

# **AST4007W**

# **Computational Methods**

*Scientific Programming in Python*

LUIS A. BALONA  
ED ELSON  
MASIMBA PARADZA  
MAYHEW STEYN

Last Updated: January 7, 2025



## About this book

This book serves as the notes for the Computational Methods module of the National Astrophysics and Space Sciences Programme (NASSP) honours course at the University of Cape Town. It was generated from the online version, found at <https://maystey.github.io/uct-nassp-cm/index.html>. The site was generated using [Jupyter Books v0.15.1](#).

The source code used to generate this book, along with the changelog, are hosted on GitHub at <https://github.com/maystey/uct-nassp-cm>. If you find errors in these notes, or if you have any suggestions, please feel free to submit an issue on the GitHub repository.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

# Table of Contents

---

Introduction	1
Programming and Python in a Nutshell	2
Control Flow Diagrams	5
The Python Standard Library	8
Python Basics	9
Variables	10
Comments	14
Type Conversion	16
Numerical Operators	19
Compound Assignment Operators	23
Introduction to Using Functions	24
Strings	28
String Formatting	36
Data Structures	40
Tuple	41
Lists	43
Dictionaries	48
Control Flow: If Statements	51
Booleans (bool)	52
Comparison Operators	53
Logical Operators	55
Inclusion Operators	57
If Statements	59
Nested If Statements	69
Control Flow: Loops	72
While Loops	73
For Loops	76
Nested Loops	85
Breaking Out of Loops	88
Else Statement and Loops	94
Defining Functions	96
return Statement	98
Function Arguments	102
Local Namespace and Variables	111
Recursive Functions	114

File I/O	118
File I/O	119
Structured Data Files	125
Scientific Packages	134
NumPy	135
Arrays	136
Array Methods and Attributes	143
2D Arrays and Matrices	145
NumPy Random Module	153
Array Conditional Statements and numpy.where()	156
Matplotlib	164
Simple Plots with Pyplot	165
Subplots	174
3D Plotting	180
Astropy	198
Units and Quantities	199
Constants	210
Numerical Methods	213
Numerical Root Finding	214
Bisection Method	215
Secant Method	221
Newton-Raphson Method	225
Comparing the Methods	228
Curve Fitting	230
Linear Regression	231
Linear Least Squares Minimization	232
Linear Chi Squared Minimization	239
Multiple Linear Least Squares Minimization	241
Non-Linear Least Squares Minimization with scipy.optimize.least_squares	255
Fitting Models to Data with scipy.optimize.curve_fit	266
Numerical Solutions to Ordinary Differential Equations	270
Euler's Method	271
Euler's Method: Truncation Error	276
Solving Coupled and Higher Order ODEs	278
Runge-Kutta Methods	291
Numerical Integration Techniques	304
Midpoint Rule	305
Trapezoidal Rule	310



# Part

---

## 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 [Py4, Py6].

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 [Py5]) to machine code [Py6].

## 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) [Py2]. 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 [Py1, Py3].

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). [Py3]. The Python Virtual Machine is always present in the Python system and is the last step of the Python Interpreter [Py1].

## References

[Py1](1,2) Compiling and linking in python — Net-informations.com. [Online; accessed 10-February-2020]. URL: <http://net-informations.com/python/iq/linking.htm>.

[Py2] Dynaimc programming language — MDN Web Docs. [Online; accessed 10-February-2020]. URL: [https://developer.mozilla.org/en-US/docs/Glossary/Dynamic\\_programming\\_language](https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_programming_language).

[Py3](1,2) Ned Batchelder. Is python interpreted or compiled? yes. — Ned Batchelder. [Online; accessed 10-February-2020]. URL: [https://nedbatchelder.com/blog/201803/is\\_python\\_interpreted\\_or\\_compiled\\_yes.html](https://nedbatchelder.com/blog/201803/is_python_interpreted_or_compiled_yes.html).

[Py4] Wikipedia contributors. Assembly language — Wikipedia, the free encyclopedia. [Online; accessed 10-February-2020]. URL: [https://en.wikipedia.org/wiki/Assembly\\_language](https://en.wikipedia.org/wiki/Assembly_language).

[Py5] Wikipedia contributors. Compiler — Wikipedia, the free encyclopedia. [Online; accessed 10-February-2020]. URL: <https://en.wikipedia.org/wiki/Compiler>.

[Py6](1,2) Wikipedia contributors. Machine code — Wikipedia, the free encyclopedia. [Online; accessed 10-February-2020]. URL: [https://en.wikipedia.org/wiki/Machine\\_code](https://en.wikipedia.org/wiki/Machine_code).

# Control Flow Diagrams

Control flow is the order in which a program is executed. A control flow diagram illustrates this with interconnected nodes. This page serves to introduce you to the format of the control flow diagrams used in this book. You may wish to come back to this page once you have encountered a control flow diagram.

The program must have a starting point, in the diagrams this is illustrated by an elliptical node:



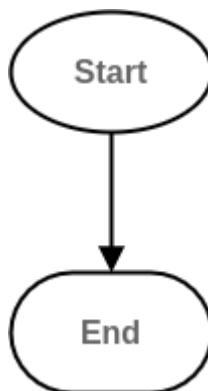
*Fig. 1 Start node of a control flow diagram.*

Control will always flow to an end point, this is illustrated as a rectangle with rounded corners:



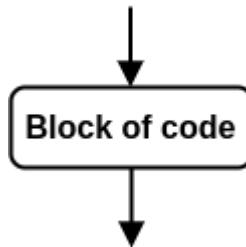
*Fig. 2 End node of a control flow diagram.*

The flow of control from one node to another is illustrated by arrows. Read the diagram by starting from the "Start" node and following the arrow from one node to the next. For example an empty program would be illustrated by:



*Fig. 3 Control flow of an empty program, start node to end node.*

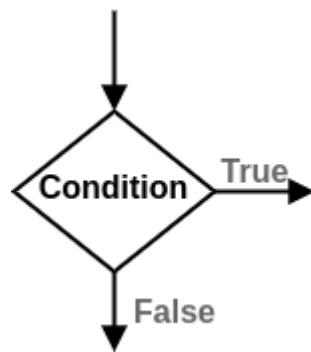
A block of code is illustrated using a rectangular node:



*Fig. 4 Control flow in and out of a block of code.*

The contents of this node are executed in the usual way before control leaves this node and flows through the rest of the diagram.

Sometimes programs require possible branching in their control flow. This is where the essence of logic comes in. These branches can be considered as stemming from questions (or rather conditions that evaluate to true or false), the different branches themselves are the potential answers. This is illustrated using a diamond shaped node:



*Fig. 5 Branching node where control follows a line depending on the outcome of the condition in question.*

If you are reading a control flow diagram that branches, start by following one branch at a time.

Sometimes it will be necessary to insert text into a control flow diagram that isn't present in the code or pseudo code. In these cases that text will be gray, and the text which actually appears in the code will be black.

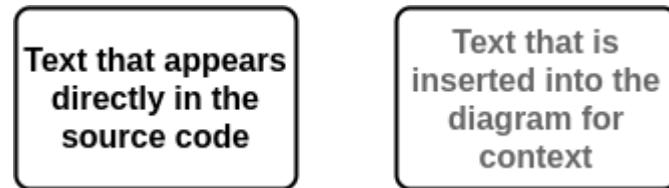


Fig. 6 Black text is directly from the source code, gray text is inserted into the diagram for context.

# Part

---

## The Python Standard Library

# Python Basics

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

# Variables

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

```
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 its name. For example, if we want to print the value to terminal:

```
print(a)
```

```
5
```

## Data Types

The information stored in memory needs to be interpreted 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:

```
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:

```
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:

```
print(type('This is a string.'))
```

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

You are not limited to using single quotes to define strings, see the chapter [Standard Library/Strings](#) for more.

## 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 separations for variable and function names.

## Assigning Variable Values to other Variable Values

You can assign the value of one variable to another:

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

```
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 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:

```
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:

```
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:

```
#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:

```
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

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:

```
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'>
```

```
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).

```
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'>
```

```
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:

```
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)
~\AppData\Local\Temp\ipykernel_6172\4259612967.py 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:

```
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)  
~\AppData\Local\Temp\ipykernel_6172\2048701317.py 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'
```

# 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:

```
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 3.333333333333335
a subtracted from c 8.0
```

These operators can also be used for integers:

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

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

and between integers and floats:

```
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.x 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:

```
print(2**3)
```

8

which can be read as  $2^3$ .

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

```
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:

```
print(16%3)
```

1

In mathematics this would be expressed as

$16 \bmod 3$ .

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

```
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:

```
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:

```
print( 2**3 + 4)
```

12

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

```
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.

# 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 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 then 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:

```
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:

```
print?
```

**Docstring:**  
`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.

**Type:** `builtin_function_or_method`

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

```
help(print)
```

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

```
print(*args, **kwargs)
    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, its default value is a space `' '`. As a first example, let's change the separation to an empty string (no space):

```
print('There', 'are', 'no', 'spaces', sep = '')
```

Therearespaces

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

```
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:

```
user_name = input('What is your name?')

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

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:

```
user_number = input('Enter a number: ')

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

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`):

```
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))
```

```
Entered: 6 <class 'int'>
```

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

# Strings

In this chapter we shall take a closer look at the string data type and some of the operations associated with it. The following page makes heavy reference to online notes by Dr. Andrew N. Harrington, [Hands-on Python 3 Tutorial \[Str1\]](#).

## String Literals

A string literal simply refers to how you specify that the data you are writing is a string. In Python this is achieved by placing quotes around the string contents. For example:

```
str_single = 'This is a string'
```

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

```
str_double = "This is a string"
```

Note that these two strings are identical.

In most cases you are free to decide which quotes you want to use. The standard for Python is to use single quotes where possible, but what's most important is that your style choice is consistent within a project.

Sometimes it is advantages to use single or double quotes specifically. For example, if you want to use double quotes inside your string this will break a double quote string literal, but not a single quote one, and vice versa.

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

```
String using single quotes, " does not break the string.
String using double quotes, ' doesn't break the string.'
```

For strings containing line breaks, you can use either `''''` (three single quotes) or `"""` (three double quotes) to enclose the string contents:

```
print(
  '''String with a
line break'''
)

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

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

Note that white space (like indentations) will show up in these strings:

```
print(
  ...
  A string with
  Indented lines
  ...
)
```

```
A string with
Indented lines
```

This can give you trouble when you are defining strings in an indented code block, in these cases you may be better off using the `\n` special character, which creates new lines.

Triple quotes can be used for single line strings as well. This may come in handy when single or double quotes are no longer an option:

```
print('''I said: "Hello world! How's it going?" ''')
```

```
I said: "Hello world! How's it going?"
```

## Concatenation

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

```
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:

```
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 its 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:

```
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:

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

```
-----
IndexError: string index out of range
```

Traceback (most recent call last)  
~\AppData\Local\Temp\ipykernel\_7184\580220801.py in <module>  
----> 1 print('Index 11', computer\_string[11])

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

```
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:

```
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.

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

The last character: r

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

```
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:

```
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`:

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

com

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

```
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:

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

puter

or simply leave the `stop_index` blank:

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

uter

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

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

ute

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`:

```
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:

```
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:

```
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.

```
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.

## References

- [Str1] Dr. Andrew N. Harrington. Hands-on python 3 tutorial. 2019. [Online; accessed 6-December-2019; released under the CC BY-NC-SA 4.0 license]. URL: <http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/index.html>.

# 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.

```
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:

```
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:

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

1, 3, 2

Note that this also allows you to repeat elements:

```
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:

```
print('You can find the point at position ({x}, {y}).'.format(x = 2, y = 6)) #Arguments w
```

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:

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

5

Now, lets see how the width affects the output:

```
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:

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

1.23

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

```
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).

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

```
'Hello World! How are you doing today?'
```

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

# 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

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:

```
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):

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

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

Like strings, tuples can be indexed and sliced:

```
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):

```
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:

```
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 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 `[]`:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```

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:

```

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]

```

# List Comprehension

If you are not familiar with for loops you may wish to read the page [Python Standard Library/Loops/For Loops](#) before returning to this section.

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**:

```
#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:

```
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 nest list comprehension:

```
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]]
```

# 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
```

```
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:

```
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:

```
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`:

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

```
-----
KeyError Traceback (most recent call last)
<ipython-input-5-c8f93a31d4a2> in <module>
----> 1 print(d[2])

KeyError: 2
```

## List 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:

```
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:

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

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



# 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 (bool)

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

```
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 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:

```
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 $\leq$ and Greater-than-equal-to $\geq$

As their names suggest, the  $\leq$  operator is related to the  $\leq$  assertion in mathematics. Similarly  $\geq$  is related to  $\geq$ .

```
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  $=$ .

```
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:

```
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:

```
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 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:

```
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:

```
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:

```
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:

```
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', not True or True and False)
```

```
True and False or True is  True
True or True and False is True
not True or True and False 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.

# Inclusion Operators

Inclusion operators are used to check if a value/object is in a collection and return boolean values.

**in**

The `in` operator returns a `True` if the left value/object is **in** the collection on the right and `False` if it is not:

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

True

```
print('a' in 'abc')
```

True

```
print(2.4 in [1, 2, 3])
```

False

**not in**

The `not in` operator returns the converse of the `in` operator:

```
print('a' not in 'abc')
```

False

```
print(7 not in [1, 2, 3, 4, 5])
```

True

# If Statements

If statements give us the ability to execute different blocks of code depending on the outcome of a logical statement (or boolean value).

The syntax for an if statement is:

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

where `block of code` following the `:` and indented is considered as the code inside the if statement. `block of code` will only be executed if condition is/evaluates to `true`.

Consider an if statement with code around it:

```
code block before  
  
if condition:  
    code block inside  
  
code block after
```

here `code block before` will be executed first, then `condition` will be evaluated. If `condition` is true, `code block inside` will be executed. Finally `code block after` will be executed (regardless of whether or not `condition` is true). Illustrated in a control flow diagram:

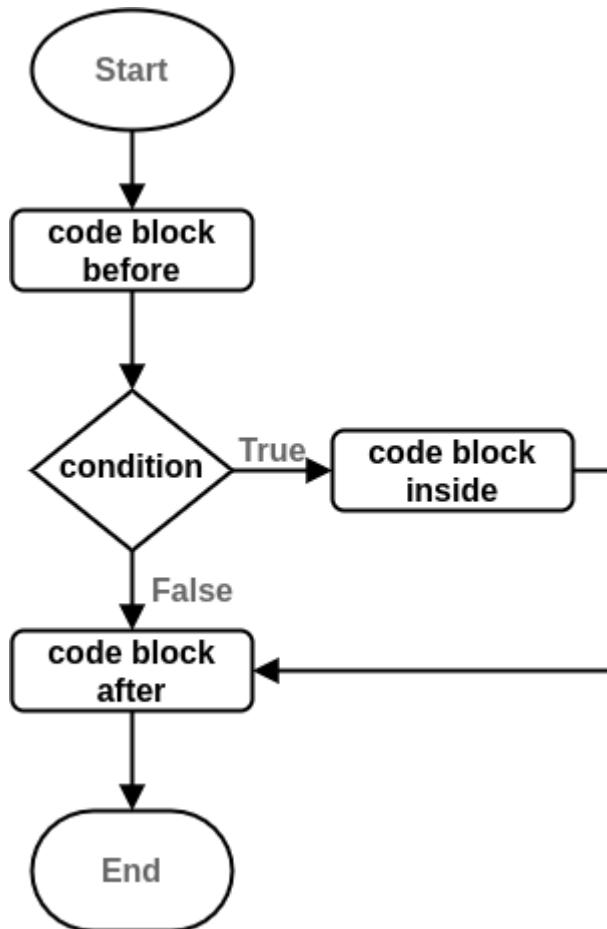


Fig. 7 Control flow diagram of the pseudo code if statement example.

## Worked Example

Let's consider the problem where we want to check if one variable is greater than the other. One solution using an if statement is:

```

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 as `a > b` would evaluate to `False` and the code block contained in the if statement would not be executed.

## 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.

This pseudo code can be illustrated by the control flow diagram:

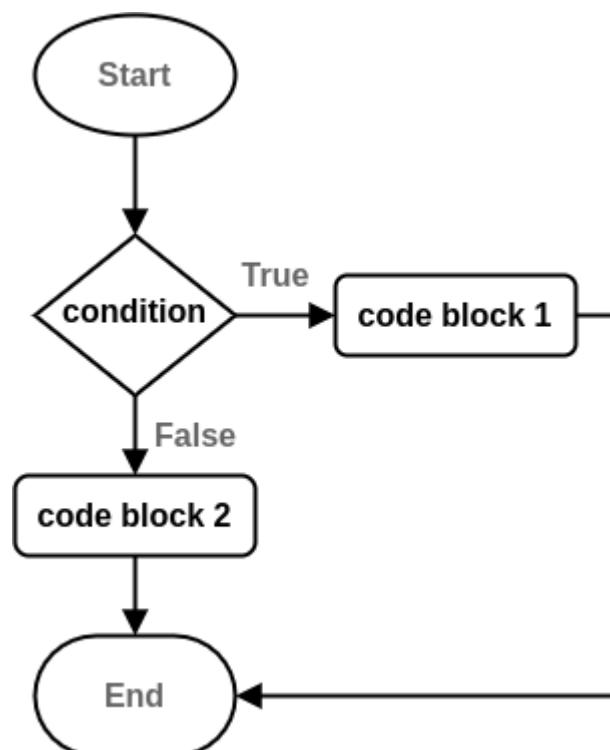


Fig. 8 Control flow diagram of the pseudo code if statement with an else part example.

### Note

The `else` statement cannot stand by itself. It requires a preceding if statement or loop for context.

## Worked Example

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

```
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

```
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 we had 2 conditions which are mutually exclusive (if one is true the other is necessarily false) and the one isn't just the converse of the other. For this we can use the `elif` part of the if statement (to be read "else if"):

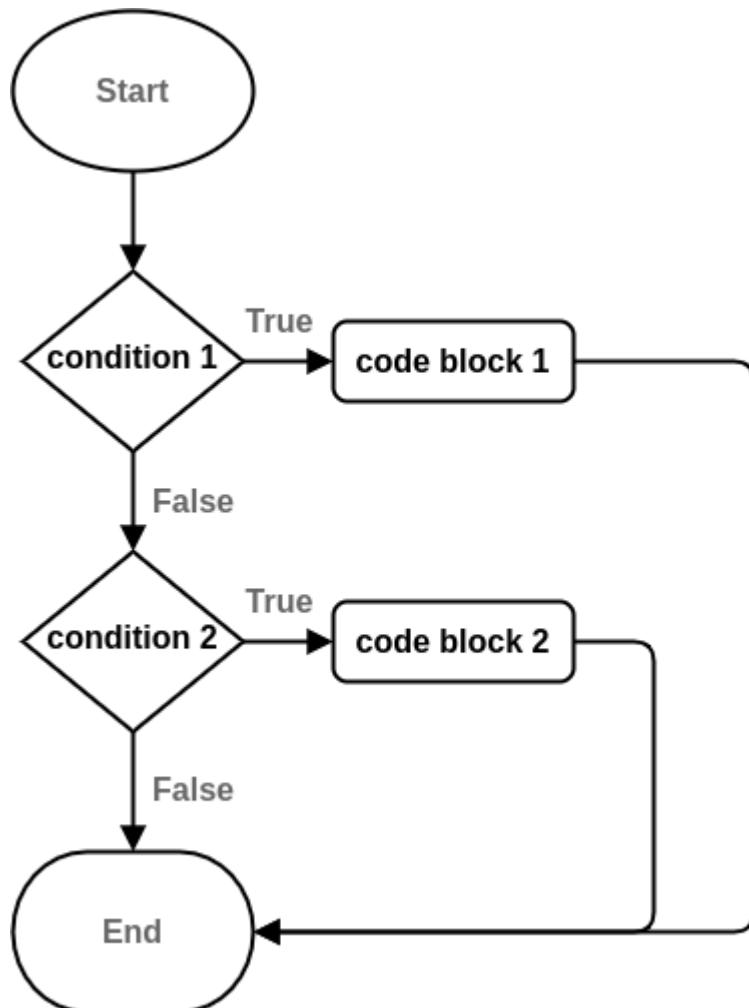
```
if condition1:
```

```

code_block_1
elif condition2:
    code_block_2

```

If `condition 1` is false, `condition 2` will be evaluated. If `condition 2` is found to be true, then `code block 2` will be executed, if not then control will move from the if statement. Illustrated as a control flow diagram:



*Fig. 9 Control flow diagram of the pseudo code example of an if statement with an elif part.*

## Worked Example

Let's continue with our worked example and change the `else` part to be more specific:

```
a = 1
b = 2

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

1 is less than 2

```
a = 1
b = 1

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

## Else After an Elif

Now, if we want to catch the case where both the conditions in the `if` and `elif` parts of the if statement are false, we can use an `else` part at the very end of the if statement. In pseudo code:

```
if condition1:
    code_block_1
elif condition2:
    code_block_2
else:
    code_block_3
```

Illustrated in a control flow diagram:

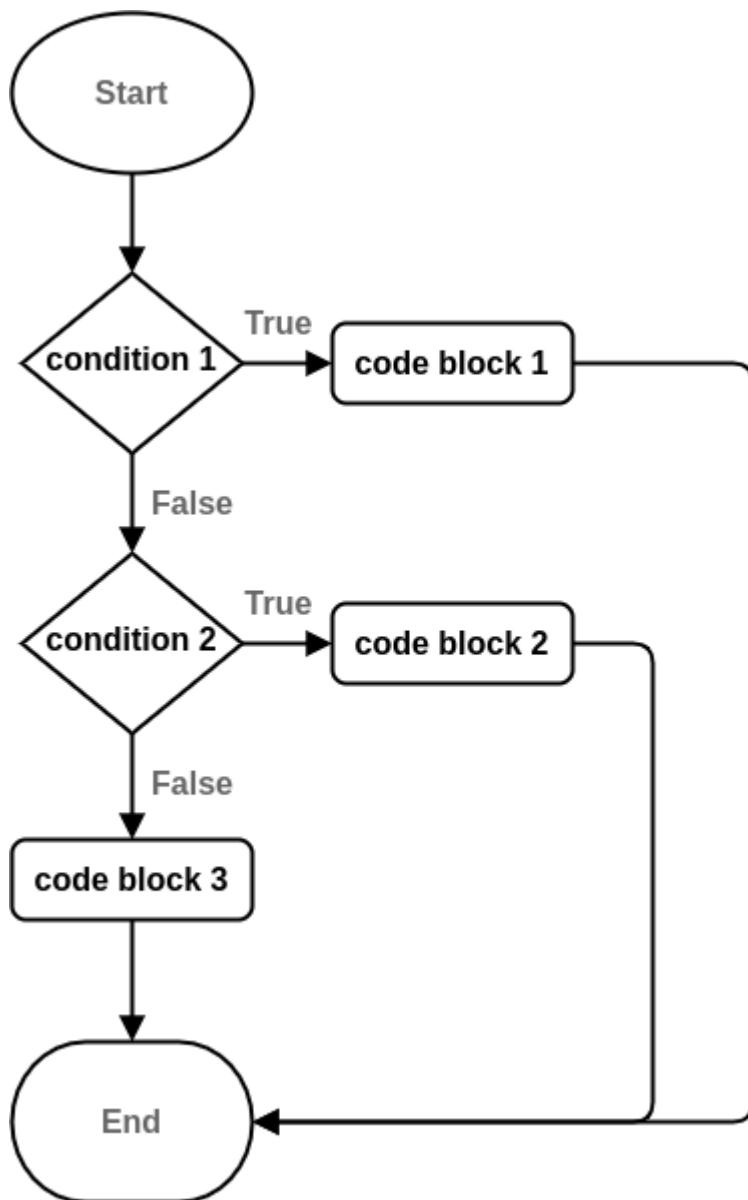


Fig. 10 Control flow diagram of the pseudo code example of an if statement with an elif and else part.

### ⚠ Warning

The `else` must **always** be the last part of the if statement and **there can only be one**.

### Worked Example

Now, let's re-introduce an `else` part to our worked example:

```
a = 1
b = 1

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 equal to 1

## Multiple Elif Parts

Though you may only use one `else` part in an if statement, you are not limited by how many `elif` parts you wish to use. For example:

```
if condition1:
    code_block_1

elif condition2:
    code_block_2

elif condition3:
    code_block_3

else:
    code_block_4
```

This can be illustrated using a control flow diagram:

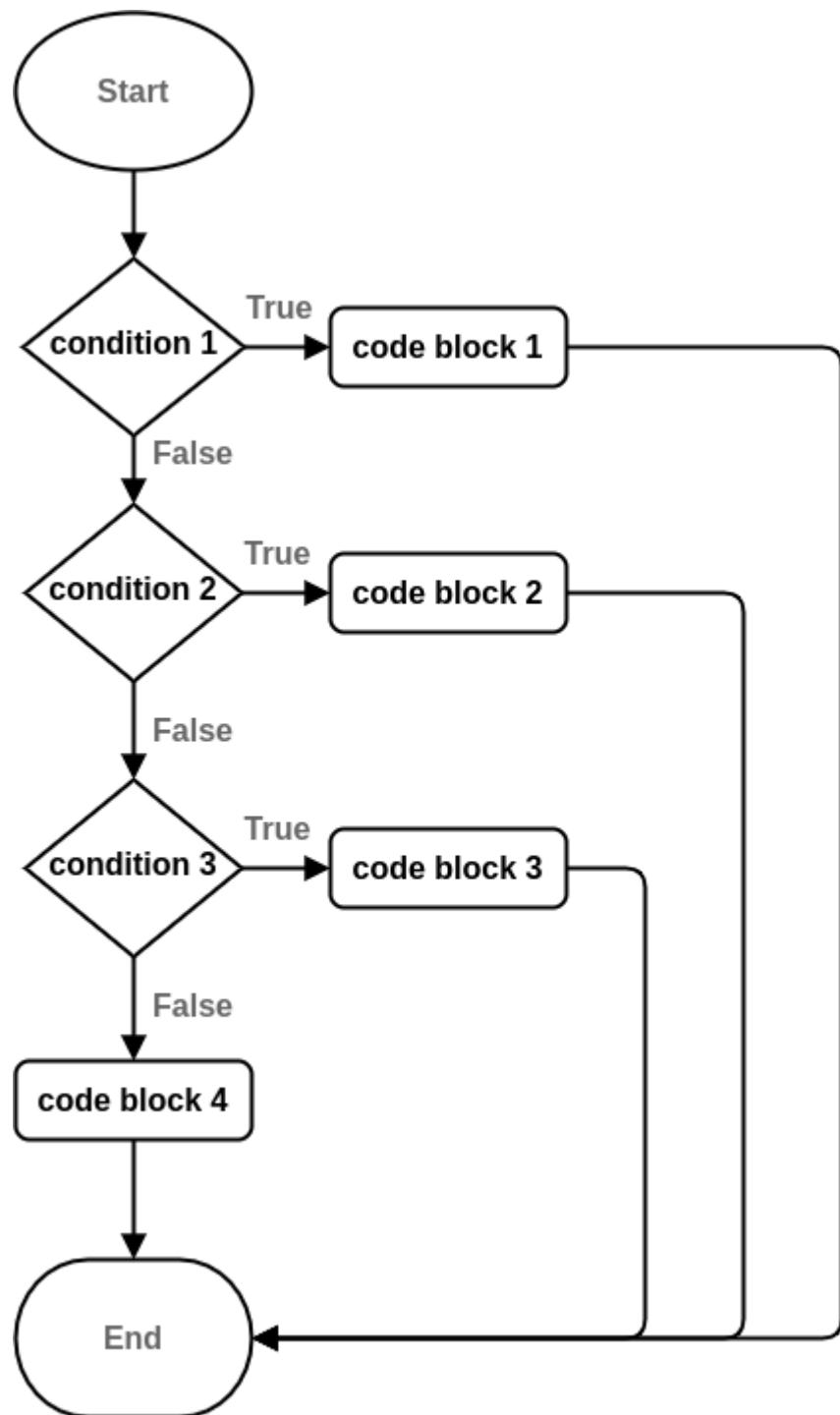


Fig. 11 Control flow diagram of the pseudo code example of an if statement with two elif parts and an else part.

## Worked Example

As an example of a script with multiple `elif` parts, let's write a script that checks if a variable is a multiple of 2, 3, or 5:

```
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')
else:
    print('Variable is not a multiple of 2, 3 or 5')
```

Variable is a multiple of 3

Note that if we put in a value that is both a multiple of 2 and 3, the script will only print out that it is a multiple of 2:

```
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')
else:
    print('Variable is not a multiple of 2, 3 or 5')
```

Variable is a multiple of 2

This is because the check to see if it's a multiple of 2 is placed before the check for 3 in the `if` statement.

# Nested If Statements

The code blocks inside the if statement can contain any valid Python code. This means that you can also nest other if statements inside an if statement. For example:

```
if condition1:  
    code_block_1  
  
    if condition2:  
        code_block_2  
    else:  
        code_block_3  
  
else:  
    code_block_4
```

which can be illustrated with the control flow diagram:

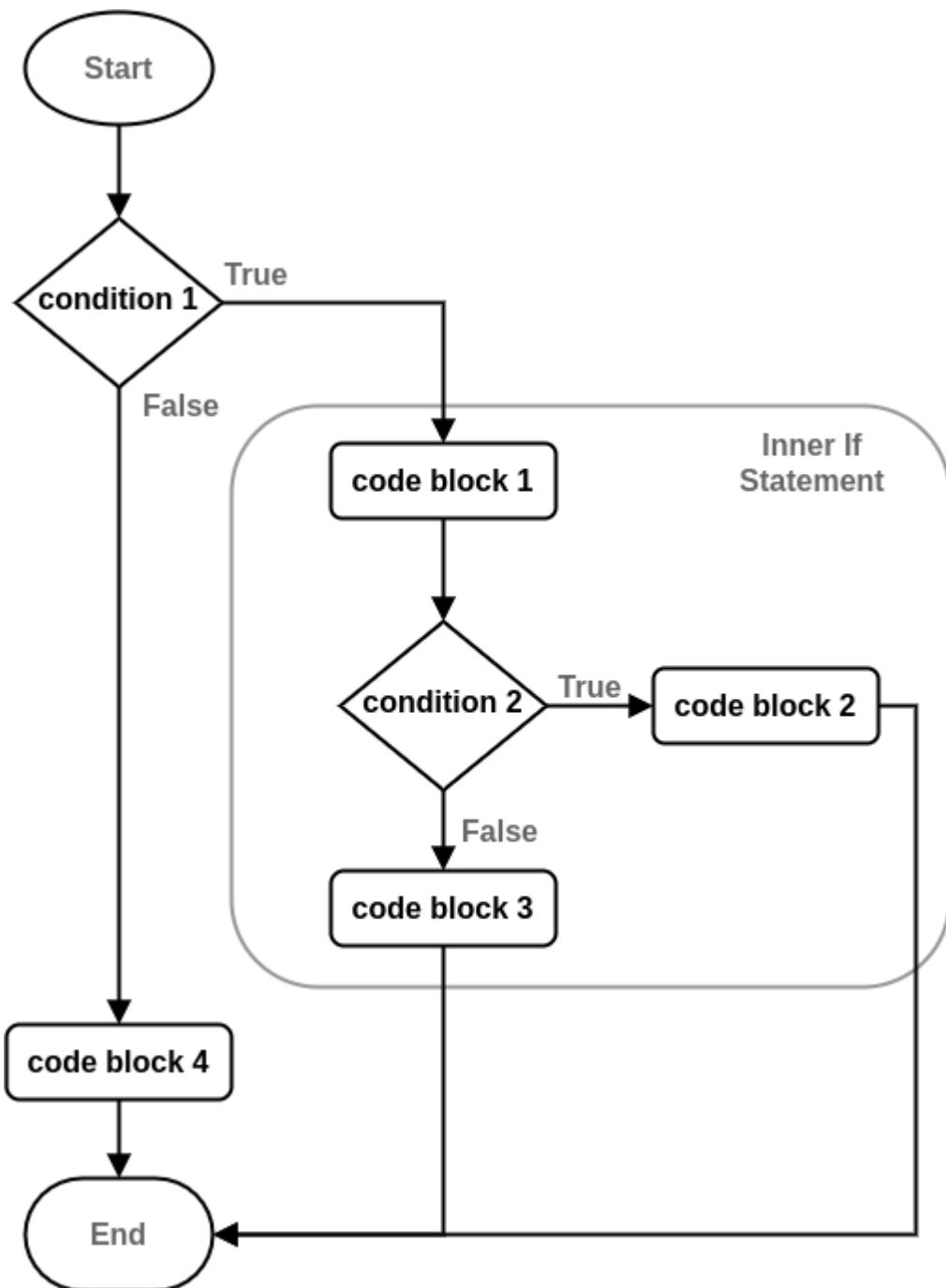


Fig. 12 Control flow diagram of the pseudo code example of an if statement nested inside an if statement.

### Hint

While there are situations which call for nested if statements, always consider whether a series of `elif`s will serve instead. Nested code blocks can make scripts harder to read.



# 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.

# While Loops

While loops are used to repeat a block of code **while** a given condition is true. If this condition is false (or becomes false), the code block will not be repeated. I will refer to individual loop repetitions as “**iterations**” (the first time the code in the loop is run is the **first iteration**, the next time is the **second iteration**, etc.).

The syntax for a **while** loop is:

```
while condition:  
    code block
```

Note the use of the **while** statement and the **:** following the condition. All the code indented after the **:** is inside the loop, represented by **code block** here. The **condition** used in the loop must either evaluate to or be a boolean value.

In each loop iteration **condition** is evaluated. If **condition** is found to be **true** then the loop will go through another iteration, executing **code block**, and checking for another iteration. If **condition** is found to be **false**, then control will leave the loop and it will not undergo another iteration.

Note that if **condition** starts as false, then **code block** will never be executed.

The while loop can be illustrated with the control flow diagram:

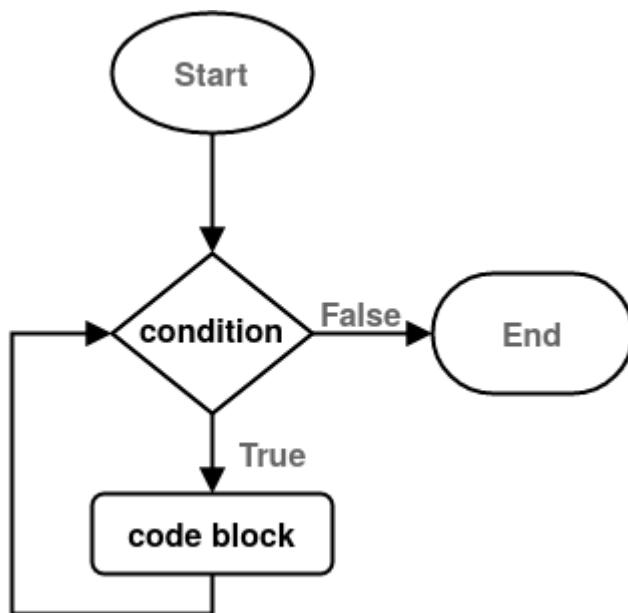


Fig. 13 Control flow diagram of the while loop.

