S}{\partial a_2} = 2 \sum_{i=0}^{n} (a_0 + a_1 x_{1i} + a_2 x_{21} - y_i) x_{2i} = 0
$$

Re-arranging the above equations and using our statistical notation yields:

$$
a_0 + a_1 \langle x_1 \rangle + a_2 \langle x_2 \rangle = \langle y \rangle
$$

,

$$
a_0 \langle x_1 \rangle + a_1 \langle x_1^2 \rangle + a_2 \langle x_1 x_2 \rangle = \langle x_1 y \rangle
$$

and

$$
a_0 \langle x_2 \rangle + a_1 \langle x_1 x_2 \rangle + a_2 \langle x_2^2 \rangle = \langle x_2 y \rangle
$$

This time algebraic manipulation is a lot more work, instead we shall use a matrix equation (which will serve us better in the more generic case to come). The matrix equation representation is:

$$
\begin{pmatrix}
1 & \langle x_1 \rangle & \langle x_2 \rangle \\
\langle x_1 \rangle & \langle x_1^2 \rangle & \langle x_1 x_2 \rangle \\
\langle x_2 \rangle & \langle x_1 x_2 \rangle & \langle x_2^2 \rangle
\end{pmatrix}
\begin{pmatrix}
a_0 \\ a_1 \\ a_2
\end{pmatrix}
=
\begin{pmatrix}
\langle y \rangle \\ \langle x_1 y \rangle \\ \langle x_2 y \rangle
\end{pmatrix}
$$

This can easily be solved numerically using:

$$
\begin{aligned}
\boldsymbol{X}\boldsymbol{A} &= \boldsymbol{Y} \\
\therefore \boldsymbol{A} &= \boldsymbol{X}^{-1}\boldsymbol{Y}
\end{aligned}
$$

Example - Cepheid Variables

You now have all you need to find the unknown coefficients for the full functional relation of the magnitude (M), period (P) and color ($B-V$) of the Cepheid variables:

$$M = a_0 + a_1 \log P + a_2(B-V)$$

using the same data file as before. (You should find the values $a_0 = -2.15$ mag, $a_1 = -3.12$ mag and $a_2 = 1.49$)

Arbitrarily Many Variables

Consider a linear functional relation between measurable variables x_1, x_2, x_3, ..., x_m and y:

$$\begin{aligned} y &= a_0 + a_1 x_1 + a_2 x_2 + \cdots + a_m x_m \\ &= a_0 + \sum_{j=1}^{m} a_j x_j \end{aligned}$$

where a_0, a_1, ... and a_m are unknown constants.

Suppose we have a data set of measured $(x_{1i}, x_{2i}, \ldots, x_{mi}, y_i)$ values for $i = 1, 2, 3, \ldots, N$. As before, we assume that the dispersion in our data from the functional relation is due to error in the y_i data points only. Therefore we can write the relation between our data points as:

$$y_i + \epsilon_i = a_0 + \sum_{j=1}^{m} a_j x_{ji}$$

The sum of errors squared can thus be written as:

$$S = \sum_{i=1}^{N} \left(a_0 + \left(\sum_{j=1}^{m} a_j x_{ji} \right) - y_i \right)^2$$

We want to find the values of a_0, a_1, ... and a_m which gives us the minimum value of S. Minimizing S with respect to a_0 gives us:

$$\frac{\partial S}{\partial a_0} = 2 \sum_{i=1}^{N} \left(a_0 + \left(\sum_{j=1}^{m} a_j x_{ji} \right) - y_i \right) = 0$$

Distributing the sum over i amongst the terms:

$$\therefore N a_0 + \left(\sum_{j=1}^{m} a_j \sum_{i=1}^{N} x_{ji} \right) - \sum_{i=1}^{N} y_i = 0$$

Dividing by N:

$$\therefore a_0 + \left(\sum_{j=1}^{m} a_j \frac{1}{N} \sum_{i=1}^{N} x_{ji} \right) - \frac{1}{N} \sum_{i=1}^{N} y_i = 0$$

Using our stats notation:

$$\therefore a_0 + \sum_{j=1}^{m} a_j \langle x_j \rangle = \langle y \rangle$$

Now, let's minimize S with respect to one of the a_k for $k = 1, 2, \ldots, m$, following a similar line of algebraic manipulation as above:

$$\frac{\partial S}{\partial a_k} = \sum_{i=1}^{N} 2x_{ki} \left(a_0 + \left(\sum_{j=1}^{m} a_j x_{ji} \right) - y_i \right) = 0$$

$$\therefore a_0 \sum_{i=1}^{N} x_{ki} + \sum_{j=1}^{m} a_j \left(\sum_{i=1}^{N} x_{ki} x_{ji} \right) - \sum_{i=1}^{N} x_{ki} y_i = 0$$

$$\therefore a_0 \langle x_k \rangle + \sum_{j=1}^{m} a_j \langle x_k x_j \rangle = \langle x_k y \rangle$$

Writing the results for a_0 and a_k ($k = 1, \ldots, m$) into a system of equations, expanding the sum over j:

$$\begin{array}{ccccccc}
a_0 & +a_1\langle x_1 \rangle & +a_2\langle x_2 \rangle & +\ldots & +a_m\langle x_m \rangle & = \langle y \rangle \\
a_0\langle x_1 \rangle & +a_1\langle x_1{}^2 \rangle & +a_2\langle x_1 x_2 \rangle & +\cdots & +a_m\langle x_1 x_m \rangle & = \langle x_1 y \rangle \\
a_0\langle x_2 \rangle & +a_1\langle x_2 x_1 \rangle & +a_2\langle x_2{}^2 \rangle & +\cdots & +a_m\langle x_2 x_m \rangle & = \langle x_2 y \rangle \\
a_0\langle x_3 \rangle & +a_1\langle x_3 x_1 \rangle & +a_2\langle x_3 x_2 \rangle & +\cdots & +a_m\langle x_3 x_m \rangle & = \langle x_3 y \rangle \\
\vdots & + \quad \vdots & + \quad \vdots & +\ddots & + \quad \vdots & = \quad \vdots \\
a_0\langle x_m \rangle & +a_1\langle x_m x_1 \rangle & +a_2\langle x_m x_2 \rangle & +\cdots & +a_m\langle x_m{}^2 \rangle & = \langle x_m y \rangle
\end{array}$$

To solve these equations numerically, we can reformulate these equations into a matrix equation:

$$\begin{pmatrix}
1 & \langle x_1 \rangle & \langle x_2 \rangle & \cdots & \langle x_m \rangle \\
\langle x_1 \rangle & \langle x_1{}^2 \rangle & \langle x_1 x_2 \rangle & \cdots & \langle x_1 x_m \rangle \\
\langle x_2 \rangle & \langle x_2 x_1 \rangle & \langle x_2{}^2 \rangle & \cdots & \langle x_2 x_m \rangle \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\langle x_m \rangle & \langle x_m x_1 \rangle & \langle x_m x_2 \rangle & \cdots & \langle x_m{}^2 \rangle
\end{pmatrix}
\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}
=
\begin{pmatrix} \langle y \rangle \\ \langle x_1 y \rangle \\ \langle x_2 y \rangle \\ \vdots \\ \langle x_m y \rangle \end{pmatrix}$$

Notice that the left most matrix is symmetric about the diagonal, this can come in handy when computing the matrix elements. As before, this equation can be solved for the a_i by inverting the left most matrix, i.e.

$$\boldsymbol{X}\boldsymbol{A} = \boldsymbol{Y}$$
$$\therefore \boldsymbol{A} = \boldsymbol{X}^{-1}\boldsymbol{Y}$$

Python Implementation

Let's work on a Python implementation of this solution. You may want to try it yourself before reading further. In order to verify our implementation we will use the Cepheid data we've used so far, though in further exercises you will be given data sets containing more variables.

We start by reading in the file. We will read the data into a 2D array. This can be achieved using the standard library as in the **Data Files** section in the **File I/O** chapter, or using `numpy.loadtxt()` (documentation here). We shall use the latter as it is far more convenient:

[43]:
```
import numpy as np

## Reading in the file

data = np.loadtxt('data/cepheid_data.csv', delimiter = ',', skiprows = 1)
```

The `data` array contains all of the data points for $y_i, x_{1i}, x_{2i}, x_{3i}, \ldots, x_{ji}, \ldots, x_{mi}$, where $i = 1, \ldots, N$ corresponds to each row of `data`. Now, we want the data in the format:

$$\texttt{data} = \begin{bmatrix} [y_1, & x_{11}, & x_{21}, & \cdots, & x_{m1} &], \\ [y_2, & x_{12}, & x_{22}, & \cdots, & x_{m2} &], \\ [y_3, & x_{13}, & x_{23}, & \cdots, & x_{m3} &], \\ [\vdots, & \vdots, & \vdots, & \ddots, & \vdots &], \\ [y_N, & x_{1N}, & x_{2N}, & \cdots, & x_{mN} &] \end{bmatrix}$$

as this will make slicing it more clear. In the case of the Cepheid variable data, however, we have our "y" variable in the central column. Therefore we shall swap column 1 and 0 to better align with our desired data structure:

[44]:
```
# Swapping data[:, 0] and data[:, 1]
# Note that this is particular to the data file we are using
# np.copy is necessaary as arrays are not passed as values by default but as␣
↪reference

data[:, 0], data[:, 1] = np.copy(data[:, 1]), np.copy(data[:, 0])
```

To extract the values of a single variable for each measurement, slice columns out of `data`. For example, the y_i are contained in the slice `data[:, 0]`, the x_{1i} are contained in `data[:, 1]`, the x_{2i} are contained in `data[:, 2]`, etc.

Note that for each of the sums along the data sets ($\sum_{i=1}^{N}$), we will be summing along the columns. For example, for the quantity:

$$\langle x_1 \rangle = \frac{1}{N} \sum_{i=1}^{N} x_{1i}$$

[45]:
```
#Using numpy.mean to calculate the expectation value
#Note that x1 = data[:,1]

x1_mean = np.mean(data[:,1])
```

To calculate an expectation value like

$$\langle x_1 x_2 \rangle = \frac{1}{N} \sum_{i=1}^{N} x_{1i} x_{2i}$$

we can use:

[46]:
```
x1_x2_mean = np.mean( data[:,1] * data[:,2] )
```

where we've made use of NumPy array's vectorized operation to multiply each element together before taking the mean of the results.

Constructing the $\boldsymbol{X}$ Matrix Before we continue, let's break down the structure of the matrix:

$$\boldsymbol{X} = \begin{pmatrix} 1 & \langle x_1 \rangle & \langle x_2 \rangle & \langle x_3 \rangle & \cdots & \langle x_m \rangle \\ \langle x_1 \rangle & \langle {x_1}^2 \rangle & \langle x_1 x_2 \rangle & \langle x_1 x_3 \rangle & \cdots & \langle x_1 x_m \rangle \\ \langle x_2 \rangle & \langle x_2 x_1 \rangle & \langle {x_2}^2 \rangle & \langle x_2 x_3 \rangle & \cdots & \langle x_2 x_m \rangle \\ \langle x_3 \rangle & \langle x_3 x_1 \rangle & \langle x_3 x_2 \rangle & \langle {x_3}^2 \rangle & \cdots & \langle x_3 x_m \rangle \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \langle x_m \rangle & \langle x_m x_1 \rangle & \langle x_m x_2 \rangle & \langle x_m x_3 \rangle & \cdots & \langle {x_m}^2 \rangle \end{pmatrix}$$

Let's construct an empty matrix for which we will fill in the entries as we go:

[47]:
```
var_count = data.shape[1]

X = np.matrix(np.ones((var_count, var_count)))
```

Note that we have created an $(m+1) \times (m+1)$ matrix, where $m+1$ is given by the length of axis-1 of `data`.

Now, as we have noted before, $\boldsymbol{X}$ is a symmetric matrix. That is for for row k and column l, $\boldsymbol{X}_{kl} = \boldsymbol{X}_{lk}$. We only need to construct one of the triangles of the matrix, the other is obtained for free.

Let's work with the upper triangle of the matrix. Here there are 3 regions with distinguishable structures

1. The first row
2. The diagonal

3. The remaining triangle

The first element of the matrix is just one. The remainder of the first row is simply the expectation value of each of the x_j:

$$\boldsymbol{X}_{00} = 1$$

and

$$\boldsymbol{X}_{0l} = \langle x_l \rangle \quad \text{where } l = 1, 2, \ldots, m$$

Note that here we are indexing $\boldsymbol{X}$ from 0 to better translate it to code:

[48]:
```
# First row and column
# We leave the first element as is

for l in range(1, var_count):
    X[0, l] = np.mean(data[:, l])

    # Setting the values for the first column
    # remember that X[k, l] = X[l, k]
    X[l, 0] = X[0, l]
```

Now, consider the triangle off of the diagonal. That is the region:

$$\begin{pmatrix} - & - & - & - & \cdots & - \\ - & - & \langle x_1 x_2 \rangle & \langle x_1 x_3 \rangle & \cdots & \langle x_1 x_m \rangle \\ - & - & - & \langle x_2 x_3 \rangle & \cdots & \langle x_2 x_m \rangle \\ - & - & - & - & \cdots & \langle x_3 x_m \rangle \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ - & - & - & - & \cdots & \langle {x_m}^2 \rangle \end{pmatrix}$$

This region exhibits the pattern:

$$\boldsymbol{X}_{kl} = \langle x_k x_l \rangle \quad \text{where } l > k$$

The diagonal has a fairly simple pattern, starting from (row, column) $(1, 1)$:

$$\boldsymbol{X}_{kk} = \langle {x_k}^2 \rangle$$

Note, however, that this is a special case of the rules for constructing region 3. We can therefore combine regions 2 and 3 with the rule:

$$\boldsymbol{X}_{kl} = \langle x_k x_l \rangle \quad \text{where } l \geq k$$

In the code this becomes:

[49]:
```
# Inner matrix

for k in range(1, var_count):
    for l in range(k, var_count):
        X[k, l] = np.mean( data[:, k] * data[:, l] )

        #Setting the value for the lower triangle
        X[l, k] = X[k, l]
```

That covers the $\boldsymbol{X}$ matrix.

Constructing the $\boldsymbol{Y}$ Matrix Now let's construct the matrix:

$$\boldsymbol{Y} = \begin{pmatrix} \langle y \rangle \\ \langle x_1 y \rangle \\ \langle x_2 y \rangle \\ \vdots \\ \langle x_m y \rangle \end{pmatrix}$$

This is fairly straight forward, with

$$\boldsymbol{Y}_{0,0} = \langle y \rangle$$

and

$$\boldsymbol{Y}_{k,0} = \langle x_k y \rangle \quad \text{where } k = 1, \ldots, m$$

[50]:
```
#Creating the Y column matrix:
Y = np.matrix( np.zeros( (var_count, 1) ) )

#First entry
Y[0, 0] = np.mean(data[:,0])

#The remainder of the entries
for k in range(1, var_count):
    Y[k, 0] = np.mean( data[:, k] * data[:, 0] )
```

Finding Matrix $\boldsymbol{A}$ (Or Solving For the a_j) Lastly, to solve for our a_j values, we consider the matrix:

$$\boldsymbol{A} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}$$

This fits into the matrix equation

$$\boldsymbol{X}\boldsymbol{A} = \boldsymbol{Y}$$

where we've already constructed $\boldsymbol{X}$ and $\boldsymbol{Y}$. All that's left is to solve the equation be inverting $\boldsymbol{X}$:

$$\boldsymbol{A} = \boldsymbol{X}^{-1}\boldsymbol{Y}$$

To achieve this numerically, we simply take the inverse of `X`, `X.I`:

[51]:
```
#Finding A:

A = X.I*Y

print(A)
```

```
[[-2.14515885]
 [-3.11733284]
 [ 1.48566643]]
```

As you can see are results agree with the specific solution for the the case of 3 variables above.