## Worked Example

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

$$\begin{aligned} T_n &= T_{n-1}^{3/4} \\ T_0 &= 100 \end{aligned}$$

Let's say 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:

```

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).

# For Loops

For loops can be used to repeat a block of code by iterating through specified values. Python for loops are technically “foreach” loops. Though the distinction is important in other programming languages, we will refer to these as “for” loops for the entirety of the book.

The syntax for a for loop is:

```
for i in iterable:  
    code block
```

As with the while loop, code indented after the `:` is inside the loop and will be executed with each loop iteration (here this code is represented by `code block`).

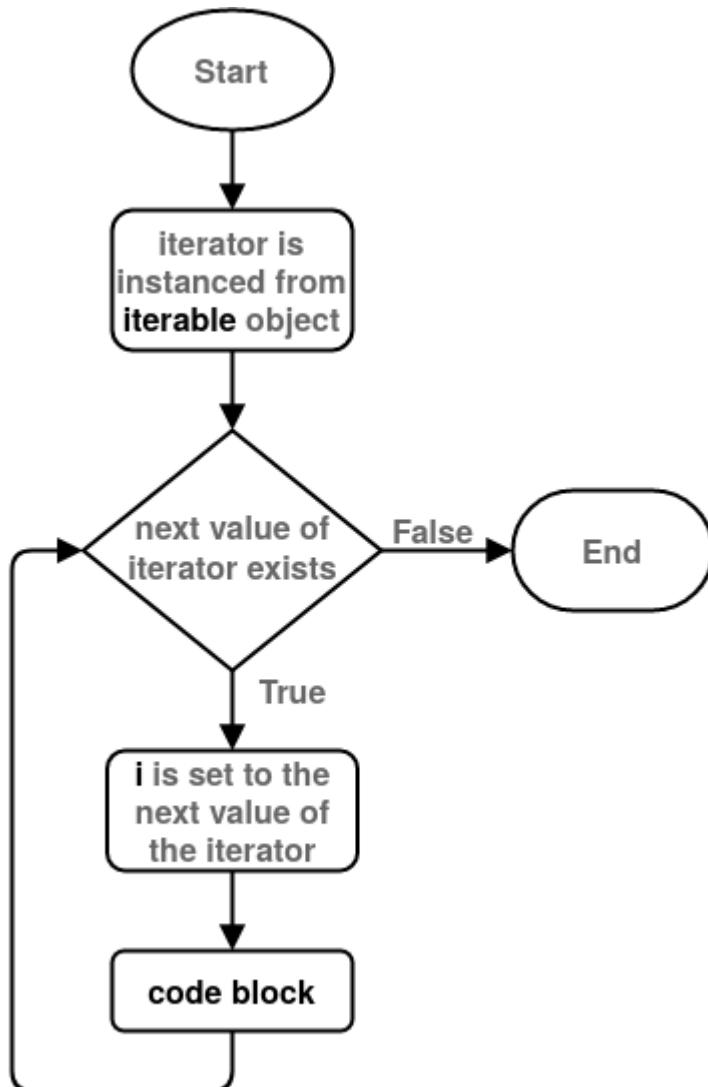
The `iterable` is an **iterable** object. These are objects that can be prompted to return a sequence of objects. Strings, lists and tuples are all examples of iterable objects.

When the loop is executed Python instances a new object from `iterable` called an **iterator**. The iterator is asked for the next object in the sequence before each loop iteration, until there isn’t a next object and control leaves the loop.

`i` is referred to as the **iteration variable**. It can be seen as a variable, and can be given any viable variable name. Before each loop iteration `i` is set to the next value of the iterator. This value can then be used in the `code block` by referring to `i` as a variable.

**Note:** be careful not to alter `iterable` inside the loop (`code block`), as this will cause an error.

The for loop can be illustrated using the control flow diagram:



*Fig. 14 Control flow diagram of the for loop.*

Note that elements of Fig. 14 have been simplified.

## Looping Through Sequences

As mentioned, sequences such as tuples, lists and strings can also be used as iterables. For example, if we want to print the individual characters of a string separately:

```
for char in 'This string':
    print(char)
```

T  
h  
i  
s  
  
s  
t  
r  
i  
n  
g

Note that we called the iteration variable `char`. We can give it any name we want.

Similarly we can print each object in a list:

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

1  
2  
3  
a  
b  
c

## The `range()` Function

If you need to loop through a sequence of values that aren't already in an instanced data collection, you can use the `range()` function. The `range()` function takes integer arguments and returns an iterable that produces a series of integers. As we shall see, the `range()` function's arguments are very similar to slicing.

Range can be used with a single argument:

```
range(stop)
```

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

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

```
0
1
2
3
4
```

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:

```
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:

```
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:

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

11  
9  
7  
5

## Useful Functions for Iterables

### `enumerate()` To Iterate Through Sequence and Index

Sometimes you want to loop through a sequence but also want to keep track of the index. The most convenient way to do this is using the `enumerate()` function:

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

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

where the iteration value is a tuple of the index and value from the original iterator (unpacked as `i` and `char` respectively).

Alternatively, the same can be achieved using the range function and indexing the iterable sequence manually:

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

```
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 can use the **zip** function:

```
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.

This can also be achieved manually:

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

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

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

## Looping Through Dictionaries

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

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

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

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

You can also loop through only the keys:

```
for k in d.keys():
    print(k, d[k])
```

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

or only the values:

```
for v in d.values():
    print(v)
```

54  
754  
42

Note that the values returned from `dictionary.values()` won't always appear in the same order. If this is a requirement it is better to use an ordered dictionary.

## Using a `while` Loop in Place of a `for` Loop

Although it's not as clean, you can use while loops in place of for loops. We will not cover how to do this in general here , a simple example is using a while loop instead of a for loop with a `range()` iterable:

```
#For loop:  
print('For loop')  
  
for i in range(5):  
    print(i)  
  
#Equivalent 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

# Nested Loops

Sometimes it is necessary to nest loops. Loops in Python can be quite slow compared to other programming languages, so unnecessary loops should be avoided.

When nesting loops, it's best to think of the inner loop as a block of code. In every iteration of the outer loop, the inner loop will execute, iterating to completion.

Consider the pseudo code example of a while loop nested in another while loop:

```
while condition1:  
    while condition2:  
        code block
```

This can be illustrated with the control flow diagram:

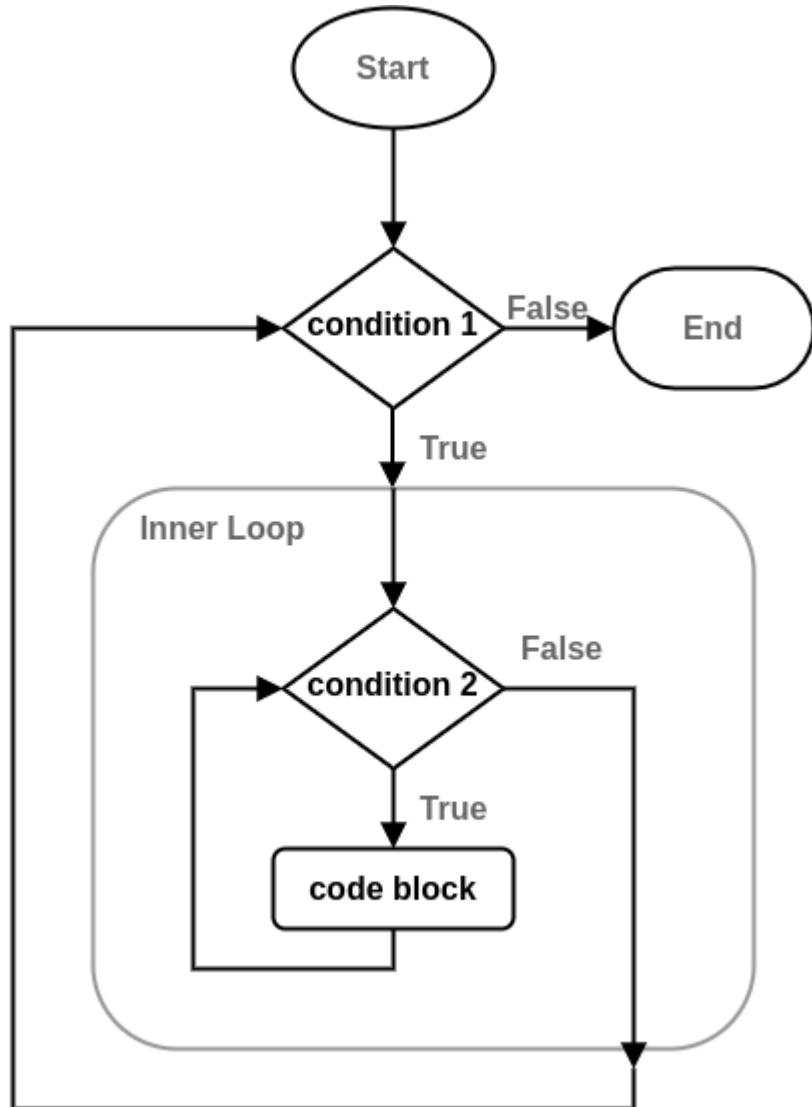


Fig. 15 Control flow diagram of the nested while loop example above.

Let's take a closer look with a more concrete example. We will nest a for loop inside a for loop and print the iteration variables of each:

```
for i in range(3):
    print('Outer loop iteration:', i)

    for j in range(4):
        print('    Inner loop iteration:', j)
```

```
Outer loop iteration: 0
    Inner loop iteration: 0
    Inner loop iteration: 1
    Inner loop iteration: 2
    Inner loop iteration: 3
Outer loop iteration: 1
    Inner loop iteration: 0
    Inner loop iteration: 1
    Inner loop iteration: 2
    Inner loop iteration: 3
Outer loop iteration: 2
    Inner loop iteration: 0
    Inner loop iteration: 1
    Inner loop iteration: 2
    Inner loop iteration: 3
```

Note that you are not limited in what you nest inside 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

The `break` statement causes control to exit the loop it is situated in. For example, if we were to put a break directly in the loop's code block:

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

0

You can see that only the first loop iteration completed.

Let's illustrate this with a pseudo code example:

```
while condition1:
    code_block_1

    if condition2:
        break

    code_block_2
```

In the example above `condition2` is evaluated as `true` in a loop iteration, then control will exit the loop before executing `code_block_2` for that iteration.

Illustrating this using a control flow diagram:

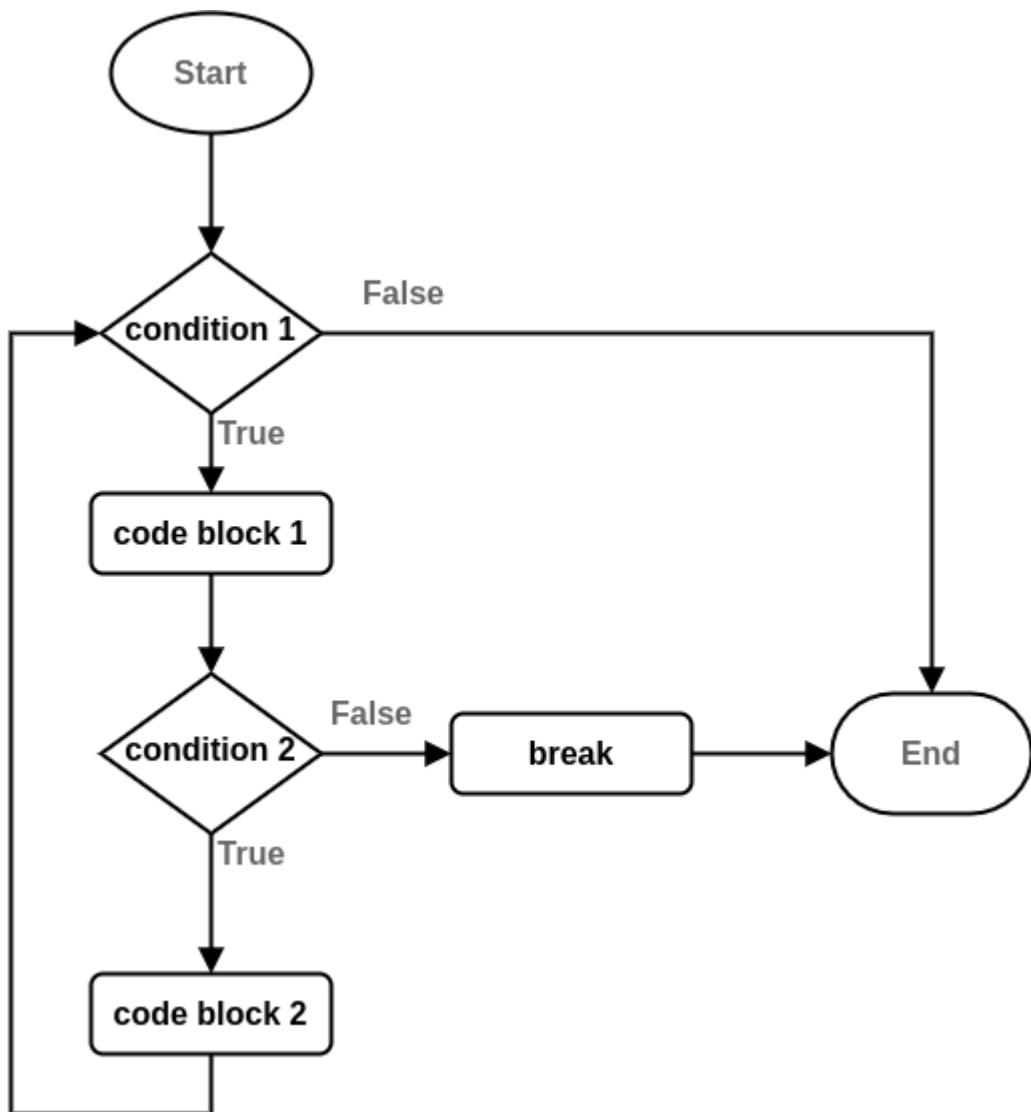


Fig. 16 Control flow diagram of the break example above.

Let's consider a similar script that loops through a sequence of numbers and stops when it reaches

5:

```
for i in range(10):
    print(i)

    if i == 5:
        break

    print('next iteration')
```

```
0
next iteration
1
next iteration
2
next iteration
3
next iteration
4
next iteration
5
```

As you can see in the example above the loop terminated before it finished iterating through `range(10)`. When `i` had a value of `5` the `break` statement was called, exiting from the loop.

Note that it doesn't matter that the `break` is nested in an if statement, it will always make control exit the nearest loop that it is nested in.

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

```

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.

Consider the pseudo code example:

```

while condition1:
    code_block_1

    if condition2:
        continue

    code_block_2

```

Here if `condition2` is found to be **true** in a loop iteration, the `continue` statement will cause control to move directly to the next loop iteration, skipping `code_block_2` for that iteration.

Illustrating this example in a control flow diagram:

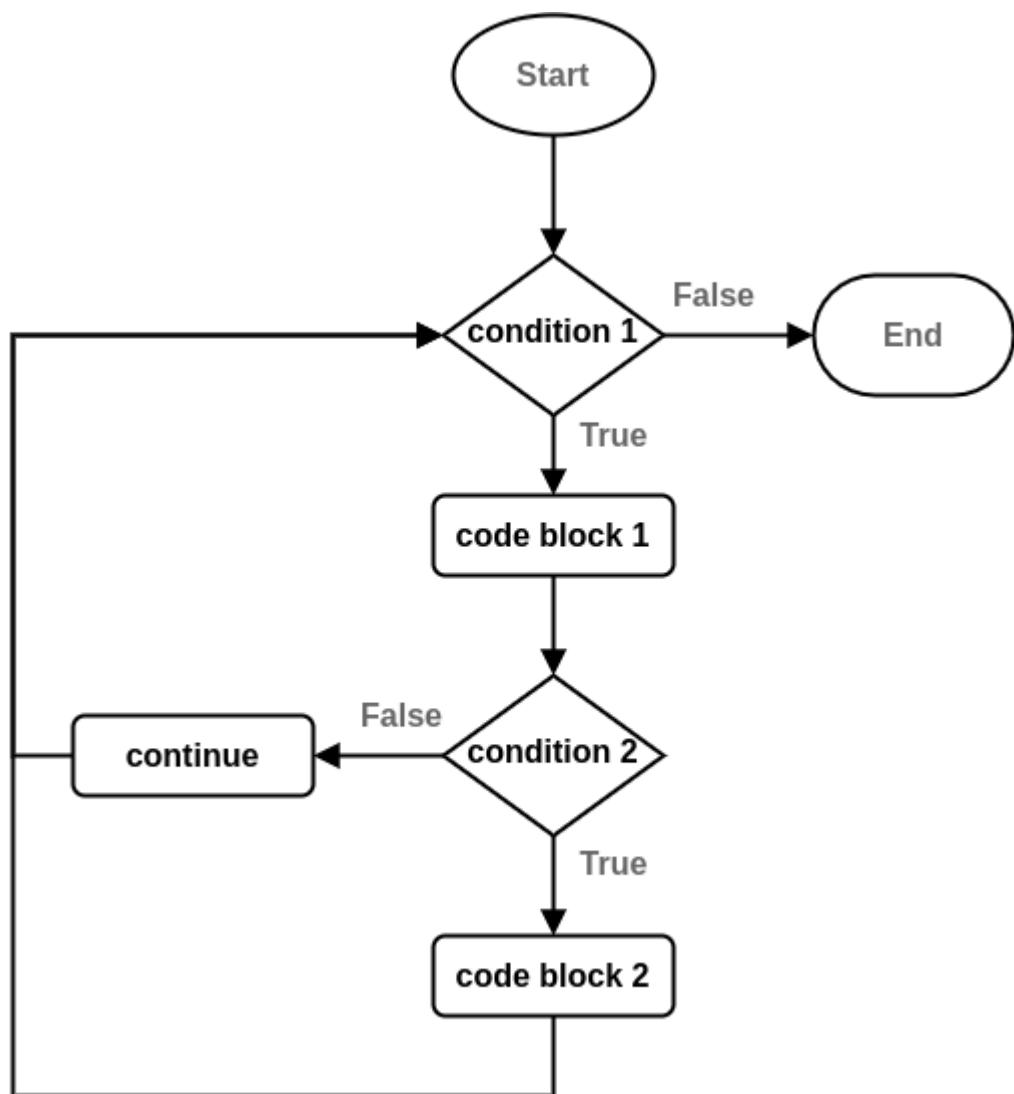


Fig. 17 Control flow diagram of the continue example above.

Let's look at real code example:

```
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 as the continue statement has caused control to 'skip' the print statement.

# 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.

```
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:

```
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:

```
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.

```
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:

```
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.

# 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

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

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

```
double(1)
```

2

double(5.5)

11.0

double('a')

'aa'

# return Statement

As was already mentioned, the `return` statement can be used to return values from a function call. There are more properties of the `return` statement that are worth covering.

## 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.

```
def none1():
    return

def none2():
    return None

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

`type(none1())`

NoneType

`type(none2())`

NoneType

`type(none3())`

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:

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

```
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 its argument is even or odd:

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

```
is_even(3)
```

False

```
is_even(6)
```

True

The else part of the function is unnecessary:

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

```
is_even(3)
```

False

## return Tuples

Although not a feature of the `return` statement itself, it's worth noting that tuple packing can be used for the value returned. This can be particularly useful when you need to return multiple values.

For example consider a function that splits a floating point number into its whole and decimal parts, returning both:

```
def floor_rem(num):
    whole = int(num)
    frac = num - whole

    return whole, frac
```

```
floor_rem(2.13)
```

(2, 0.1299999999999999)

Note that tuple unpacking can also be used to collect the return values of the function into multiple variables:

```
num = 3.14

whole, frac = floor_rem(num)

print('Number:', num)
print('Whole part:', whole)
print('Decimal part:', frac)
```

```
Number: 3.14
Whole part: 3
Decimal part: 0.1400000000000012
```

# 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.

```
def arg3(arg1, arg2, arg3):  
    print(arg1)  
    print(arg2)  
    return arg3
```

```
arg3(1, 2, 3)
```

```
1  
2
```

```
3
```

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

```
var1 = 45  
arg3(var1, 3*4, 7)
```

```
45  
12
```

```
7
```

Note that for the function defined above, all of the arguments need to be passed into the functions:

```
arg3('a', 'b')
```

```
-----
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_11308\4179836295.py in <module>
----> 1 arg3('a', 'b')

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

## Default Values for Function Arguments

You can assign default values to arguments by using the `=` operator, for example:

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

(If you are unfamiliar with f-strings `f''`, see the page on [The Python Standard Library/Strings/String Formatting](#))

If `hello()` is called without passing arguments, the values defined in the function will be used:

```
hello()
```

```
'Hello World! How are you today?'
```

alternatively you pass in new values:

```
hello('reader', 'feeling')
```

```
'Hello reader! How are you feeling?'
```

Not all arguments need be replaced:

```
hello('reader')
```

```
'Hello reader! How are you today?'
```

You don't need to assign default values for every argument, for example:

```
def hello_hello(num, name = 'World', time = 'today'):
    return f"Hello {num*'hello '}{name}! How are you {time}?"
```

Note that the `num` argument needs to have a value passed into it, but the others do not:

```
hello_hello(1)
```

```
'Hello hello World! How are you today?'
```

Also note that an argument without a default value cannot be defined after an argument that does:

```
def hello_again(name = 'World', time = 'today', num):
    return f"Hello {num*'hello '}{name}! How are you {time}?"
```

```
File "C:\Users\mayhe\AppData\Local\Temp\ipykernel_11308\438166721.py", line 1
    def hello_again(name = 'World', time = 'today', num):
               ^
SyntaxError: non-default argument follows default argument
```

## Positional and Keyword Arguments

In the examples above, we have been entering argument values directly into the function in the order in which they are defined. These are called **positional arguments**. Alternatively, arguments can be passed in by name:

```
arg3(arg1 = 1, arg2 = 2, arg3 = 3)
```

1  
2

3

These are called **keyword arguments**. Keyword arguments can be passed into the function in any order:

```
arg3(arg2 = 1, arg3 = 2, arg1 = 3)
```

3  
1

2

Both positional and keyword arguments can be used in the same function call:

```
arg3(1, arg2 = 2, arg3 = 3)
```

1  
2

3

Note that positional arguments cannot be passed into a function after keyword arguments:

```
arg3(arg2 = 1, arg1 = 2, 3)
```

```
File "C:\Users\mayhe\AppData\Local\Temp\ipykernel_11308\2581517664.py", line 1
    arg3(arg2 = 1, arg1 = 2, 3)
               ^
SyntaxError: positional argument follows keyword argument
```

## Unpacking Data Structures As Arguments

You can unpack data structures to be passed into functions as individual arguments, this can be useful if you need to store the values of arguments to be used in functions later. Tuples and lists can be unpacked using the `*` operator, and will behave like positional arguments:

```
print( *(1, 2, 3) ) #Unpacking a tuple as 3 positional arguments
print( *['a', 'b', 'c'] ) #Unpacking a list as 3 positional arguments
```

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

Dictionaries can be unpacked as keyword arguments using the `**` operator, using our `hello` function from above:

```
hello_hello(2, **{'name' : 'everybody', 'time' : 'you all doing'}) #Unpacking a dictionary
```

```
'Hello hello hello everybody! How are you you all doing?'
```

## Defining Functions That Take Arbitrary Many Positional and Keyword Arguments

You can define functions that take an arbitrary number of positional or keyword arguments. So far we have encountered this in the `print()` function which can take an arbitrary amount of positional arguments.

# Positional Arguments

You can collect arbitrarily many positional arguments by using the `*` operator before the argument name. Non-keyword arguments that are passed into the function from the position of this argument on will be collected as a tuple and passed in as this argument. For example:

```
def argn(*args):
    for arg in args:
        print(arg)
```

```
argn(1, 2)
```

```
1
2
```

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

```
1
2
3
4
5
```

As implied above, any positional arguments that are defined before the `*args` argument will be collected separately:

```
def arg2n(arg1, arg2, *args):
    print('Arg1:', arg1)
    print('Arg2:', arg2)

    print('Args:')
    for arg in args:
        print('    ', arg)
```

```
arg2n('a', 'b', 1, 2, 3)
```

```
Arg1: a
Arg2: b
Args:
  1
  2
  3
```

Note that arguments defined after `*args` must be passed into the function as keyword arguments:

```
def arg2n1(arg1, arg2, *args, arg3):
    print('Arg1:', arg1)
    print('Arg2:', arg2)

    print('Args:')
    for arg in args:
        print('    ', arg)

    print('Arg3:', arg3)
```

```
arg2n1(1, 2, 3, 4, 5)
```

```
-----
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_11308\220825871.py in <module>
----> 1 arg2n1(1, 2, 3, 4, 5)
```

```
TypeError: arg2n1() missing 1 required keyword-only argument: 'arg3'
```

```
arg2n1(1, 2, 3, 4, 5, arg3 = 6)
```

```
Arg1: 1
Arg2: 2
Args:
  3
  4
  5
Arg3: 6
```

If you define arguments with default values, these must appear after the `*args` argument:

```
#TODO: rename or replace this with something a little more sensible
def arg2nkw(arg1, arg2, *args, kwarg1 = 'kwarg1', kwarg2 = 'kwarg2'):
    print('Arg1:', arg1)
    print('Arg2:', arg2)

    print('Args:')
    for arg in args:
        print('    ', arg)

    print('Kwarg1:', kwarg1)
    print('Kwarg2:', kwarg2)
```

```
arg2nkw('a', 'b', 1, 2, 3, 4, 5, kwarg1 = 'c', kwarg2 = 'd')
```

```
Arg1: a
Arg2: b
Args:
    1
    2
    3
    4
    5
Kwarg1: c
Kwarg2: d
```

## Keyword Arguments

You can collect arbitrary keyword arguments using the `**`

```
def kwargs(arg1, arg2, **kwargs):
    print('Arg1', arg1)
    print('Arg2', arg2)
    print(kwargs)
```

```
kwargs(1, 2, kwarg1 = 2, kwarg3 = 3)
```

```
Arg1 1
Arg2 2
{'kwarg1': 2, 'kwarg3': 3}
```

Note that no other arguments can be used after `**kwargs`:

```
def kwargs_arg(arg1, **kwargs, arg2):  
    pass
```

```
File "C:\Users\mayhe\AppData\Local\Temp\ipykernel_11308\1522776780.py", line 1  
  def kwargs_arg(arg1, **kwargs, arg2):  
          ^
```

```
SyntaxError: invalid syntax
```

# Local Namespace and Variables

Variable names are stored in a **namespace**, and must be unique within a namespace. Variables that are defined in the main body of a script are stored in the **global namespace** and are referred to as **global** variables.

The global namespace is accessible both in the script body and inside of functions:

```
x = 5  
def get_x():  
    return x
```

```
get_x()
```

```
5
```

however, when variables are defined inside of the function (including arguments) these are only accessible within that function. For example:

```
def make_var():  
    func_var = 4  
    return func_var
```

```
make_var()
```

```
4
```

```
func_var
```

```
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_13780\2269113617.py in <module>
      1 func_var

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

This is because a function has its own namespace, referred to as a **local namespace**. Variables in the global namespace are still accessible inside the local namespace, but variables defined in the local namespace are not available in the global namespace (or any other namespaces that encompasses it).

If a variable is defined in a local namespace with the same name as a variable in the global namespace, then the local variable will be used by default.

For example, if we were to define `func_var` as a global variable, `make_var()` will instance a local variable instead of reassigning a new value to the global variable:

```
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
```

In other words Python will check the local namespace **before** the global namespace.

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

```
def arg_var(x):
    return x
```

```
x = 5  
arg_var(2)
```

2

### ⚠ Warning

Although variable names are restricted to namespaces, the objects that they represent are not. This is not so important to keep in mind with numbers and strings (which are passed in by value), but it is important for objects that can be altered, such as arrays and dictionaries.

For example, if we were to pass a dictionary into a function, any alterations made to that dictionary inside the function would remain outside (as there is only one dictionary object being used).

```
def add_to_dict(d):  
    d['key'] = 'value'  
    return d  
  
d = {}  
  
print('Dictionary before the function call:', d)  
print('Function return:', add_to_dict(d))  
print('Dictionary after the function call:', d)
```

```
Dictionary before the function call: {}  
Function return: {'key': 'value'}  
Dictionary after the function call: {'key': 'value'}
```

# 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

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:

```
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 its recursion.

Putting this function into action:

```
factorial(-1)
```

n must be greater than or equal to 0

```
factorial(0.5)
```

n must be an integer

```
factorial(0)
```

1

```
factorial(1)
```

1

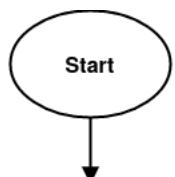
```
factorial(5)
```

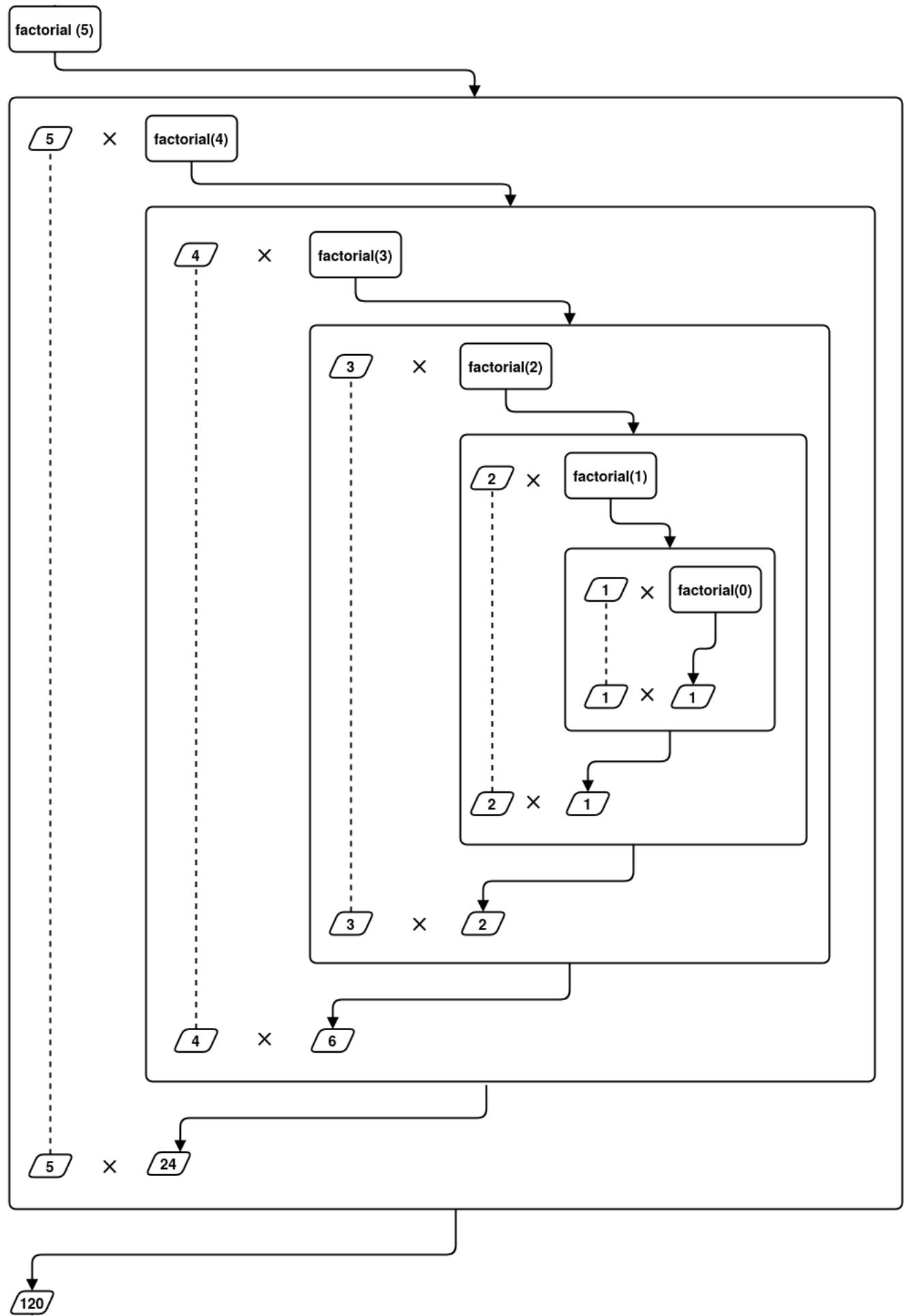
120

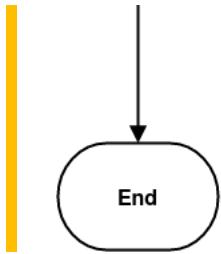
```
factorial(10)
```

3628800

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







## 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

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

## 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

```
open(filename, mode = 'r')
```

## File Modes

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

Character	Meaning
'r'	open for reading ( <b>default</b> )
'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 ( <b>default</b> )
'+'	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:

```
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'`:

```
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:

```
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:

```
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.

```
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:

```
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:

```
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:

```
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:

```
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()
```

# 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.

```
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 [The Python Standard Library/Strings/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
0.7693696294505615,0.6956825506034864,0.7183493500977277
0.8975979010256552,0.7818314824680298,0.6234898018587336
1.0258261726007487,0.8551427630053461,0.5183925683105252
```

Or in a more presentable format:

<b>x</b>	<b>sin(x)</b>	<b>cos(x)</b>
0.000000	0.000000e+00	1.000000
0.128228	1.278772e-01	0.991790
0.256457	2.536546e-01	0.967295
0.384685	3.752670e-01	0.926917
...	...	...
5.898500	-3.752670e-01	0.926917
6.026729	-2.536546e-01	0.967295
6.154957	-1.278772e-01	0.991790
6.283185	-2.449294e-16	1.000000

## 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:

```
'a b c d'.split()
```

```
['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:

```
'a,b,c,d'.split(',')
```

```
['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.

```
#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 separator
        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:

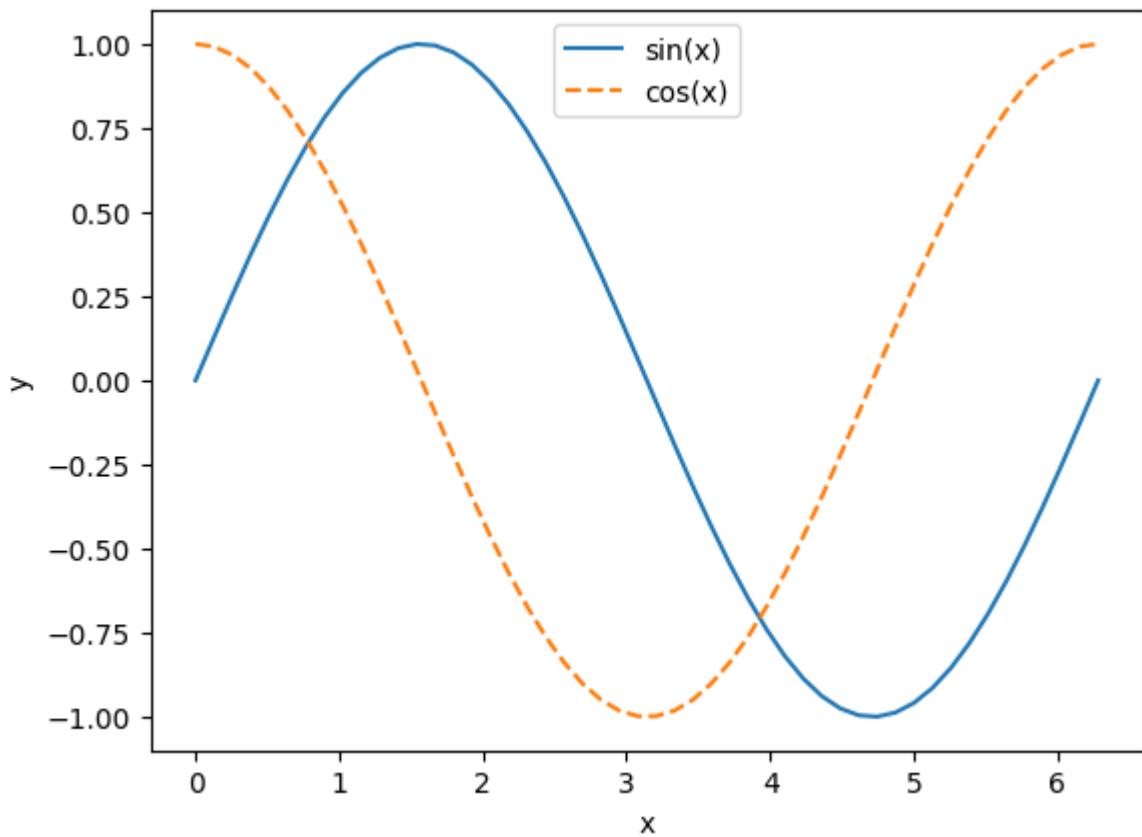
```
import matplotlib.pyplot as plt

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).

```

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
0.0202020202020204	0.1421338109037403	0.0004081216202428324
0.04040404040404041	0.20100756305184242	0.0016324864809713297
0.06060606060606061	0.24618298195866548	0.0036730945821854917
0.08080808080808081	0.2842676218074806	0.006529945923885319
0.101010101010102	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.

```
#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 separator
        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:

```

import matplotlib.pyplot as plt

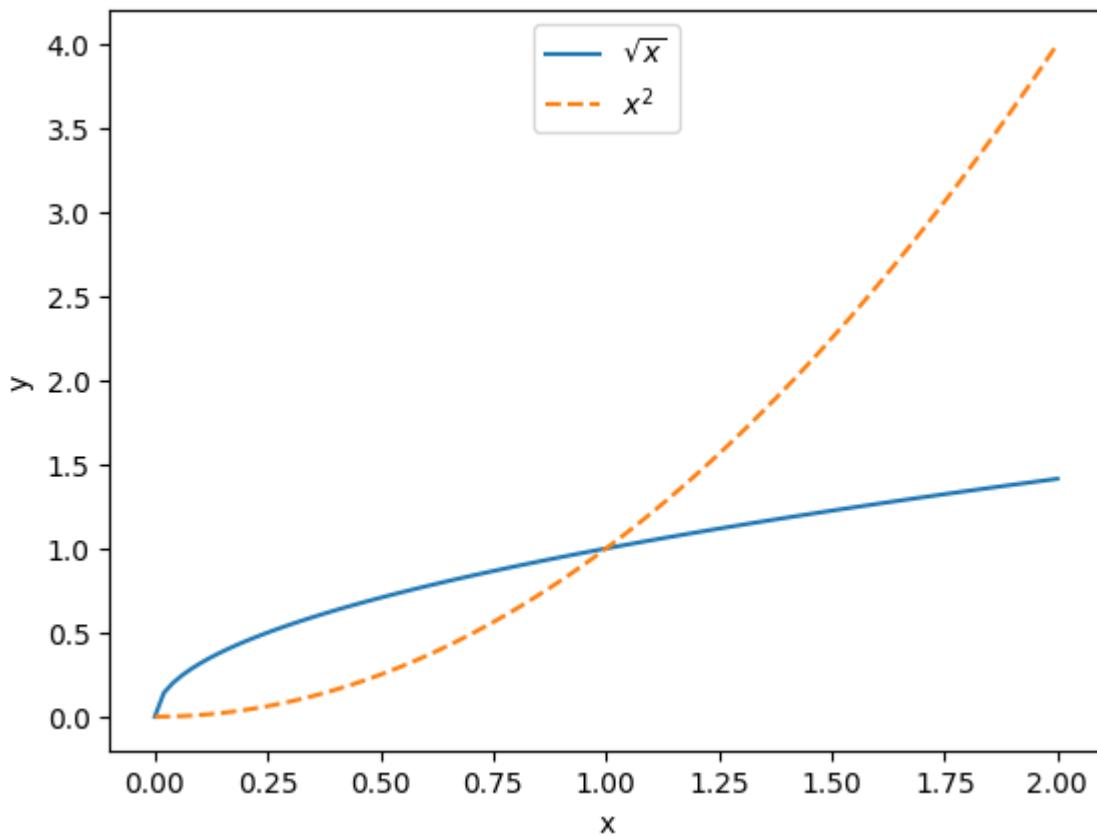
fig, ax = plt.subplots()

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:

```
#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)
```

```
C:\Users\mayhe\anaconda3\envs\jb\lib\site-packages\ipykernel_launcher.py:13: DeprecationWarning:
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html
del sys.path[0]
```

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

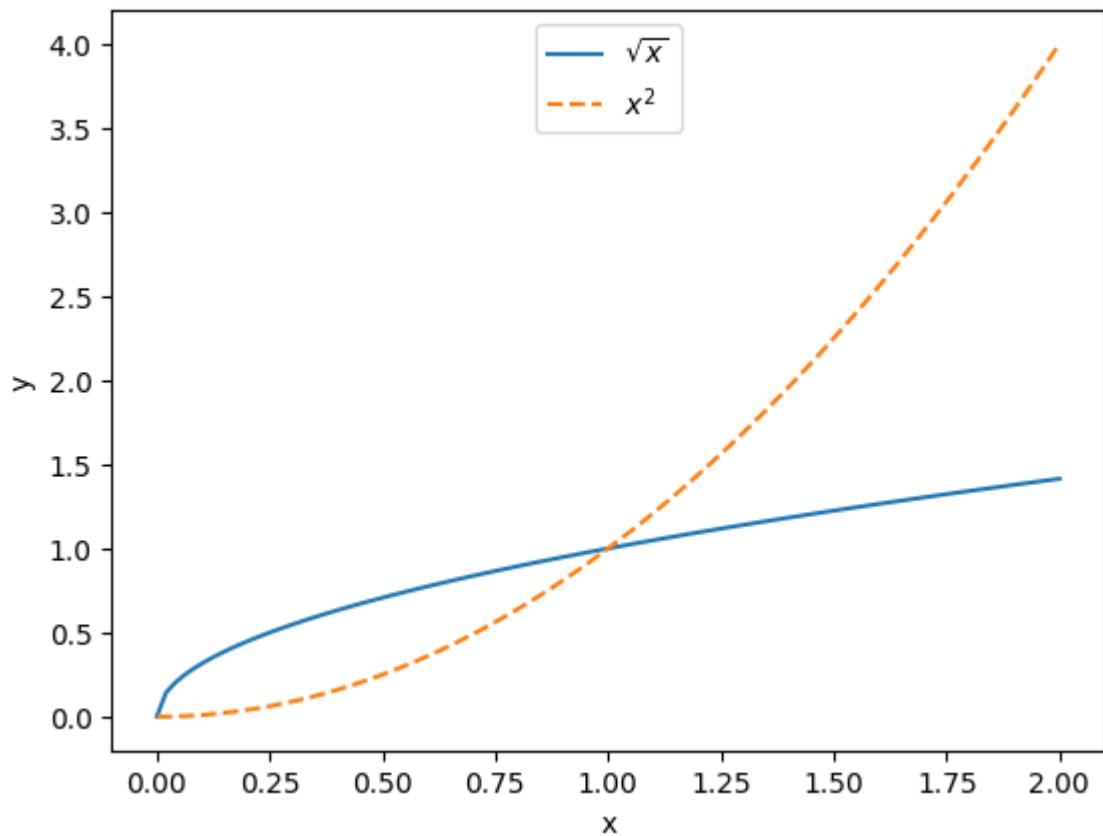
```
import matplotlib.pyplot as plt

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()
```



## Part

---

### Scientific Packages

# Numpy

The NumPy package provides us with arrays and matrices (efficient data structures), special 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`:

```
import numpy as np
```

# 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:

```
#Array of integers
np.array([1, 2, 3, 4])
```

```
array([1, 2, 3, 4])
```

```
#Array pf strings
np.array(['a', 'b', 'c'])
```

```
array(['a', 'b', 'c'], dtype='<U1')
```

Remember that arrays are homogeneous:

```
#Trying to create an array with different types
np.array([1, 2.3, 'x'])
```

```
array(['1', '2.3', 'x'], dtype='|U32')
```

## Indexing and Slicing

As said before, arrays can be indexed and sliced similarly to lists and strings

```
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:

```
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:

```
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:

```
2*np.array([1, 2, 3, 4])
```

```
array([2, 4, 6, 8])
```

```
np.array([1, 4, 5]) + 1
```

```
array([2, 5, 6])
```

Array on array operations are also performed element by element:

```
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.

```
#More efficient:
print(arr1, '+', arr2, 'is', arr1 + arr2)
```

```
[1 2 3 4] + [2 4 6 8] is [ 3 6 9 12]
```

```
#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:

```
#1 argument: stop
np.arange(5)
```

```
array([0, 1, 2, 3, 4])
```

```
#2 arguments: start, stop
np.arange(1, 5)
```

```
array([1, 2, 3, 4])
```

```
#3 arguments: start, stop, step
np.arange(1, 10 ,2)
```

```
array([1, 3, 5, 7, 9])
```

Unlike the `range()` function, `np.arange()` also allows for floating point values:

```
np.arange(2.3, 3, 0.1)
```

```
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.

```
np.linspace(0, 1, 10)
```

```
array([0.          , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

## np.zeros()

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:

```
np.zeros(5)
```

```
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.

```
np.ones(7)
```

```
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

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:

```
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:

```
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.

```
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).

```
print('Average height: ', heights.mean())
print('Standard deviation of heights: ', heights.std())
```

Average height: 1.712684356  
 Standard deviation of heights: 0.18476698650385862

```
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

```
import numpy as np
```

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:

```
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:

```
print(np.array([[1, 2, 3], [4, 5]]))
```

```
[list([1, 2, 3]) list([4, 5])]
```

```
C:\Users\mayhe\anaconda3\envs\jb\lib\site-packages\ipykernel_launcher.py:1: VisibleDeprecationWarning:  
    """Entry point for launching an IPython kernel.
```

## Shape and Size

Now would be a good time to talk about the distinction between the `shape` and `size` attributes of an array.

```
arr = np.array(
    [[0, 1],
     [0, 1],
     [0, 1]])
```

The `size` of the array is a count of how many elements the array contains.

```
arr.size
```

6

The `shape` of an array is a tuple which tells you the length of each axis:

```
arr.shape
```

(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:

```
np.ones( (3, 6) )
```

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

```
np.zeros( (5, 2) )
```

```
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:

```
arr = np.array(
    [[1, 2, 3, 4],
     [5, 6, 7, 8],
     [9, 10, 11, 12]])
)
```

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

7

```
arr[2, -1]
```

12

You can slice multidimensional arrays by separating the slice along each axis by commas:

```
arr[:, 1:3]
```

```
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:

```
#Slicing the first row
arr[0, :]
```

```
array([1, 2, 3, 4])
```

```
#Slicing the third column
arr[:, 2]
```

```
array([ 3,  7, 11])
```

```
#Slicing the last column
arr[:, -1]
```

```
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):

```
arr.T
```

```
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:

```
np.matrix(
    [[1, 2],
     [3, 4],
     [5, 6]])
)
```

```
matrix([[1, 2],
       [3, 4],
       [5, 6]])
```

To generate large, structured matrices, you can use some of the array generating functions:

```
np.matrix(np.ones( (2, 3) ))
```

```
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`.

```
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):

```
mat1.T + mat2
```

```
matrix([[2., 3.],
       [4., 5.],
       [6., 7.]])
```

```
mat1.T/mat2
```

```
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:

```
mat1*mat2
```

```
matrix([[ 9., 12.],
       [ 9., 12.]])
```

```
mat2*mat1
```

```
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.

```
mat = np.matrix(
    [[1, 0, 1],
     [0, 1, 0],
     [1, 0, 2]])
)
```

```
mat.I
```

```
matrix([[ 2.,  0., -1.],
       [ 0.,  1.,  0.],
       [-1.,  0.,  1.]])
```

```
mat*mat.I
```

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

```
mat.I*mat
```

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



# NumPy Random Module

The `numpy.random` module provides us with random number generators (RNG). You can find the documentation [here](#). As the 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:

```
np.random.random()
```

```
0.1602130782023068
```

If the length or shape is specified, `random()` returns an array of random numbers:

```
np.random.random(5)
```

```
array([0.89758914, 0.7492066 , 0.01365653, 0.36307593, 0.4039456 ])
```

```
np.random.random((2, 3))
```

```
array([[0.41927877, 0.36197376, 0.71224223],
       [0.06707082, 0.31950669, 0.83969188]])
```

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  $[18, 30)$ :

```
np.random.random(4)*(30 -18) + 18
```

```
array([18.70153721, 23.11515985, 19.60246616, 28.96641049])
```

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:

```
#Random numbers from 1 up to 10
np.random.randint(1, 10)
```

```
1
```

Again, you can specify a size or shape of the output array:

```
np.random.randint(1, 10, 3)
```

```
array([8, 7, 9])
```

```
np.random.randint(1, 10, (2, 4))
```

```
array([[4, 2, 1, 6],
       [5, 4, 9, 3]])
```

# 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()

## Comparison and Bitwise Operations on Arrays

We can apply comparison operators to arrays:

```
a1 = np.array([1, 2, 3, 4, 5])
a2 = np.array([2, 1, 5, 6, 4])
a1 < a2
```

```
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 **Standard Library/If Statements/Comparison Operators** won't work. For example using `and`:

```
a1 < a2 and a1 > 2
```

```
-----
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5188\1014758179.py 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()
```

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:

```
(a1 < a2) & (a1 > 2)
```

```
array([False, False, True, True, False])
```

Note that the comparisons must be grouped in brackets for this to work:

```
a1 < a2 & a1 > 2
```

```
-----
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5188\2655709546.py in <module>
----> 1 a1 < a2 & a1 > 2
```

**ValueError:** The truth value of an array with more than one element is ambiguous. Use a.any()

## Worked Example - Random Points in a Region

We can use `np.where()` to check which points in an array lie inside or outside of region.

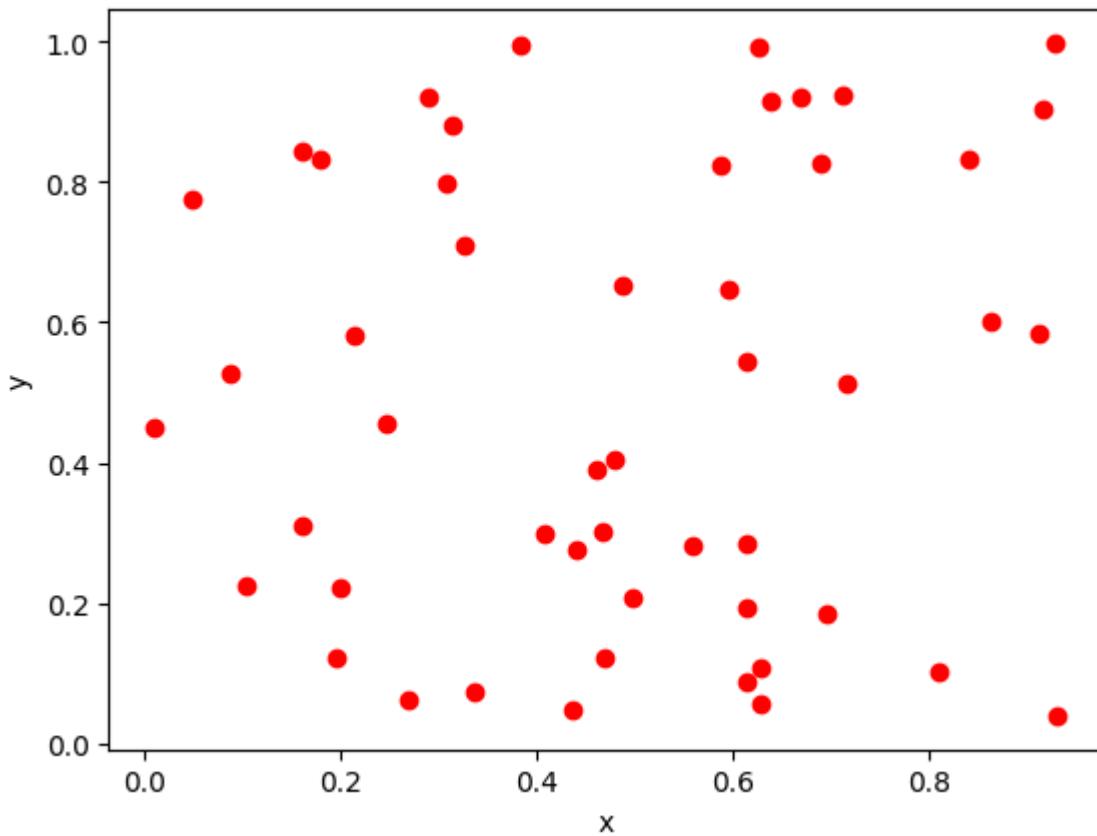
First let's generate an array of 50 random points in 2D space:

```
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  $(x_0, y_0), (x_1, y_1), \dots, (x_{49}, y_{49})$ , `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:

```

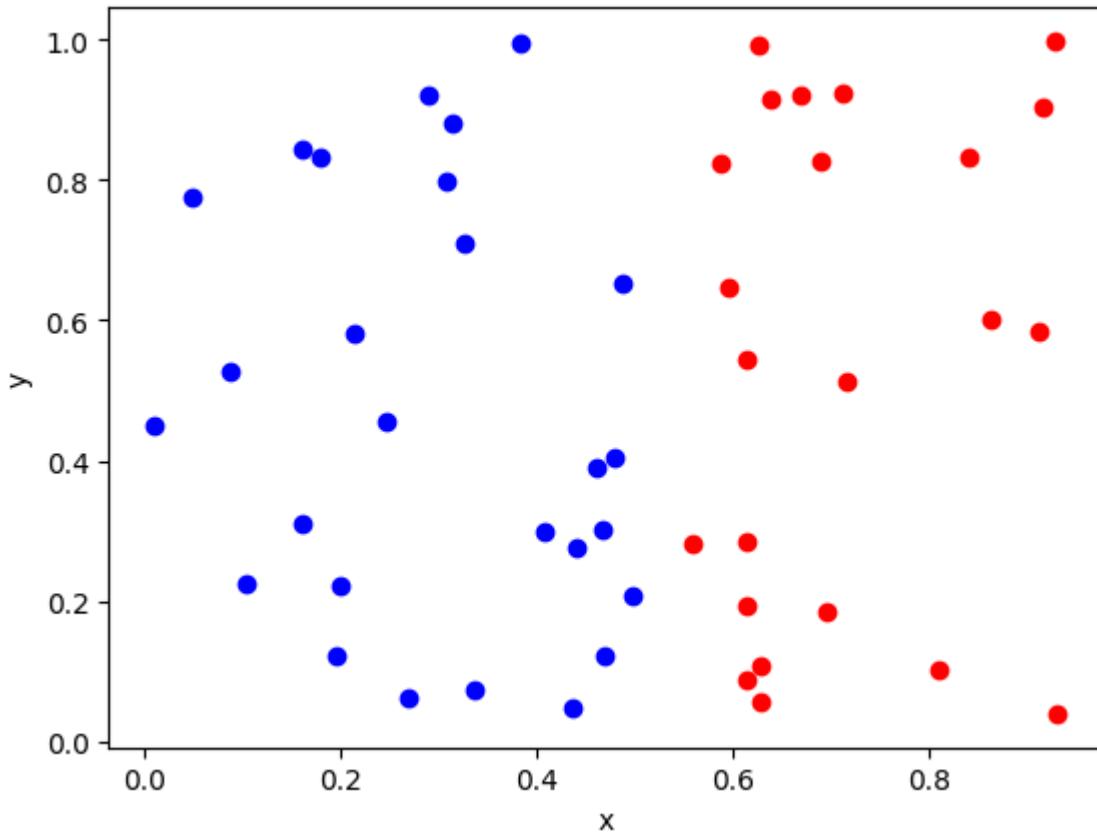
is_left = points[0, :] < 0.5 #True where left of 0.5

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')

```

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.

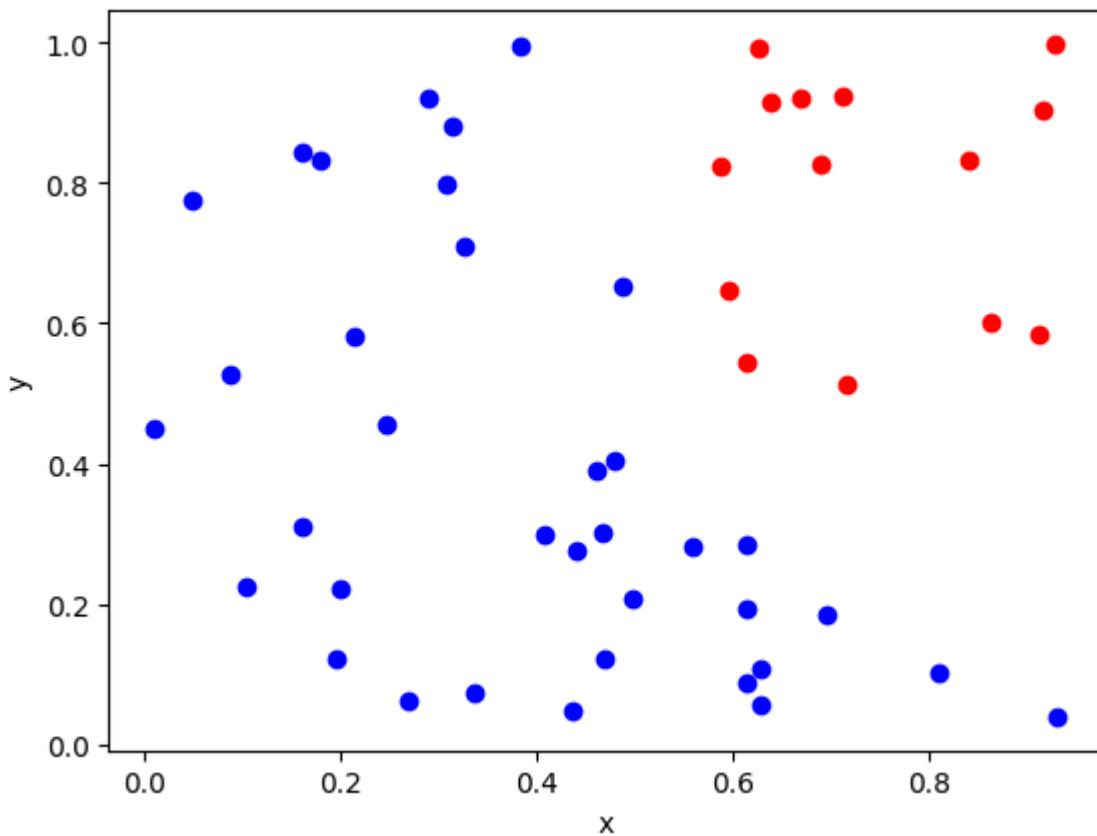
Now, lets plot the points right of 0.5 and above 0.5 (the top left square) as red and the rest as blue (remember the **bitwise and**):

```
#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')
```

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 an array of indices.

```
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[arr_where])
```

```
arr1: [10 11 12 13 14 15 16 17 18 19]
Indices where arr1 is greater than 15: (array([6, 7, 8, 9], dtype=int64),)
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.

```
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.]
```

## Worked Example - Piecewise Defined Functions

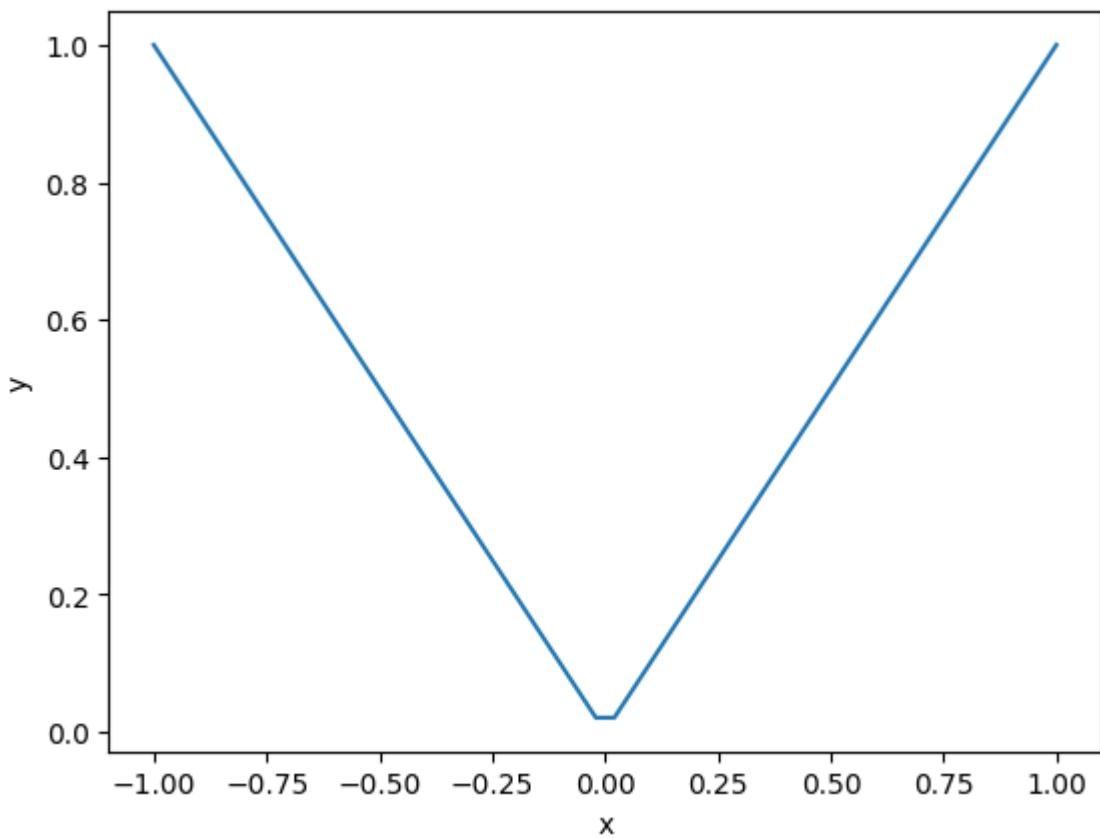
One use for `np.where()` is to define a piecewise defined function that works on arrays.

As a first example, let's use `np.where()` to plot the absolute value function (you should really use `np.abs()` for this):

$$y = \begin{cases} -x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

```
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:

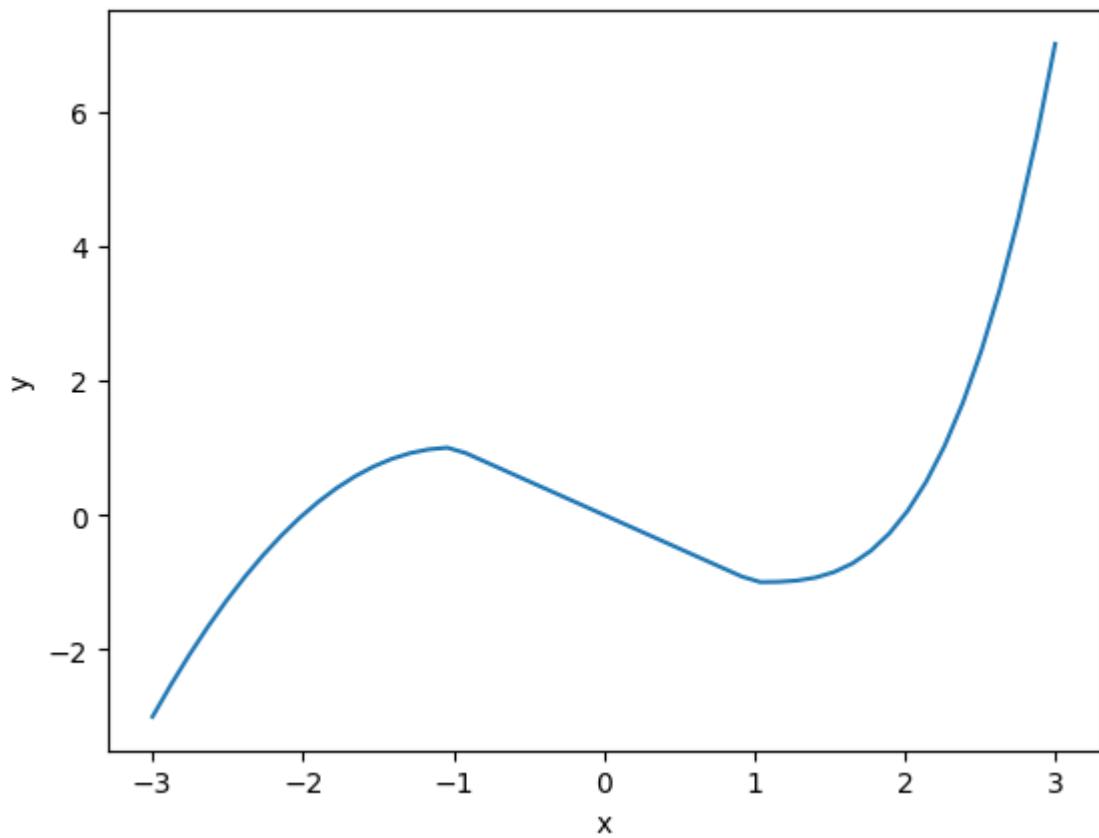
```
x = np.linspace(-3, 3)

#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

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

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:

```
import matplotlib.pyplot as plt
```

As a first example, let's plot the line  $y = x^2$ :

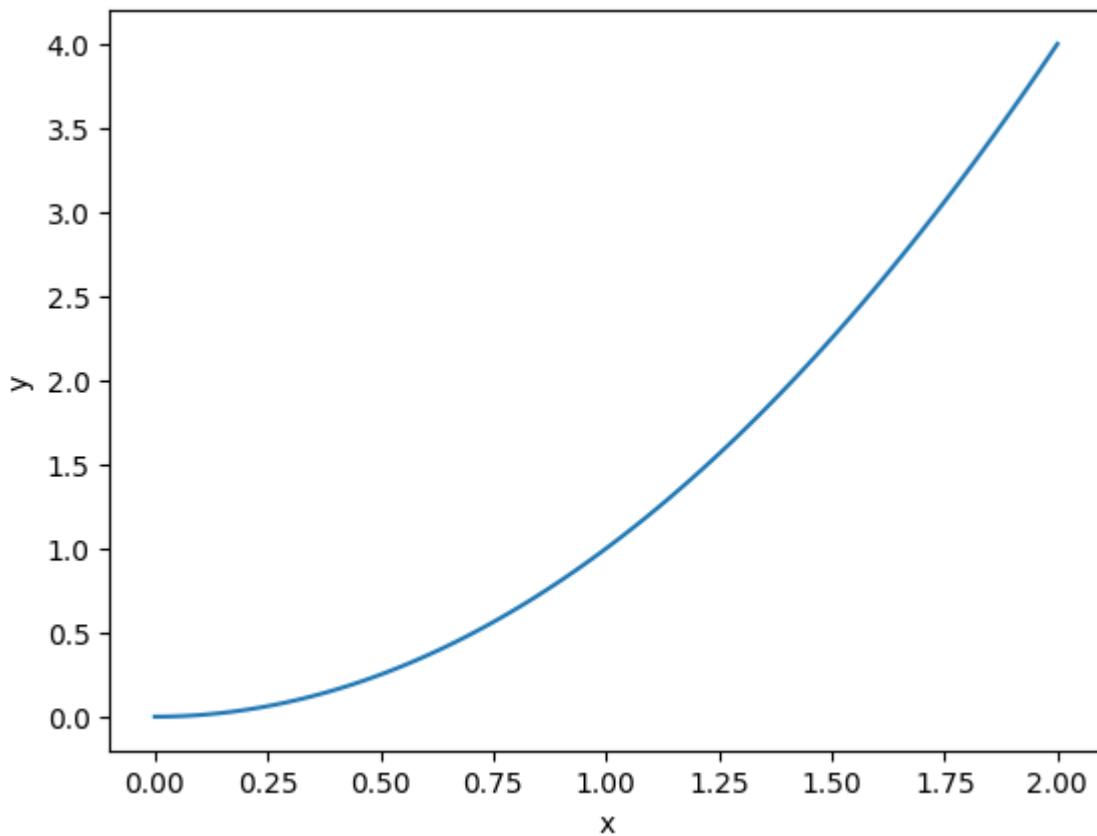
```
import numpy as np
import matplotlib.pyplot as plt

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 page [Scientific Packages/Matplotlib/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')
```

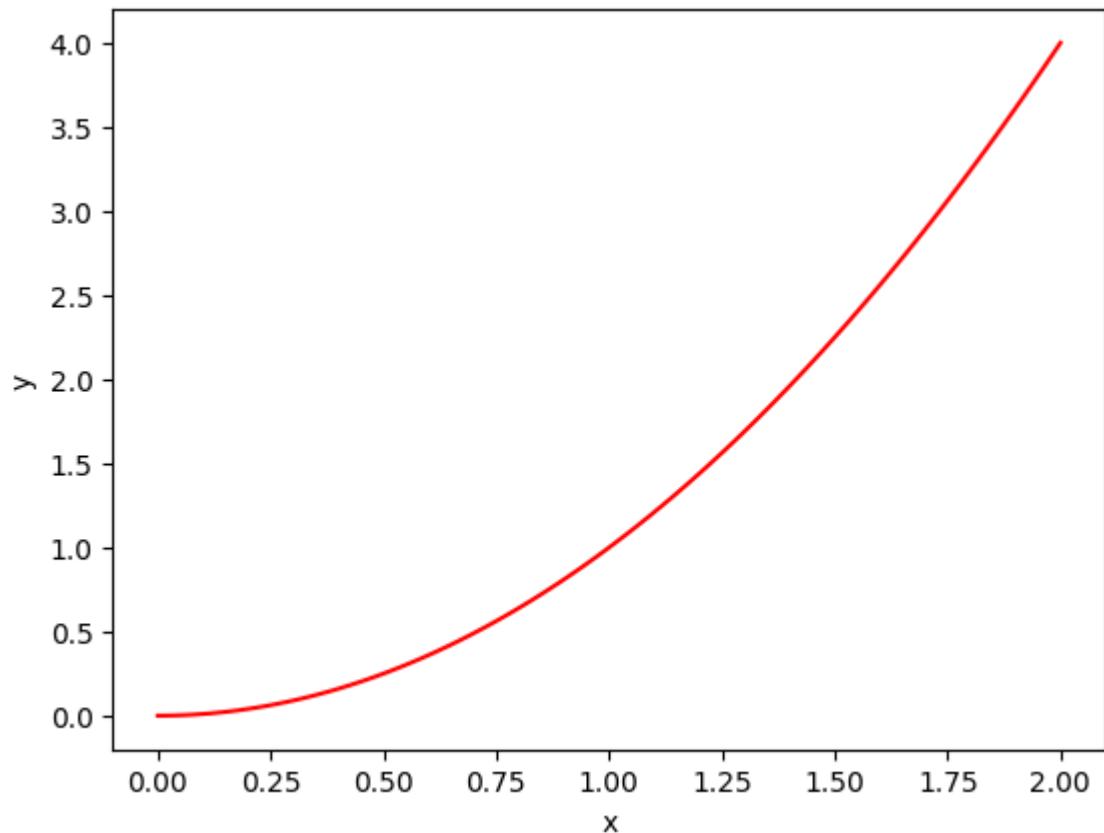
where the examples above both produce red lines, for example:

```
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:

## line styles



## 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):

```

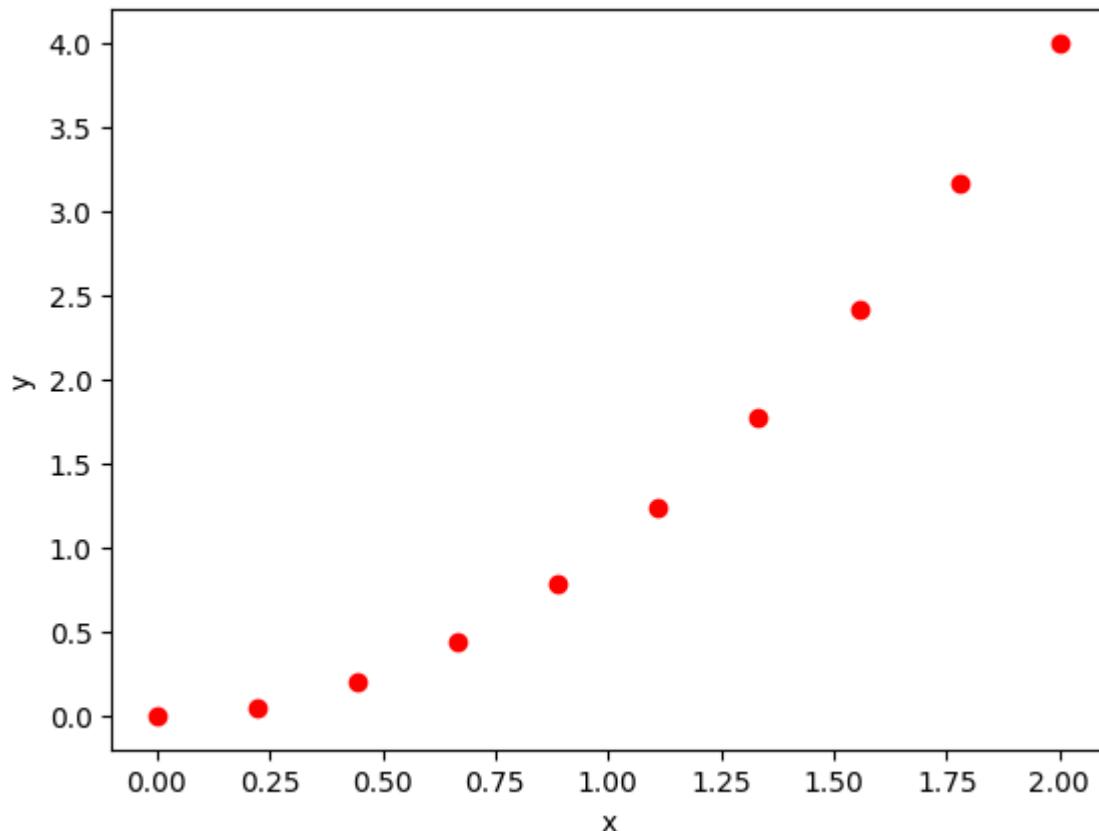
x = np.linspace(0, 2, 10)
y = x*x

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:

```
x = np.linspace(0, 2, 100)

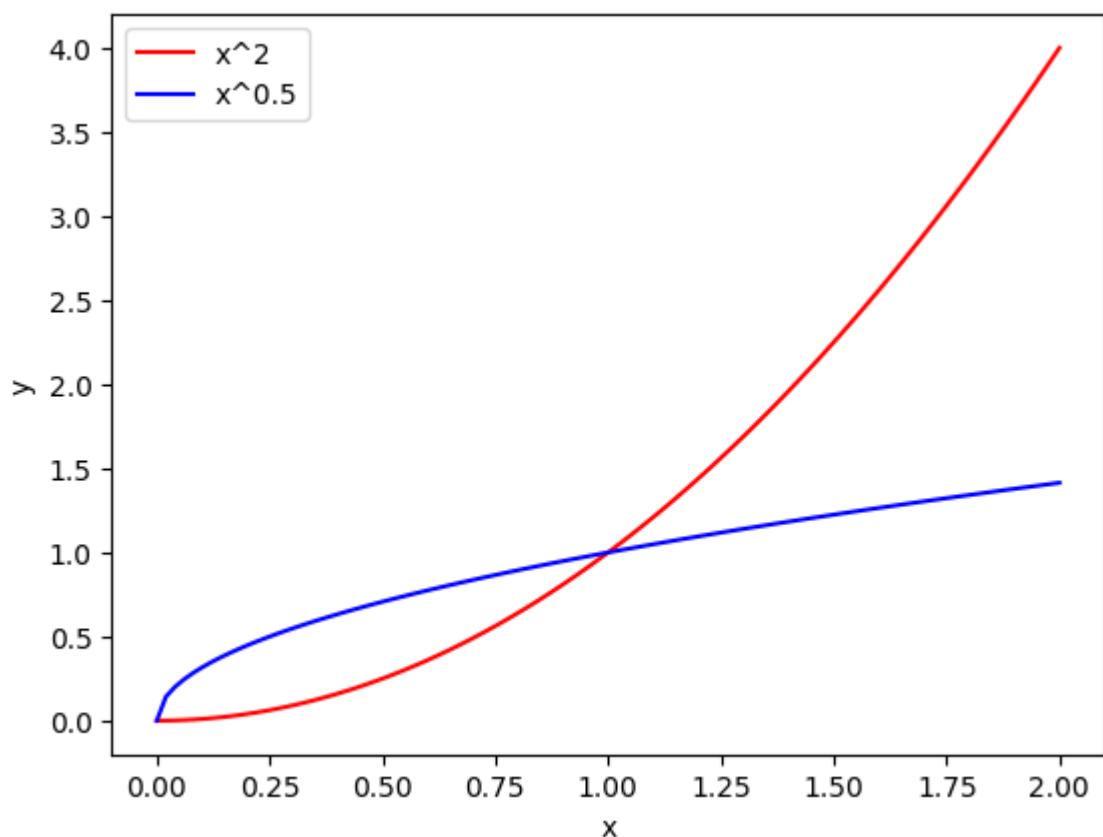
plt.plot(x, x*x, 'r', label = 'x^2')

plt.plot(x, np.sqrt(x), 'b', label = 'x^0.5')

plt.xlabel('x')
plt.ylabel('y')

plt.legend()

plt.show()
```