Putting it all together: Let's gather all of the code cells together into a single script. We will also merge the loops together for efficiency:

[24]:
```
import numpy as np

#Reading the data
data = np.loadtxt('data/cepheid_data.csv', delimiter = ',', skiprows = 1)

# Swapping data[:, 0] and data[:, 1]
# Note that this is particular to the data file we are using
# np.copy is necessaary as arrays are not passed as values by default but as␣
 ↪reference
data[:, 0], data[:, 1] = np.copy(data[:, 1]), np.copy(data[:, 0])

#Creating empty X and Y matrices
var_count = data.shape[1]

X = np.matrix(np.ones( (var_count, var_count) ))
Y = np.matrix( np.zeros( (var_count, 1) ) )

#Filling the X and Y matrices

Y[0, 0] = np.mean(data[:,0])

for k in range(1, var_count):
    #First row and column of X
```

```
    X[0, k] = np.mean(data[:, k])
    X[k, 0] = X[0, k]

    #Y
    Y[k, 0] = np.mean( data[:, k] * data[:, 0] )

    #Inner matrix of X
    for l in range(k, var_count):
        X[k, l] = np.mean( data[:, k] * data[:, l] )
        X[l, k] = X[k, l]

#Calculating A

A = X.I*Y

print(A)
```

```
[[-2.14515885]
 [-3.11733284]
 [ 1.48566643]]
```

Nonlinear Least Squares Minimization with scipy.optimize.leastsq

Least Squares Minimization with `scipy.optimize.leastsq`

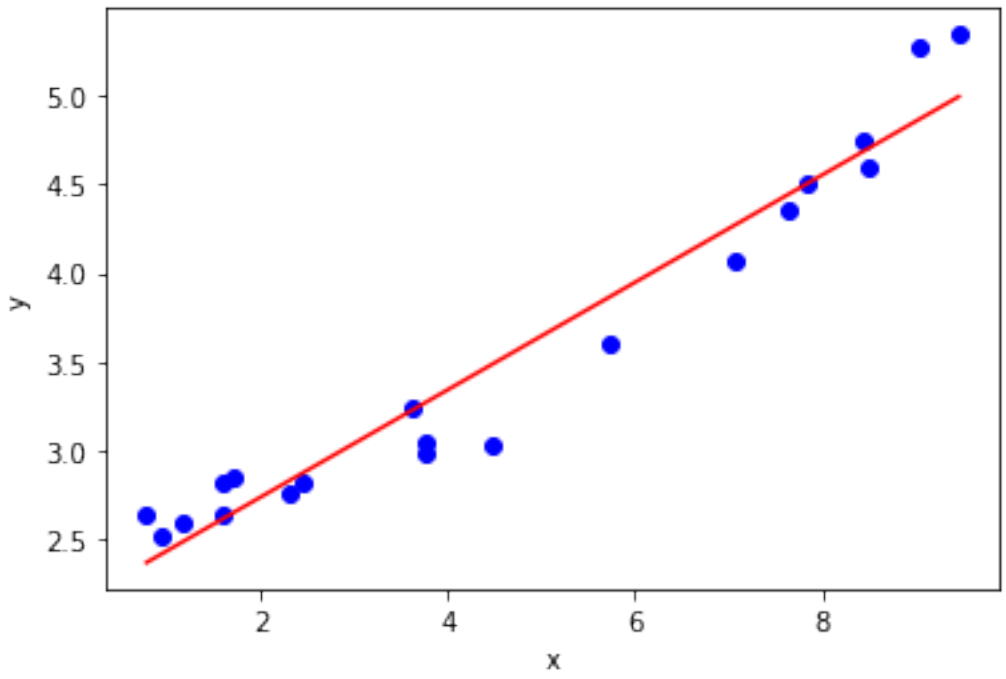
So far we have only considered functional relationships that are linear in the unknown constants. Non-linear cases are far more complicated and generally require numerical solutions. We will use a function from the SciPy module `scipy.optimize`, which contains functions for minimization, least squares and root finding techniques.

An Example of a Nonlinear Model

Consider the data found in the data file **nonlinear_data.csv** on GitHub , plotted below:

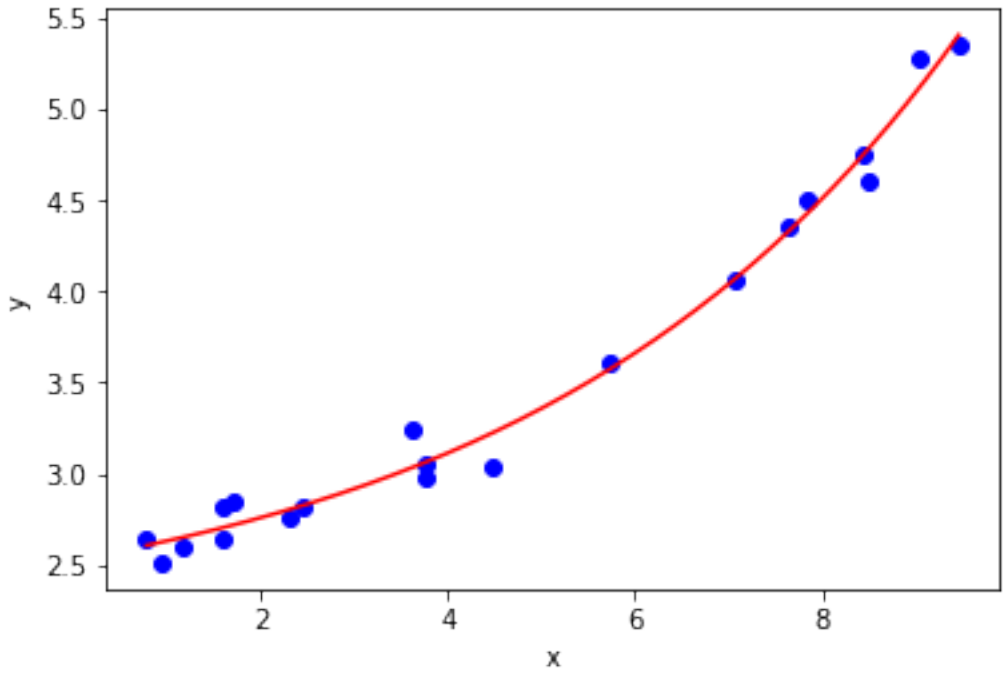
Though the data may appear to follow a linear trend, this is not the case. Consider the linear fit below:

[10]:
```
from scipy.optimize import leastsq
import numpy as np
import matplotlib.pyplot as plt

#Remove input

def lin_f(a,x):
    return a[0] + a[1]*x

def lin_err(a, x, y):
    return lin_f(a, x) - y

a0 = [1.5, 0.2]

a, success = leastsq(lin_err, a0, args = (xdata, ydata))

#Plotting the fit and data
x = np.linspace(xdata.min(), xdata.max(), 1000)

plt.plot(xdata, ydata, 'bo')
plt.plot(x, lin_f(a, x), 'r-')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Instead a more appropriate functional relation is an exponential function:

$$y = a_0 + a_1 e^{a_2 x}$$



Note that this functional relation is non-linear in a_2. Applying the method of least squares minimization to this functional relation will not yield an analytic solution, therefore a numerical method is required. We shall not be implementing this numerical method ourselves, instead using a function from **SciPy** to solve our problem. In short the numerical minimization technique involves following the negative gradient (or an approximation of this) from a given starting point, until a local minimum is found (essentially the solution is captured here).

Nonlinear Least Squares Minimization with `scipy.optimize.leastsq`

The SciPy module `scipy.optimize` contains functions for minimization, least squares and root finding techniques. Of particular interest to us now is the `leastsq` function (documentation here), which we shall use to perform nonlinear least squares minimization.

The call signature of `leastsq`, including only the arguments of immediate interest to us, is:

```
leastsq(func, x0, args = () )
```

The `x0` argument is an initial guess for the unknown parameters we are trying to find (required by the numerical minimization technique). In our case this is an initial case of the a_j constants.

The keyword argument `args` is a tuple of the variables or data we are fitting the model to. In our case x and y. The order in which these variables are presented is up to you, but must correspond to the order they are used in `fun`. Each element of this tuple should be an array or list of data points, for instance `(xdata, ydata)`.

The `func` argument is a callable object (function). It is referred to as the residual. It is the sum of the residuals squared that will be minimized.For the sum of errors squared the residual is equivalent to our error terms (ϵ_i).

$$S^2 = \sum_{i=1}^{N} \text{func(arguments)}^2$$

The call signature of `func` is:

```
func(params, *args)
```

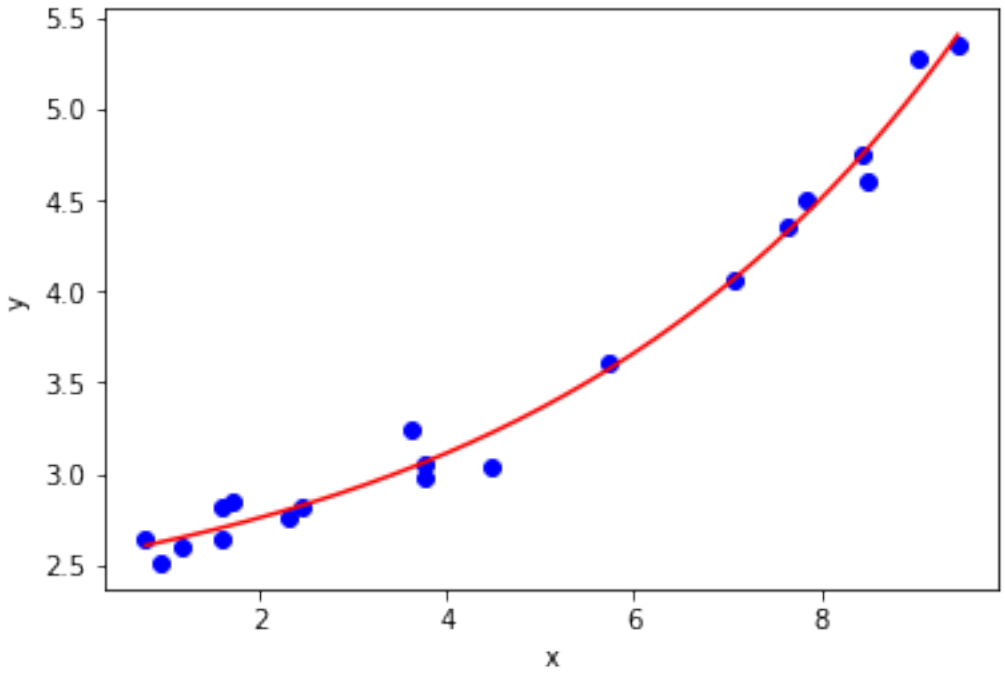
where `params` is a list or array of the parameters we are trying to find (a_j), and `args` is the tuple of the data for our variables (x and y).

The return value of the `leastsq` function (if it is only given the arguments listed above) is a tuple containing the solution for the a_j and an integer flag (for which a value between 1 and 4 indicates the solution was found).

Putting this all together, we can solve the problem from above:

[51]:
```
import numpy as np
import matplotlib.pyplot as plt

#importing scipy.optimize.leastsq only
from scipy.optimize import leastsq
```

```
#The model to fit to the data
def f(a, x):
    return a[0] + a[1] * np.exp(a[2] * x)

#Residuals (in this case the error term)
def err(a, x, y):
    return f(a, x) - y

#Reading the data
# The `unpack` keyword argument seperates the columns into individual arrays
xdata, ydata = np.loadtxt('data/nonlinear_data.csv', delimiter = ',', unpack =␣
 ↪True)

#Performing the fit
a0 = [1.5, 0.6, 0.2] #initial guess

a, success = leastsq(err, a0, args = (xdata, ydata))

#Plotting the fit and data
x = np.linspace(xdata.min(), xdata.max(), 1000)

plt.plot(xdata, ydata, 'bo')
plt.plot(x, f(a, x), 'r-')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

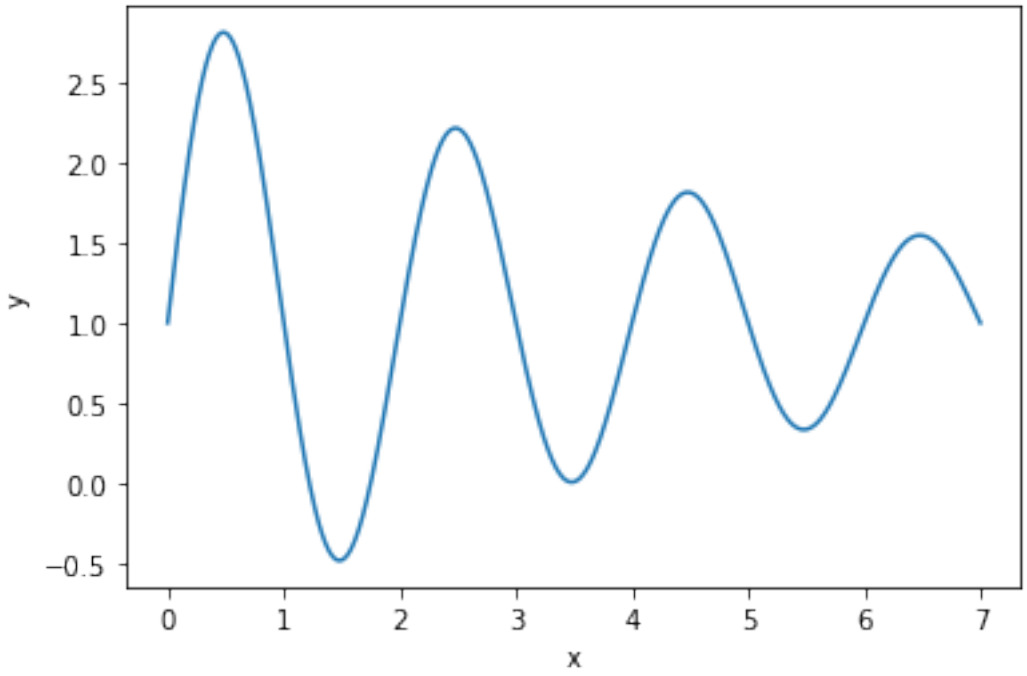


Solutions Converging on Local Minima

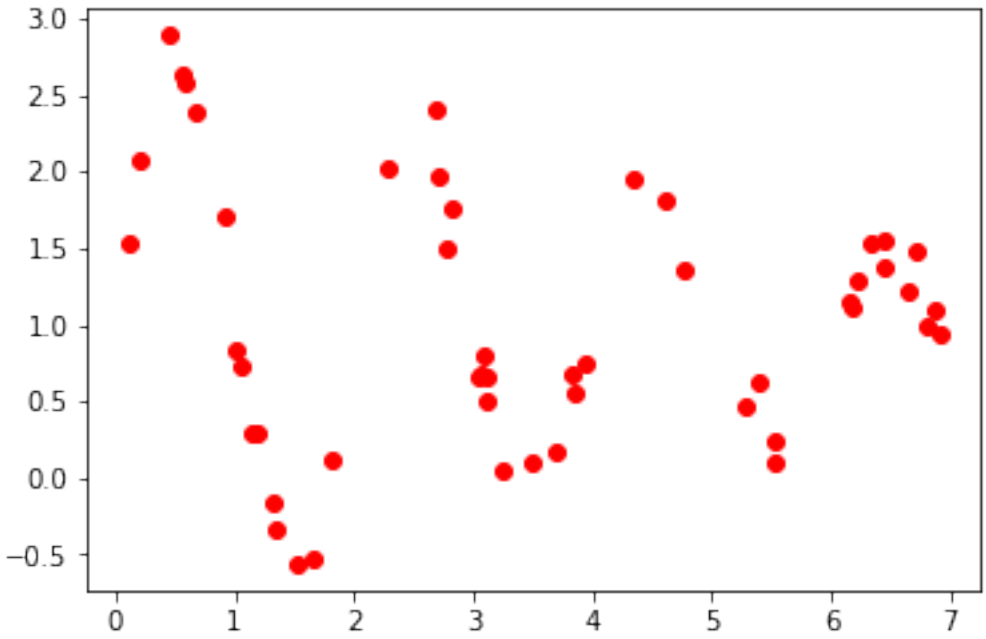
As mentioned before, the numerical algorithm is complete once it has minimized the objective function (the sum of errors squared in out case) to a **local minimum**. It is possible for the solution to not represent the global minimum, which is the ideal solution to obtain.

Let's take a relatively simple example to illustrate this. Consider the functional relation:

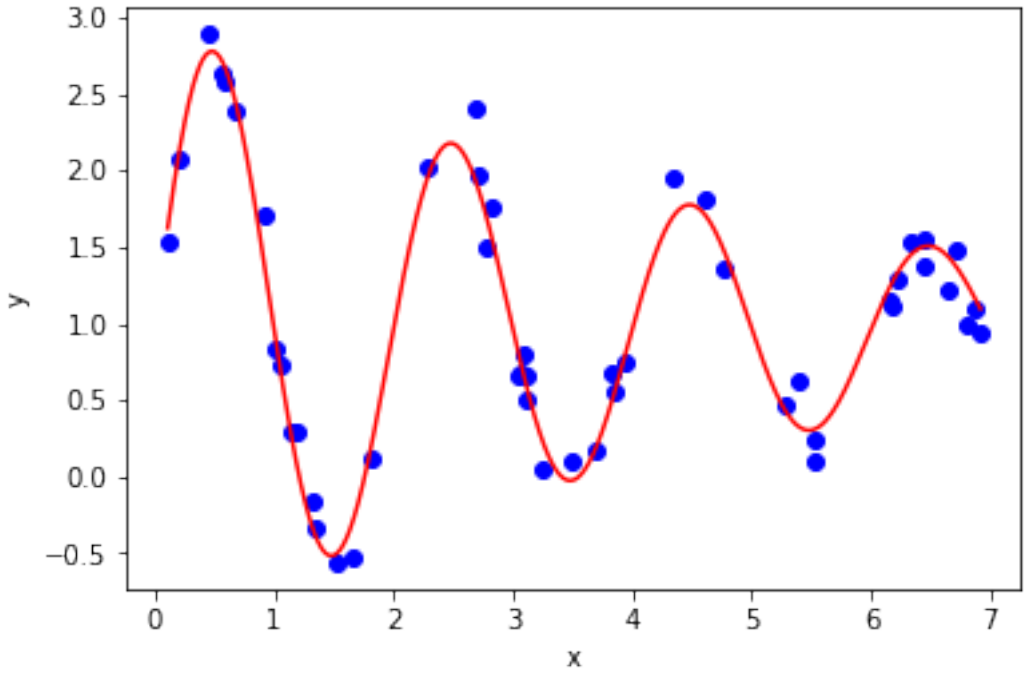
$$y = a_0 + a_1 e^{-a_2 x} \sin(a_3 x)$$



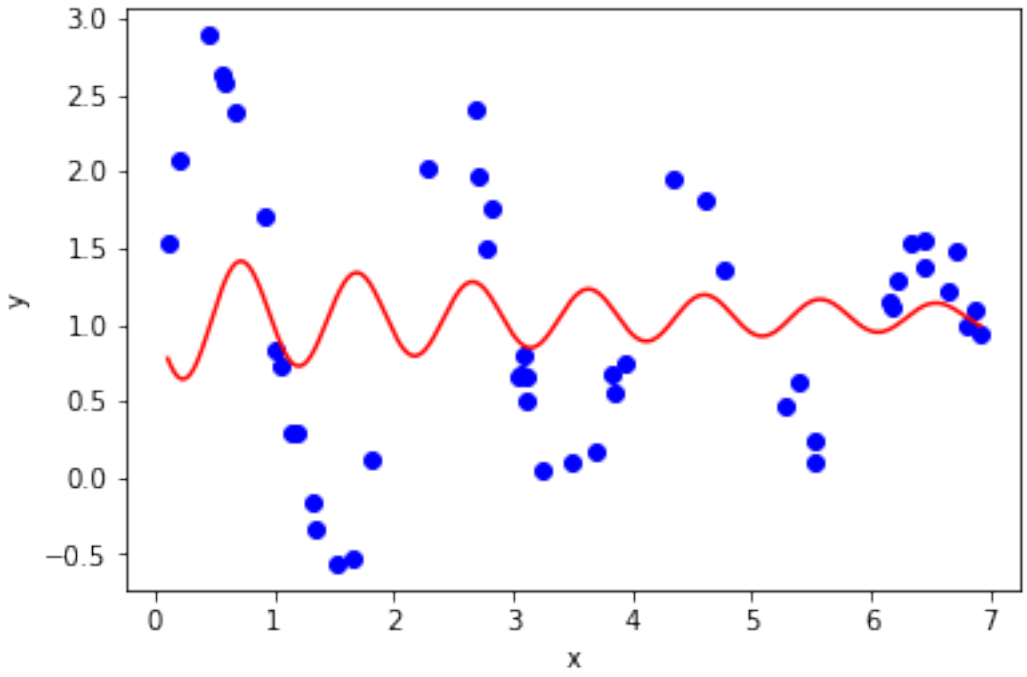
Given a set of data characterized by this relation:

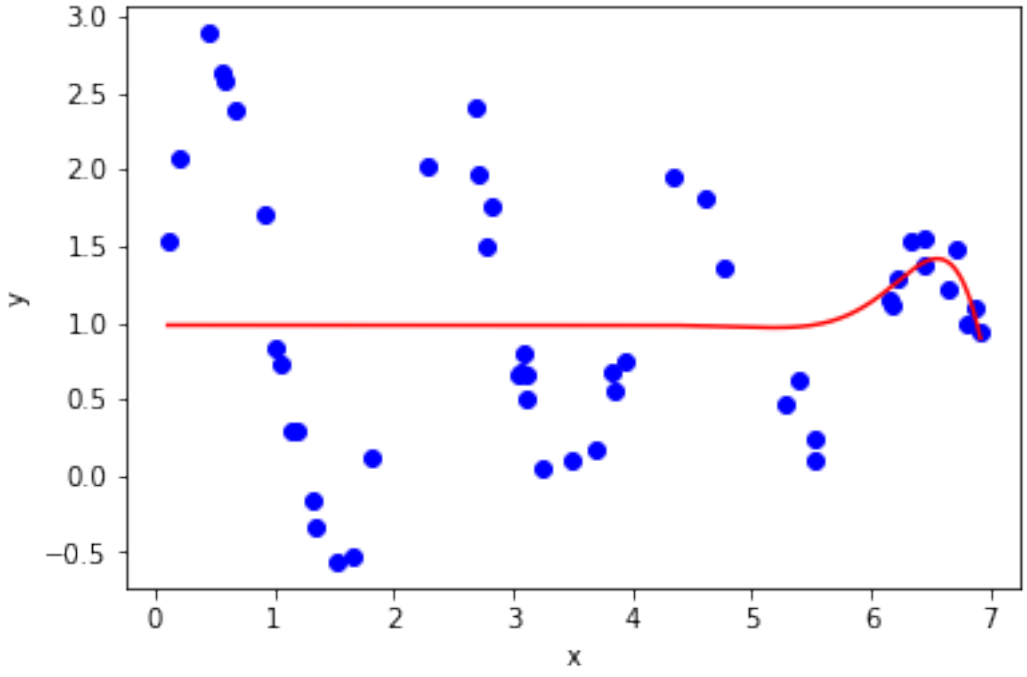


It is relatively easy to find a good fit using `leastsq`:

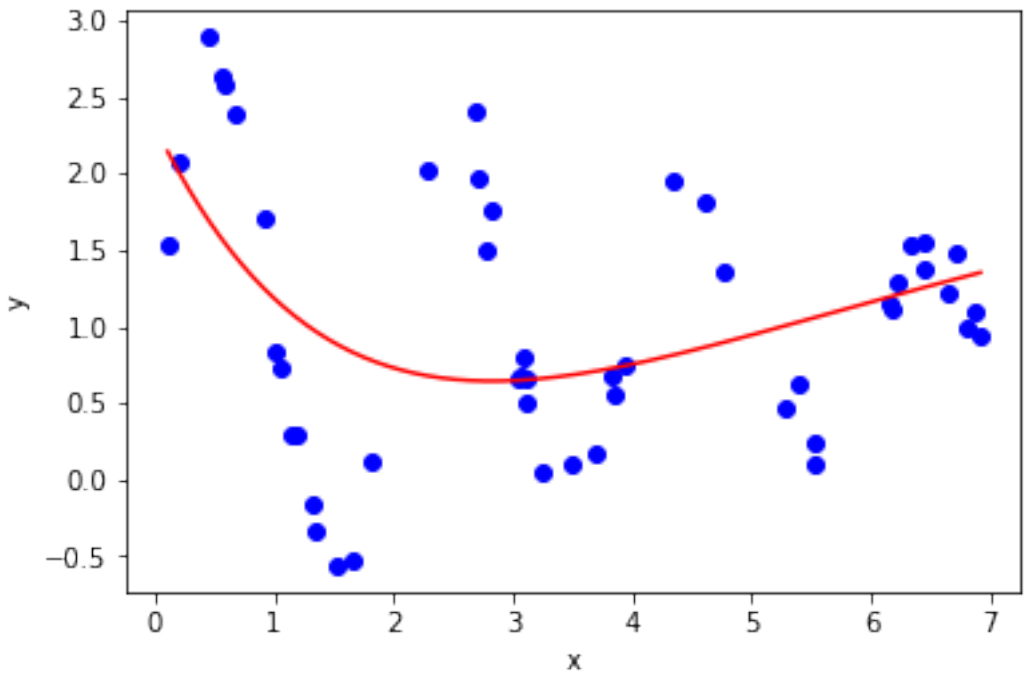


Here are a couple of examples of solutions that returns a supposedly successful solution, but have obviously not converged to the best fit.





Here is an example of a solution that has not succeded (returned an integer flag greater than 4):



If your model does not fit, try varying the initial guess for the fit parameters.

Numerical Solutions to ODEs

Numerical Solutions to ODEs

Numerical Solutions to Ordinary Differential Equations

In many cases you will come across ordinary differential equations (ODEs) with no analytic solution. In this chapter we will explore numerical methods that we can use to solve ODEs that can be expressed in the form:

$$\frac{dy}{dx} = f(x, y)$$

with a given initial value for $y(x_0)$.

We shall also look at ODEs of higher order:

$$\frac{d^n y}{dx^n} = f\left(x, y, \frac{dy}{dx}, \frac{d^2y}{dx^2}, \ldots, \frac{d^{n-1}y}{dx^{n-1}}\right)$$

with given initial conditions for $y(x_0)$, $\frac{d}{dx}y(x_0)$, $\frac{d^2}{dx^2}y(x_0)$, $\frac{d^3}{dx^3}y(x_0)$, ..., $\frac{d^{n-1}}{dx^{n-1}}y(x_0)$.

These are called initial value problems. To solve them you to set as many initial conditions as the order of the equation.

Euler's Method

Euler's Method

Given a first order ODE of the form:

$$\frac{dy}{dx} = y' = f(x, y)$$

where the value for $y(x = x_0) = y_0$ is known. If we wanted to approximate the solution for $y(x_1) = y_1$ at the point $x_1 = x_0 + h$, we can use the Taylor approximation (expanding around x_0):

$$y_1 = y_0 + y'|_{x_0} h + y''|_{x_0} \frac{h^2}{2!} + y'''|_{x_0} \frac{h^3}{3!} + \dots$$

For a small value of h $(0 < h < 1)$, we can neglect high order powers of h without incurring too much error:

$$\begin{aligned} y_1 &\approx y_0 + y'h \\ &\approx y_0 + hf(x_0, y_0) \end{aligned}$$

Now if we used this approximation to find the next value of y at $x_2 = x_1 + h$, y_2,:

$$y_2 \approx y_1 + hf(x_1, y_1)$$

and again for $x_3 = x_2 + h$, y_3:

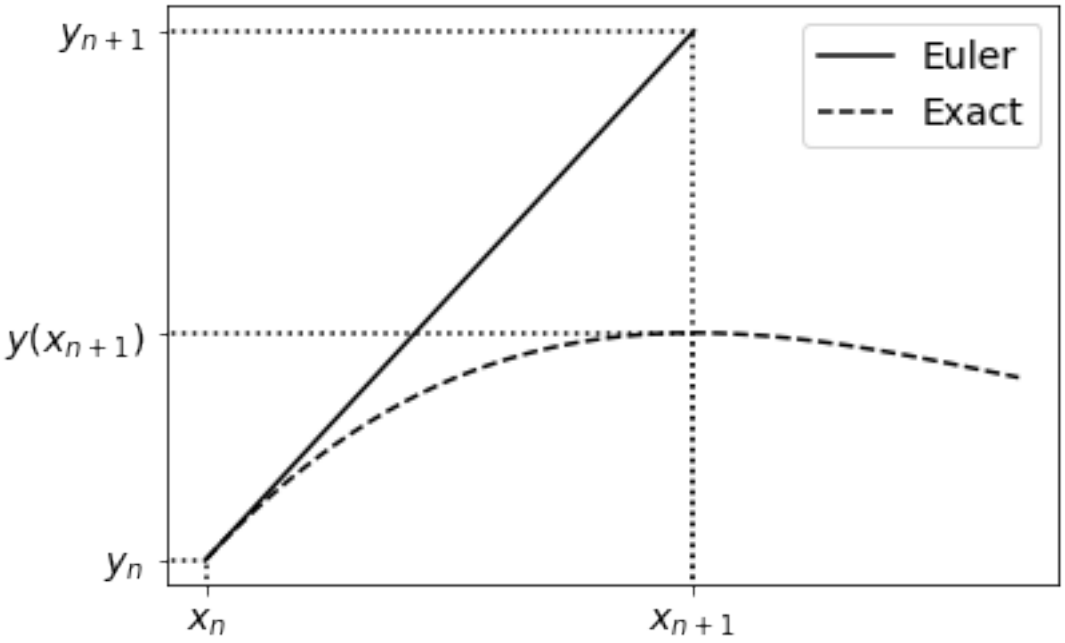
$$y_3 \approx y_2 + hf(x_2, y_2)$$

This method can be iterated n times to find:

$$y_n \approx y_{n-1} + hf(x_{n-1}, y_{n-1})$$

Geometric Interpretation

Another way to see the Euler method is as approximating the solution $y(x)$ as a straight line over the interval $[x_n, x_n + h]$, passing through the point (x_n, y_n) with a gradient of $f(x_n, y_n)$ (the tangent of y at that point):



Worked Example

Consider the ODE:

$$\frac{dy}{dx} = y - xy^2$$

with the given initial conditions: $y = 0.1$ at $x = 0$.

Let's say we want to know the value of y at $x = 10$. We shall **choose** a step size of $h = 0.05$ when integrating this out.