# 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:

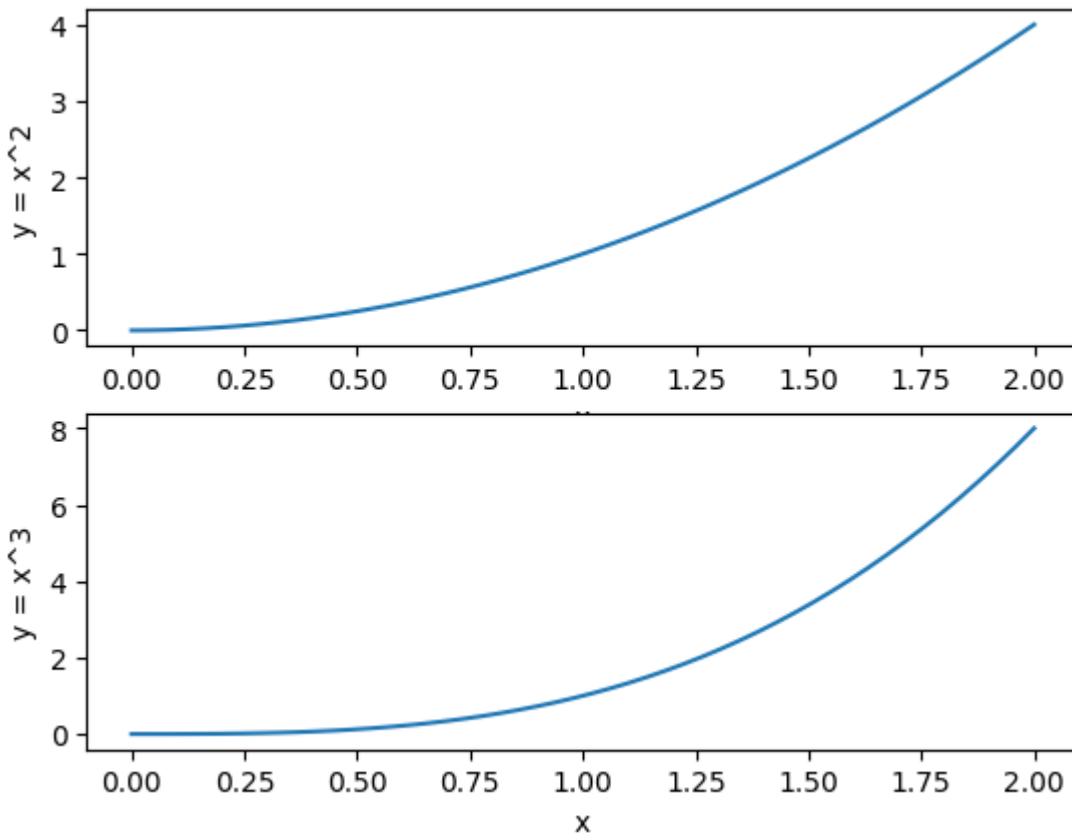
```
x = np.linspace(0, 2)

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 and columns as before.

Let's recreate the previous plot using this function:

```

x = np.linspace(0, 2)

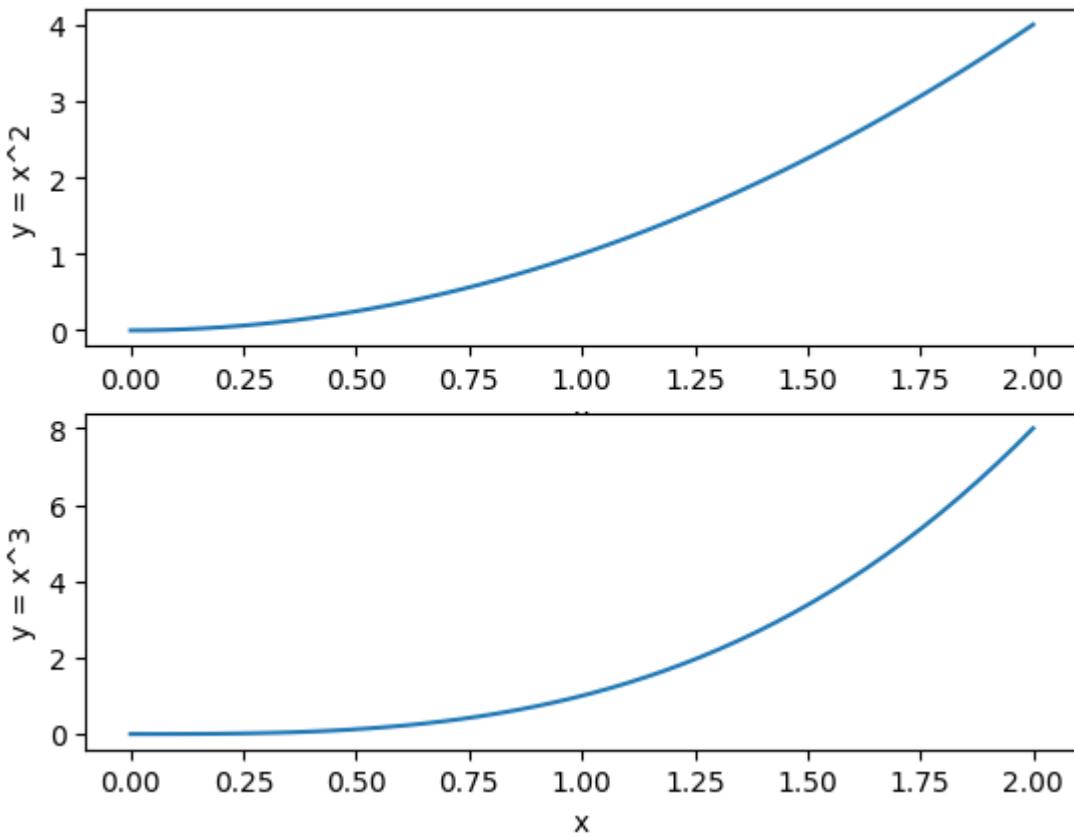
fig, ax = plt.subplots(2, 1)

#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:

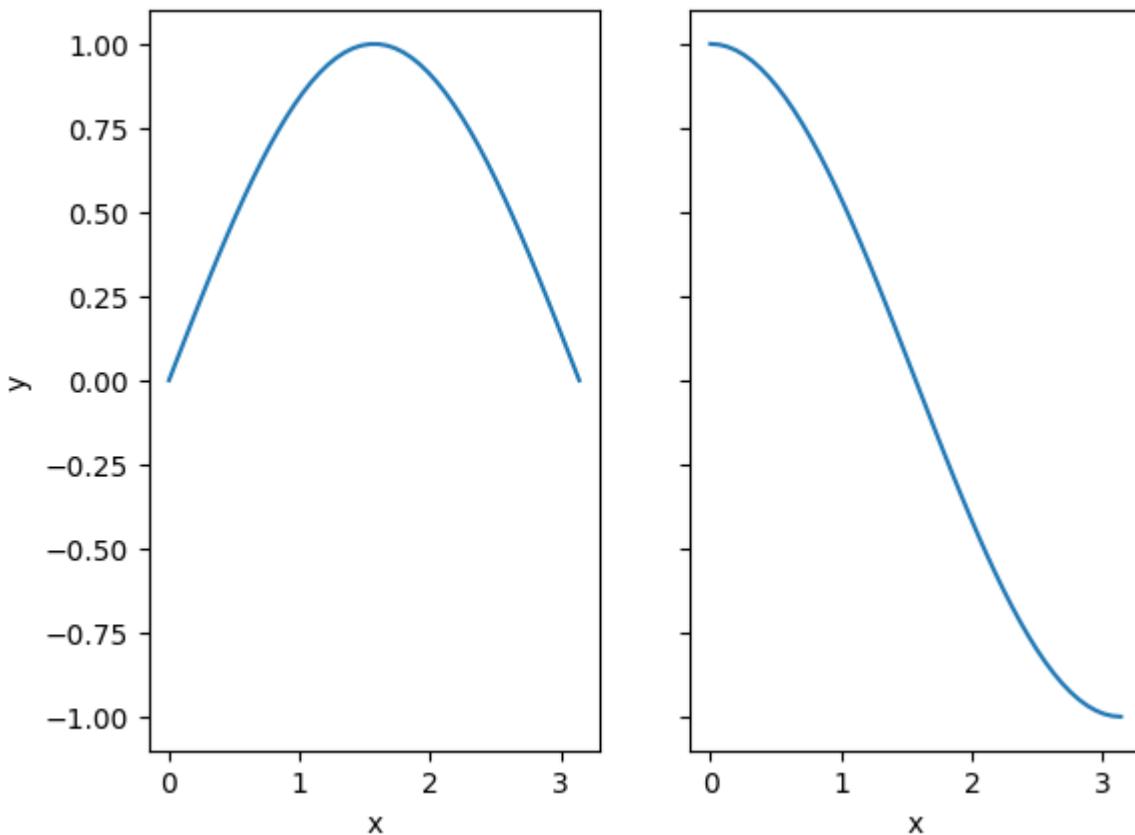
```
x = np.linspace(0, np.pi)
fig, ax = plt.subplots(1, 2, sharey = True)

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.

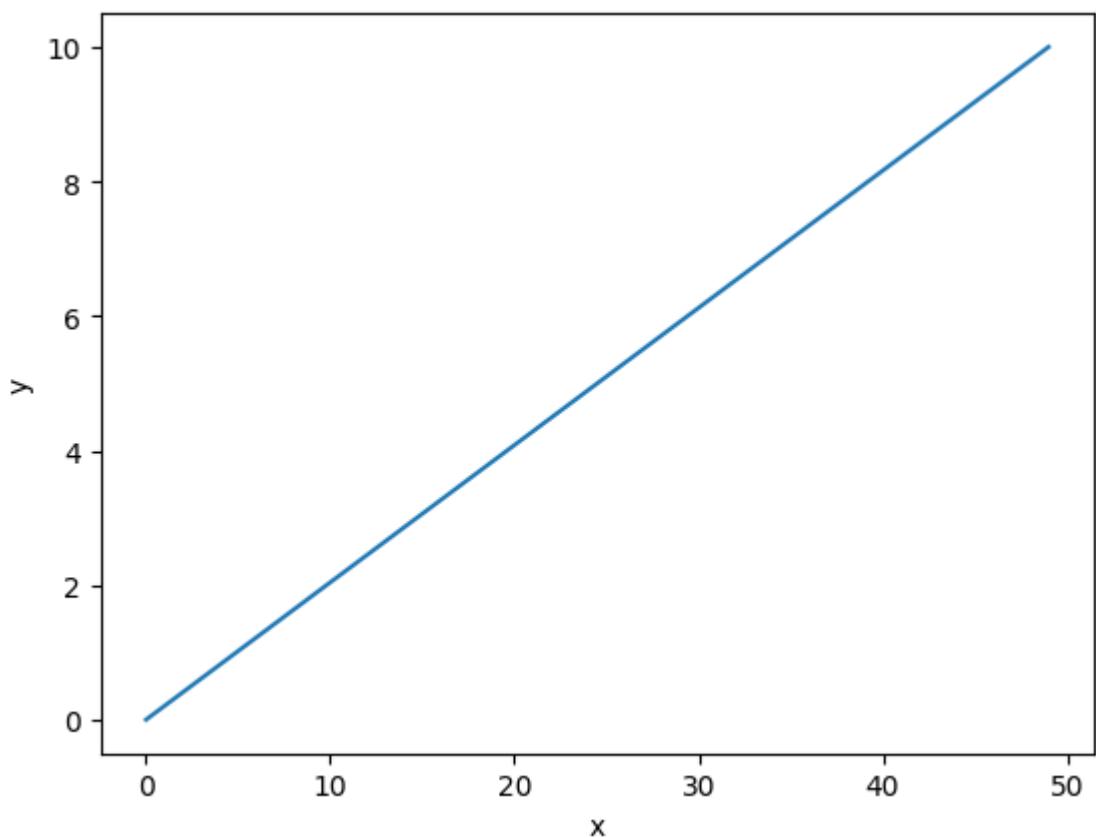
## Using Subplots For General Plots

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:

```
fig = plt.figure()
ax = fig.add_subplot()

ax.plot(np.linspace(0, 10))
ax.set_xlabel('x')
ax.set_ylabel('y')

plt.show()
```



# 3D Plotting

To use Matplotlib's 3D plotting functionality you first need to import the `mplot3d` module:

```
from mpl_toolkits import mplot3d
```

Then you need to pass the keyword argument `projection="3d"` into any of Matplotlib's axis creating functions. For example:

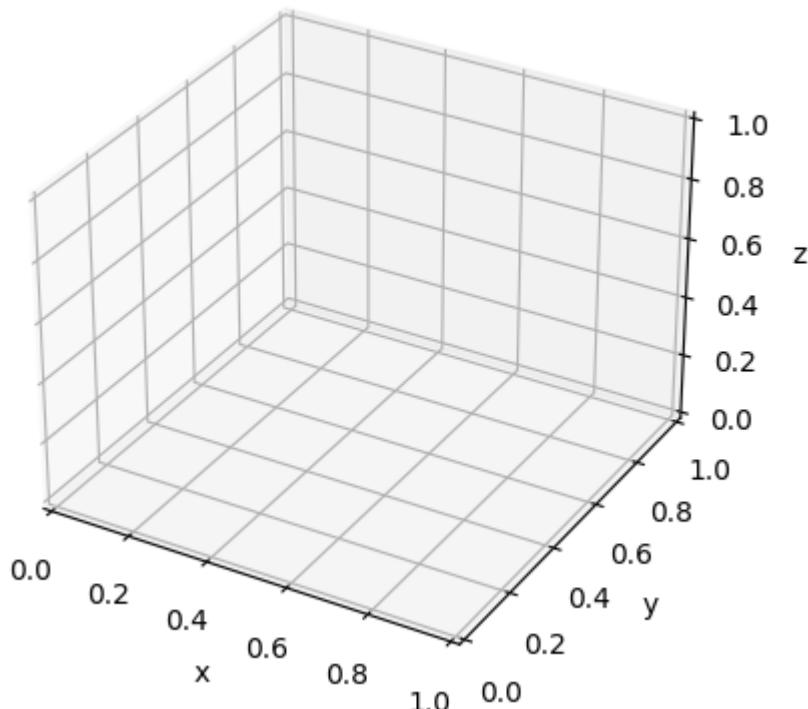
```
from mpl_toolkits import mplot3d

import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

plt.show()
```



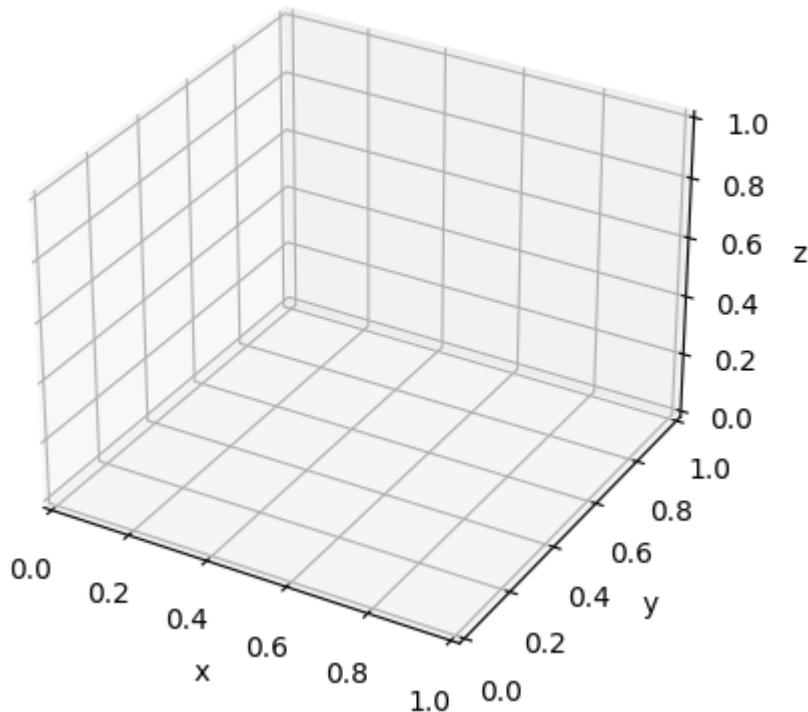
Note that this is a little less straight forward for the `plt.subplots()` function:

```
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

plt.show()
```



## Rotating the Viewing Angle

You can rotate the viewing angle of the figure programmatically using the `view_init()` function:

```
view_init(elev=None, azim=None)
```

where

- `elev` is the elevation angle in the vertical plane in degrees
- `azim` is the angle in the horizontal plane in degrees

```

from mpl_toolkits import mplot3d

import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 2, subplot_kw=dict(projection='3d'), figsize=(10,8))

# Vertical Rotation
ax[0].set_xlabel('x')
ax[0].set_ylabel('y')
ax[0].set_zlabel('z')

ax[0].set_title('Vertical Rotation')

ax[0].view_init(elev=5) #Vertical rotation

# Horizontal Rotation
ax[1].set_xlabel('x')
ax[1].set_ylabel('y')
ax[1].set_zlabel('z')

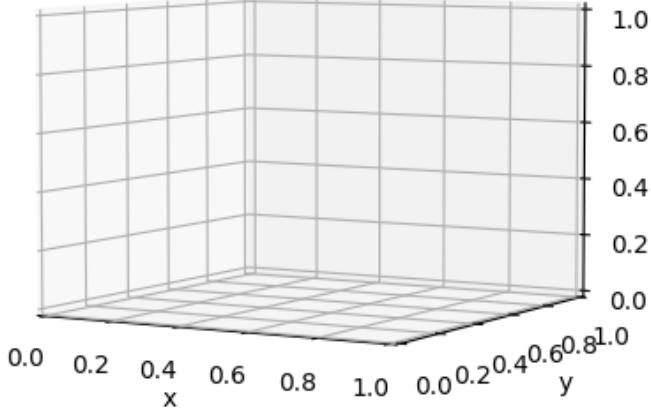
ax[1].set_title('Horizontal Rotation')

ax[1].view_init(azim=30) #Horizontal rotation

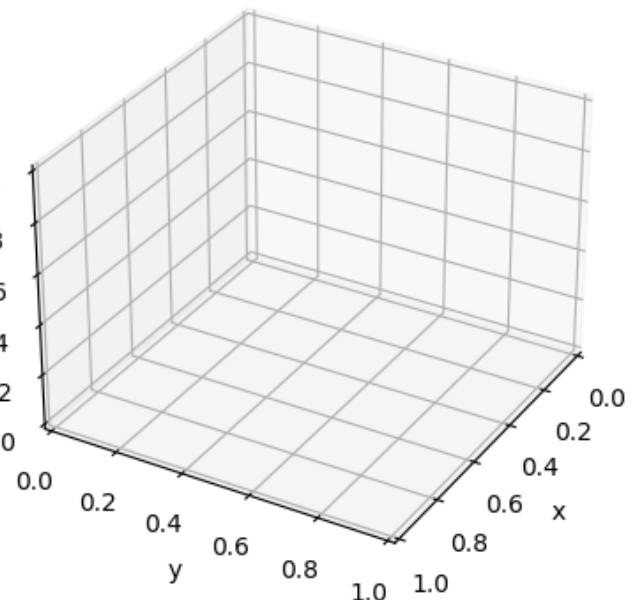
plt.show()

```

Vertical Rotation



Horizontal Rotation



Of course vertical and horizontal rotation can be combined.

# 3D Plotting Functions

Now you can use the [plotting functions](#) available in the `mplot3d` module. We shall cover a few here, but note that these are not the full extend of what is available.

## Plotting a line with `plot()` or `plot3D()`

You can plot a line on a 3D axis using `plot()` or `plot3D()`, which has the call signature:

```
plot(xs, ys, zs)
```

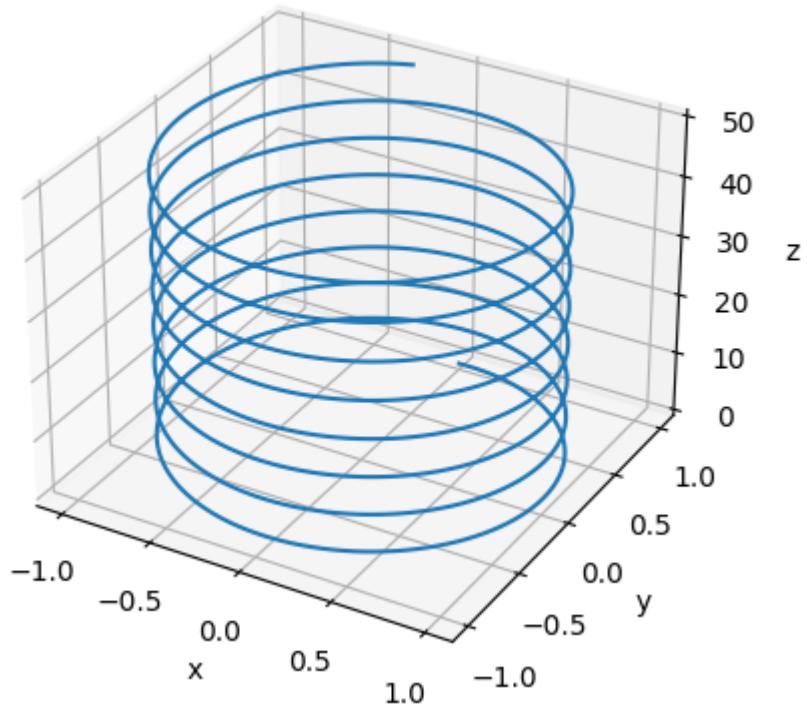
where `xs`, `ys` and `zs` are 1D array-like objects that contain the  $x$ ,  $y$  and  $z$  coordinates of each point/vertex making up the line. Note that you can use the same keyword arguments as the 2D `plot()` function.

### Worked Example - Plotting a 3D Spiral Line

For example, let's plot a spiral, defined by:

$$\begin{aligned}x(s) &= \sin(s) \\y(s) &= \cos(s) \\z(s) &= s\end{aligned}$$

```
from mpl_toolkits import mplot3d  
  
import numpy as np  
import matplotlib.pyplot as plt  
  
fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))  
  
s = np.linspace(0, 50, 1000)  
  
ax.plot(np.sin(s), np.cos(s), s)  
  
ax.set_xlabel('x')  
ax.set_ylabel('y')  
ax.set_zlabel('z')  
  
plt.show()
```



## Plotting surfaces with `plot_surface()`

The `plot_surface()` function creates a surface plot using a grid of points (or vertices) arranged to form quadrilaterals.

```
plot_surface(X, Y, Z)
```

The arguments  $\text{x}$ ,  $\text{y}$  and  $\text{z}$  are 2D arrays containing the  $x$ ,  $y$  and  $z$  coordinates of the points on the grid respectively. Pairs of adjacent points make the edges of the quadrelaterals.

## Worked Example - Plotting a Rectangle

For this first example, let's plot a square with one side elevated. We will give it points with the  $(x, y, z)$  coordinates:

1.  $(0, 0, 0)$
2.  $(1, 0, 0)$
3.  $(0, 1, 1)$
4.  $(1, 1, 1)$

These coordinates will be stored in separate arrays for  $x$ ,  $y$  and  $z$  in the following order:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
from mpl_toolkits import mplot3d

import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots(subplot_kw=dict(projection='3d'), figsize=(10,8))

X = np.array([
    [0, 1],
    [0, 1]
])

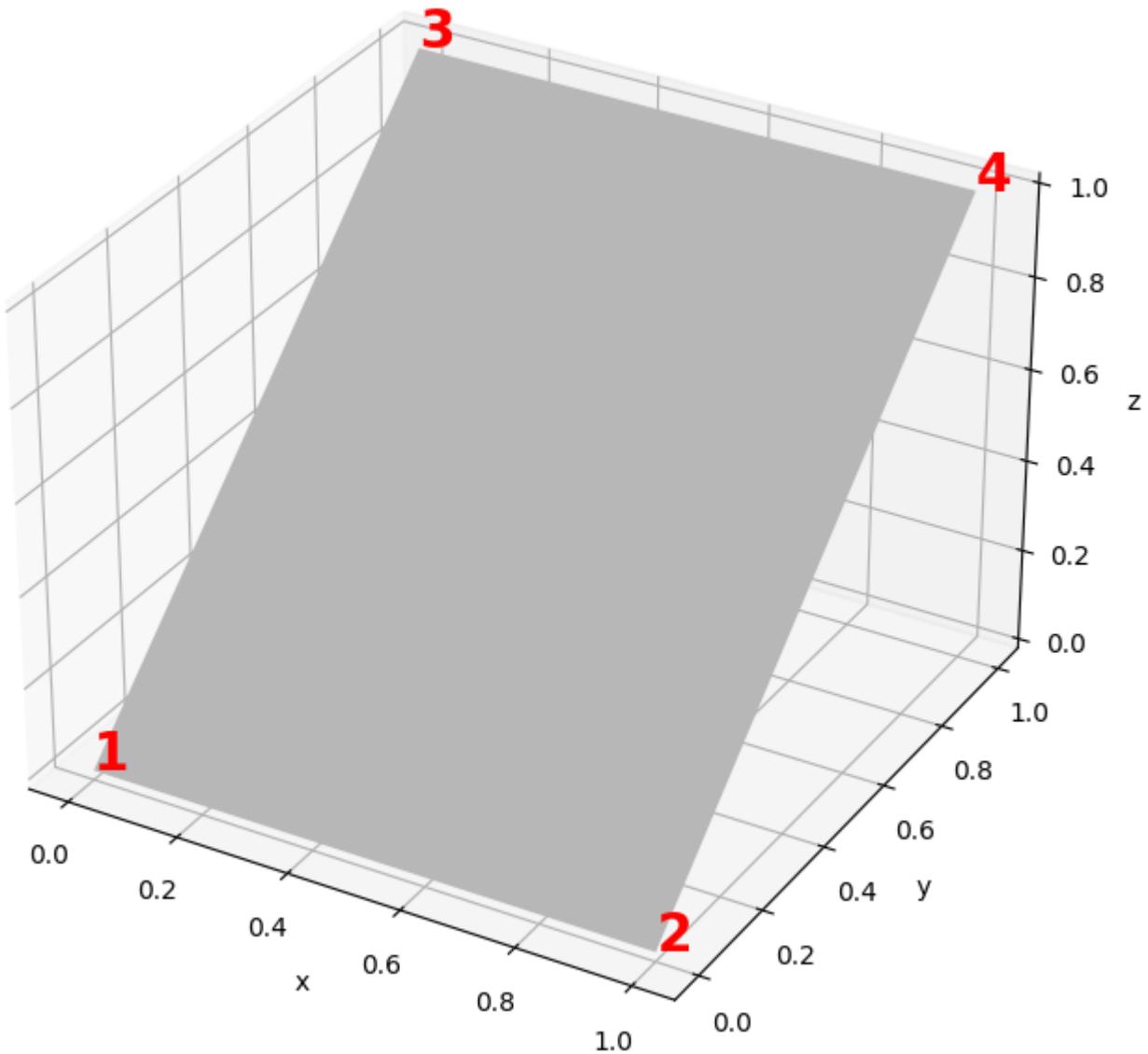
Y = np.array([
    [0, 0],
    [1, 1]
])

Z = np.array([
    [0, 0],
    [1, 1]
])

ax.plot_surface(X, Y, Z)

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

plt.show()
```



Note that the code used to select the color of the surface and create the numerical annotations is not included above.

Now, we will often create surfaces where  $Z(X, Y)$  is some function of  $X$  and  $Y$ , where  $X$  and  $Y$  can be treated as independent variables. In these cases we can generate a grid of  $x$  and  $y$  coordinates, and use these to calculate the corresponding  $z$  values.

To create this grid we will use the `numpy.meshgrid()` function. For our interests, the call signature of `meshgrid()` is:

```
meshgrid(*xi)
```

where `xi` is sequence of arrays. Here we are only interested in passing 2 arrays into `meshgrid()` (one for a sequence of  $x$  values and another for a sequence of  $y$  values) to produce a 2D grid.

Consider the arrays `x` and `y` where:

$$\begin{aligned}x &= [x_1, x_2, x_3, \dots, x_n] \\y &= [y_1, y_2, y_3, \dots, y_m]\end{aligned}$$

if we were to put these into `meshgrid()`:

```
X, Y = np.meshgrid(x, y)
```

then the resulting 2D arrays would be of the form:

$$X = \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_n \\ x_1 & x_2 & x_3 & \cdots & x_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1 & x_2 & x_3 & \cdots & x_n \end{bmatrix}$$

and

$$Y = \begin{bmatrix} y_1 & y_1 & y_1 & \cdots & y_1 \\ y_2 & y_2 & y_2 & \cdots & y_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_m & y_m & y_m & \cdots & y_m \end{bmatrix}$$

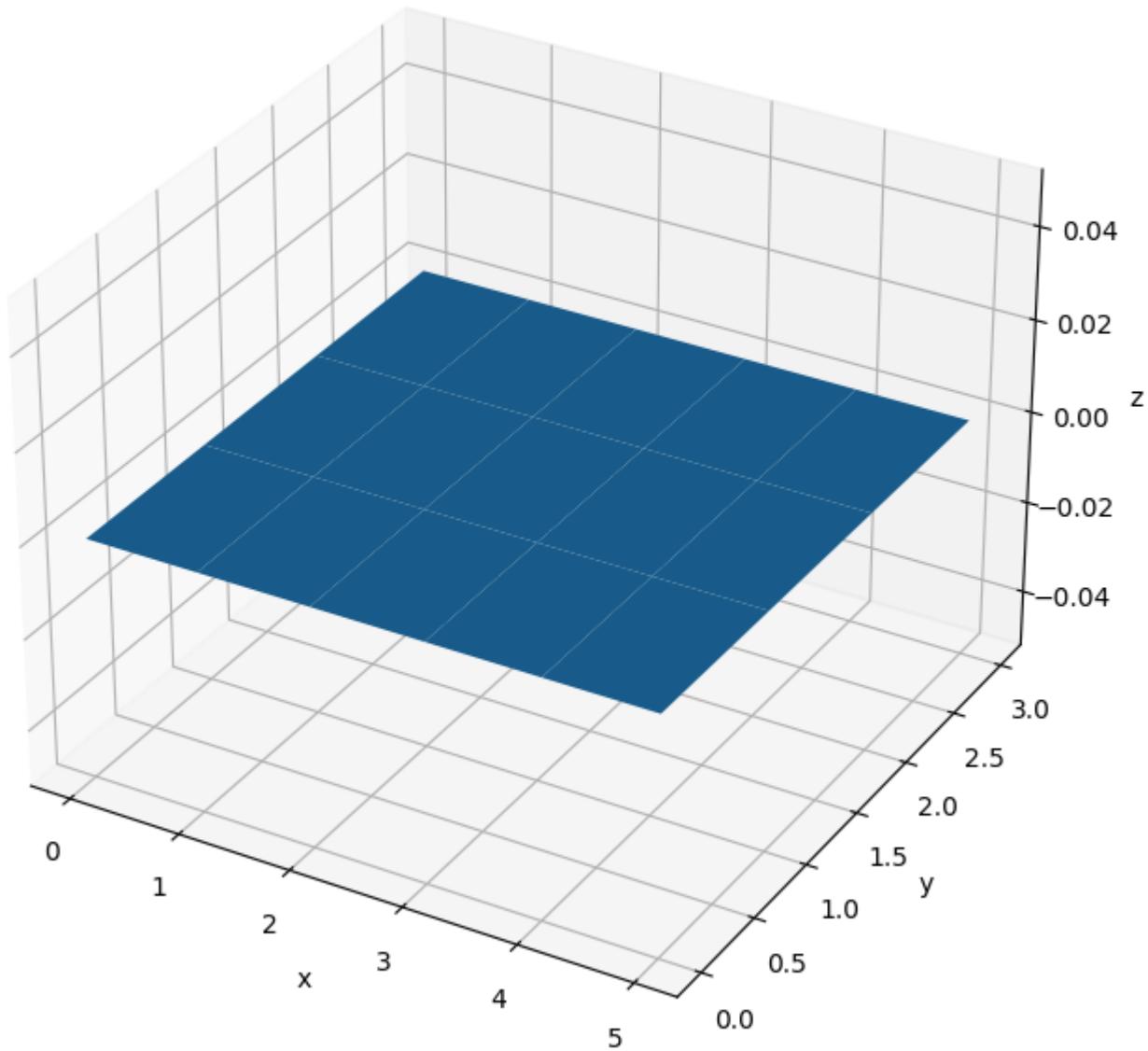
both with  $m$  rows and  $n$  columns.