What we need to do is recursively apply Euler steps until we have reached the desired x:

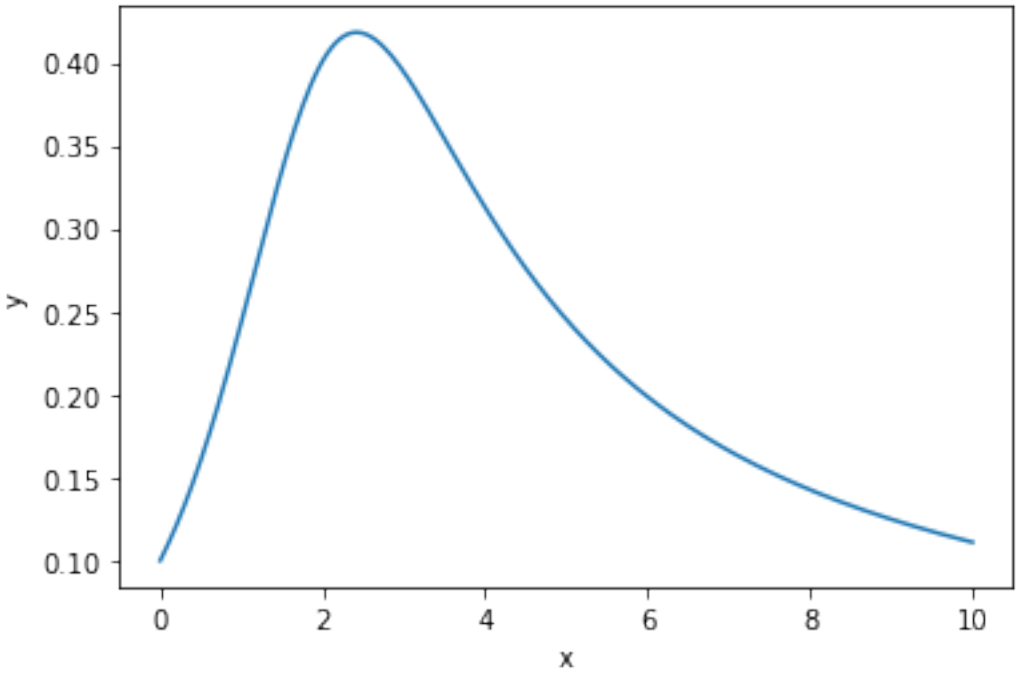
[5]:
```
x, y = 0, 0.1 #initial conditions

h = 0.05 #step size

x_end = 10 #the value of x for which we want to know y

#The ODE function
def f(x,y):
```

```
    return y - x*y*y

#Iterating through the Euler method until x >= x_end:
while x < x_end:
    y = y + h*f(x,y)
    x = x + h #Note, we don't want to update x before it's used in the line␣
 ↪above

print('y at x = 10 is', y)
```

```
y at x = 10 is 0.11102901561046892
```

Now, it is often important for us to visualize the solution for $y(x)$ over the interval, rather than only finding the value of $y(x = 10)$. We could alter the solution above to append the values to an array (as would be the best solution if we didn't know how many iterations we needed), but instead we will create an array of x values on the interval, as this is known to us before perform the Euler solution:

[9]:
```
import numpy as np
import matplotlib.pyplot as plt

x0, y0 = 0, 0.1
h = 0.05
x_end = 10

#The ODE function
def f(x,y):
    return y - x*y*y

#Constructing the arrays:
x_arr = np.arange(x0, x_end + h, h) #make sure it goes up to and including x_end

y_arr = np.zeros(x_arr.shape)
y_arr[0] = y0

#Performing the Euler method, note we don't use the last x value in the update␣
 ↪calculations
for i,x in enumerate(x_arr[:-1]):
    y_arr[i+1] = y_arr[i] + h*f(x, y_arr[i])

#Plotting the solution
fig, ax = plt.subplots()

ax.plot(x_arr, y_arr)
ax.set_xlabel('x')
ax.set_ylabel('y')

plt.show()
```


0.40
0.35
0.30
0.25
0.20
0.15
0.10
y
0
2
4
6
8
10
x

Truncation Error in Euler's Method

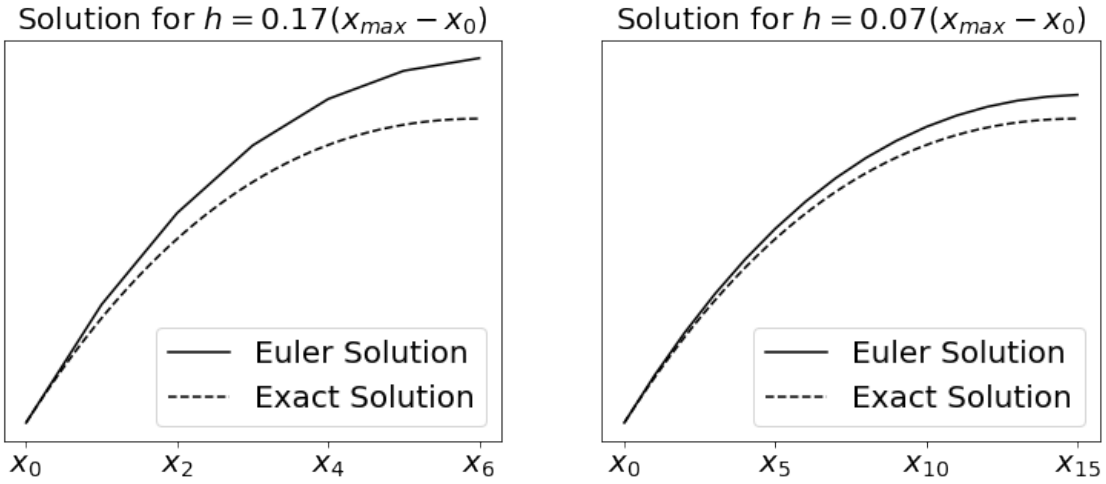
Euler's Method: Truncation Error

Like all numerical methods, Euler's method has systemic error. This is introduced when we discard the higher order terms in the Taylor expansion. The **local** truncation error is thus:

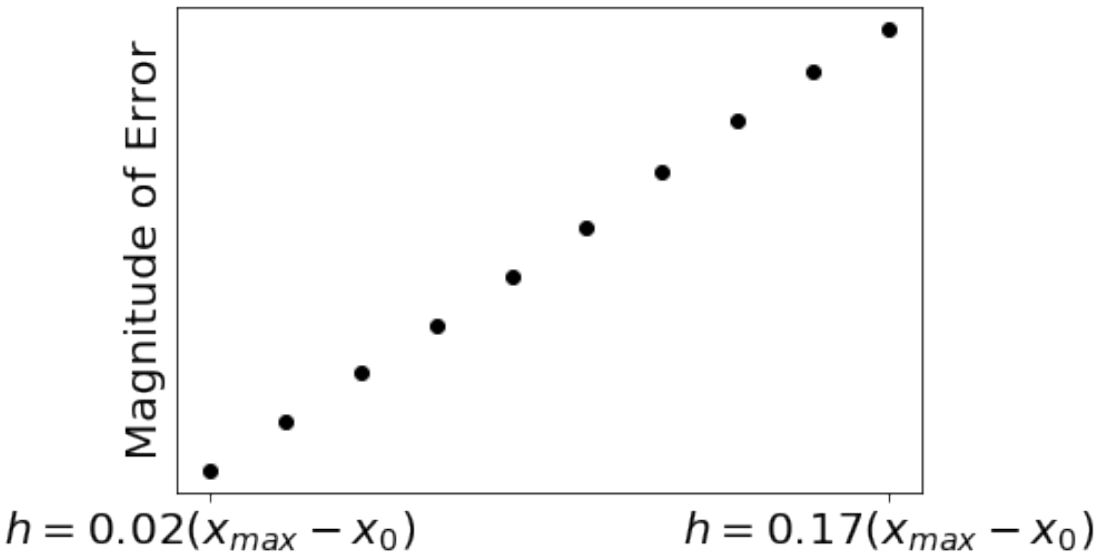
$$E_{n+1} = \frac{1}{2}y''(x_n)h^2 + (h^3)$$

If you are unfamiliar with the notation for (h^3) (big O notation), in this case it stands for all the terms where the lowest order of h is 3. The h^3 term is more relevant than higher order terms for $0 < h < 1$.

The **local** truncation error is associated with a single integration step. It is far more useful, however, to consider the **global** truncation error, which is the error accumulated over multiple integration steps. The global truncation error is (h) {% cite efferson-numerical-methods %}. The derivation for the bounds of the error are beyond the scope of the course. As this error approximately scales linearly with h, reducing the size of h will generally reduce the global error:



We can illustrate the relationship between the global error and h directly by looking at the magnitude of error at the same final x value for different h values:



There is a limit to how much reducing h will help you. If h is too small you could introduce floating point errors, that is when operations require more precision than afforded by the float data type. Reducing the size of h also means that you will have more steps to integrate to a final x, which increases the computational time.

References

{% bibliography –cited %}

Solving Higher Order ODEs

Solving Higher Order ODEs

Second Order Differential Equations

In general, if we wish to solve an ODE of the form

$$\frac{d^2y}{dx^2} = f\left(x, y, \frac{dy}{dx}\right)$$

with initial conditions $y(x = x_0) = y_0$ and $y'(x = x_0) = y'_0$, we can transform these into a system of coupled first order equations by introducing the variable:

$$v = \frac{dy}{dx}$$

which gives us the equations:

$$\frac{dy}{dx} = v$$
$$\frac{dv}{dx} = f(x, y, v)$$

with the initial conditions

As the ODE for y depends on v and the ODE for v depends on y, these equations need to be integrated simultaneously.

Worked Example

Consider second order ODE:

$$\frac{d^2y}{dt^2} + 10\frac{dy}{dt} + 100y = 100|\sin(t)|$$

which we wish to solve for the initial conditions $y = 0.1$, $dy/dx = -0.5$ at $t = 0$.

Firstly let's rearrange the equation to make y'' the subject:

$$\frac{d^2y}{dt^2} = 100|\sin(t)| - 10\frac{dy}{dt} - 100y$$

We start by introducing the variables:

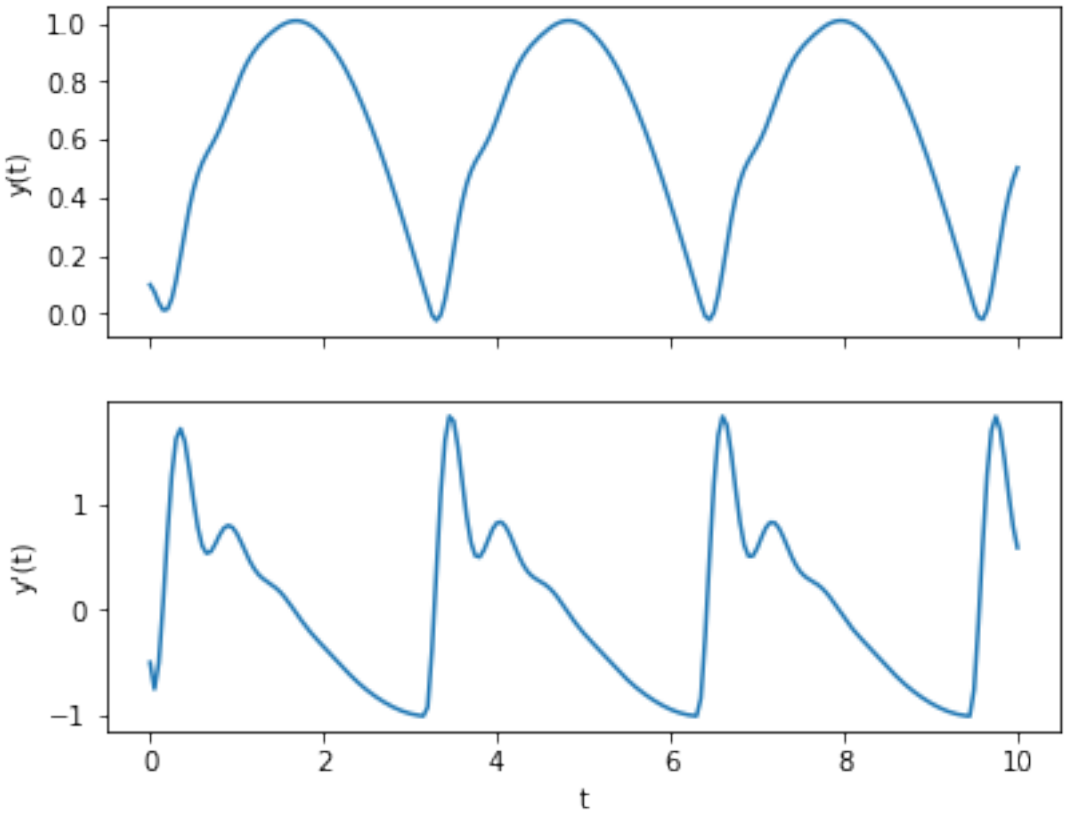
$$v_0 = y$$
$$v_1 = \frac{dy}{dt} = \frac{dv_0}{dt}$$

in order to reduce the second order ODE to a coupled system of two first order ODEs:

$$\frac{dv_0}{dt} = v_1$$
$$\frac{dv_1}{dt} = 100|sin(t)| - 10v_1 - 100v_0$$

[8]:
```
import numpy as np
import matplotlib.pyplot as plt

t0, y0, v0 = 0, 0.1, -0.5 #initial conditions
h = 0.05 #step size
t_end = 10


#The ODE function
def f1(t, y, v):
    return v

def f2(t, y, v):
    return 100*np.abs(np.sin(t)) - 10 * v - 100 * y


#Constructing the arrays:
t_arr = np.arange(t0, t_end + h, h) #make sure it goes up to and including x_end

y_arr = np.zeros(t_arr.shape)
v_arr = np.zeros(t_arr.shape)

#Setting the initial conditions
y_arr[0] = y0
v_arr[0] = v0
```

```
#Performing the Euler method, note we don't use the last x value in the update␣
 ↪calculations
for i,t in enumerate(t_arr[:-1]):
    y_arr[i + 1] = y_arr[i] + h * f1(t, y_arr[i], v_arr[i])
    v_arr[i + 1] = v_arr[i] + h * f2(t, y_arr[i], v_arr[i])

##Plotting both curves
fig, ax = plt.subplots(2,1, sharex = True, figsize = (6.4, 5))

ax[0].plot(t_arr, y_arr)

ax[0].set_ylabel('y(t)')

ax[1].plot(t_arr, v_arr)

ax[1].set_xlabel('t')
ax[1].set_ylabel("y'(t)")

plt.show()
```



In the solution above we used separate variables to store the values for $y(x)$ and $v(x)$. In the

example below, we shall see that it is more practical to store these values in a single 2D array.

Higher Order Differential Equations

We can extend this technique of creating a system of coupled first order equations to an ODE of arbitrary order:

$$\frac{d^n y}{dx^n} = f\left(x, \frac{dy}{dx}, \frac{d^2y}{dx^2}, \frac{d^3y}{dx^3}, \ldots, \frac{d^{n-1}y}{dx^{n-1}}\right)$$

with initial conditions

$$y(x = x_0) = y_0 \qquad \frac{dy}{dx}(x = x_0) = y_0' \qquad \frac{d^2y}{dx^2}(x = x_0) = y_0'' \qquad \ldots \qquad \frac{d^{n-1}y}{dx^{n-1}}(x = x_0) = y_0^{(n-1)}$$

We start by introducing the variables:

$$v_0 = y \qquad v_1 = \frac{dy}{dx} \qquad v_2 = \frac{d^2y}{dx^2} \qquad \ldots \qquad v_{n-1} = \frac{d^{n-1}y}{dx^{n-1}}$$

we can transform the order n ODE to a set of n first order coupled differential equations:

$$\begin{aligned}
\frac{dv_0}{dx} &= v_1 \\
\frac{dv_1}{dx} &= v_2 \\
\frac{dv_2}{dx} &= v_3 \\
&\vdots \\
\frac{dv_{n-2}}{dx} &= v_{n-1} \\
\frac{dv_{n-1}}{dx} &= f(x, v_0, v_1, v_2, v_3, \ldots, v_{n-2}, v_{n-1})
\end{aligned}$$

As the subscripts suggest, it is practical to store the v_i values in a vector.

These equations can be integrated simultaneously, and the solution for y given by v_0.

Worked Example

Consider the order 3 ODE:

$$\frac{d^3y}{dx^3} + y\frac{d^2y}{dx^2} = 0$$

with the initial conditions $y = 1$, $\frac{d}{dx}y = 0.5$ and $\frac{d^2}{dx^2}y = 0.7$ at $x = 0$.