Let's consider an example of a grid like this generated from the arrays:

$$\begin{aligned}x &= [0, 1, 2, 3, 4, 5] \\y &= [0, 1, 2, 3]\end{aligned}$$

used to generate a flat surface with  $z = 0$ .

```
from mpl_toolkits import mplot3d  
  
import numpy as np  
import matplotlib.pyplot as plt  
  
fig, ax = plt.subplots(subplot_kw=dict(projection='3d'), figsize=(10,8))  
  
x = np.arange(0, 6)  
y = np.arange(0, 4)  
  
X, Y = np.meshgrid(x, y)  
Z = np.zeros(X.shape)  
  
ax.plot_surface(X, Y, Z)  
  
ax.set_xlabel('x')  
ax.set_ylabel('y')  
ax.set_zlabel('z')  
  
plt.show()
```

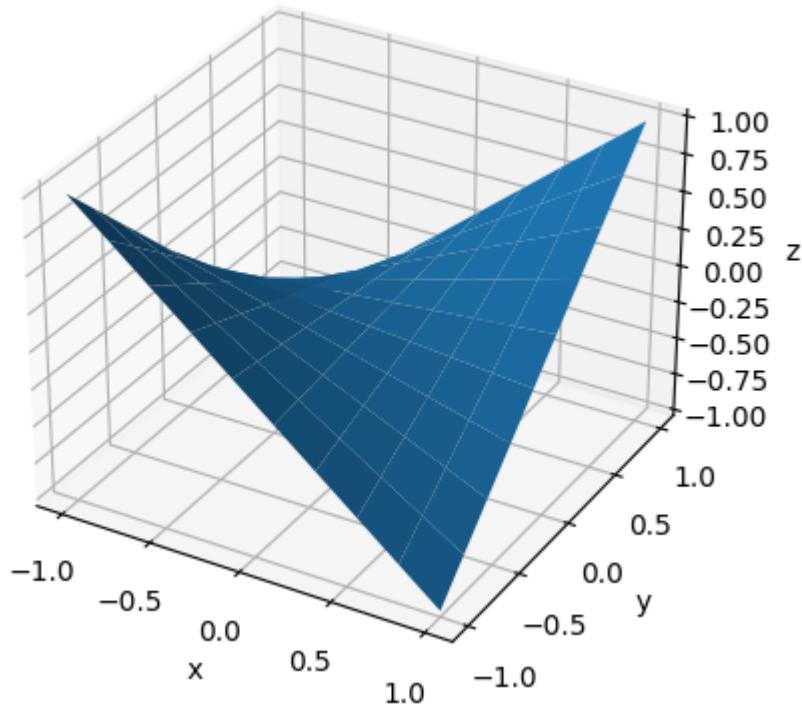


### Worked Example - Plotting a Surface Using a Rectangular Grid

Let's consider a simple surface:

$$z = xy$$

```
from mpl_toolkits import mplot3d  
  
import numpy as np  
import matplotlib.pyplot as plt  
  
fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))  
  
x = np.linspace(-1, 1, 10)  
y = np.linspace(-1, 1, 10)  
  
X, Y = np.meshgrid(x, y)  
  
Z = X * Y  
  
ax.plot_surface(X, Y, Z)  
  
ax.set_xlabel('x')  
ax.set_ylabel('y')  
ax.set_zlabel('z')  
  
plt.show()
```



## Worked Example - Plotting a Surface Using a Radial Grid

You may not always want to use a rectangular grid. For example, if we have a radially symmetric surface (or just a surface defined using polar coordinates):

$$z = 1 - (x^2 + y^2)$$

then we may want to use a radial grid, by creating a rectangular grid of  $r$  and  $\theta$  coordinates, where:

$$\begin{aligned}x &= r \cos(\theta) \\y &= r \sin(\theta)\end{aligned}$$

and then calculating an  $x, y$  grid from this. Note that  $x, y$  and  $z$  can be considered as parameterized by  $r$  and  $\theta$ .

```
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))

r = np.linspace(0, 10, 10)
theta = np.linspace(0, 2 * np.pi, 20)

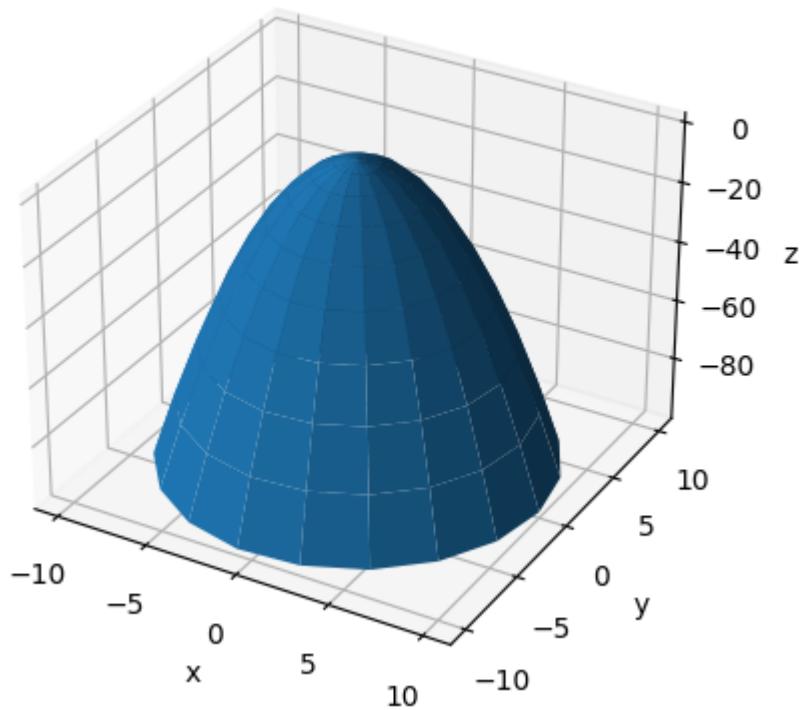
R, THETA = np.meshgrid(r, theta)

X = R * np.cos(THETA)
Y = R * np.sin(THETA)
Z = 1 - R * R # X**2 + Y**2 == R**2

ax.plot_surface(X, Y, Z)

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

plt.show()
```



There's much more that you can do with these surface plots, such as coloring them in using a colormap.

## Plotting wireframes with `plot_wireframe()`

The `plot_wireframe()` function is similar to the `plot_surface()` function, except it produces a plot of the edges of the quadrilaterals only, with the faces unfilled.

### Worked Example - Plotting a Simple Wireframe

Let's consider the same surface as before:

$$z = xy$$

```
from mpl_toolkits import mplot3d

import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 2, subplot_kw=dict(projection='3d'), figsize=(10,8))

x = np.linspace(-1, 1, 10)
y = np.linspace(-1, 1, 10)

X, Y = np.meshgrid(x, y)

Z = X * Y

#Surface Plot
ax[0].plot_surface(X, Y, Z)

ax[0].set_title('Surface Plot')

ax[0].set_xlabel('x')
ax[0].set_ylabel('y')
ax[0].set_zlabel('z')

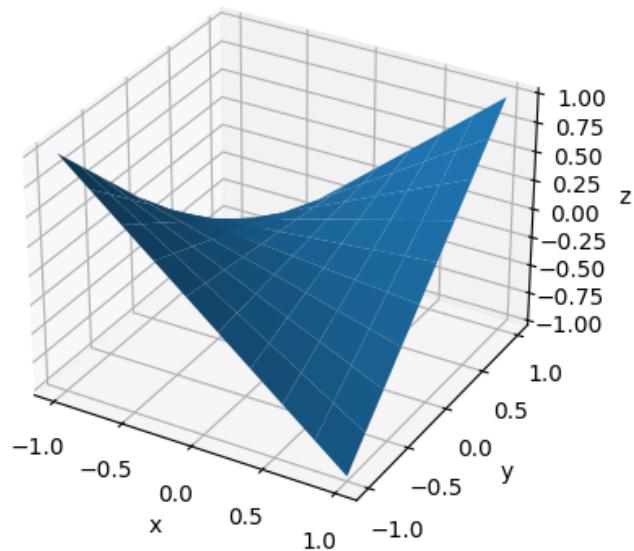
#Wireframe plot
ax[1].plot_wireframe(X, Y, Z)

ax[1].set_title('Wireframe Plot')

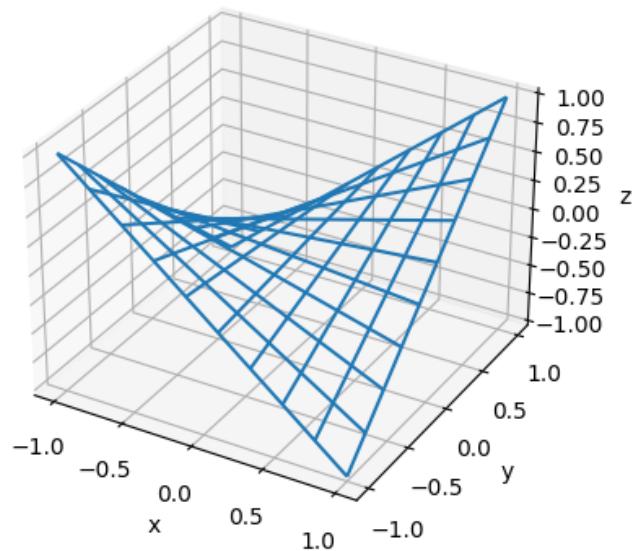
ax[1].set_xlabel('x')
ax[1].set_ylabel('y')
ax[1].set_zlabel('z')

plt.show()
```

Surface Plot



Wireframe Plot



# Astropy

In this final chapter we introduce the Astropy package. In this module we only provide a brief introduction, but you will be making more use of the package later in the AST4007W course.

Astropy is an open-source and community-developed Python package. It contains core astronomy-related functionality. It was started in 2011 to bring together developers across the field of astronomy in order to coordinate the development of a common set of Python tools for astronomers.

The website can be found at <https://www.astropy.org/>, which also contains links to examples, tutorials and the documentation.

## Supported File Formats

Astropy provides support for domain-specific file formats such as:

- Flexible Image Transport System (FITS) files
- Virtual Observatory (VO) tables
- common ASCII table formats,
- unit and physical quantity conversions
- physical constants specific to astronomy
- celestial coordinate and time transformations,
- world coordinate system (WCS) support
- e.t.c

We cover working with the FITS files in the final tutorial.

# Units and Quantities

The `astropy.units` module gives us the tools to work with units and quantities (values paired with units). The documentation can be found [here](#).

The `astropy.units.Unit` class is used to represent base units of measurement, such as meters, seconds, kilograms, ect). These units can be manipulated, for example to determine conversion factors from one set of units to another.

You can also define your own units, either as standalone base units or by composing other units together. It is also possible to decompose units into its base units.

As an example, consider the SI unit for mass : kilograms (kg)

```
from astropy import units as u
u.kg
```

kg

## Quantities

Quantities are created by multiplying a float/integer with units, for example a mass:

```
mass = 3.0*u.kg
mass
```

3 kg

The `value` and `unit` of the quantity can be accessed separately:

```
mass.value
```

```
3.0
```

```
mass.unit
```

kg

Quantities can be combined mathematically, the units combine accordingly. For example, consider the force:

```
acceleration = 4 * u.m/(u.s*u.s)
force = mass*acceleration
force
```

$$12 \frac{\text{kg m}}{\text{s}^2}$$

## Converting Units

Units can be converted to like units. For example, let's convert the force calculated above into Newtons:

```
force_N = force.to(u.N)
force_N
```

12 N

The conversion above is still within the SI system. We can also convert to units from other systems. For example, converting the force to dynes from the cgs system:

```
force_dyn = force.to(u.dyn)
force_dyn
```

1200000 dyn

Astropy gives us a more convenient way to convert between unit systems without requiring us to do any book keeping. Again, lets convert the force in SI units to the cgs units:

```
force_cgs = force.cgs
force_cgs
```

1200000 dyn

## Quantities and NumPy Arrays

Astropy quantities support NumPy arrays. The whole array will have the same units.

```
import numpy as np
distances = np.array([1, 2, 3, 4]) * u.m
distances
```

[1, 2, 3, 4] m

```
distances.to(u.cm)
```

[100, 200, 300, 400] cm

```
distances/(1 * u.s)
```

$$[1, 2, 3, 4] \frac{\text{m}}{\text{s}}$$

## Compose Units

Above we converted our force from  $\text{kg} \cdot \text{m} \cdot \text{s}^{-2}$  to N directly. We can let Astropy compose those units for us:

```
force.unit.compose()
```

```
[Unit("N"), Unit("100000 dyn")]
```

As you can see this gives us all the standard options available.

## Decompose Units

You can also decompose units back into base units:

```
force_N.decompose()
```

$$12 \frac{\text{kg m}}{\text{s}^2}$$

## Dimensionless Quantities

Astropy understands dimensionless quantities. For example, lets take the ratio of 2 lengths:

```
r1 = 10 * u.m
r2 = 2 * u.m

r1 / r2
```

5

Sometimes you will need to decompose units first:

```
force1 = 3 * u.N
force2 = 6 * u.kg * u.m / u.s / u.s

force_ratio = force1 / force2
force_ratio
```

$$0.5 \frac{s^2 N}{kg m}$$

```
force_ratio.decompose()
```

0.5

## Find Equivalent Units

You can find equivalent units by using the `find_equivalent_units()` method. For example, for milliseconds (ms):

```
u.ms.find_equivalent_units()
```

<b>Primary name</b>	<b>Unit definition</b>	<b>Aliases</b>
a	3.15576e+07 s	annum
d	86400 s	day
fortnight	1.2096e+06 s	
h	3600 s	hour, hr
min	60 s	minute
s	irreducible	second
sday	86164.1 s	
wk	604800 s	week
yr	3.15576e+07 s	year

## Set Equivalencies

In some contexts we want to use certain units interchangeably, that are not actually equivalent.

For example, the energy of a photon ( $E$ ) is related to its wavelength ( $\lambda$ ) and frequency ( $\nu$ ) as:

$$E = h\nu = hc/\lambda$$

where  $h$  is Planck's constant. Radio astronomers refer to the energy of an emission line of neutral hydrogen being 1420 MHz or 21 cm. These units are obviously not that of energy, but in this context they can be treated as such.

Astropy allows for this kind of treatment of our units:

```
wavelength = 21.106 * u.cm  
wavelength.to(u.MHz, equivalencies = u.spectral())
```

1420.4134 MHz

## Finding Equivalent Units with Set Equivalencies

You can also find equivalent within the context of the equivalencies:

```
freq = 1 * u.Hz
freq.unit.find_equivalent_units(equivalencies = u.spectral())
```

<b>Primary name</b>	<b>Unit definition</b>	<b>Aliases</b>
AU	1.49598e+11 m	au, astronomical_unit
Angstrom	1e-10 m	AA, angstrom
Bq	1 / s	becquerel
Ci	3.7e+10 / s	curie
Hz	1 / s	Hertz, hertz
J	kg m2 / s2	Joule, joule
Ry	2.17987e-18 kg m2 / s2	rydberg
cm	0.01 m	centimeter
eV	1.60218e-19 kg m2 / s2	electronvolt
earthRad	6.3781e+06 m	R_earth, Rearth
erg	1e-07 kg m2 / s2	
jupiterRad	7.1492e+07 m	R_jup, Rjup, R_jupiter, Rjupiter
k	100 / m	Kayser, kayser
lyr	9.46073e+15 m	lightyear
m	irreducible	meter
micron	1e-06 m	
pc	3.08568e+16 m	parsec
solRad	6.957e+08 m	R_sun, Rsun

# Fractional Units

Sometimes we'll need to work with fractional units (units with a fractional power). Floats can work for this, but it is better to use the Standard Library's `fraction.Fraction` objects:

```
from fractions import Fraction
T = 3000.0 * u.K
T
```

3000 K

```
T ** Fraction(3/2)
```

$164316.77 \text{ K}^{3/2}$

# Defining Your Own Units

You can define your own units derived from other units. For example, lets make a unit called a baker's fortnight (bf), which is a fortnight (14 days) and an extra day:

```
bakers_fortnight = u.def_unit('bf', 15 * u.day)
bakers_fortnight
```

bf

```
time = 3 * bakers_fortnight
time
```

3 bf

```
time.to(u.day)
```

45 d

# Constants

The `astropy.constants` module contains measured physical constants, for example the gravitational constant:

```
from astropy import constants as c  
print(c.G)
```

```
Name      = Gravitational constant  
Value     = 6.6743e-11  
Uncertainty = 1.5e-15  
Unit      = m3 / (kg s2)  
Reference = CODATA 2018
```

These constants can be treated as quantities.

See the [documentation](#) for the list of constants available:

Name	Value	Unit	Description
G	6.6743e-11	m3 / (kg s2)	Gravitational constant
N_A	6.02214076e+23	1 / (mol)	Avogadro's number
R	8.31446262	J / (K mol)	Gas constant
Ryd	10973731.6	1 / (m)	Rydberg constant
a0	5.29177211e-11	m	Bohr radius
alpha	0.00729735257	Fine-structure constant	
atm	101325	Pa	Standard atmosphere
b_wien	0.00289777196	m K	Wien wavelength displacement law constant
c	299792458	m / (s)	Speed of light in vacuum
e	1.60217663e-19	C	Electron charge
eps0	8.85418781e-12	F/m	Vacuum electric permittivity
g0	9.80665	m / s2	Standard acceleration of gravity
h	6.62607015e-34	J s	Planck constant
hbar	1.05457182e-34	J s	Reduced Planck constant
k_B	1.380649e-23	J / (K)	Boltzmann constant
m_e	9.1093837e-31	kg	Electron mass
m_n	1.6749275e-27	kg	Neutron mass
m_p	1.67262192e-27	kg	Proton mass
mu0	1.25663706e-06	N/A2	Vacuum magnetic permeability
muB	9.27401008e-24	J/T	Bohr magneton

Name	Value	Unit	Description
sigma_T	6.65245873e-29	m2	Thomson scattering cross-section
sigma_sb	5.67037442e-08	W / (K4 m2)	Stefan-Boltzmann constant
u	1.66053907e-27	kg	Atomic mass
GM_earth	3.986004e+14	m3 / (s2)	Nominal Earth mass parameter
GM_jup	1.2668653e+17	m3 / (s2)	Nominal Jupiter mass parameter
GM_sun	1.3271244e+20	m3 / (s2)	Nominal solar mass parameter
L_bol0	3.0128e+28	W	Luminosity for absolute bolometric magnitude 0
L_sun	3.828e+26	W	Nominal solar luminosity
M_earth	5.97216787e+24	kg	Earth mass
M_jup	1.8981246e+27	kg	Jupiter mass
M_sun	1.98840987e+30	kg	Solar mass
R_earth	6378100	m	Nominal Earth equatorial radius
R_jup	71492000	m	Nominal Jupiter equatorial radius
R_sun	695700000	m	Nominal solar radius
au	1.49597871e+11	m	Astronomical Unit
kpc	3.08567758e+19	m	Kiloparsec
pc	3.08567758e+16	m	Parsec

# Part

---

## Numerical Methods

# 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

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.

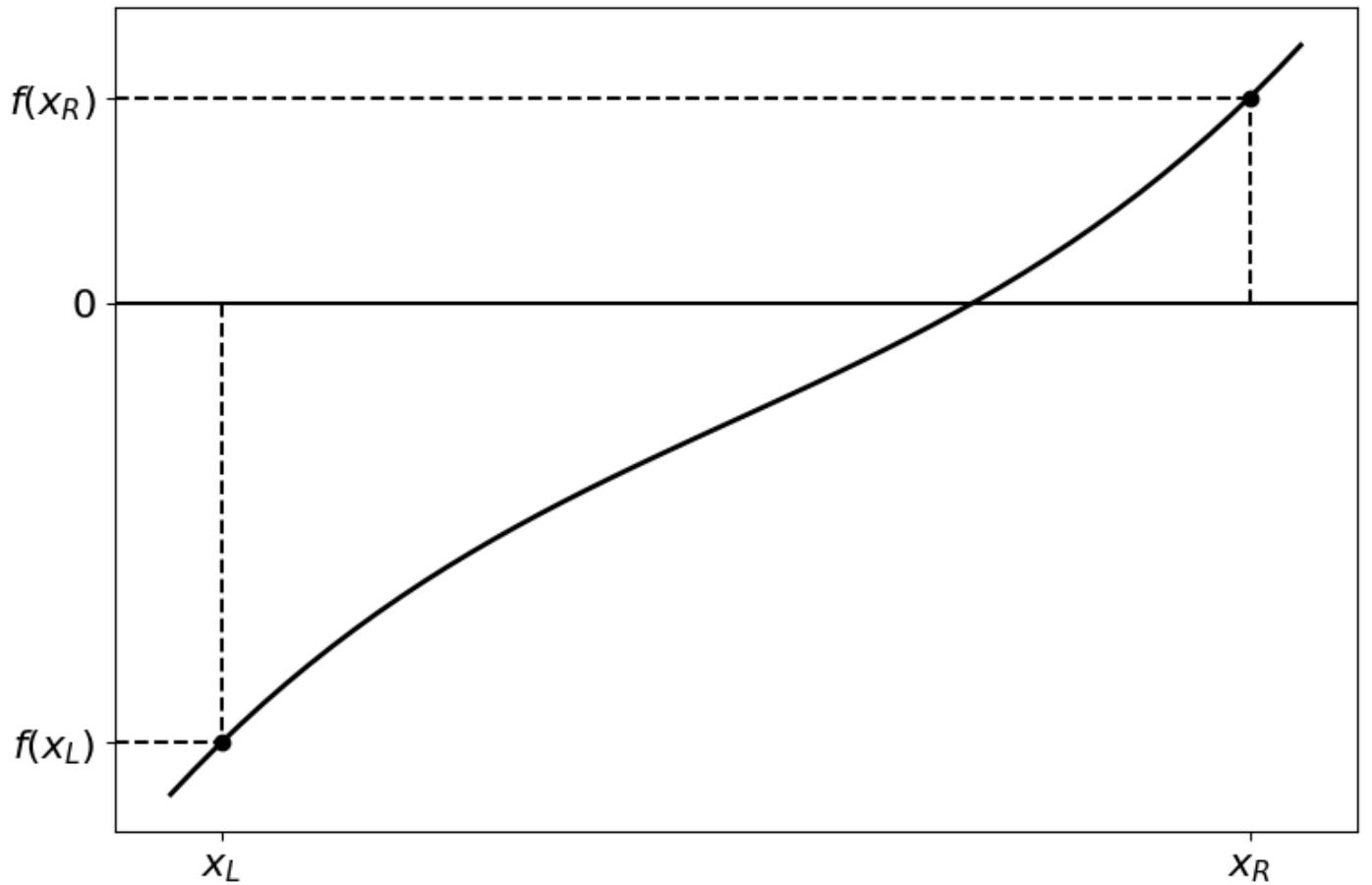
The method can be summarized as:

- 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

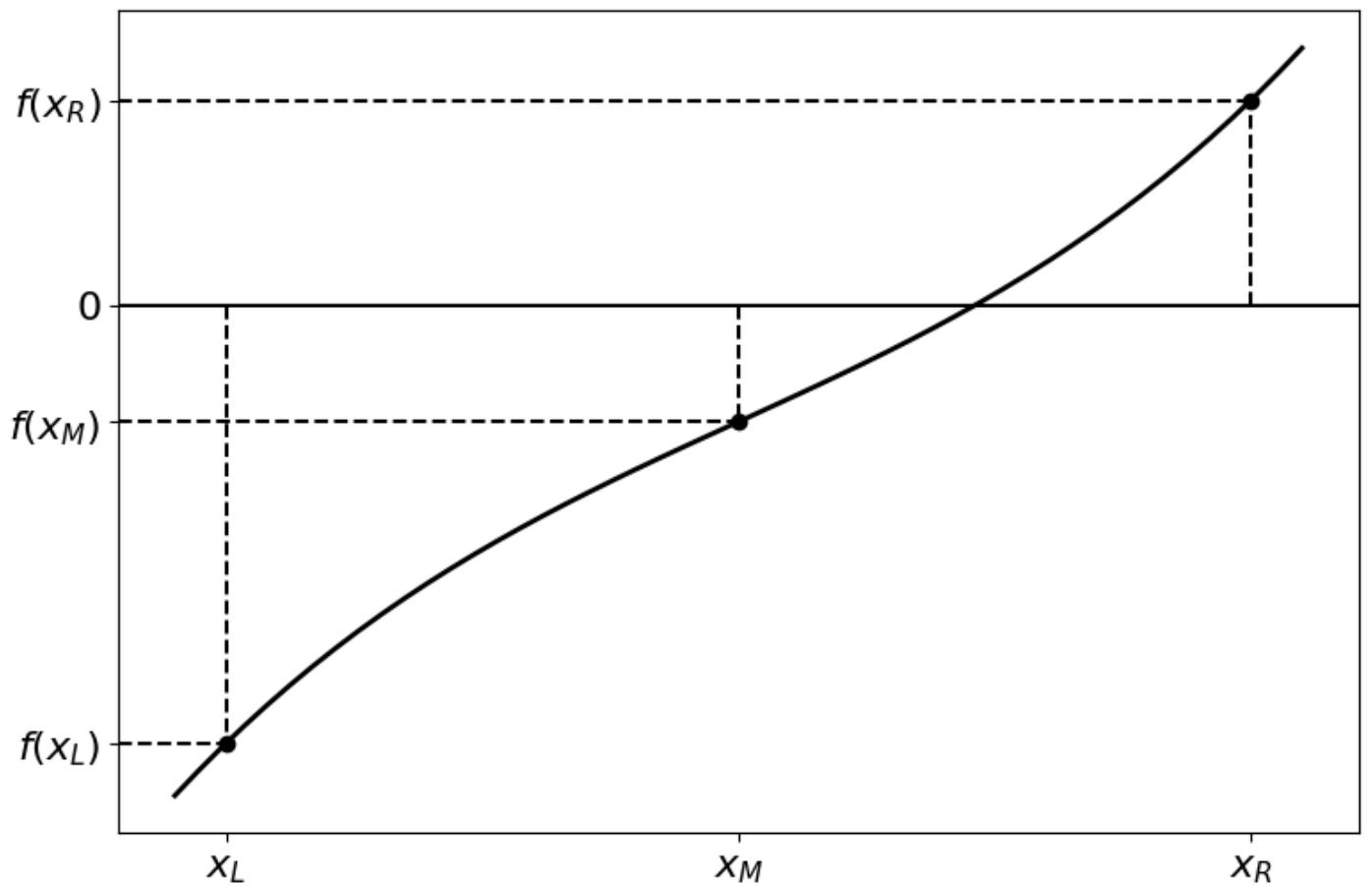
Note that with this technique you end up with an interval that contains the root, rather than an approximation for the root itself. Also note that this method will always converge on a root if one exists in the interval.

## 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 introducing the midpoint  $x_M = \frac{1}{2}(x_L + x_R)$  and calculating  $f(x_M)$ :

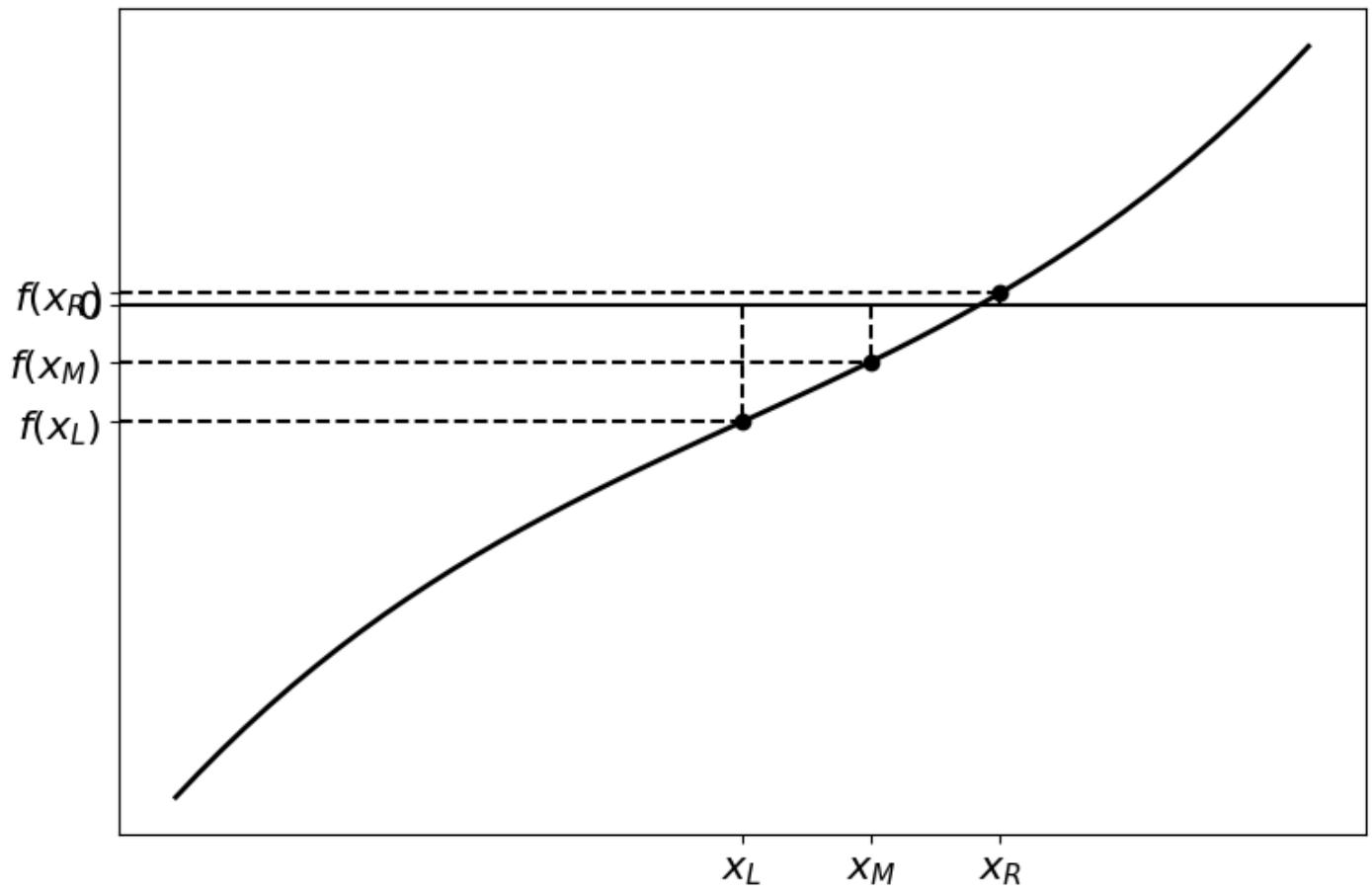
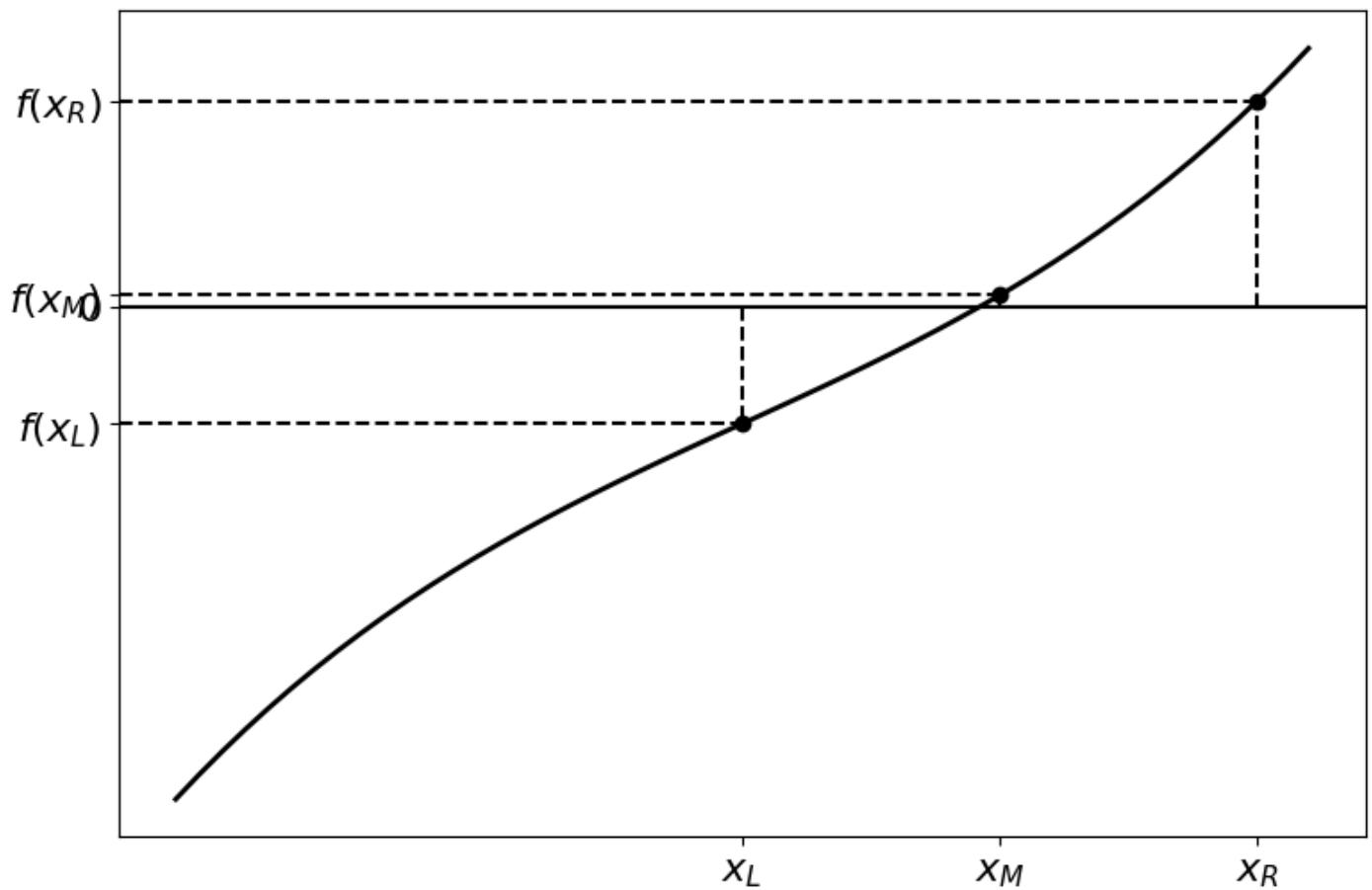


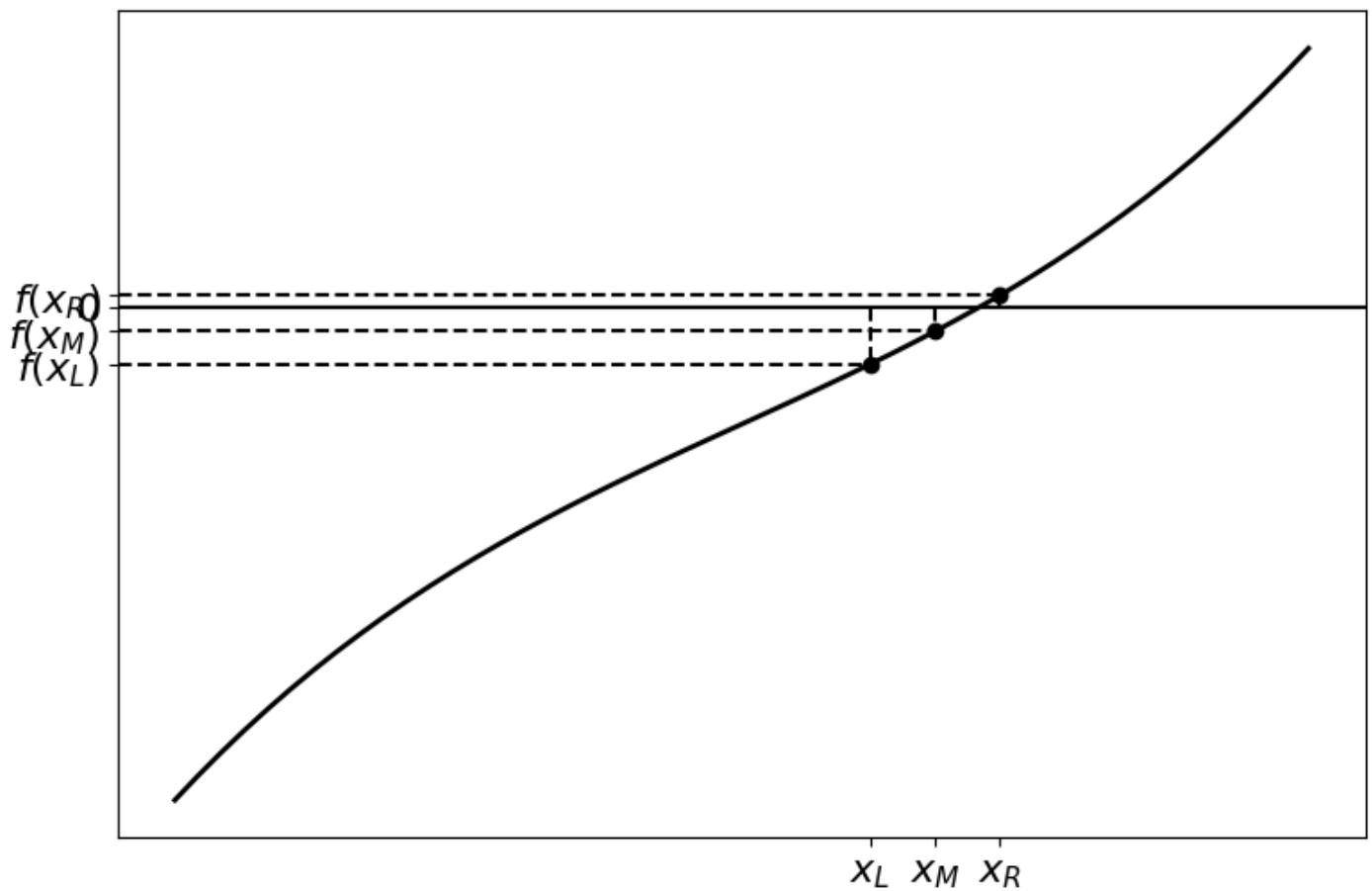
Now we need to figure out which interval contains the root. We can do this by checking if the function value changes signs at the ends of the interval, i.e. which of  $f(x_L)$  and  $f(x_R)$  is the opposite sign of  $f(x_M)$ ? An easy way to check this is to check if the product of  $f(x_L) \times f(x_M)$  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.