To solve this we introduce the variables:

$$
\begin{aligned}
y_0 &= y \\
y_1 &= \frac{dy}{dx} \\
y_2 &= \frac{d^2y}{dx^2}
\end{aligned}
$$

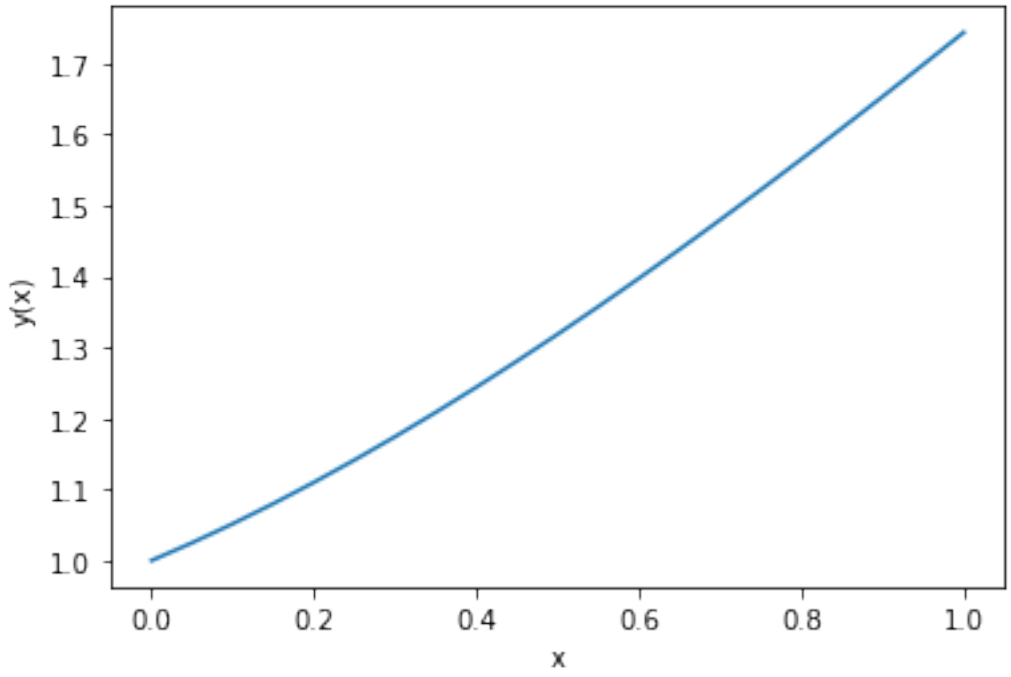
This gives us the system of equations:

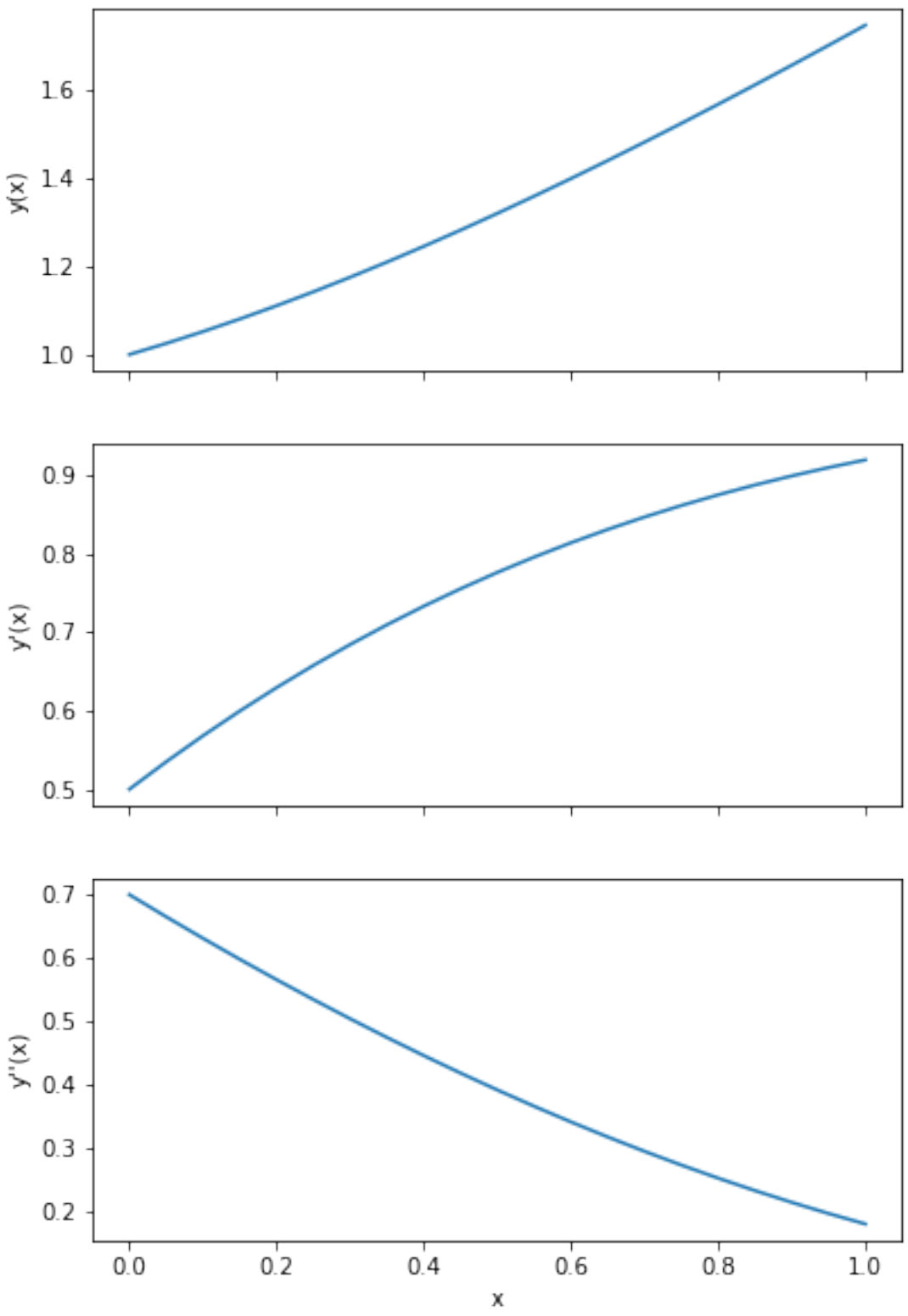
$$
\begin{aligned}
\frac{dy_0}{dx} &= y_1 \\
\frac{dy_1}{dx} &= y_2 \\
\frac{dy_2}{dx} &= -y_0 y_2
\end{aligned}
$$

The solution in Python is:

[20]:
```
import numpy as np
import matplotlib.pyplot as plt

x0, y0 = 0, [1, 0.5, 0.7] #initial conditions
h = 0.05
x_end = 1

##The ODE function (takes y as an array and returns an array of values)

def f(x, y):
    return np.array([
        y[1],
        y[2],
        - y[0] * y[2],
    ])


#Contructing the arrays
x_arr = np.arange(x0, x_end + h, h)

y_arr = np.zeros((x_arr.size, len(y0))) #Using y instead of v as there is no␣
 ↪ambiguity
y_arr[0, :] = y0 #setting the initial conditions
```

```
#Performing the Euler method, note we don't use the last x value in the update␣
↪calculations
for i,x in enumerate(x_arr[:-1]):
    y_arr[i+1,:] = y_arr[i, :] + h*f(x, y_arr[i, :])


##Plotting the solution for y(x) only
fig1, ax1 = plt.subplots()

ax1.plot(x_arr, y_arr[:, 0])

ax1.set_xlabel('x')
ax1.set_ylabel('y(x)')

plt.show()

##Plotting the solutions to the derivatives
fig2, ax2 = plt.subplots(len(y0),1, sharex = True, figsize = (6.4, 10))


for i in range(len(y0)):
    ax2[i].plot(x_arr, y_arr[:, i])

    ax2[i].set_ylabel('y{}(x)'.format("'"*i))

ax2[-1].set_xlabel('x')

plt.show()
```


1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
y(x)
0.0
0.2
0.4
0.6
0.8
1.0
x


y(x)
1.0
1.2
1.4
1.6
y'(x)
0.5
0.6
0.7
0.8
0.9
y''(x)
0.2
0.3
0.4
0.5
0.6
0.7
0.0
0.2
0.4
0.6
0.8
1.0
x

Runge-Kutta Methods

Runge-Kutta Methods

The aforementioned Euler's method is the simplest single step ODE solving method, but has a fairly large error. The Runge-Kutta methods are more popular due to their improved accuracy, in particular 4th and 5th order methods.

Outline of the Derivation

The idea behind Runge-Kutta is to perform integration steps using a weighted average of Euler-like steps. The following outline {% cite efferson-numerical-methods %} is not a full derivation of the method, as this requires theorems outside the scope of this course.

Second Order Runge-Kutta

We shall start by looking at second order Runge-Kutta methods. We want to solve an ODE of the form

$$\frac{dy}{dx} = f(x, y)$$

on the interval $[x_i, x_{i+1}]$, where $x_{i+1} = x_i + h$, with a given initial condition $y(x = x_i) = y_i$. That is we wish to determine the value of $y(x_{i+1}) = y_{i+1}$. We start by calculating the gradient of y at 2 places:

- The start of the interval: (x_i, y_i)
- A point inside the interval, for which we approximate the y value using Euler's method: $(x_i + \alpha h, y_i + \alpha h f(x_i, y_i))$, for some choice of α.

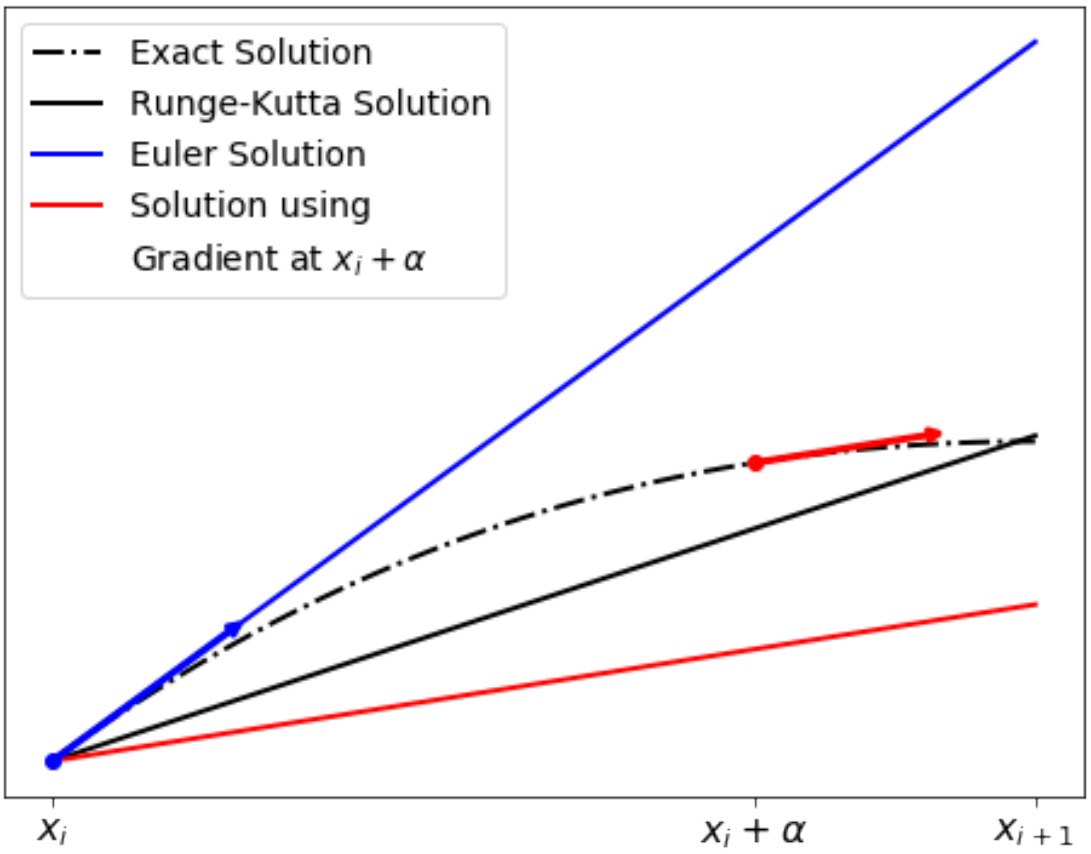
We then approximate the value of y_{i+1} using Euler's method with each of these gradients:

- $y_{i+1} \approx y_i + h f(x_i, y_i)$
- $y_{i+1} \approx y_i + h f(x_i + \alpha h, y_i + \alpha h f(x_i, y_i))$

The final approximation of y_{i+1} is calculated by taking a weighted average of these two approximations:

$$y_{i+1} \approx y_i + c_1 h f(x_i, y_i) + c_2 h f(x_i + \alpha h, y_i + \alpha h f(x_i, y_i))$$

where $c_1 + c_2 = 1$ is required.



Now, how do we go about choosing good values for c_1, c_2 and α? If we Taylor expand the left-hand side of the equation above, and the last term on the right-hand side gives us the relation:

$$\alpha = \frac{1}{2c_2}$$

This still gives us a free choice of one of the parameters. Two popular choices are:

The trapezoid rule: $c1 = c2 = \frac{1}{2}$ and $\alpha = 1$, which yields:

$$y_{i+1} = y_i + \tfrac{1}{2}h\left[f(x_i, y_i) + f(x_i + h, y_i + hf(x_i, y_i))\right]$$

The midpoint rule: $c1 = 0$, $c2 = 1$ and $\alpha = \frac{1}{2}$, which yields:

$$y_{i+1} = y_i + hf\left(x_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}hf(x_i, y_i)\right)$$

Both of these methods have an accumulated error of (h^2), as opposed to Euler's method with (h)

Fourth Order Runge-Kutta (RK4)

As mentioned, the more popular Runge-Kutta method is the fourth order (for which we will not cover the derivation):

$$y_{i+1} = y_i + \frac{1}{6}h\ (k_1 + 2k_2 + 2k_3 + k_4)$$

where the k values are the slopes:

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}hk_1\right) \\ k_3 &= f\left(x_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}hk_2\right) \\ k_4 &= f(x_i + h, y_i + k_3) \end{aligned}$$

k_1 is gradient value at the left of the interval. k_2 is the gradient at the midpoint of the interval, approximated using k_1. The k_3 value is the gradient at the midpoint of the interval using k_2 to approximate it. k_4 is the value of the gradient at the right end of the interval using k_3 to approximate it.

This method has an accumulated error of (h^4)

Worked Example

Consider the ordinary differential equation:

$$\frac{dy}{dx} = \frac{1}{1 + x^2}$$

with the initial condition $y = 1$ at $x = 0$.

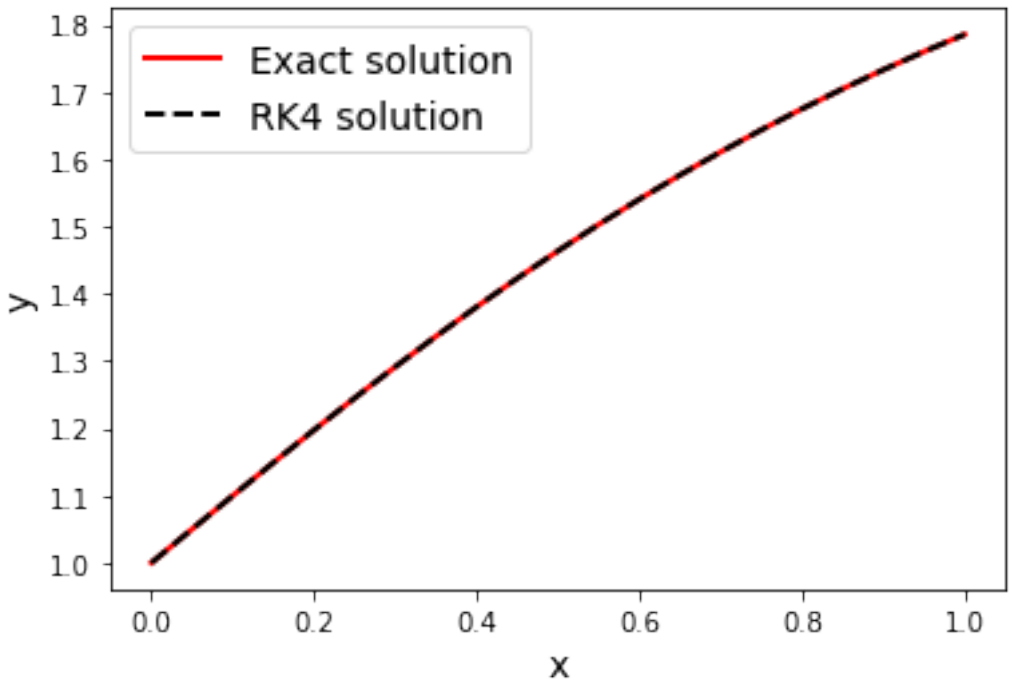
This has the exact solution:

$$y = 1 + \arctan(x)$$

which we can compare are results to.