## 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  $|b - a|/2$  and after the  $n$ -th step the error is  $|b - a|/2^n$ . Thus, for a specified tolerance, the number of steps required is:

$$\begin{aligned} \text{tolerance} &= \frac{|b - a|}{2^n} \\ \therefore 2^n &= \frac{|b - a|}{\text{tolerance}} \\ \therefore n &= \log_2 \left( \frac{|b - a|}{\text{tolerance}} \right) \end{aligned}$$

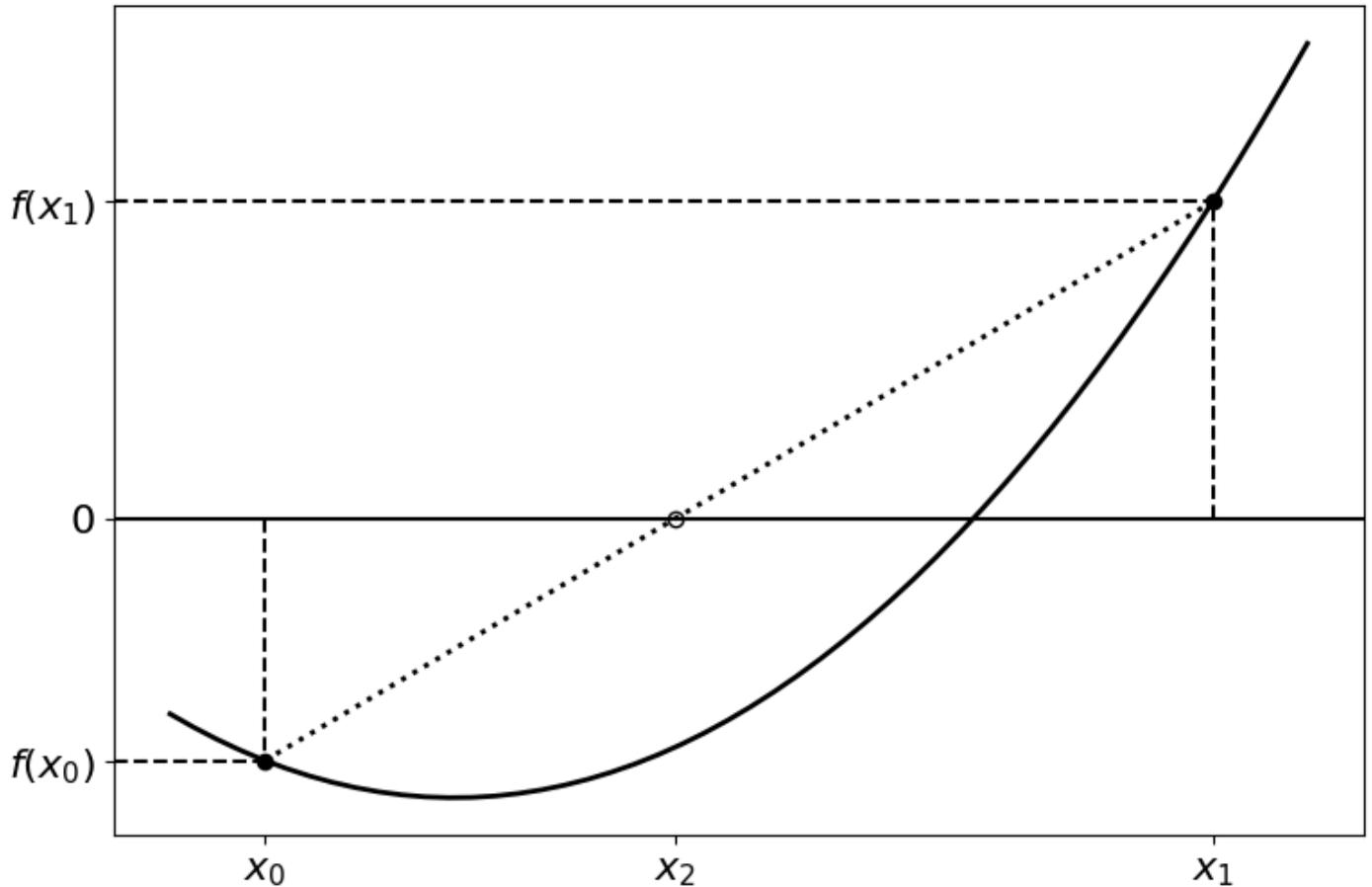
This value is rounded up to an integer.

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

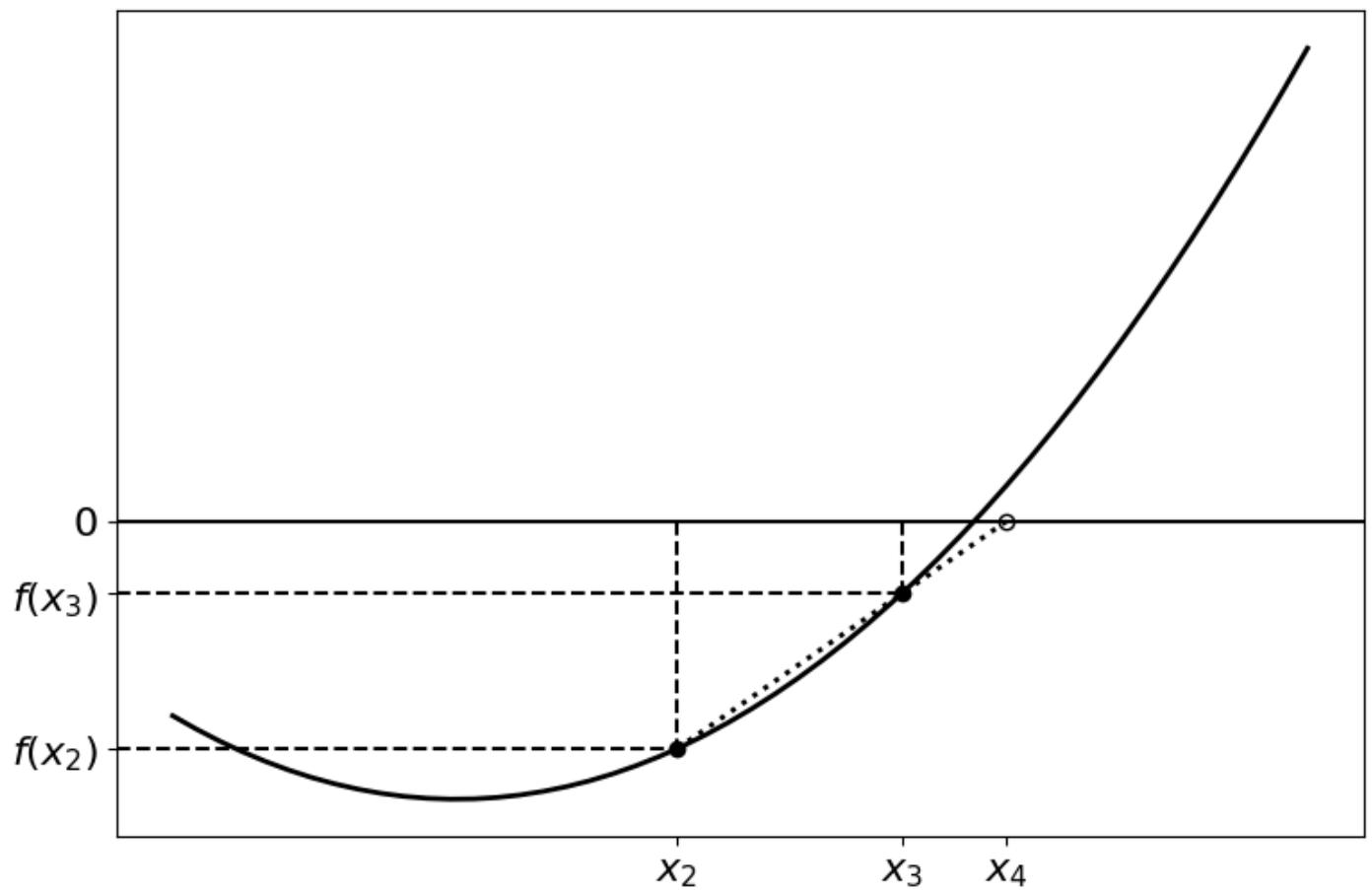
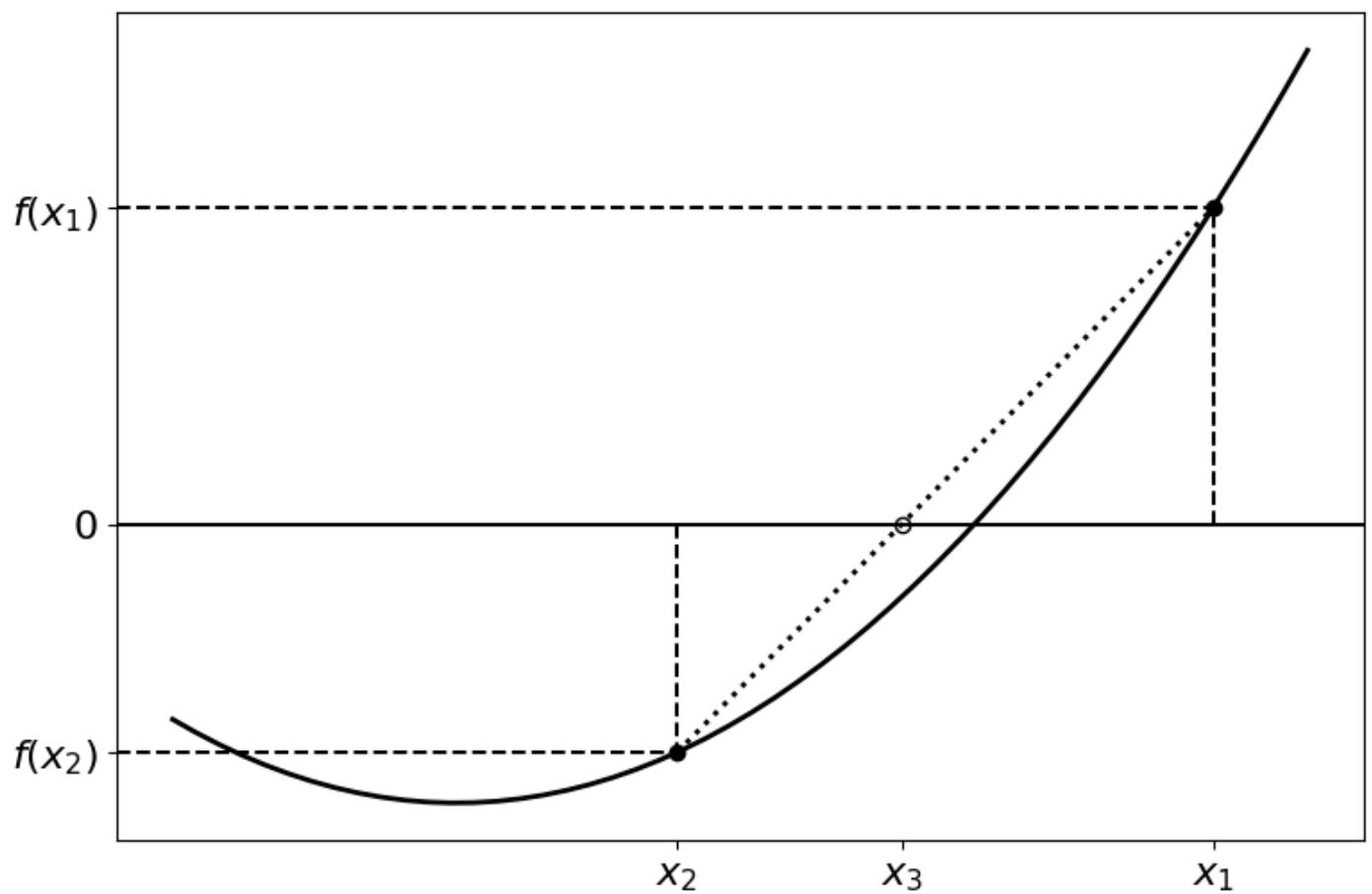
In the secant method we construct a line running between two points on the curve  $(x_0, f(x_0))$  and  $(x_1, f(x_1))$ , and find where it intersects with the  $x$ -axis:  $x_2$ :

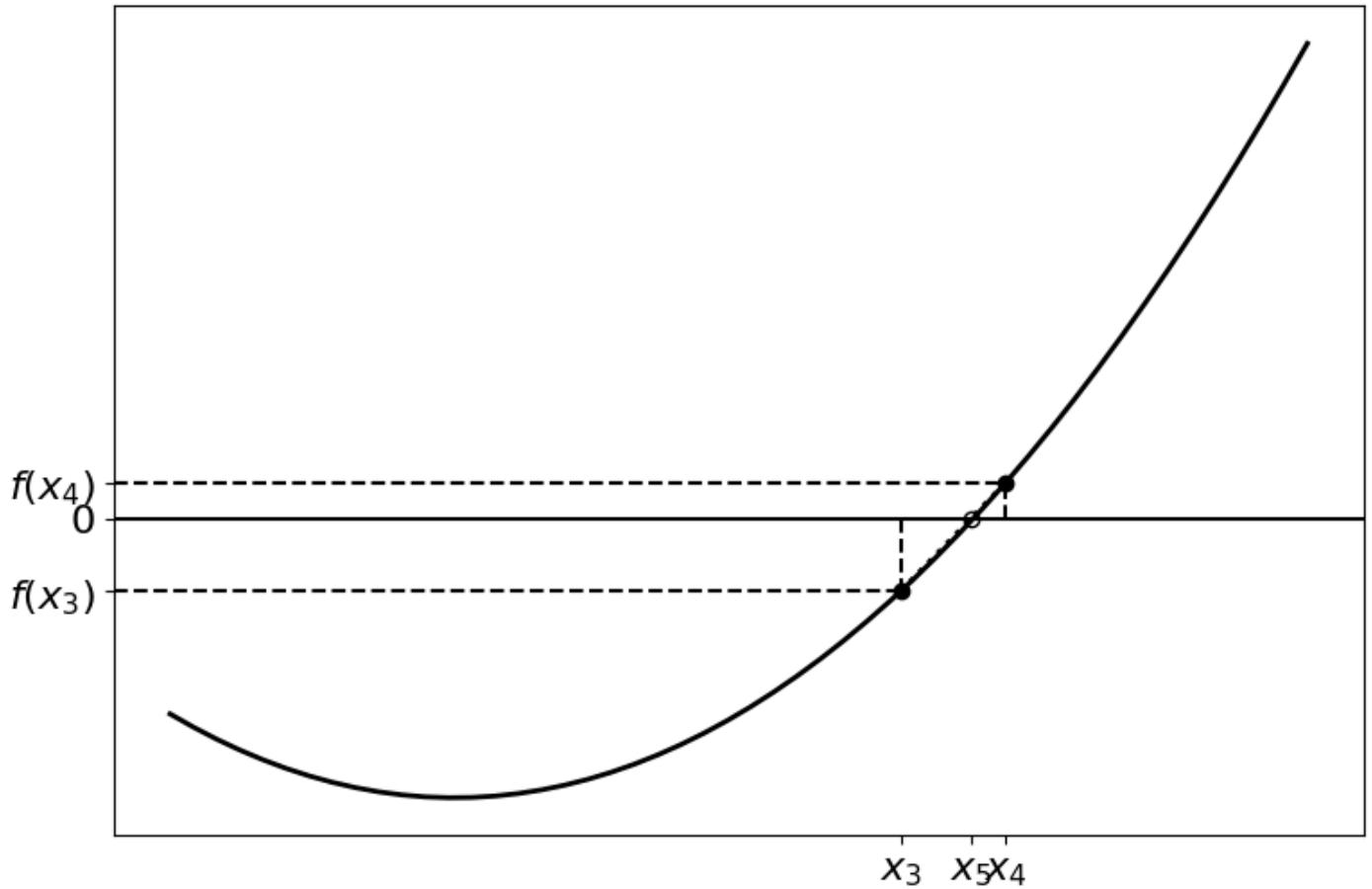


It is assumed that the new point is closer to the root. The justification behind this is beyond the scope of this course.

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:





To calculate the intersection  $x_n$  for the line constructed from the previous two points  $(x_{n-2}, f(x_{n-2}))$  and  $(x_{n-1}, f(x_{n-1}))$  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 your 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.

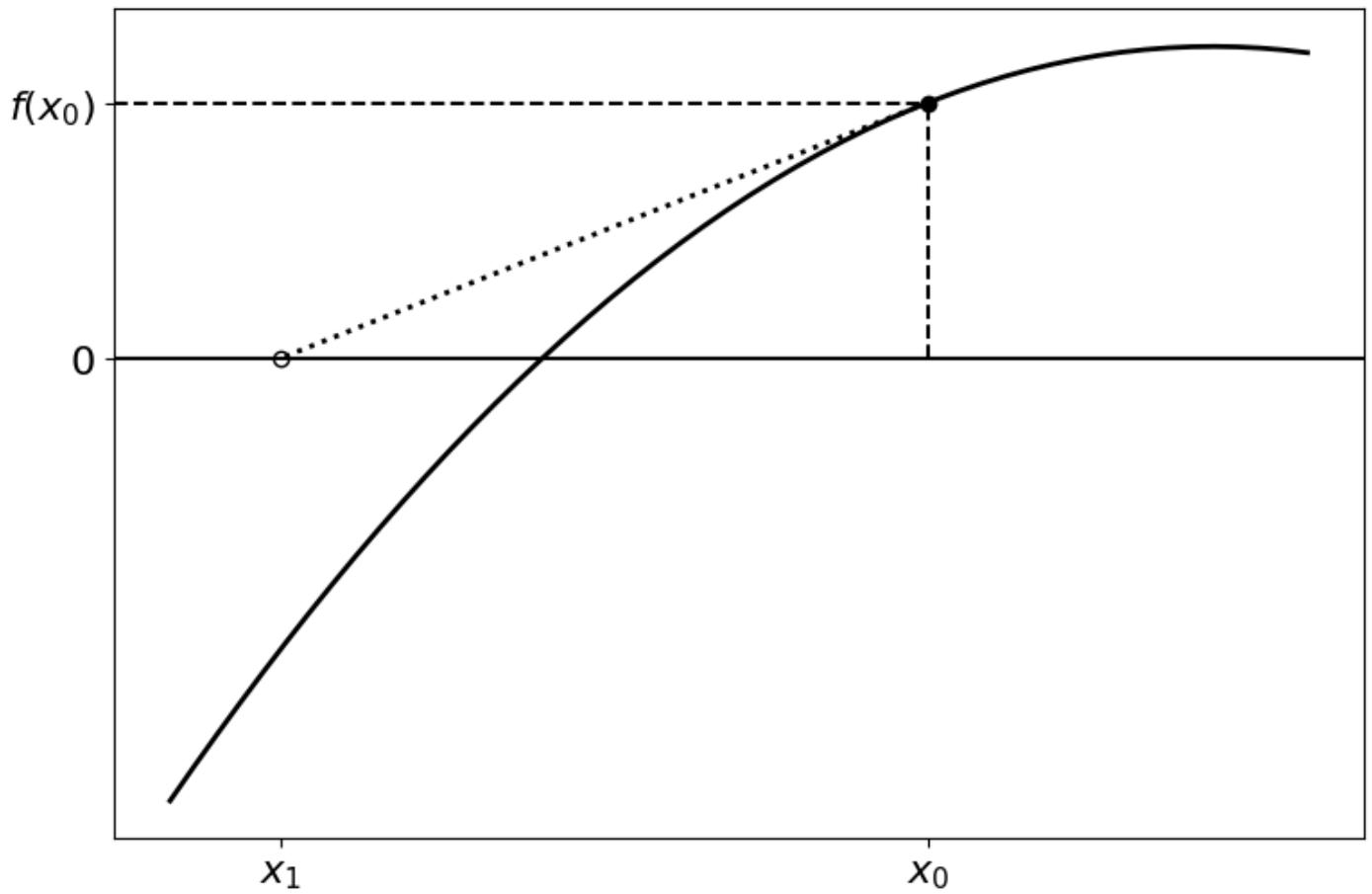
## Instability

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.

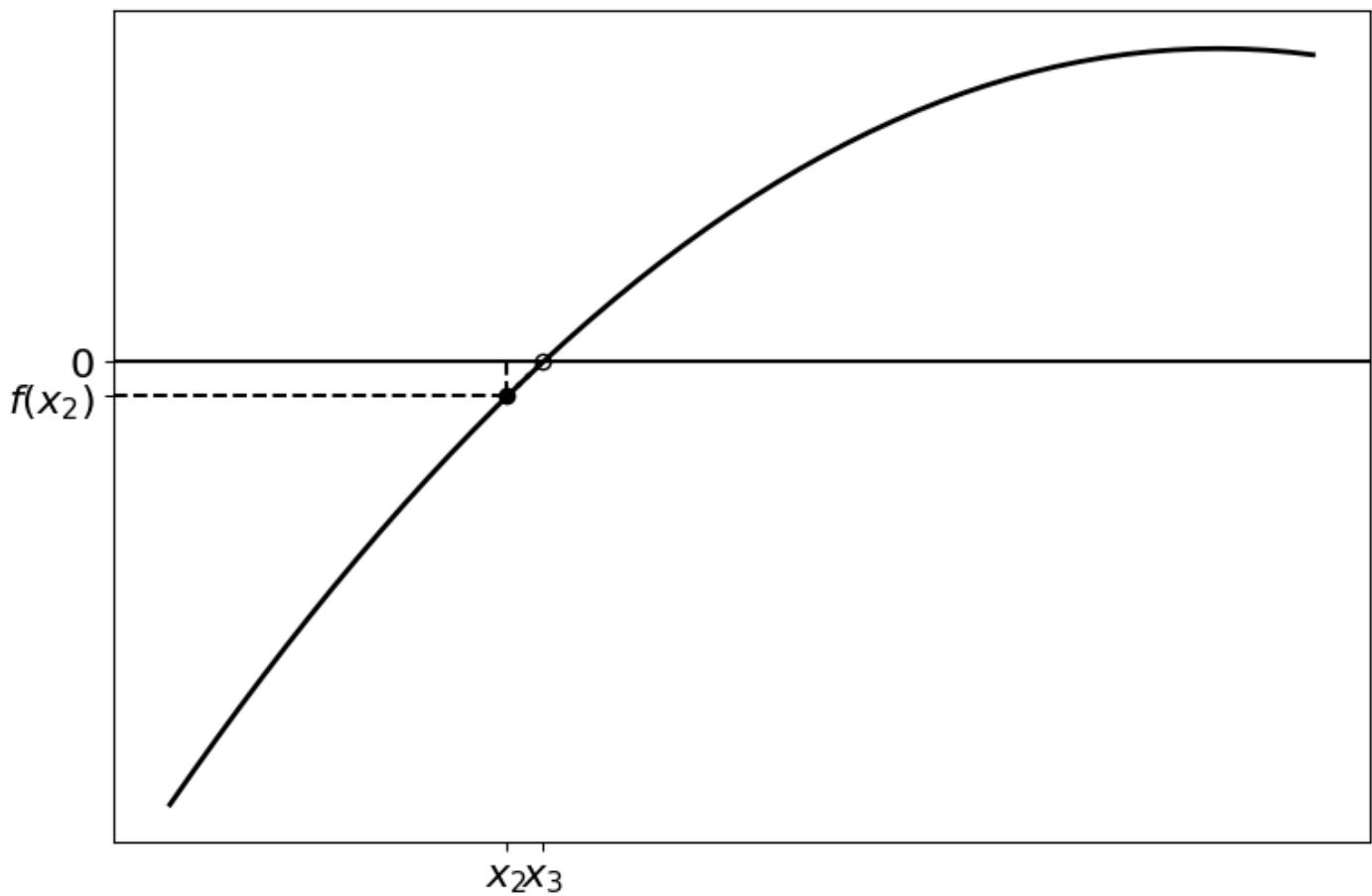
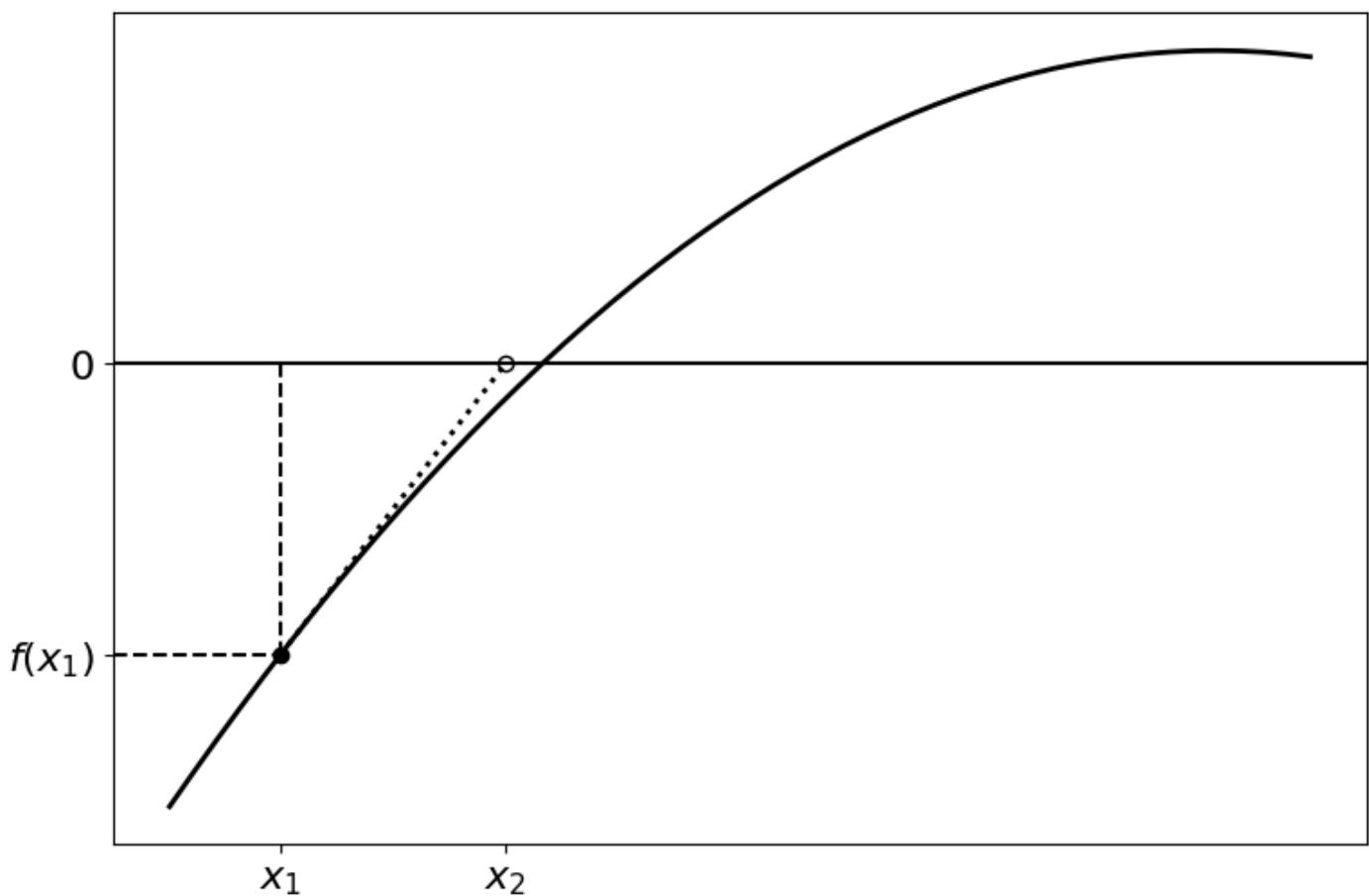
It is also possible for the solution to converge to a different root if they are in close proximity.

# Newton-Raphson Method

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  $x = x_n$ :

$$\begin{aligned} \frac{0 - f(x_{n-1})}{x_n - x_{n-1}} &= f'(x_{n-1}) \\ \therefore x_n &= x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \end{aligned}$$

## 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

	<b>Bisection</b>	<b>Secant</b>	<b>Newton-Raphson</b>
<b>Convergence</b>	Will always converge to a root inside the interval, as long as the function is well behaved.	May not converge to a root if the function has stationary points near it. May converge on neighboring roots.	
<b>Rate of Convergence</b>	Relatively slow convergence.	Fast convergence	
<b>Complexity</b>	Only requires the function, which must simply return values for given arguments on the interval.		Requires knowledge of the first derivative of the function.

# Curve Fitting

In this chapter we will cover fitting mathematical models to data sets, starting with linear regression algorithms, and leading up to using SciPy's non-linear curve fitting functions.

# Linear Regression

In linear regression the relationship between a dependent variable and one or more independent variables is modelled by a linear relation, such as:

$$y = a_0 + a_1x_1 + a_2x_2 + \cdots + a_mx_m$$

where  $y$  is the dependent variable, the  $x_j$  are the independent variables, and the  $a_j$  are scalar parameters. Note that it is the linearity of the  $a_j$  parameters that is important, any non-linear combination of  $x_j$  that do not cause a non-linear combination of  $a_j$  can be dealt with by redefining the  $x_j$  variables. The  $a_j$  parameters are often unknown and need to be determined from paired measurements of the  $y$  and  $x_j$  variables.

## Statistical Notation

Going forward we will use some statistical notation to clean up our equations.

For a data set of values for a variable  $x$ ,  $x_i$  ( $i = 1, 2, 3, \dots, N$ ):

- Expected value of  $f(x)$  (which can be considered the average of  $f(x)$ ):

$$\langle f(x) \rangle = \frac{1}{N} \sum_{i=1}^N f(x_i)$$

- Expected value of  $x$  is the mean of the data set:

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

- Standard deviation (which is an indication of the spread of the data):

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

# Linear Least Squares Minimization

## The Problem

We propose a linear functional relation between 2 measurable variables,  $x$  and  $y$ :

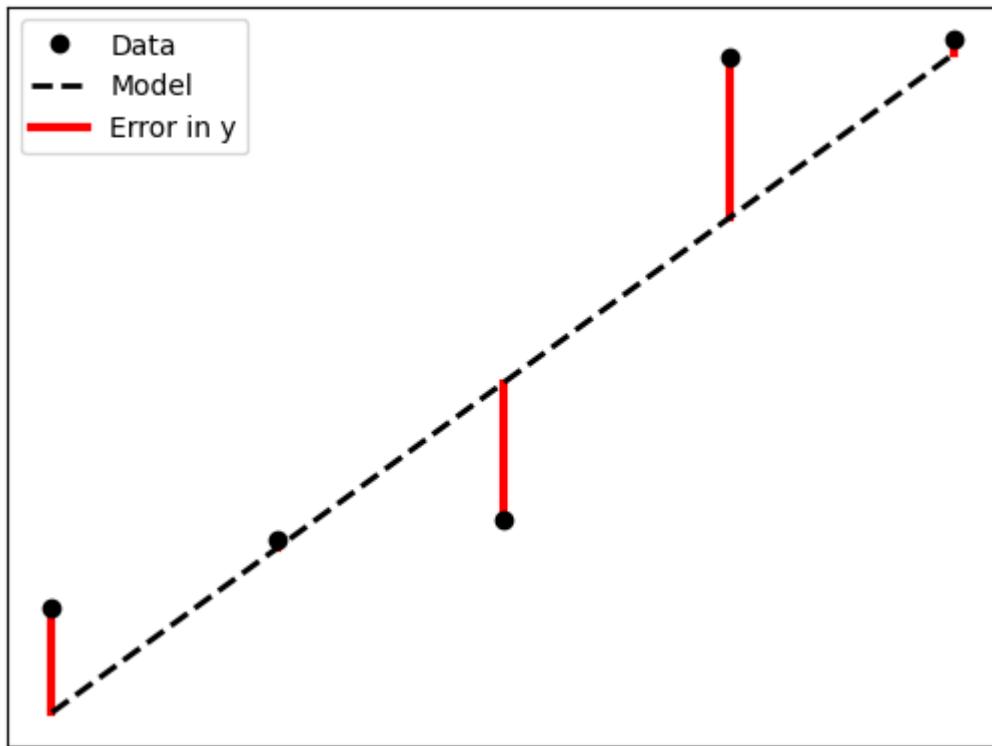
$$y = a_0 + a_1 x$$

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  $i = 1, 2, 3, \dots, N$ .

If we assume that the  $x_i$  are free of error, we can introduce error terms  $\epsilon_i$  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  $\epsilon_i^2$ :

$$S = \sum_{i=1}^N (a_0 + a_1 x_i - y_i)^2$$

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  $a_0$ :

$$\begin{aligned}\frac{\partial S}{\partial a_0} &= 2 \sum_{i=1}^N (a_0 + a_1 x_i - y_i) = 0 \\ \therefore a_0 N + a_1 \sum_{i=1}^N x_i - \sum_{i=1}^N y_i &= 0 \\ \therefore a_0 + a_1 \langle x \rangle &= \langle y \rangle\end{aligned}$$

and  $a_1$ :

$$\begin{aligned}\frac{\partial S}{\partial a_1} &= 2 \sum_{i=1}^N (a_0 + a_1 x_i - y_i) x_i = 0 \\ \therefore a_0 \sum_{i=1}^N x_i + a_1 \sum_{i=1}^N x_i^2 - \sum_{i=1}^N x_i y_i &= 0 \\ a_0 \langle x \rangle + a_1 \langle x^2 \rangle &= \langle xy \rangle\end{aligned}$$

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:

$$\begin{aligned}a_1 &= \frac{\langle xy \rangle - \langle x \rangle \langle y \rangle}{\langle x^2 \rangle - \langle x \rangle^2} \\ a_0 &= \langle y \rangle - a_1 \langle x \rangle\end{aligned}$$

## 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 the variance of  $y$  as:

$$\begin{aligned}\sigma_y^2 &= \frac{1}{N} \sum_{i=1}^N \epsilon_i^2 \\ &= \frac{1}{N} \sum_{i=1}^N (a_0 + a_1 x_i - y_i)^2\end{aligned}$$

## 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/exercise10.1/) or on [GitHub](#). The data file contains measurements of:

- $\log P$
- $M$
- $B - V$  (color, not using yet)

We will determine  $a_0$  and  $a_1$  under 2 different assumptions:

1. The error in the data is associated with  $M$
2. The error in the data is associated with  $\log P$  (this will require us to re-arrange things)

### Solution:

You are encouraged to attempt this yourself before continuing.

We start by reading in the file. We will read the data into a 2 arrays. This can be achieved using the standard library as in the page [Python Standard Library/File IO/Data Files](#), or using `numpy.loadtxt()` (documentation [here](#)). We shall use the latter as it is far more convenient:

```
import numpy as np
import matplotlib.pyplot as plt

logP, M, color = np.loadtxt('./data/cepheid_data.csv', delimiter = ',', skiprows = 1,
```

The keyword arguments used above are:

- `delimiter`: the string used to separate the data columns
- `skiprows`: the number of rows to skip from the data file (in this case the header)
- `unpack`: this makes `loadtxt` return each column in the data file as arrays, as apposed to the default of a single 2D array.

As we will be performing 2 minimizations, we will define a function to determine  $a_0$  and  $a_1$ , and  $\sigma_y$ :

```
def least_squares(x, y):
    mean_x = np.mean(x)
    mean_y = np.mean(y)

    #Note that the expectation values can be calculated using the mean function
    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 sigma(a0, a1, x, y):
    return np.sqrt(np.mean((a0 + a1*x - y)**2))
```

## Error in $M$

Let's estimate the coefficients for the relation:

$$M = a_0 + a_1 \log P$$

assuming the error resides primarily in  $M$ .

```
a0, a1 = least_squares(logP, M)
```

### Error in $\log P$

Now, if we assumed that the error resides primarily in  $\log P$ , we want to apply least squares minimization to the relation:

$$\log P = b_0 + b_1 M$$

to find the values for the coefficients  $b_0$  and  $b_1$ . These values can be used to calculate  $a_0$  and  $a_1$  by re-arranging the relation to put  $M$  as the subject:

$$M = -\frac{b_0}{b_1} + \frac{1}{b_1} \log P$$

which gives us:

$$a_0 = -\frac{b_0}{b_1}, \quad a_1 = \frac{1}{b_1}$$

```
b0, b1 = least_squares(M, logP)
```

### Plotting the solutions

Instead of printing out the values of the coefficients, let's visualize them by plotting the linear relations.

```

fontsize = 16
linewidth = 2

x = np.array([logP[0], logP[-1]]) #for the relation

y_M = a0 + a1*x #error in M
y_P = -b0/b1 + x/b1 #error in logP

fig_ceph, ax = plt.subplots()

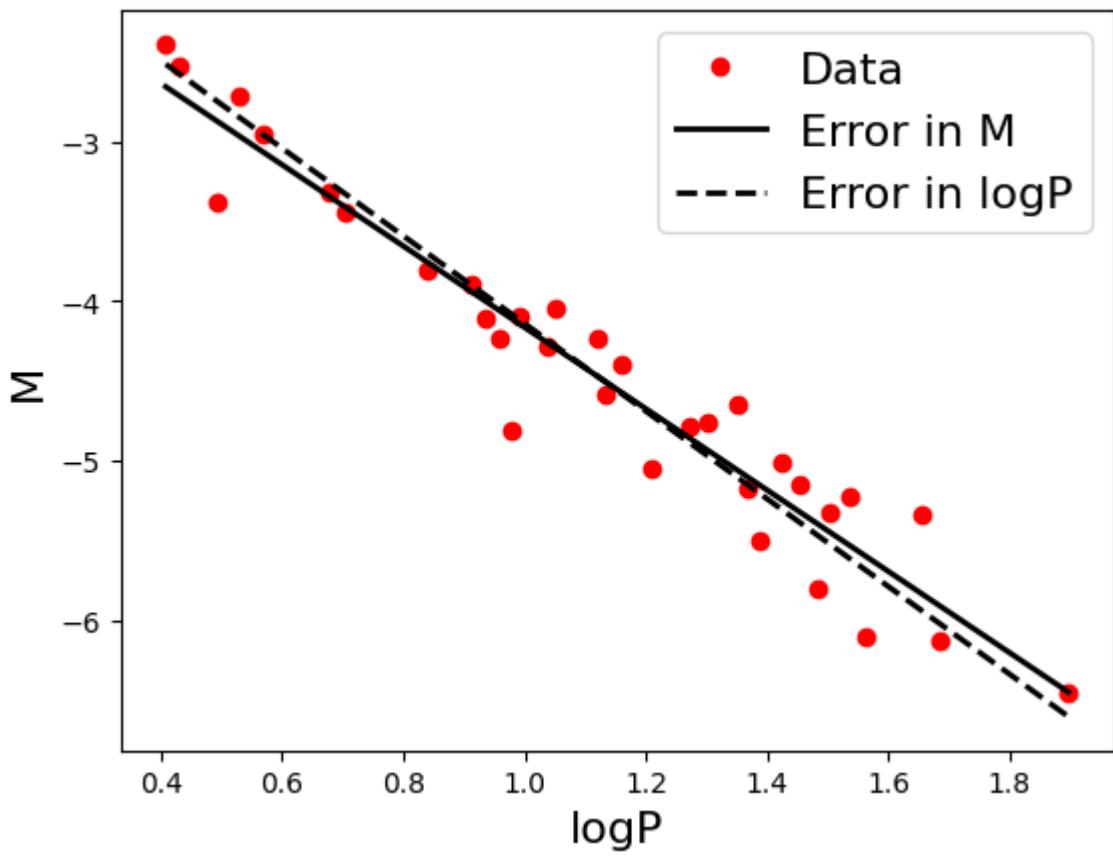
ax.plot(logP, M, 'ro', label = 'Data')
ax.plot(x, y_M, 'k', label = 'Error in M', lw = linewidth)
ax.plot(x, y_P, 'k--', label = 'Error in logP', lw = linewidth)

ax.set_xlabel('logP', fontsize = fontsize)
ax.set_ylabel('M', fontsize = fontsize)

ax.legend(fontsize = fontsize)

plt.show()

```



# Linear Chi Squared Minimization

For 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:  $\sigma_i$ .