[7]:
```
import numpy as np
import matplotlib.pyplot as plt

x0, y0 = 0, 1 #initial conditions
h = 0.05
x_end = 1

#Differential equation
def f(x, y):
    return 1/(1 + x*x)

#Exact solution
```

```
def y_exact(x):
    return 1 + np.arctan(x)

#Constructing the arrays:
x_arr = np.arange(x0, x_end + h, h) #make sure it goes up to and including x_end

y_arr = np.zeros(x_arr.shape)
y_arr[0] = y0

#Runge-Kutta method
for i,x in enumerate(x_arr[:-1]):

    #k values
    k1 = f(x, y_arr[i])
    k2 = f(x + 0.5*h, y_arr[i] + 0.5*h*k1)
    k3 = f(x + 0.5*h, y_arr[i] + 0.5*h*k2)
    k4 = f(x + h, y_arr[i] + k3)

    #update
    y_arr[i+1] = y_arr[i] + h/6*(k1 + 2*k2 + 2*k3 + k4)


#Plotting the solution
fig, ax = plt.subplots()

ax.plot(x_arr, y_exact(x_arr), '-r', label = 'Exact solution', linewidth = 2)
ax.plot(x_arr, y_arr, '--k', label = 'RK4 solution', linewidth = 2)
ax.set_xlabel('x', fontsize = 14)
ax.set_ylabel('y', fontsize = 14)

ax.legend(fontsize = 14)

plt.show()
```



High Order ODEs

As we have discussed in a previous page, higher order ODEs can be reduced to a collection of coupled first order ODEs, for example:

$$\frac{dy_0}{dx} = f_0(x, y_0, y_1, \ldots, y_{n-1}) \tag{1}$$
$$\frac{dy_1}{dx} = f_1(x, y_0, y_1, \ldots, y_{n-1}) \tag{2}$$
$$\frac{dy_2}{dx} = f_2(x, y_0, y_1, \ldots, y_{n-1}) \tag{3}$$
$$\vdots \tag{4}$$
$$\frac{dy_{n-1}}{dx} = f_{n-1}(x, y_0, y_1, \ldots, y_{n-1}) \tag{5}$$

As we have seen, the Euler's method solution for this is fairly simple. For the RK4 method, things are slightly more complicated. We must decide how to calculate the k values.

$$y_{j,i+1} = y_{j,i} + \tfrac{h}{6}(k_{1,j} + 2k_{2,j} + 2k_{3,j} + k_{4,j})$$

Note that the y_j variables are not explicitly dependent on each other, but on the independant variable x. Thus we do not have free choice over which y_j values to use when examining another for a particular value of x. For any change in x, we expect simultenous change in all of the y_j. For

this reason, when calculating the k_j values for a particular y_j, we need to consider the changes in the other y_l.

$$\begin{array}{lllllll}
k_{1,j} & = f_j(x_i, & y_{0,i}\ , & \ldots, & y_{j,i}\ , & \ldots, & y_{n-1,i}\ ) \\
k_{2,j} & = f_j\big(x_i + \frac{1}{2}h, & y_{0,i} + \frac{1}{2}k_{1,0}\ , & \ldots, & y_{j,i} + \frac{1}{2}k_{1,j}, & \ldots, & y_{n-1,i} + \frac{1}{2}k_{1,n-1}\ \big) \\
k_{3,j} & = f_j\big(x_i + \frac{1}{2}h, & y_{0,i} + \frac{1}{2}k_{2,0}\ , & \ldots, & y_{j,i} + \frac{1}{2}k_{2,j}, & \ldots, & y_{n-1,i} + \frac{1}{2}k_{2,n-1}\ \big) \\
k_{4,j} & = f_j(x_i + h, & y_{0,i} + k_{3,0}\ , & \ldots, & y_{j,i} + k_{3,j}, & \ldots, & y_{n-1,i} + k_{3,n-1}\big)
\end{array}$$

This looks more complicated then it is to apply in practice. All we need to do is vectorize the solution, as on the previous page. We can represent all the y_j as a vector $\vec{y}$, i.e

$$\vec{y} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

the ODE can thus be represented as:

$$\frac{d\vec{y}}{dx} = \vec{f}(x, \vec{y}) = \begin{pmatrix} f_0(x, \vec{y}) \\ f_1(x, \vec{y}) \\ \vdots \\ f_{n-1}(x, \vec{y}) \end{pmatrix}$$

and an update step as:

$$\vec{y}_{i+1} = \vec{y}_i + \tfrac{1}{6}h(\vec{k_1} + 2\vec{k_2} + 2\vec{k_3} + \vec{k_4})$$

where:

$$\vec{k_m} = \begin{pmatrix} k_{0,m} \\ k_{1,m} \\ \vdots \\ k_{n-1,m} \end{pmatrix}$$

Note that we can write:

$$\begin{pmatrix} y_{0,i} + \frac{1}{2}hk_{1,0} \\ \vdots \\ y_{j,i} + \frac{1}{2}hk_{1,j} \\ \cdots \\ y_{n-1,i} + \frac{1}{2}hk_{1,n-1} \end{pmatrix} = \vec{y}_i + \tfrac{1}{2}h\vec{k1}$$

with this in mind, we can simply write the k values as:

$$
\begin{aligned}
\vec{k_1} &= \vec{f}(x_i, \vec{y_i}) \\
\vec{k_2} &= \vec{f}\left(x_i + \tfrac{1}{2}h, \vec{y_i} + \tfrac{1}{2}h\ \vec{k_1}\right) \\
\vec{k_3} &= \vec{f}\left(x_i + \tfrac{1}{2}h, \vec{y_i} + \tfrac{1}{2}h\ \vec{k_2}\right) \\
\vec{k_4} &= \vec{f}\left(x_i + h, \vec{y_i} + h\ \vec{k_3}\right)
\end{aligned}
$$

Worked Example Consider the third order differential equation:

$$\frac{d^4y}{dx^4} = -12xy - 4x^2\frac{dy}{dx}$$

with the initial conditions: $y(x=0) = 0$, $y'(0) = 0$ and $y''(0) = 2$.

This has an exact solution of:

$$y(x) = e^{-x^2}$$

which we shall use to test our numerical result.

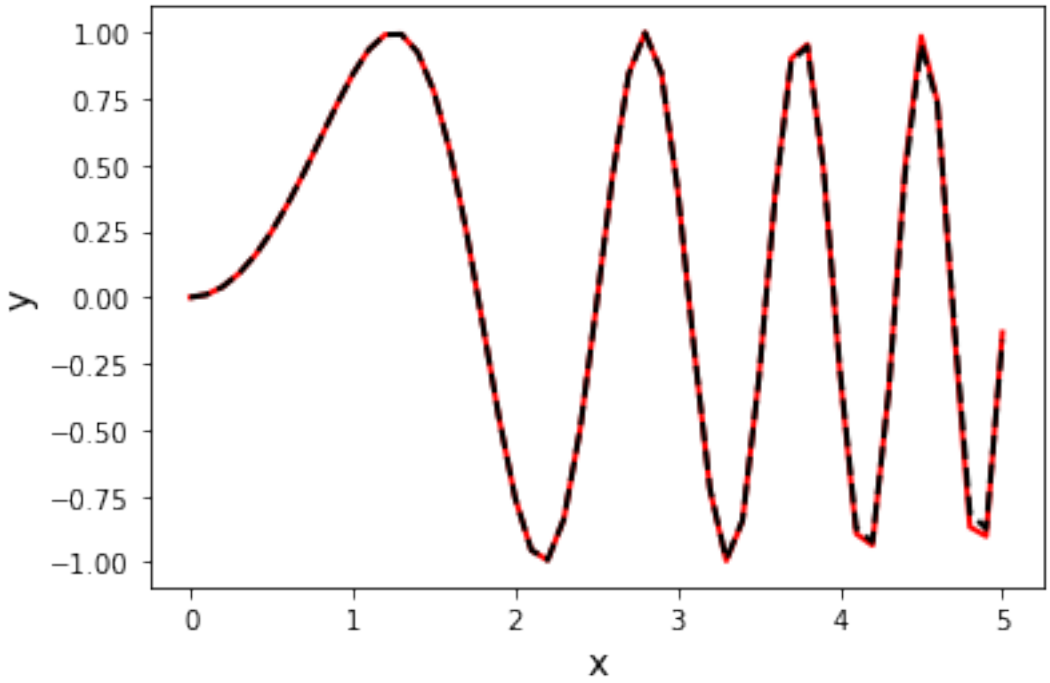
We shall solve this up to $x = 5$ with steps of size $h = 0.1$.

First we reduce this to a system of first order equations by introducing the variables $y_0(x) = y(x)$, $y_1(x) = y'(x)$ and $y_2(x) = y''(x)$:

$$
\begin{aligned}
\frac{dy_0}{dx} &= y_1 \\
\frac{dy_1}{dx} &= y_2 \\
\frac{dy_2}{dx} &= -12xy_0 - 4x^2y_1
\end{aligned}
$$

[25]:
```
import numpy as np
import matplotlib.pyplot as plt

x0, y0 = 0, [0, 0, 2] #initial conditions
h = 0.1
x_end = 5

def f(x, y):
    #Important! This must return an array!
    return np.array([
        y[1],
        y[2],
        -12*x*y[0] - 4*x*x*y[1]
    ])
```

```
def y_exact(x):
    return np.sin(x*x)

#Constructing the arrays:
x_arr = np.arange(x0, x_end + h, h) #make sure it goes up to and including x_end

y_arr = np.zeros((x_arr.size, len(y0)))
y_arr[0, :] = y0

#Runge-Kutta method
for i,x in enumerate(x_arr[:-1]):
    y = y_arr[i,:]

    #k values
    k1 = f(x, y)
    k2 = f(x + 0.5*h, y + 0.5*h*k1)
    k3 = f(x + 0.5*h, y + 0.5*h*k2)
    k4 = f(x + h, y + h*k3)

    #update
    y_arr[i+1, :] = y + h/6*(k1 + 2*k2 + 2*k3 + k4)


#Plotting the solution
fig, ax = plt.subplots()

ax.plot(x_arr, y_exact(x_arr), 'r-', linewidth = 2)
ax.plot(x_arr, y_arr[:, 0], 'k--', linewidth = 2)
ax.set_xlabel('x', fontsize = 14)
ax.set_ylabel('y', fontsize = 14)

plt.show()
```



References

{% bibliography –cited %}

Numerical Root Finding Techniques

Numerical Root Finding Techniques

Numerical Root Finding

A common problem we have to solve is finding the solution of equations of the form:

$$f(x) = 0$$

In the case where a simple analytic solution does not exist, numerical solutions can be employed. We shall take a look at three of these techniques: the bisection method, the secant method and the Newton Raphson method.

All of these methods require that the function f is continuous around the root.

Bisection Method

Bisection Method

The bisection method is what is known as a bracketing root finding method. To use this method the root must not be a turning point of f, or rather f does not change sign as it passes through the root, and that there is only one root in the chosen interval.

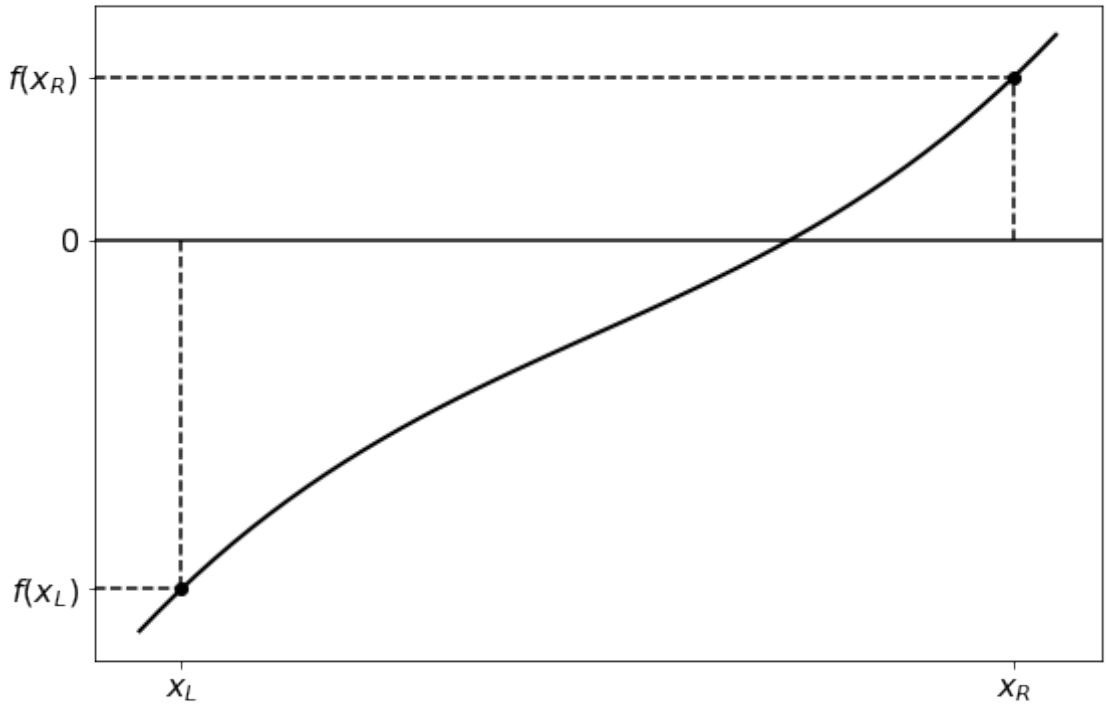
The method can be summarized as:

- Start with a bracket $[x_L, x_R]$ around the root.
- Halve the bracket, introducing the midpoint x_M, giving you two brackets: $[x_L, x_M]$ and $[x_M, x_R]$
- Keep the bracket that contains the root and discard the one that doesn't
- Repeat the process until you are satisfied with the precision of your solution

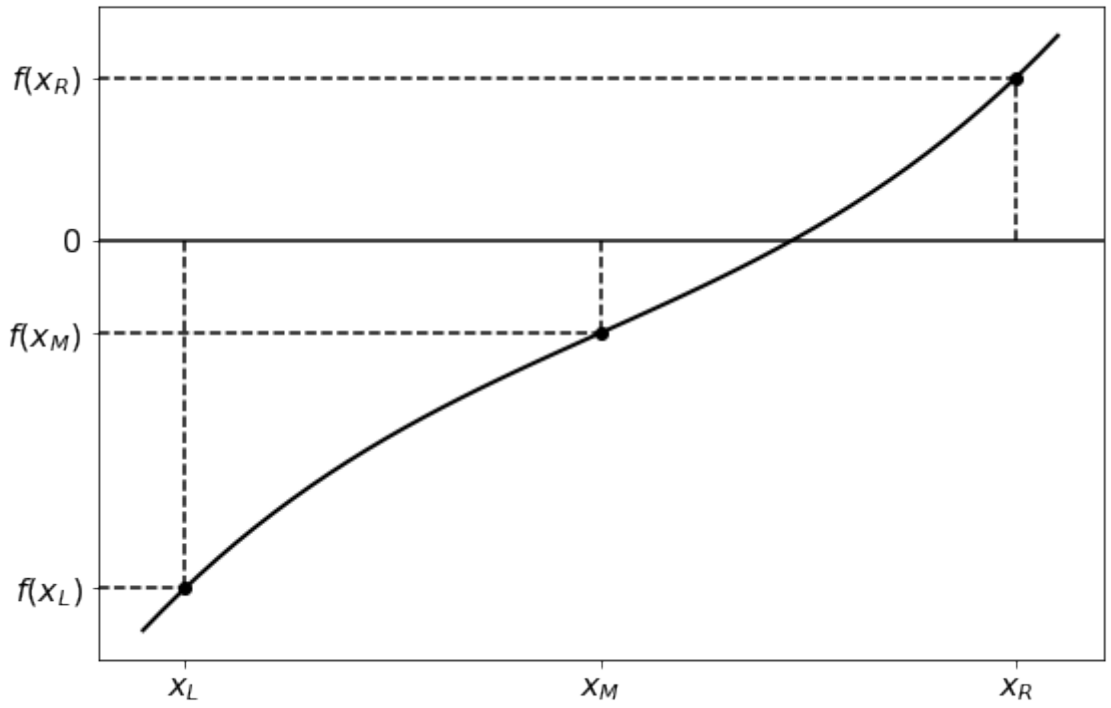
Note that with this technique you end up with an interval that contains the root, rather than an approximation for the root itself. Also note that this method will always converge on a root if one exists in the interval.