We now want to weight the contribution that each error value  $\epsilon_i$  gives to the total error by the uncertainties  $\sigma_i$ . 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  $\chi^2$  value of the data:

$$\chi^2 = \sum_{i=1}^N \left( \frac{\epsilon_i}{\sigma_i} \right)^2$$

where each error value is weighted by dividing it by the uncertainty. Note that if all of the  $\sigma_i$  were constant, we'd be dealing with least squares (the multiplicative factor will drop out in the minimization)

## With 1 Independent Variable

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  $\sigma_i$  as the uncertainty of the  $y_i$  values for  $i = 1, \dots, N$ ,  $\chi^2$  can now be written as:

$$\begin{aligned}\chi^2 &= \sum_{i=1}^N \left( \frac{\epsilon_i}{\sigma_i} \right)^2 \\ &= \sum_{i=1}^N \left( \frac{a_0 + a_1 x_i - y_i}{\sigma_i} \right)^2\end{aligned}$$

Minimizing  $\chi^2$  with respect to  $a_0$  and  $a_1$ , will yield:

$$\begin{aligned}a_0 &= \left( \left\langle \frac{y}{\sigma^2} \right\rangle \left\langle \frac{x^2}{\sigma^2} \right\rangle - \left\langle \frac{x}{\sigma^2} \right\rangle \left\langle \frac{xy}{\sigma^2} \right\rangle \right) / D \\ a_1 &= \left( \left\langle \frac{1}{\sigma^2} \right\rangle \left\langle \frac{xy}{\sigma^2} \right\rangle - \left\langle \frac{x}{\sigma^2} \right\rangle \left\langle \frac{y}{\sigma^2} \right\rangle \right) / D\end{aligned}$$

where

$$D = \left\langle \frac{1}{\sigma^2} \right\rangle \left\langle \frac{x^2}{\sigma^2} \right\rangle - \left\langle \frac{x}{\sigma^2} \right\rangle^2$$

Note that in practice the  $1/N$  factors of the expectation values cancel out.

# Multiple Linear Least Squares Minimization

In [Numerical Methods/Curve Fitting/Linear Regression/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 parameters to be determined.

On this page we will look at the more generic case, where we solve the problem for an arbitrary number of independent variables and parameters.

## Two Independent 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  $(x_{1,i}, x_{2,i}, y_i)$  pairs for  $i = 1, 2, 3, \dots, N$ . If we attribute the dispersion of this data from the functional relation to error in the  $y_i$  terms,  $\epsilon_i$ , then we can relate the data points with:

$$\begin{aligned} y_i + \epsilon_i &= a_0 + a_1 x_{1,i} + a_2 x_{2,i} \\ \therefore \epsilon_i &= a_0 + a_1 x_{1,i} + a_2 x_{2,i} - y_i \end{aligned}$$

The sum of errors squared is given by:

$$\begin{aligned} S &= \sum_{i=1}^N \epsilon_i^2 \\ &= \sum_{i=1}^N (a_0 + a_1 x_{1,i} + a_2 x_{2,i} - y_i)^2 \end{aligned}$$

We want to minimize  $S$  with respect to each of the constants,  $a_0$ ,  $a_1$  and  $a_2$ :

$$\frac{\partial S}{\partial a_0} = 2 \sum_{i=0}^n (a_0 + a_1 x_{1,i} + a_2 x_{2,i} - y_i) = 0$$

,

$$\frac{\partial S}{\partial a_1} = 2 \sum_{i=0}^n (a_0 + a_1 x_{1,i} + a_2 x_{2,i} - y_i) x_{1,i} = 0$$

and

$$\frac{\partial S}{\partial a_2} = 2 \sum_{i=0}^n (a_0 + a_1 x_{1,i} + a_2 x_{2,i} - y_i) x_{2,i} = 0$$

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:

$$\begin{aligned} \mathbf{XA} &= \mathbf{Y} \\ \therefore \mathbf{A} &= \mathbf{X}^{-1}\mathbf{Y} \end{aligned}$$

## Worked 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  $a_2 = 1.49$ ). You may wish to use NumPy matrices, which have the inverse property I ( $\mathbf{X}^{-1}$  is achieved using X.I, for the appropriately defined X matrix), and can be multiplied directly

Try this yourself before reading the solution that follows.

### Solution:

As in the **Numerical Methods/Linear Regression Algorithms/Linear Least Squares Minimization** worked example, we shall read in the data using numpy.loadtxt():

```
import numpy as np
import matplotlib.pyplot as plt

logP, M, color = np.loadtxt('./data/cepheid_data.csv', delimiter = ',', skiprows = 1,
```

Here we have:

$$\begin{aligned}y &= M \\x_1 &= \log P \\x_2 &= (B - V)\end{aligned}$$

We will now proceed to construct the matrices using the data. Starting with simpler  $\mathbf{Y}$  matrix:

```
#Defining Y as a column matrix
Y = np.matrix([
    [np.mean(M)],
    [np.mean(logP * M)],
    [np.mean(color * M)]
])
```

Note that  $\mathbf{X}$  is symmetric about the diagonal, i.e.  $\mathbf{X}_{k l} = \mathbf{X}_{l k}$ , we will make use of this to reduce the number of calculations we need to perform.

```
X = np.matrix(np.ones( (3, 3) )) #Using an array generating function

# X[0, 0] is just 1, so we leave it

#The first row
X[0, 1] = np.mean(logP)
X[0, 2] = np.mean(color)

#The first column (which is the transpose of the first row)
X[1, 0] = X[0, 1]
X[2, 0] = X[0, 2]

#The diagonal
X[1, 1] = np.mean(logP * logP)
X[2, 2] = np.mean(color * color)

#The off-diagonal elements
X[1, 2] = np.mean(logP * color)
X[2, 1] = X[1, 2]
```

Now we calculate the  $\mathbf{A}$  matrix:

```
A = X.I * Y

#Printing the constants

for i in range(3):
    print(f'a{i+1} =', '{:.3}'.format(A[i,0]) )
```

```
a1 = -2.15
a2 = -3.12
a3 = 1.49
```

## Arbitrarily Many Independent Variables

Consider a linear functional relation between measurable variables  $x_1, x_2, x_3, \dots, x_m$  and  $y$ :

$$\begin{aligned}y &= a_0 + a_1x_1 + a_2x_2 + \cdots + a_mx_m \\&= a_0 + \sum_{j=1}^m a_jx_j\end{aligned}$$

where  $a_0, a_1, \dots$  and  $a_m$  are unknown constants.

Suppose we have a data set of measured  $(x_{1,i}, x_{2,i}, \dots, x_{mi}, y_i)$  values for  $i = 1, 2, 3, \dots, N$ . 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:

$$y_i + \epsilon_i = a_0 + \sum_{j=1}^m a_jx_{j,i}$$

The sum of errors squared can thus be written as:

$$S = \sum_{i=1}^N \left( a_0 + \left( \sum_{j=1}^m a_jx_{j,i} \right) - y_i \right)^2$$

We want to find the values of  $a_0, a_1, \dots$  and  $a_m$  which gives us the minimum value of  $S$ . Minimizing  $S$  with respect to  $a_0$  gives us:

$$\frac{\partial S}{\partial a_0} = 2 \sum_{i=1}^N \left( a_0 + \left( \sum_{j=1}^m a_j x_{j,i} \right) - y_i \right) = 0$$

Distributing the sum over  $i$  amongst the terms:

$$\therefore N a_0 + \left( \sum_{j=1}^m a_j \sum_{i=1}^N x_{j,i} \right) - \sum_{i=1}^N y_i = 0$$

Dividing by  $N$ :

$$\therefore a_0 + \left( \sum_{j=1}^m a_j \frac{1}{N} \sum_{i=1}^N x_{j,i} \right) - \frac{1}{N} \sum_{i=1}^N y_i = 0$$

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  $k = 1, 2, \dots, m$ , following a similar line of algebraic manipulation as above:

$$\begin{aligned} \frac{\partial S}{\partial a_k} &= \sum_{i=1}^N 2x_{k,i} \left( a_0 + \left( \sum_{j=1}^m a_j x_{j,i} \right) - y_i \right) = 0 \\ \therefore a_0 \sum_{i=1}^N x_{k,i} + \sum_{j=1}^m a_j \left( \sum_{i=1}^N x_{k,i} x_{j,i} \right) - \sum_{i=1}^N x_{k,i} y_i &= 0 \\ \therefore a_0 \langle x_k \rangle + \sum_{j=1}^m a_j \langle x_k x_j \rangle &= \langle x_k y \rangle \end{aligned}$$

Writing the results for  $a_0$  and  $a_k$  ( $k = 1, \dots, m$ ) into a system of equations, expanding the sum over  $j$ :

$$\begin{aligned}
a_0 + a_1 \langle x_1 \rangle + a_2 \langle x_2 \rangle + \dots + a_m \langle x_m \rangle &= \langle y \rangle \\
a_0 \langle x_1 \rangle + a_1 \langle x_1^2 \rangle + a_2 \langle x_1 x_2 \rangle + \dots + a_m \langle x_1 x_m \rangle &= \langle x_1 y \rangle \\
a_0 \langle x_2 \rangle + a_1 \langle x_2 x_1 \rangle + a_2 \langle x_2^2 \rangle + \dots + a_m \langle x_2 x_m \rangle &= \langle x_2 y \rangle \\
a_0 \langle x_3 \rangle + a_1 \langle x_3 x_1 \rangle + a_2 \langle x_3 x_2 \rangle + \dots + a_m \langle x_3 x_m \rangle &= \langle x_3 y \rangle \\
&\vdots + \vdots + \vdots + \ddots + \vdots = \vdots \\
a_0 \langle x_m \rangle + a_1 \langle x_m x_1 \rangle + a_2 \langle x_m x_2 \rangle + \dots + a_m \langle x_m^2 \rangle &= \langle x_m y \rangle
\end{aligned}$$

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.

$$\begin{aligned}
\mathbf{X}\mathbf{A} &= \mathbf{Y} \\
\therefore \mathbf{A} &= \mathbf{X}^{-1}\mathbf{Y}
\end{aligned}$$

## 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 we shall design it to support any number of  $x_j$  variables.

We will start by designing a function that takes the data in as two arguments:

$$\mathbf{y\_data} = [y_1, y_2, y_3, \dots, y_N]$$

$$\mathbf{x\_data} = \begin{bmatrix} [x_{1,1}, x_{1,2}, \dots, x_{1,N}], \\ [x_{2,1}, x_{2,2}, \dots, x_{2,N}], \\ [x_{3,1}, x_{3,2}, \dots, x_{3,N}], \\ [\vdots, \vdots, \ddots, \vdots], \\ [x_{m,1}, x_{m,2}, \dots, x_{m,N}] \end{bmatrix}$$

as NumPy arrays.

## Calculating Expectation Values

Note that for each of the sums along the data sets ( $\sum_{i=1}^N$ ), we will be summing along the rows. For example, for the quantity:

$$\langle x_1 \rangle = \frac{1}{N} \sum_{i=1}^N x_{1,i}$$

we can use:

```
np.mean(x_data[0, :])
```

and for

$$\langle x_1 x_2 \rangle = \frac{1}{N} \sum_{i=1}^N x_{1,i} x_{2,i}$$

we can use

```
np.mean(x_data[0, :] * x_data[1, :])
```

where we have made use of NumPy array's vectorized operations.

## Constructing the $\mathbf{X}$ Matrix

Now, let's break down the structure of the matrix:

$$\mathbf{X} = \begin{pmatrix} 1 & \langle x_1 \rangle & \langle x_2 \rangle & \langle x_3 \rangle & \cdots & \langle x_m \rangle \\ \langle x_1 \rangle & \langle x_1^2 \rangle & \langle x_1 x_2 \rangle & \langle x_1 x_3 \rangle & \cdots & \langle x_1 x_m \rangle \\ \langle x_2 \rangle & \langle x_2 x_1 \rangle & \langle x_2^2 \rangle & \langle x_2 x_3 \rangle & \cdots & \langle x_2 x_m \rangle \\ \langle x_3 \rangle & \langle x_3 x_1 \rangle & \langle x_3 x_2 \rangle & \langle x_3^2 \rangle & \cdots & \langle x_3 x_m \rangle \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \langle x_m \rangle & \langle x_m x_1 \rangle & \langle x_m x_2 \rangle & \langle x_m x_3 \rangle & \cdots & \langle x_m^2 \rangle \end{pmatrix}$$

the dimensions of this matrix is  $(m + 1) \times (m + 1)$ , we can get the value for  $m$  from the dimensions of the `x_data` array:

```
m = x_data.shape[0]
```

from this we can create a matrix of ones, which we will populate later:

```
X = np.array(np.ones((m+1, m+1)))
```

Now, as we have noted before,  $\mathbf{X}$  is a symmetric matrix. That is for for row  $k$  and column  $l$ ,  $\mathbf{X}_{kl} = \mathbf{X}_{lk}$ . 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  $x_j$ :

$$\mathbf{X}_{00} = 1$$

and

$$\mathbf{X}_{0l} = \langle x_l \rangle \quad \text{where } l = 1, 2, \dots, m$$

Note that here we are indexing  $\mathbf{X}$  from 0 to better translate it to code (keep in mind that the `x_data` array also starts with a 0 index, so the  $x_l$  data is in row  $l - 1$ ):

```
for l in range(m):
    X[0, l + 1] = np.mean(x_data[l, :])

    # Setting the values for the first column
    # remember that X[k, 1] = X[1, k]
    X[l + 1, 0] = X[0, l + 1]
```

Now, consider the triangle off of the diagonal. That is the region:

$$\begin{pmatrix} - & - & - & - & \cdots & - \\ - & - & \langle x_1 x_2 \rangle & \langle x_1 x_3 \rangle & \cdots & \langle x_1 x_m \rangle \\ - & - & - & \langle x_2 x_3 \rangle & \cdots & \langle x_2 x_m \rangle \\ - & - & - & - & \cdots & \langle x_3 x_m \rangle \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ - & - & - & - & \cdots & - \end{pmatrix}$$

This region exhibits the pattern:

$$\mathbf{X}_{kl} = \langle x_k x_l \rangle \quad \text{where } 1 \leq k \leq m \text{ and } l > k$$

The diagonal has a fairly simple pattern, starting from (row, column) (1, 1):

$$\mathbf{X}_{kk} = \langle x_k^2 \rangle \quad \text{where } 1 \leq k \leq m$$

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:

$$\mathbf{X}_{kl} = \langle x_k x_l \rangle \quad \text{where } 1 \leq k \leq m \text{ and } l \geq k$$

In the code this becomes:

```
# Inner matrix

for k in range(m):
    for l in range(k, m):
        X[k + 1, l + 1] = np.mean( x_data[k, :] * x_data[l, :] )
```

```
#Setting the value for the lower triangle
X[1 + 1, k + 1] = X[k + 1, 1 + 1]
```

That covers the  $\mathbf{X}$  matrix.

## Constructing the $\mathbf{Y}$ Matrix

Now let's construct the matrix:

$$\mathbf{Y} = \begin{pmatrix} \langle y \rangle \\ \langle x_1 y \rangle \\ \langle x_2 y \rangle \\ \vdots \\ \langle x_m y \rangle \end{pmatrix}$$

This is fairly straight forward, with

$$\mathbf{Y}_{0,0} = \langle y \rangle$$

and

$$\mathbf{Y}_{k,0} = \langle x_k y \rangle \quad \text{where } k = 1, \dots, m$$

```
#Creating the Y column matrix:
Y = np.matrix( np.zeros( (m + 1, 1) ) )

#First entry
Y[0, 0] = np.mean(y_data)

#The remainder of the entries
for k in range(m):
    Y[k + 1, 0] = np.mean( x_data[k, :] * y_data )
```

## Finding Matrix $\mathbf{A}$ (Or Solving For the $a_j$ )

Lastly, to solve for our  $a_j$  values, we consider the matrix:

$$\mathbf{A} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}$$

This fits into the matrix equation

$$\mathbf{X}\mathbf{A} = \mathbf{Y}$$

where we've already constructed  $\mathbf{X}$  and  $\mathbf{Y}$ . All that's left is to solve the equation by inverting  $\mathbf{X}$ :

$$\mathbf{A} = \mathbf{X}^{-1}\mathbf{Y}$$

To achieve this numerically, we simply take the inverse of  $\mathbf{X}$ ,  $\mathbf{X.I}$ :

```
#Finding A:
```

```
A = X.I*Y
```

This  $\mathbf{A}$  matrix is a column matrix. As a bonus, if we wanted to turn it into a one-dimensional array, we can use the `flatten()` method:

```
np.array(A).flatten()
```

## Putting It All Together

Putting this all together into a function:

```

import numpy as np
import matplotlib.pyplot as plt

def least_squares(y_data, x_data):
    m = x_data.shape[0]

    X = np.matrix(np.ones((m+1, m+1)))

    #First row
    for l in range(m):
        X[0, l + 1] = np.mean(x_data[l, :])

        # Setting the values for the first column
        # remember that X[k, 1] = X[l, k]
        X[l + 1, 0] = X[0, l + 1]

    # Upper triangle
    for k in range(m):
        for l in range(k, m):
            X[k + 1, l + 1] = np.mean( x_data[k, :] * x_data[l, :] )

        #Setting the value for the lower triangle
        X[l + 1, k + 1] = X[k + 1, l + 1]

    #Creating the Y column matrix:
    Y = np.matrix( np.zeros( (m + 1, 1) ) )

    #First entry
    Y[0, 0] = np.mean(y_data)

    #The remainder of the entries
    for k in range(m):
        Y[k + 1, 0] = np.mean( x_data[k, :] * y_data )

    #Finding A:
    A = X.I*Y

    return np.array(A).flatten()

```

## Worked Example - Applying the Solution to the Cepheid Variables Data

Now, lets apply this function to the Cepheid Variables data to find the unknown coefficients for the functional relation of the magnitude ( $M$ ), period ( $P$ ) and color ( $B - V$ ):

$$M = a_0 + a_1 \log P + a_2(B - V)$$

We'll unpack the data as in the [Numerical Methods/Linear Regression Algorithms/Linear Least Squares Minimization](#) example, and then pack it into the structure that is required by the function.

```
#Reading the data
logP, M, color = np.loadtxt('./data/cepheid_data.csv', delimiter = ',', skiprows = 1,
a_arr = least_squares(M, np.array([logP, color]))

for i, a in enumerate(a_arr):
    print(f'a_{i} =', '{:.3}'.format(a))
```

```
a_0 = -2.15
a_1 = -3.12
a_2 = 1.49
```

Which matches the results from the worked example above.

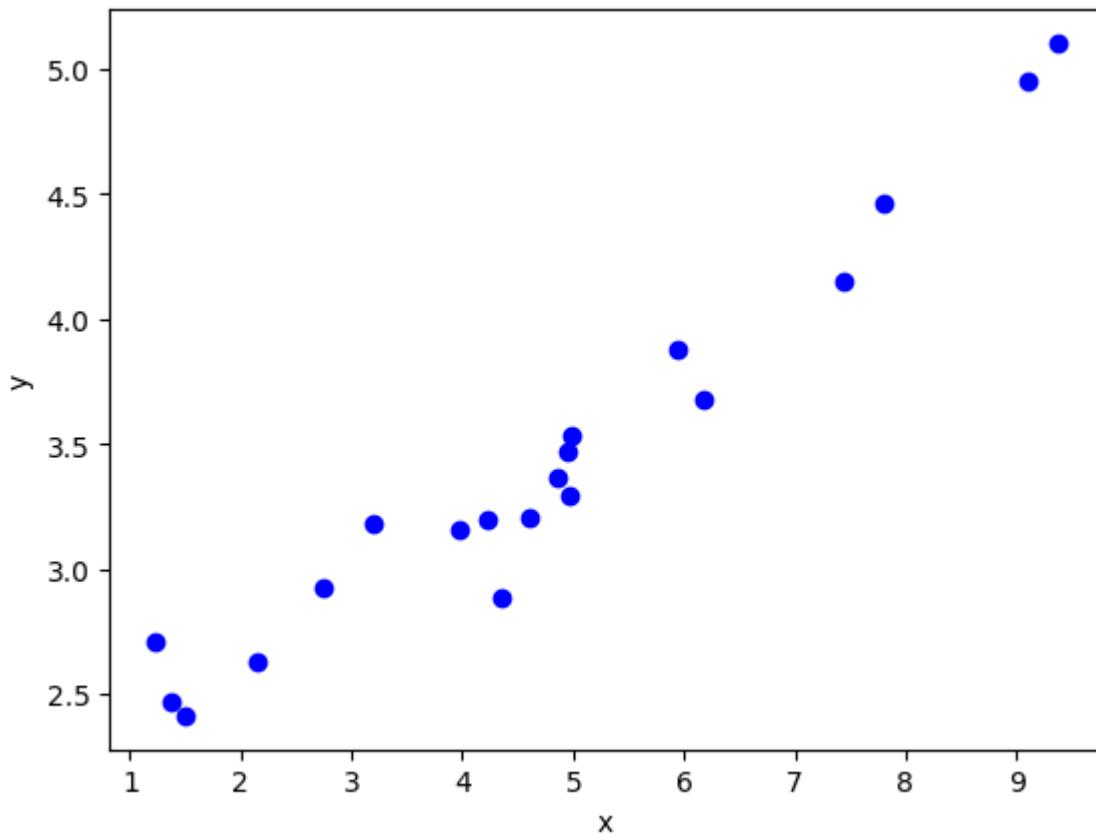
It is left as an exercise to the reader to implement a solution for multiple linear  $\chi^2$  minimization.

# Non-Linear Least Squares Minimization with `scipy.optimize.least_squares`

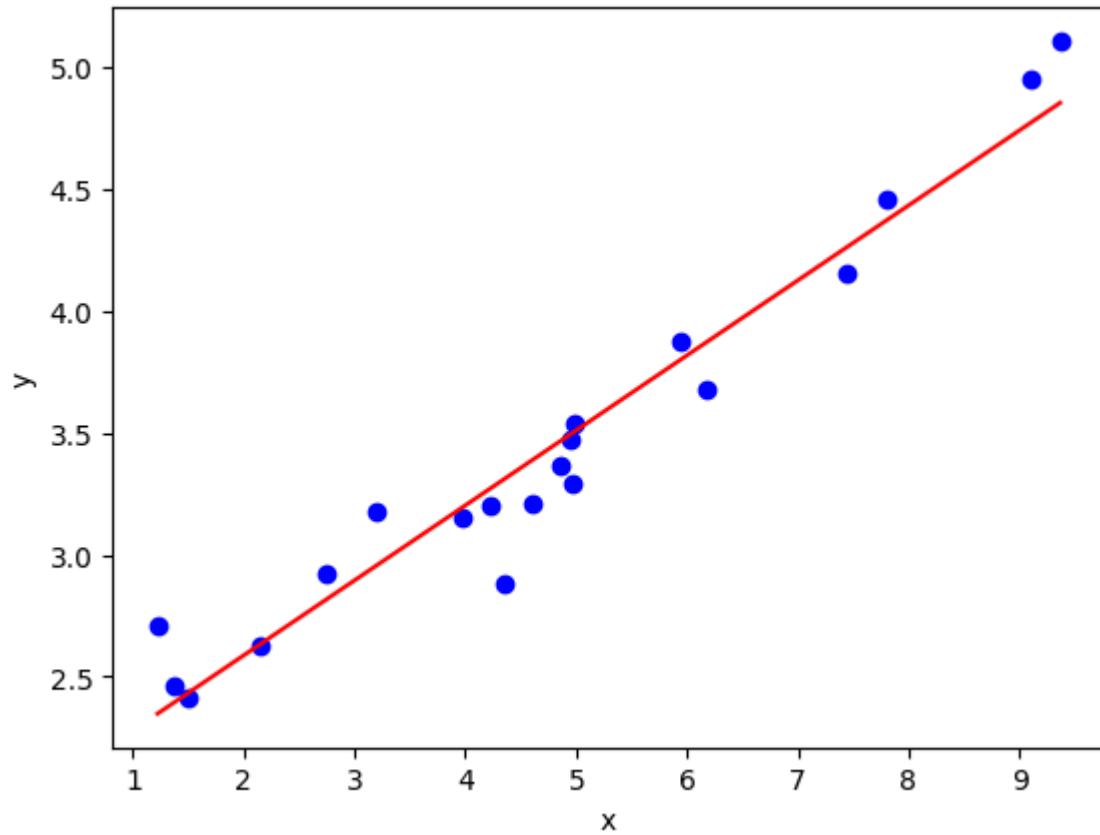
So far we have only considered functional relationships that are linear in the unknown constants. Non-linear cases are far more complicated and generally require purely numerical solutions. For non-linear cases, we can use the `least_squares` function from the `scipy.optimize` submodule.

## An Example of a Nonlinear Model

Consider the data found in the data file `nonlinear_data.csv` on [GitHub](#), plotted below:

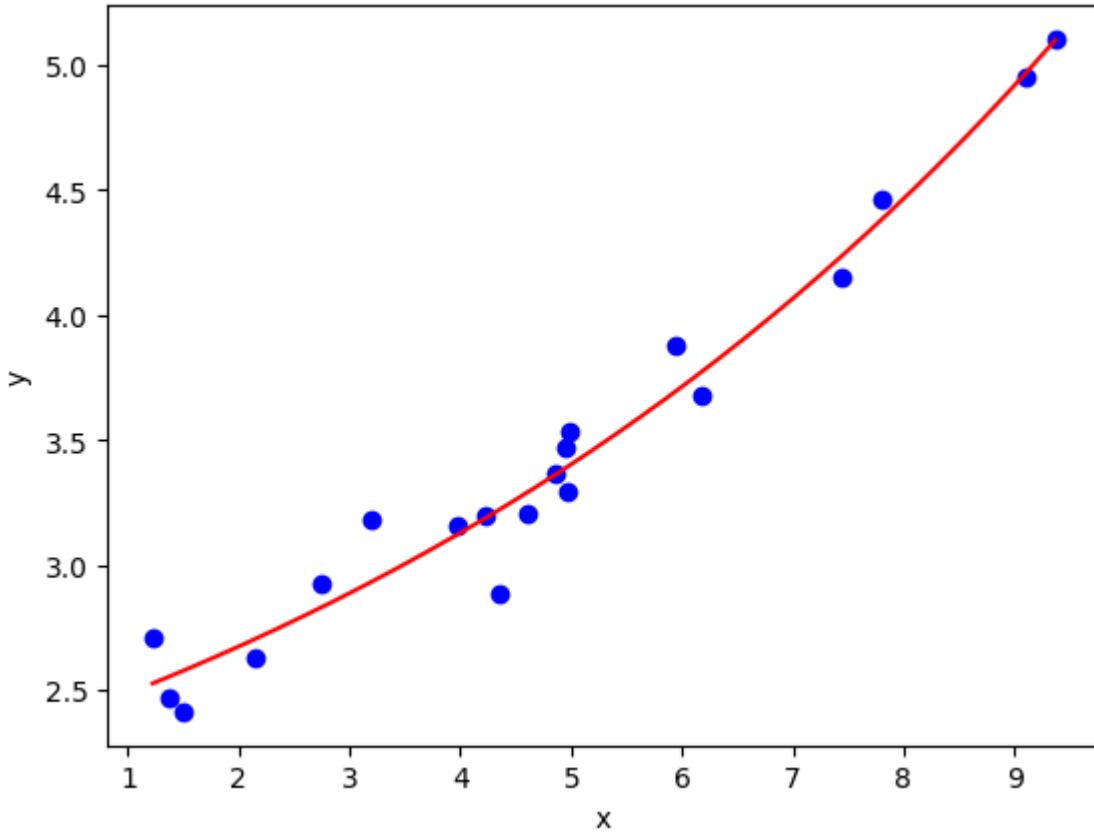


Though the data may appear to follow a linear trend, this is not the case. Consider the linear fit below:



and compare this to a fit using an exponential function as the relation:

$$y = a_0 + a_1 e^{a_2 x}$$



Note that this functional relation is non-linear for  $a_2$ . Applying the method for 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 the aforementioned function from **SciPy** to solve our problem. In essence 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 (which is taken as the solution).

## Nonlinear Least Squares Minimization with `scipy.optimize.least_squares`

Unlike for the linear case, finding the  $a_j$  values which best fit the data will require starting with an initial guess for these values. If possible, it is advised to visualize the model produced by the initial fit and compare it to the data. As we will demonstrate, in certain cases it is possible that the algorithm will not converge on a desired solution if inappropriate initial values are used.

The call signature of `least_squares` (including only the arguments of immediate interest to us) is:

```
least_squares(fun, x0, ..., bounds = (-np.inf, np.inf), ..., args = (), kwargs = ())
```

where

- `fun` is a callable object (function) referred to as the “residual”. The squared sum of the return values from `fun` is the quantity that is minimized. In the case of least squares minimization, this is the error for each  $y$  variable.

The call signature of `fun` is `fun(x, *args, **kwargs)`, where

- `x` is an array of the  $a_j$  values
- `args` is a tuple of additional arguments (the same `args` argument passed into the `least_squares` function will be used here)
- `kwargs` is a dictionary of additional keyword arguments (the same `kwargs` argument passed into the `least_squares` function will be used here)
- `x0` is an array of the initial guess for our  $a_j$  values.
- `args` a tuple of optional additional arguments to pass into `fun`. We will mostly use this to pass in the  $y$  and  $x_j$  data.
- `kwargs` a dictionary of optional additional keyword arguments to pass into `fun`.
- `bounds` is a tuple of array-like values. If limits should be imposed on allowed  $a_j$  values, then you can set these here. The structure of the limits are  $([a_0 \text{ min}, a_1 \text{ min}, \dots, a_m \text{ min}], [a_0 \text{ max}, a_1 \text{ max}, \dots, a_m \text{ max}])$ . You can use `np.inf` if you want to leave a value unbounded.

The return value of the `least_squares` function is an object with the following fields of interest to us:

- `x`: an array of the solution found for the  $a_j$  values.
- `success`: boolean, True if the solution has converged.

## Worked Example

Let's use `least_squares` to fit the functional relation:

$$\begin{aligned}y &= a_0 + a_1 e^{a_2 x} \\&= f(x; \vec{a})\end{aligned}$$

to the `nonlinear_data.csv` data.

First we shall define the functional relation:

```
def f(a, x):
    return a[0] + a[1] * np.exp(a[2] * x)
```

and use this to define the residuals. For regular least-squares, we will use the error as the residuals:

$$\epsilon = f(x; \vec{a}) - y$$

in Python this looks like:

```
def err(a, x, y):
    return f(a, x) - y
```

Note that, if we wanted to make this a bit more generic, we could pass the function `f` into the `err` function as an argument.