In Depth

Let's look at the steps of the method more in depth, starting with choosing our interval such that it contains the root:



We now divide the interval in half by intruding the midpoint $x_M = \frac{1}{2}(x_L + x_R)$ and calculating $f(x_M)$:

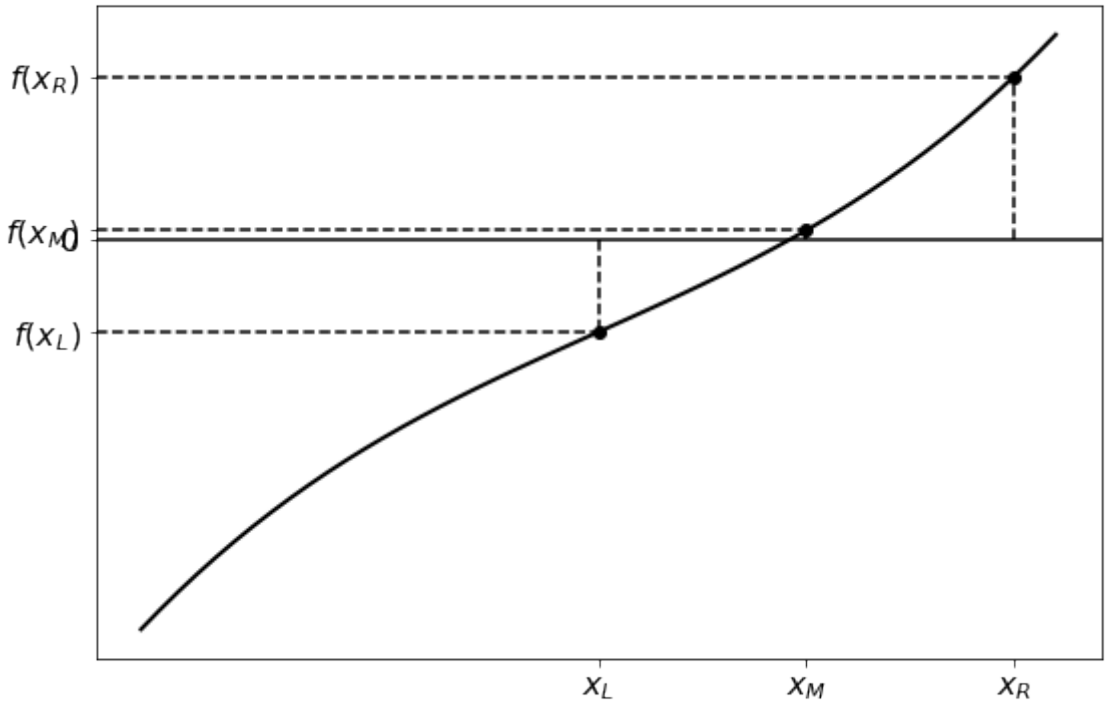


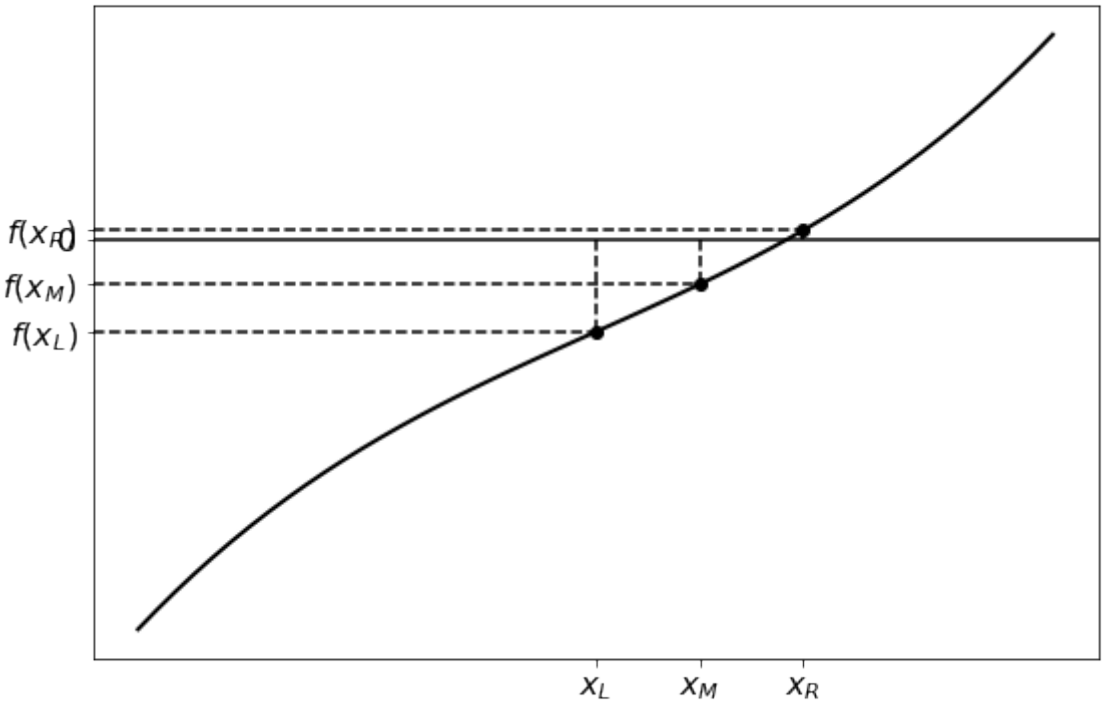
Now we need to figure out which interval contains the root. We can do this by checking if the function value changes signs at the ends of the interval, i.e. which of $f(x_L)$ and $f(x_R)$ is the opposite sign of $f(x_M)$? An easy way to check this is to check if the product of $f(x_L) \times f(x_M)$ or $f(x_M) \times f(x_R)$ is negative. If the product is negative then the sign has changed in that interval and it is the one we choose.

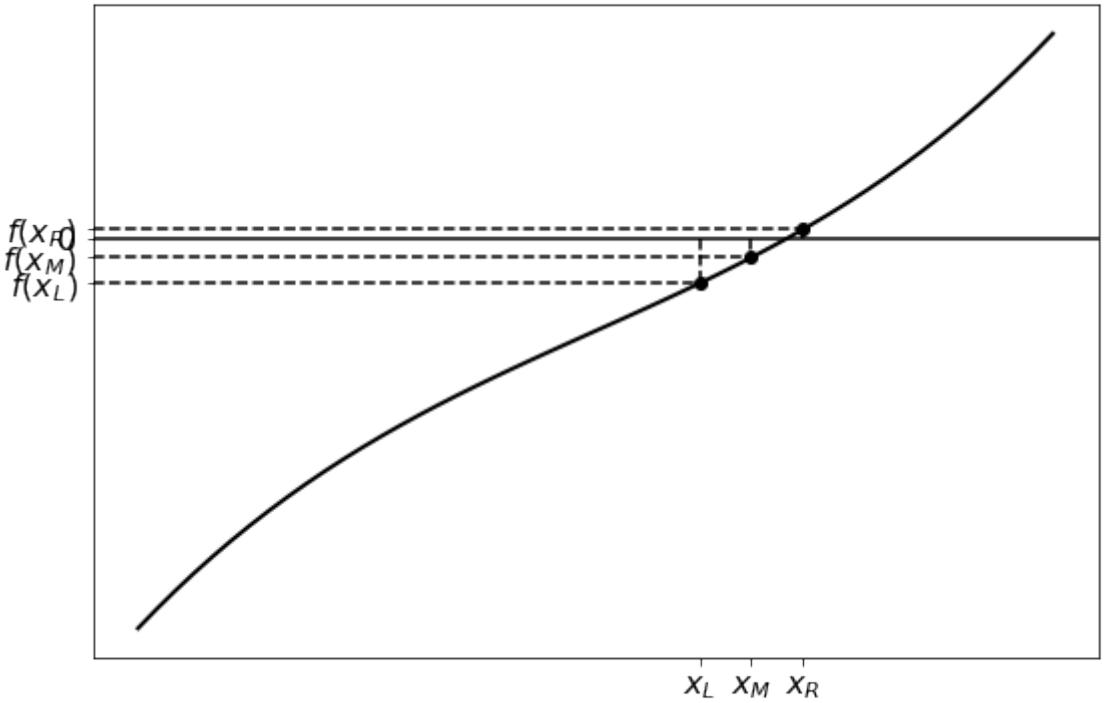
In the figure above, the right interval $[x_M, x_R]$ contains the root.

If we want a more precise answer we can keep applying this technique to our chosen interval. Each time we are left with an interval that was half as big as the last, improving the precision of our solution.

Subsequent iterations of the bisection method are illustrated in the figures below.




$f(x_R)$
0
$f(x_M)$
$f(x_L)$
x_L
x_M
x_R


$f(x_R)$
0
$f(x_M)$
$f(x_L)$
x_L
x_M
x_R

Precision of The Result

The error of our solution is the size of the last interval. Because the length of our interval is halved every step, we can calculate how many steps are needed to achieve a particular accuracy, given the length of our initial interval. After the first step the error is $|x_R - x_L|/2$ and after the n-th step the error is $|x_R - x_L|/2^n$. Thus, for a specified tolerance, the number of steps required is:

$$\begin{aligned} \text{tolerance} &= \frac{|x_R - x_L|}{2^n} \\ \therefore 2^n &= \frac{|x_R - x_L|}{\text{tolerance}} \\ \therefore n &= \log_2 \left(\frac{|x_R - x_L|}{\text{tolerance}} \right) \end{aligned}$$

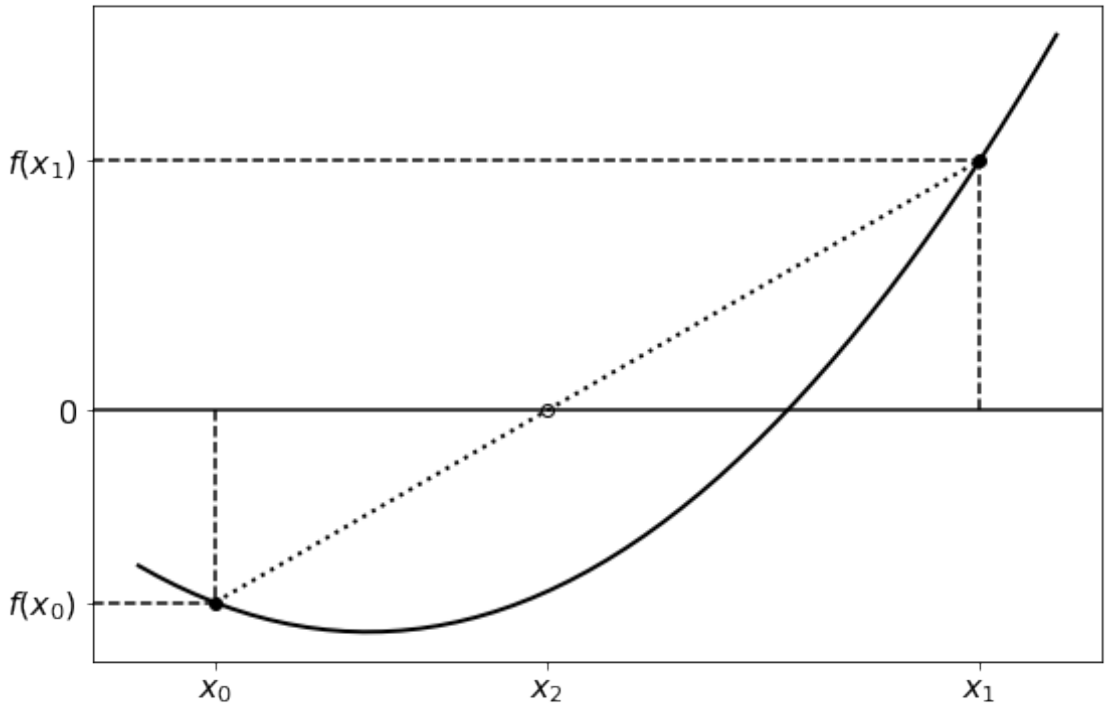
This value is rounded up to an integer.

As we know the number of iterations required to reach a given tolerance, we can use a `for` loop instead of a `while` loop (though both are perfectly acceptable). Note that the number of iterations depends on the size of the starting interval, so it helps to narrow this down before relying on the root finding technique.

Secant Method

Secant Method

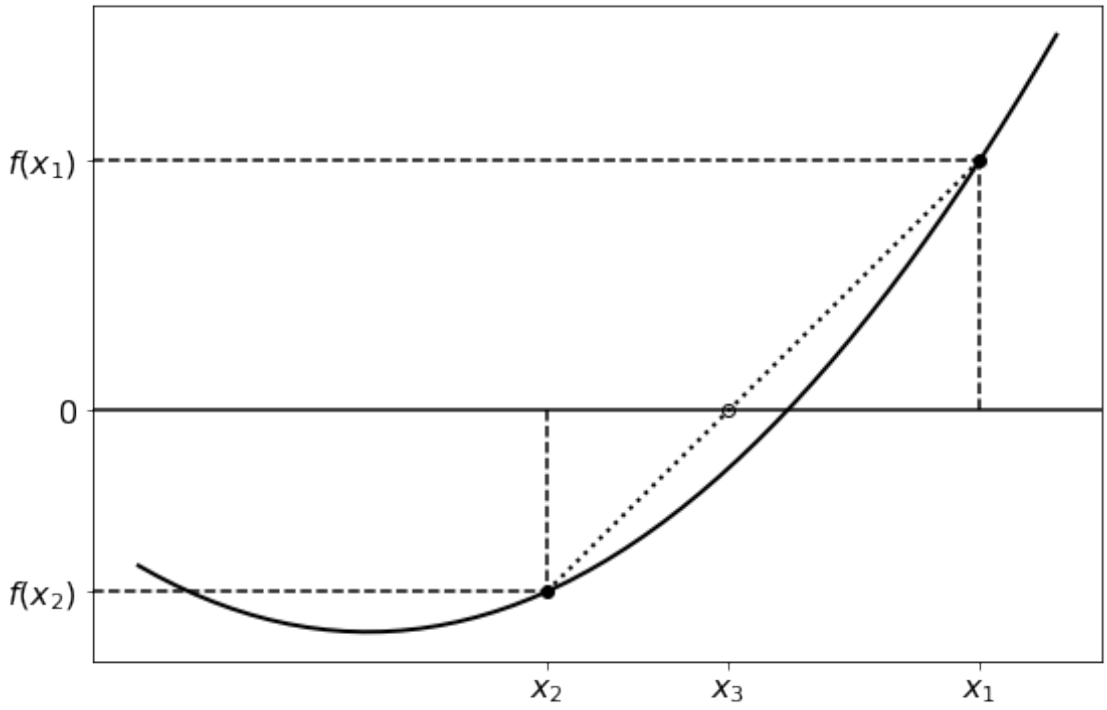
In the secant method we construct a line running between two points on the curve $(x_0, f(x_0))$ and $(x_1, f(x_1))$, and find where it intersects with the x-axis: x_2:

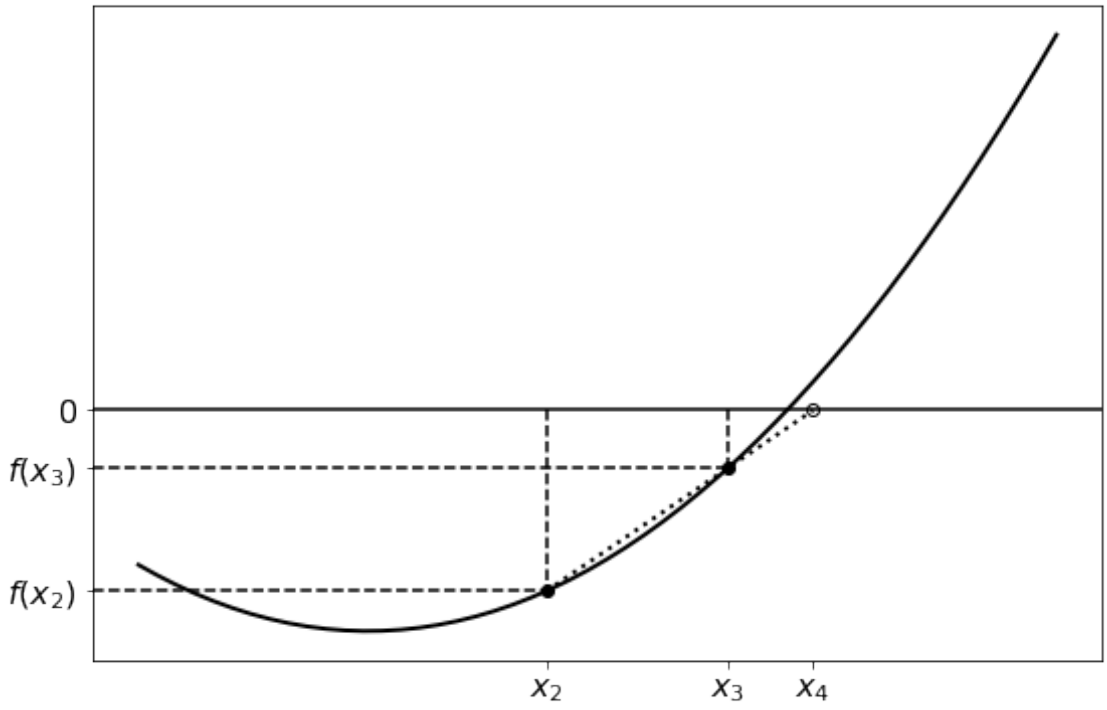


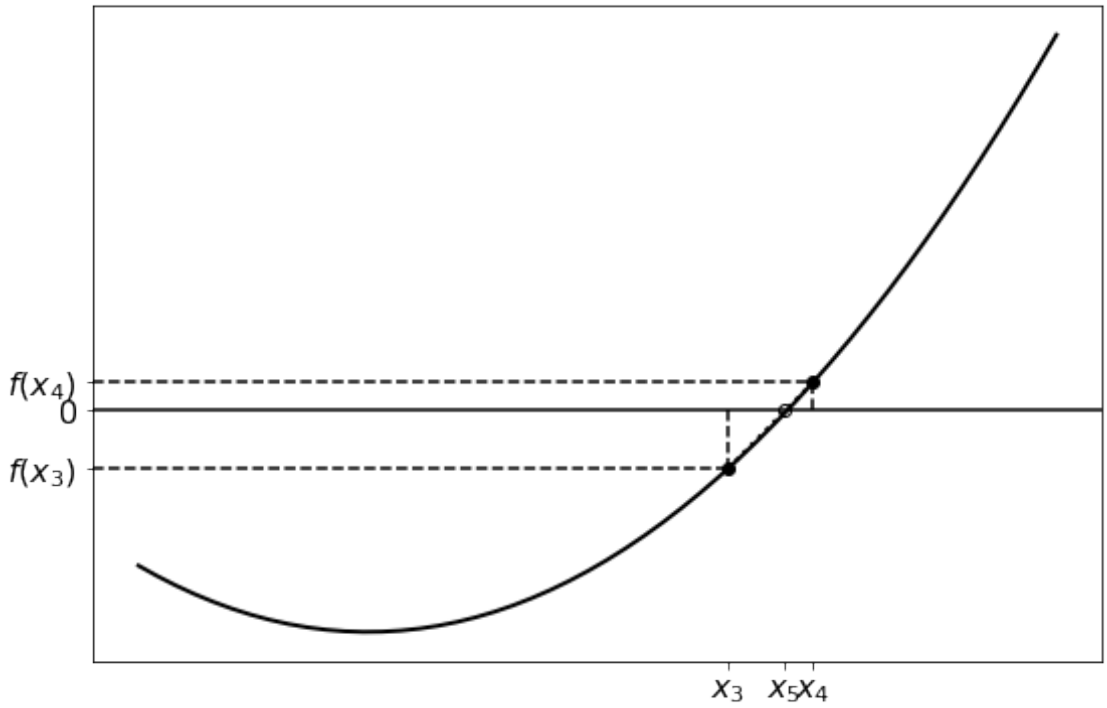
It is assumed that the new point is closer to the root. The justification behind this is beyond the scope of this course.

Note that the starting values x_0 and x_1 can bracket the root, though they need not. You should choose points that are close to the root you desire to find, especially if the function has multiple roots.

We can continue in this fashion, constructing a line between $(x_1, f(x_1))$ and $(x_2, f(x_2))$, and finding the point where this line intersects with the x-axis, x_3. We can continue using the last two points to find the new one, all the while getting closer to the root with each point, as illustrated with the following figures:


$f(x_1)$
0
$f(x_2)$
x_2
x_3
x_1


0
$f(x_3)$
$f(x_2)$
x_2
x_3
x_4



To calculate the intersection x_n for the line constructed from the previous two points $(x_{n-2}, f(x_{n-2}))$ and $(x_{n-2}, f(x_{n-2}))$ we find the equation of the line:

$$y = \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}(x - x_{n-1}) + f(x_{n-1})$$

At the x-intercept $y = 0$ and $x = x_n$:

$$\begin{aligned} 0 &= \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}(x_n - x_{n-1}) + f(x_{n-1}) \\ \therefore x_n &= x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} \end{aligned}$$

Precision

In general the secant method converges far faster than the bisection method, however it is not possible to predict how many iterations are required to achieve a given precision. The precision of the solution can be determined by measuring the convergence of you solution. For a given tolerance, you have reached your required precision when:

$$|x_n - x_{n-1}| < \text{tolerance}$$

Practically you can use a `while` loop to achieve this.

Instability

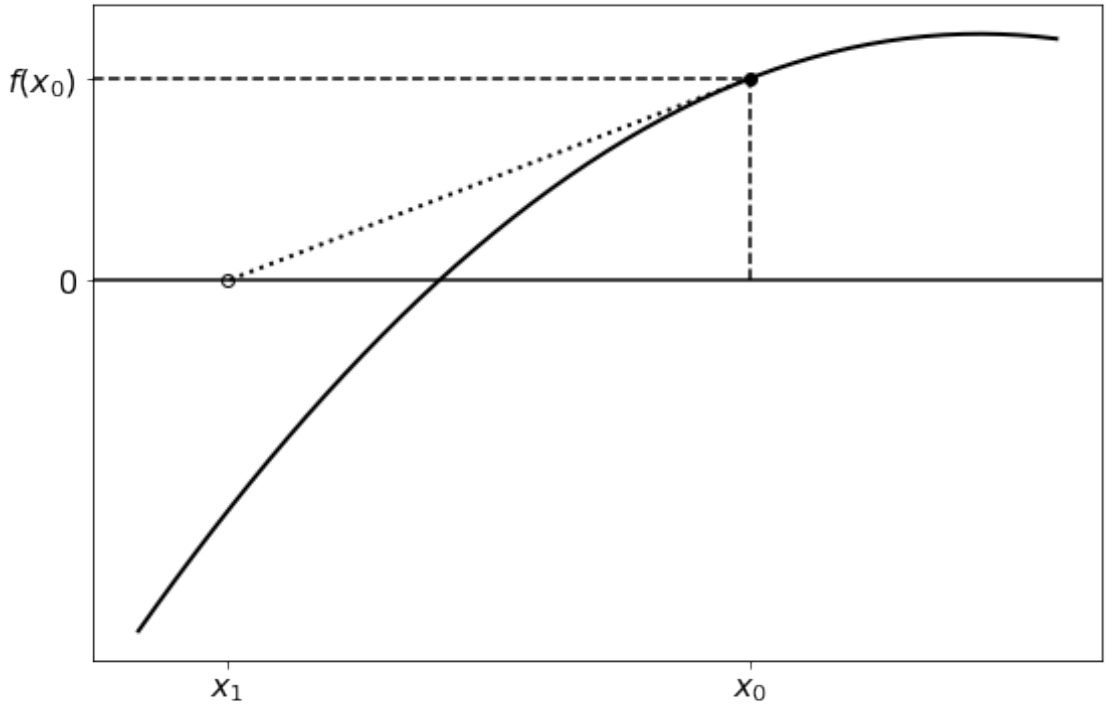
Unlike the bisection method, the secant method isn't always guaranteed to converge, depending on the characteristics of $f(x)$. For example, if there is a stationary point, or if the gradient of $f(x)$ approaches 0 the constructed line can become nearly horizontal, causing the next value of x_n to diverge.

It is also possible for the solution to converge to a different root if they are in close proximity.

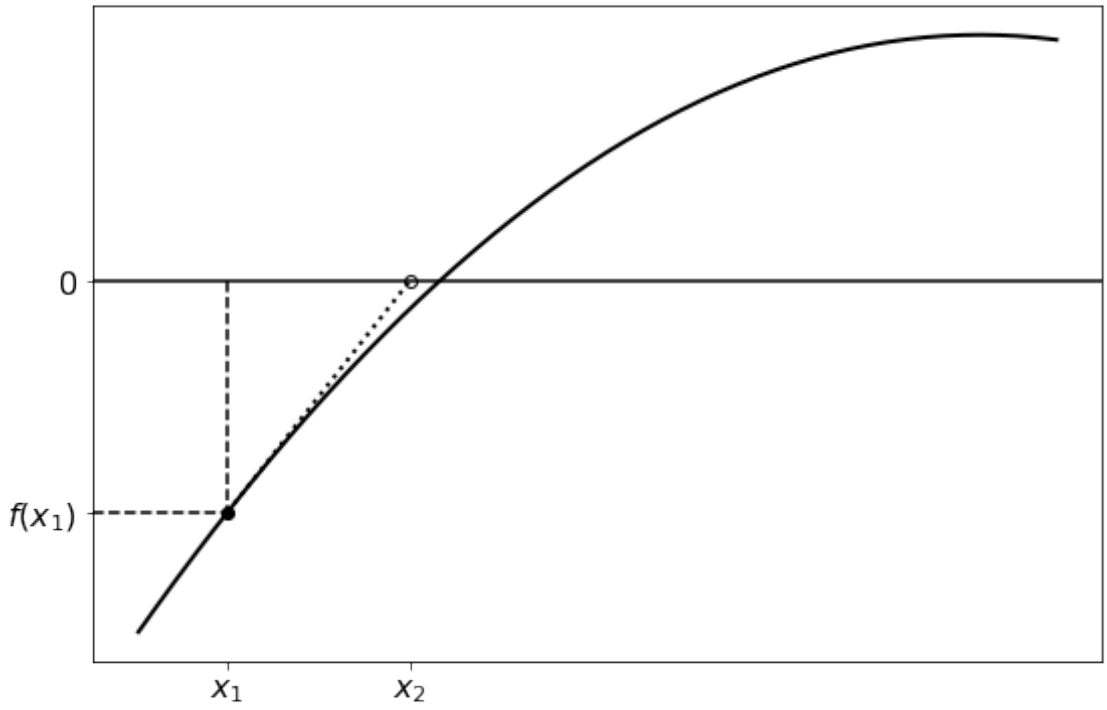
Newton-Raphson Method

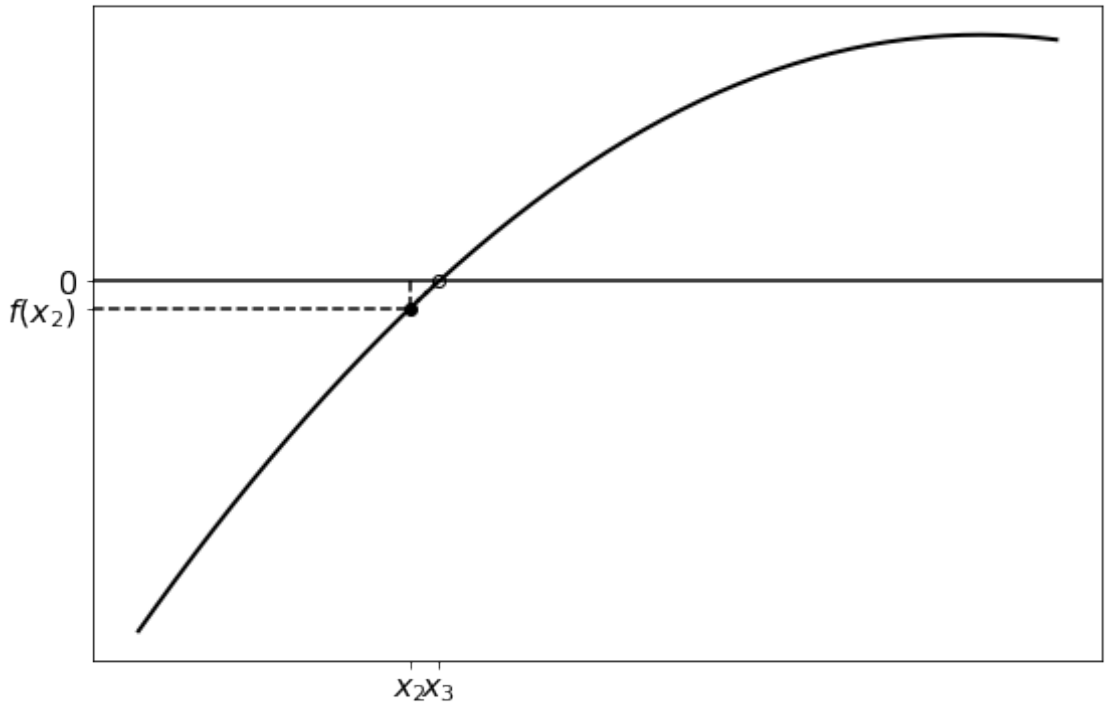
Newton-Raphson Method

The Newton-Raphson method is similar to the secant method, except here we construct a straight line that passes through a point $(x_0, f(x_0))$ with a gradient of $f'(x_0)$, the tangent of $f(x)$ at that point. The next point, x_1, is the intersection of this line with the x-axis:



As before, this process can be repeated with x_1, and the rest of the points after it, converging closer to the root. Further iterations are illustrated in the following figures:





To calculate the point x_n using the previous point x_{n-1}, we start by constructing the line running

through $(x_{n-1}, f(x_{n-1}))$:

$$\frac{y - f(x_{n-1})}{x - x_{n-1}} = f'(x_{n-1})$$

at the x-intercept, $y = 0$ and $x = x_n$:

$$\frac{0 - f(x_{n-1})}{x_n - x_{n-1}} = f'(x_{n-1})$$
$$\therefore x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

Precision

Similarly to the secant method, the precision for the Newton-Raphson method can be set for a given tolerance by finding n such that:

$$|x_n - x_{n-1}| < \text{tolerance}$$

Instability

The Newton-Raphson method suffers from much the same issues as the secant method.

Comparing the Methods

Let's compare the three root finding algorithms we have covered to each other.

Bisection Method

The bisection method starts with an interval that is known to contain the root. The size of this interval is halved with each iteration (improving the precision. For a desired tolerance (or precision), it is possible to calculate how many iterations the Bisection method will take.

If f is continuous on the interval, the interval only contains one root, and the function changes signs as it passes through the root, then the root is guaranteed to be found.

Secant Method

The Secant method requires two points near the root to start off with. If it will converge to the root, then it generally converges quicker than the bisection method, although it's not possible to calculate how many iterations the method will need for a given tolerance.

It is possible for this method not to converge, especially in the case where the gradient of f becomes shallow, which would cause one of the calculated points to shoot off.

It is also possible for this method to converge on a different root if there is one nearby.

Newton-Raphson Method

The Newton-Rhaphson method is similar to the secant method, except it makes use of the derivative of f.

As for the secant method, the Newton-Raphson method converges to the root faster than the bisection method. Also like the secant method, it is possible the method not to converge, or to converge on another nearby root.

In Summary

<table>
<tr><th></th><th>Bisection</th><th>Secant</th><th>Newton-Raphson</th></tr>
<tr><td>Convergence</td><td>Will always converge to a root inside the interval, as long as the function is well behaved.</td><td colspan="2">May not converge to a root if the function has stationary points near it. May converge on neighboring roots.</td></tr>
<tr><td>Rate of Convergence</td><td>Relatively slow convergence.</td><td colspan="2">Fast convergence.</td></tr>
<tr><td>Complexity</td><td colspan="2">Only requires the function, which must simply return values for given arguments on the interval.</td><td>Requires knowledge of the first derivative of the function.</td></tr>
</table>