Putting this into practice:

```
import numpy as np
import matplotlib.pyplot as plt

#Importing scipy.optimize.leastsq only
from scipy.optimize import least_squares

#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

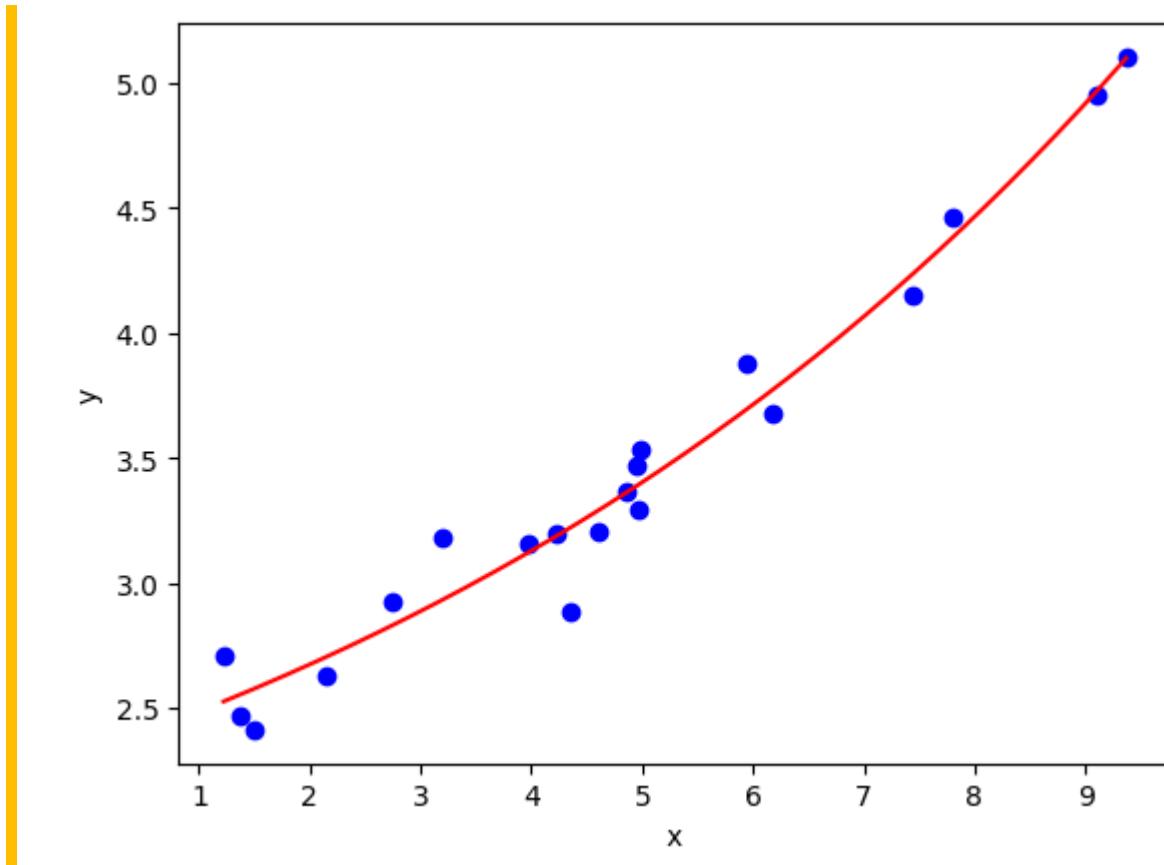
#Reading the data
# The `unpack` keyword argument separates the columns into individual arrays
xdata, ydata = np.loadtxt('data/nonlinear_data.csv', delimiter = ',', unpack = True)

#Performing the fit
a0 = [1.5, 0.6, 0.2] #initial guess

fit = least_squares(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(fit.x, x), 'r-')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

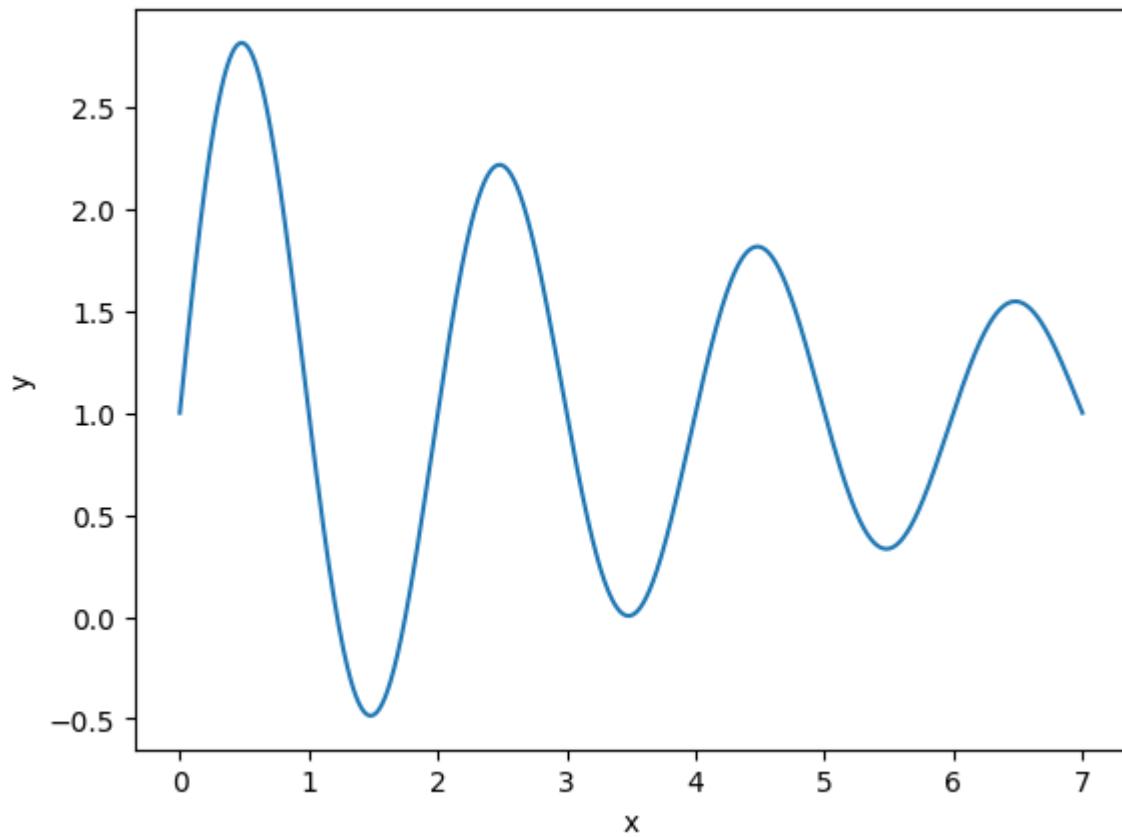


## Solutions Converging on Local Minima

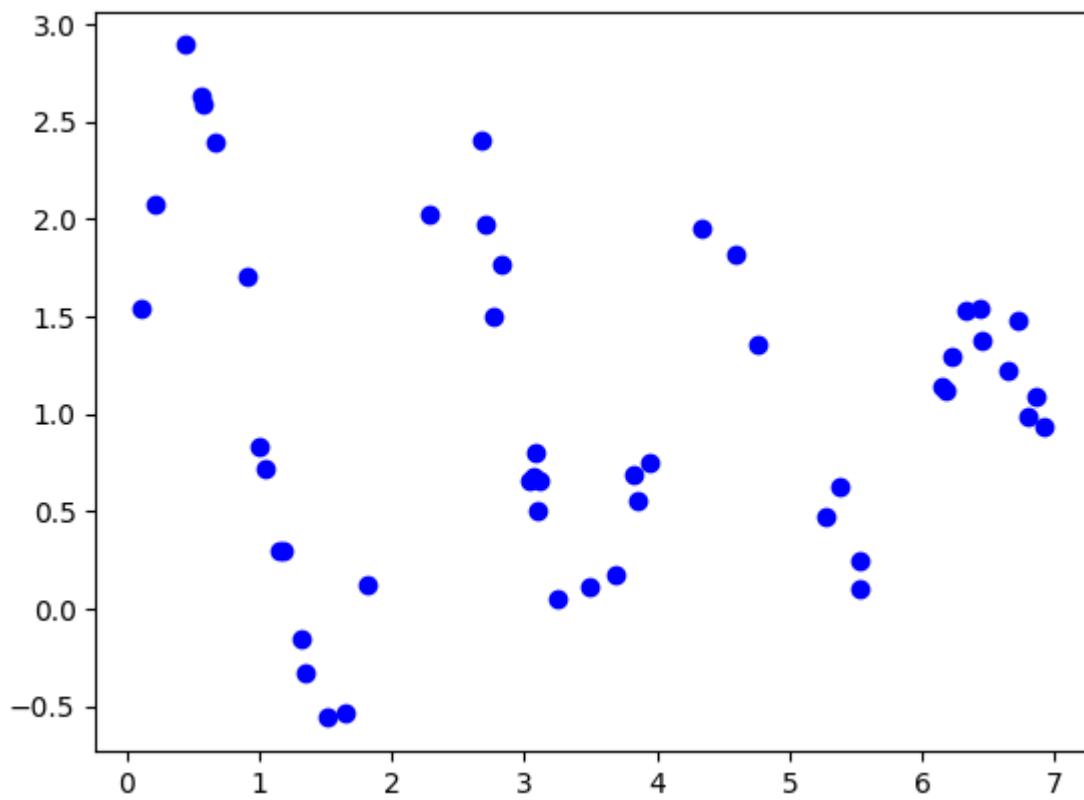
As mentioned before, the numerical algorithm is complete once it has minimized the objective function (the sum of errors squared in our case) to a **local minimum**. It is possible for the solution to not represent the global minimum, which is the ideal solution to obtain.

Let's take a relatively simple example to illustrate this. Consider the functional relation:

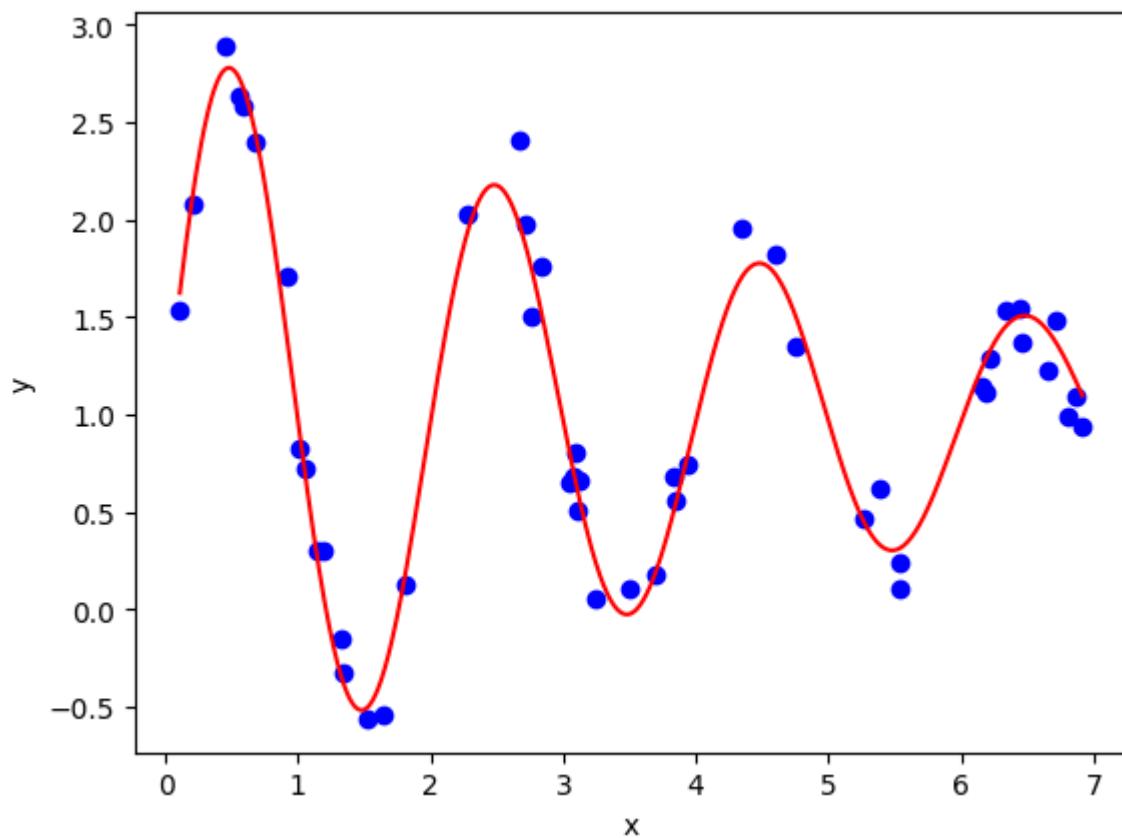
$$y = a_0 + a_1 e^{-a_2 x} \sin(a_3 x)$$



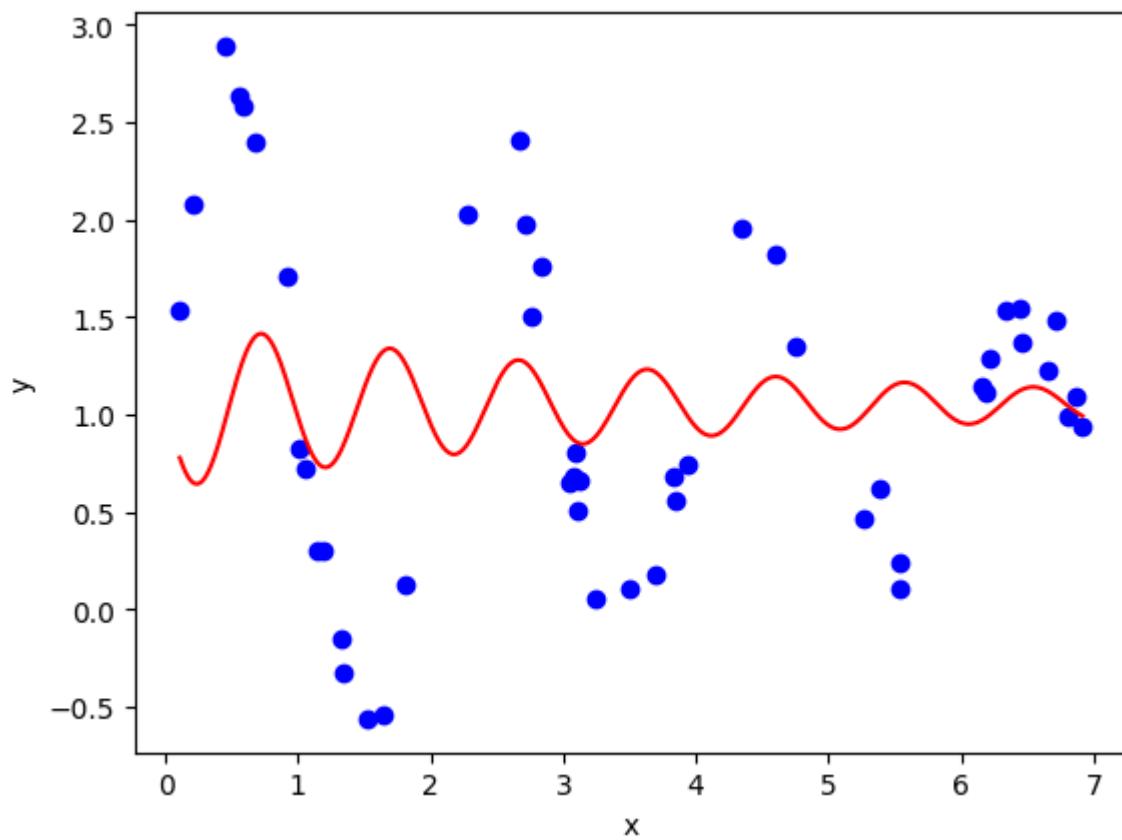
Given a set of data characterized by this relation:

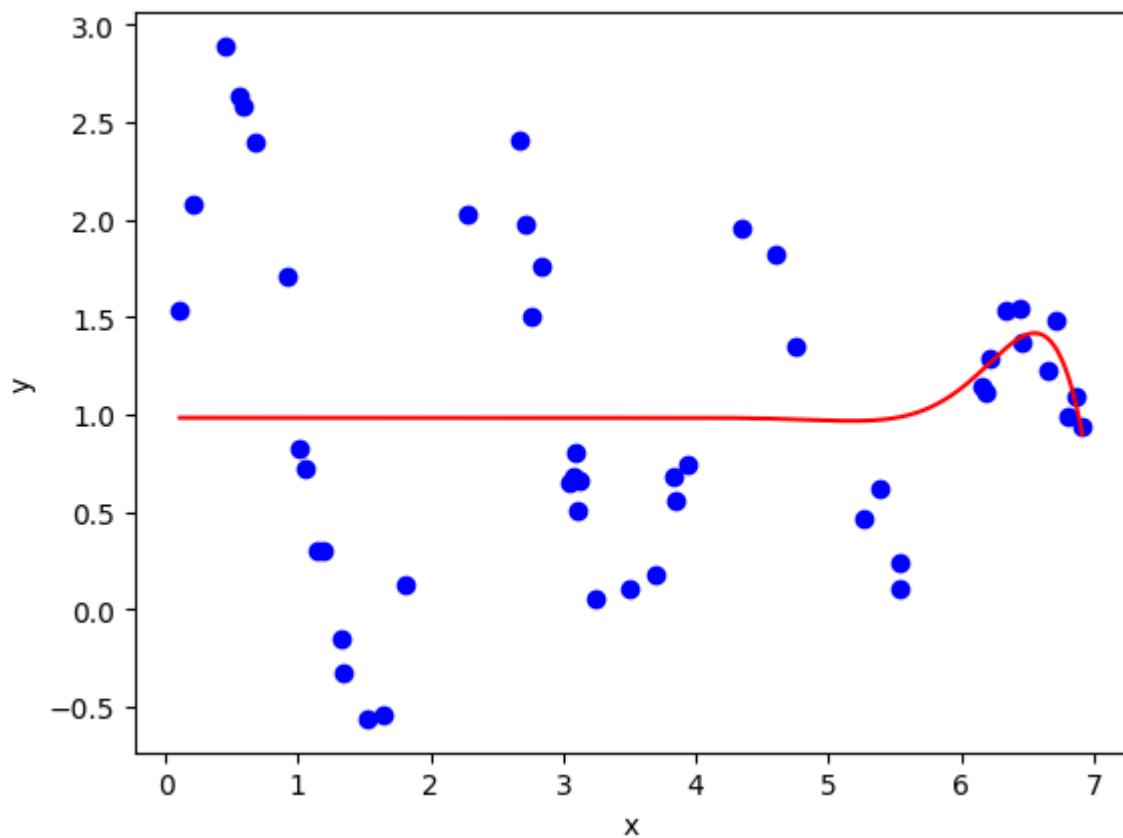


It is relatively easy to find a good fit using `leastsq`:

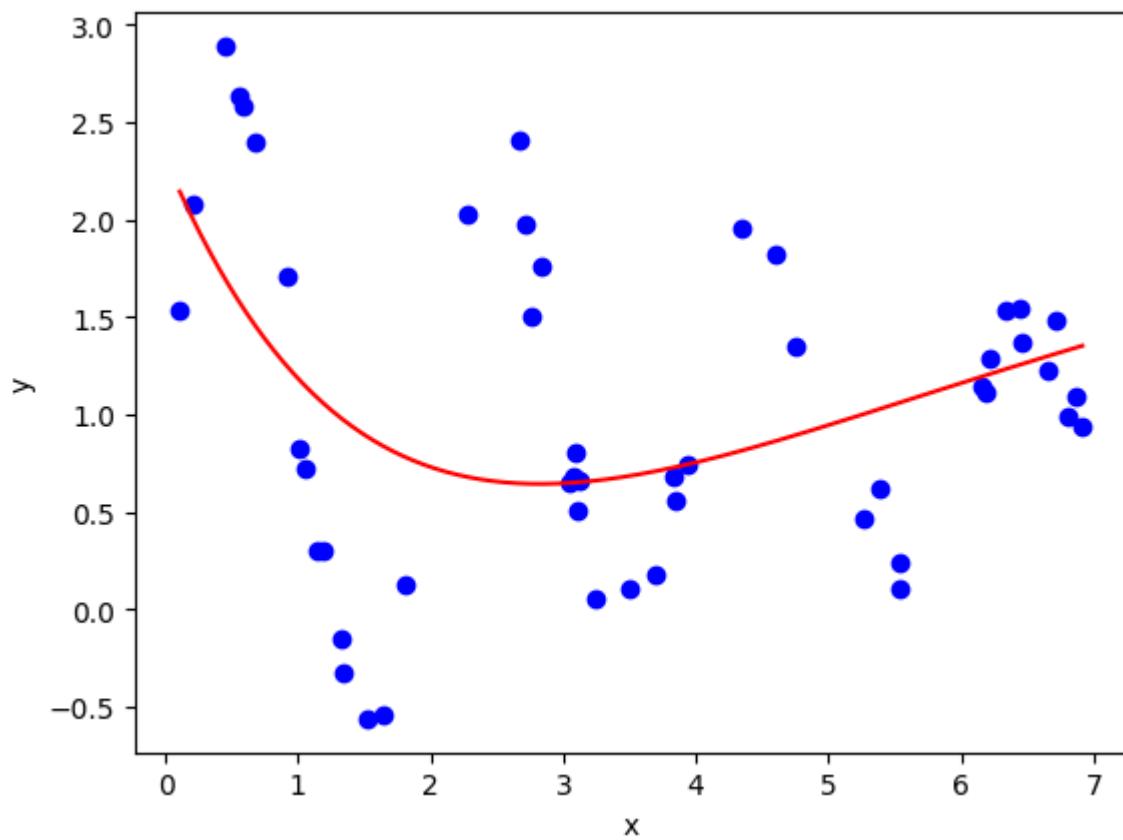


Here are a couple of examples of solutions that returns a supposedly successful solution, but have obviously not converged to the best fit.





Here is an example of a solution that has not succeeded:



If your model does not fit, try varying the initial guess for the fit parameters.

# Fitting Models to Data with `scipy.optimize.curve_fit`

The `curve_fit` function uses non-linear least squares minimization to fit a function to data (making use of the `least_squares` function). Though we demonstrated how to do this on the previous page using the `least_squares` function, the `curve_fit` function is far more convenient to use for this purpose.

The call signature of `curve_fit` (including only the arguments of immediate interest to us) is:

```
curve_fit(f, xdata, ydata, p0=None, sigma=None, ..., bounds=(- np.inf, np.inf))
```

where

- `f` is the function for the model being fitted to the data, i.e.  $y = f(x, \vec{a})$ . It has the call signature `f(xdata, *params)`, where `params` are the model parameters which need to be found for the fit (in our previous notation, the  $a_j$  values).
- `xdata` is the data for the independent variable.
- `ydata` is the data for the dependent variable.
- `p0` is the initial guess for the model parameters ( $a_j$ ).
- `sigma` is the uncertainty in the `ydata`. If this is not `None`, then `curve_fit` will use  $\chi^2$  minimization.
- `bounds` is a tuple of 2 arrays for the lower and upper bounds of the parameters (the same use as in `least_squares`).

The `curve_fit` function returns a tuple where the first element is an array of the values for the model parameters which best fit the data.

## Worked Example

Let's use `curve_fit` to fit the previous functional relation:

$$\begin{aligned}y &= a_0 + a_1 e^{a_2 x} \\&= f(x; \vec{a})\end{aligned}$$

to the **nonlinear\_data.csv** data.

Unlike when using `least_squares`, we don't have to define the residual function, only the functional relation:

```
def f(x, a):  
    return a[0] + a[1] * np.exp(a[2] * x)
```

```

import numpy as np
import matplotlib.pyplot as plt

# importing scipy.optimize.leastsq only
from scipy.optimize import curve_fit

#The model to fit to the data
def f(x, *a):
    return a[0] + a[1] * np.exp(a[2] * x)

#Reading the data
# The `unpack` keyword argument seperates the columns into individual arrays
xdata, ydata = np.loadtxt('data/nonlinear_data.csv', delimiter = ',', unpack = True)

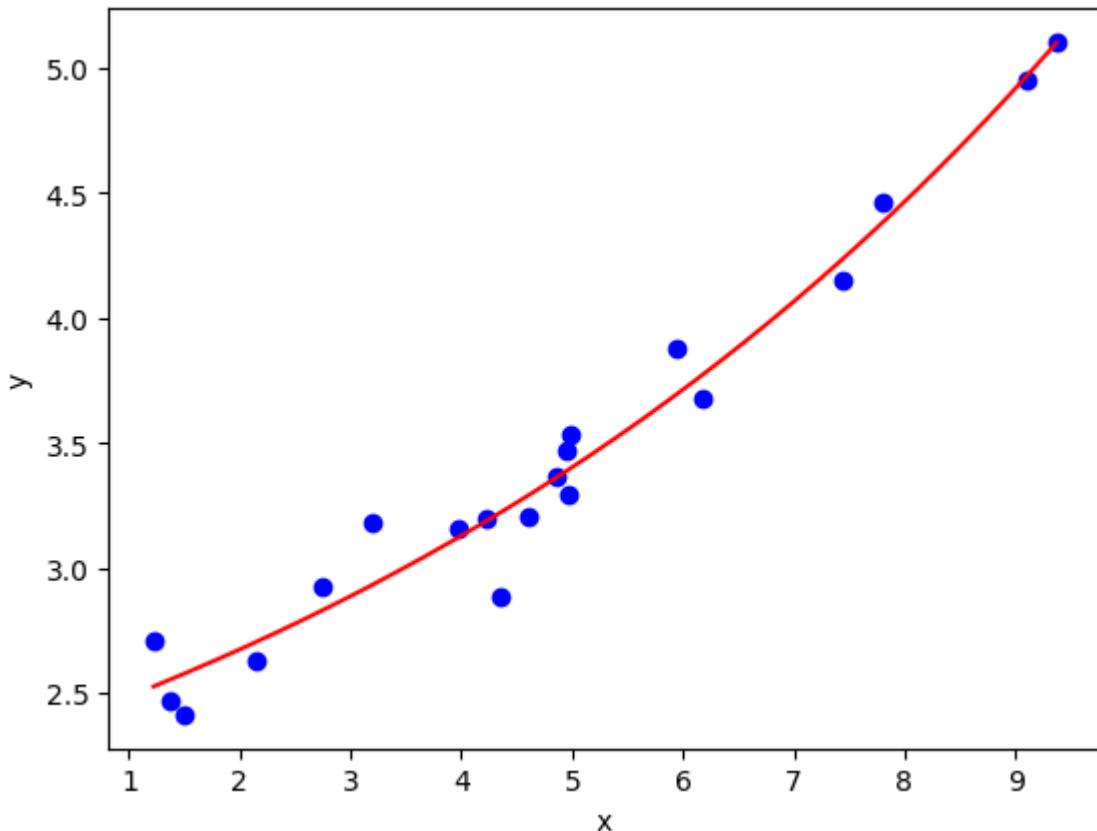
#Performing the fit
a0 = [1.5, 0.6, 0.2] #initial guess

fit = curve_fit(f, xdata, ydata, a0)

#Plotting the fit and data
x = np.linspace(xdata.min(), xdata.max(), 1000)

plt.plot(xdata, ydata, 'bo')
plt.plot(x, f(x, *fit[0]), 'r-')
plt.xlabel('x')
plt.ylabel('y')
plt.show()

```





# 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:

$$\frac{d^n y}{dx^n} = f\left(x, y, \frac{dy}{dx}, \frac{d^2y}{dx^2}, \dots, \frac{d^{n-1}y}{dx^{n-1}}\right)$$

with given initial conditions for  $y(x_0), \frac{dy}{dx}(x_0), \frac{d^2y}{dx^2}(x_0), \frac{d^3y}{dx^3}(x_0), \dots, \frac{d^{n-1}y}{dx^{n-1}}(x_0)$ .

These are called initial value problems. To solve them you need to set as many initial conditions as the order of the equation.

# 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  $x_3 = x_2 + h$ ,  $y_3$ :

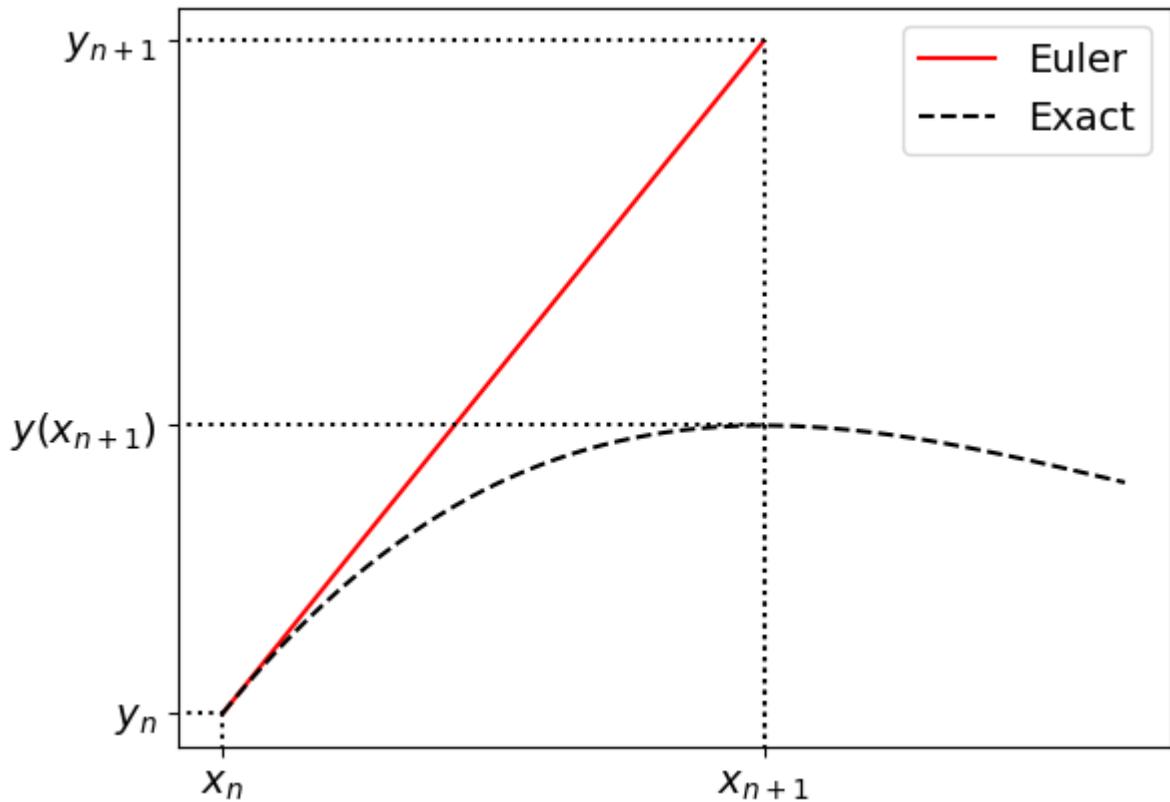
$$y_3 \approx y_2 + hf(x_2, y_2)$$

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:

$$\frac{dy}{dx} = -2(x - 1)y$$

with the given initial conditions:  $y = 1$  at  $x = 0$ .

This has an analytic solution of:

$$y(x) = e^{(x-1)^2+1}$$

which we can compare our numerical solutions to.

Let's say we want to find the value of  $y$  at  $x = 1$  numerically (it's  $e$ ). 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  $x$ :

```
import numpy as np

x, y = 0, 1 #initial conditions

h = 0.05 #step size

x_end = 1 #the value of x for which we want to know y

#The ODE function
def f(x,y):
    return - 2 * (x - 1) * 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 = 1 approximated using Euler: ', y)
print('y at x = 1: ', np.e)
```

```
y at x = 1 approximated using Euler:  2.7617285451021716
y at x = 1:  2.718281828459045
```

Now, it is often important for us to visualize the solution for  $y(x)$  over the interval, rather than only finding the value of  $y$  at the end of it. We will plot the numerical solution for  $y$  on the interval  $0 < x < 5$  along with the exact solution for comparison. 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:

```
import numpy as np
import matplotlib.pyplot as plt

x0, y0 = 0, 1
h = 0.05
x_end = 5

#The ODE function
def f(x,y):
    return -2 * (x - 1) * 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

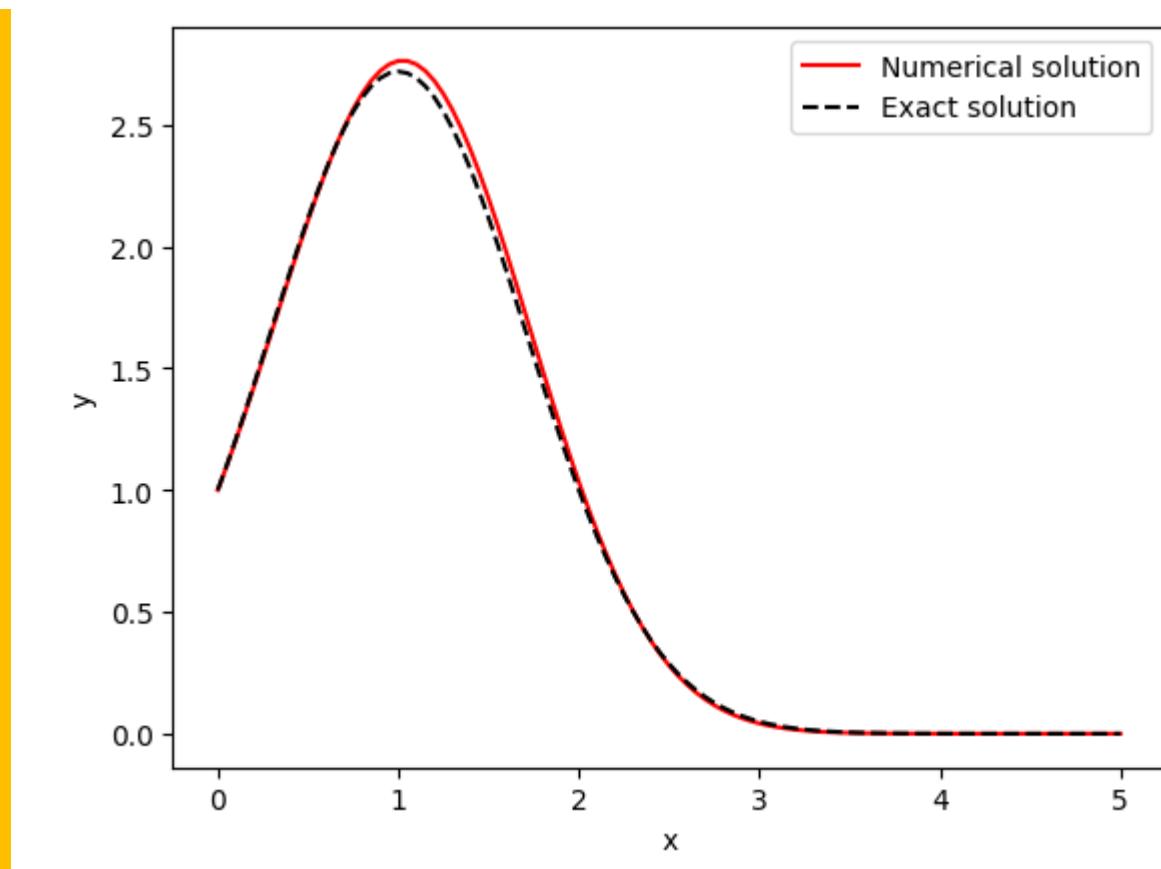
#Performing the Euler method, note we don't use the last x value in the update calculation
for i,x in enumerate(x_arr[:-1]):
    y_arr[i+1] = y_arr[i] + h*f(x, y_arr[i])

#Plotting the solution
fig, ax = plt.subplots()

ax.plot(x_arr, y_arr, color = 'red', label = 'Numerical solution')
ax.plot(x_arr, np.exp( -(x_arr - 1)**2 + 1 ), 'k--', label = 'Exact solution')
ax.set_xlabel('x')
ax.set_ylabel('y')

ax.legend()

plt.show()
```



# Euler's Method: Truncation Error

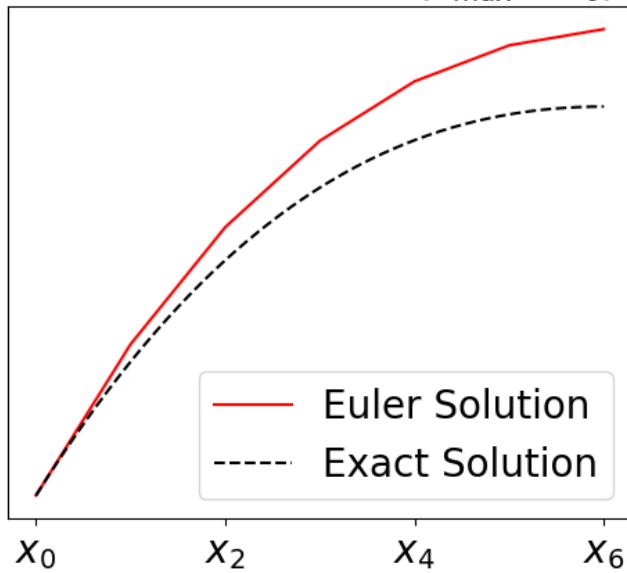
Like all numerical methods, Euler's method has systemic error. This is introduced when we discard the higher order terms in the Taylor expansion. The **local** truncation error is thus:

$$E_{n+1} = \frac{1}{2}y''(x_n)h^2 + O(h^3)$$

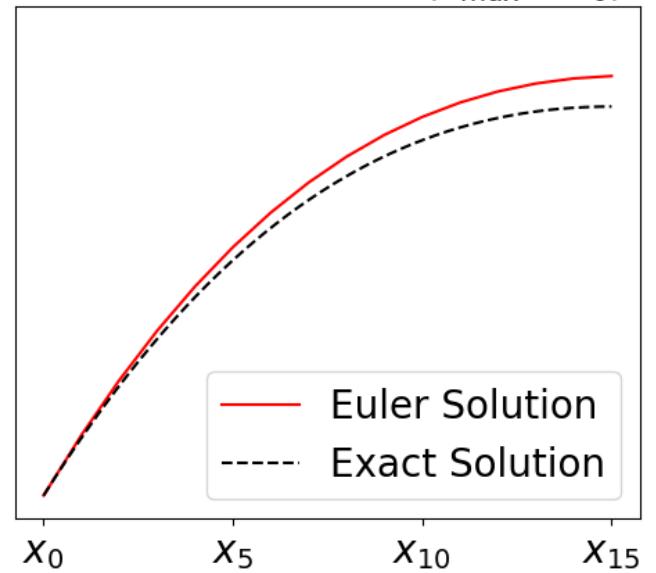
If you are unfamiliar with the notation for  $O(h^3)$  (big O notation), in this case it means the dominant terms are proportional to  $h^3$  (higher order terms of  $h$  will be less dominant for  $0 < h < 1$ ).

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  $O(h)$  [EulerErr1]. 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:

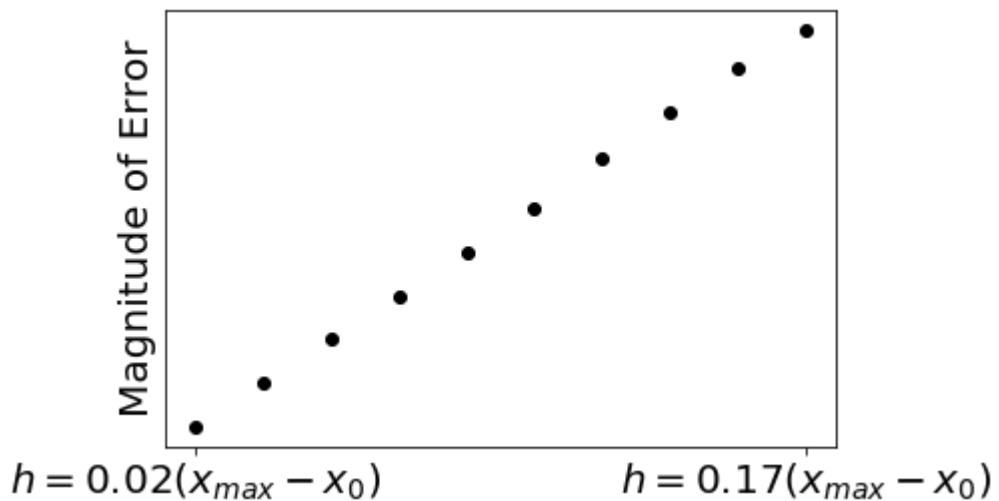
Solution for  $h = 0.17(x_{max} - x_0)$



Solution for  $h = 0.07(x_{max} - x_0)$



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

- [EulerErr1] James F. Epperson. *An Introduction to Numerical Methods and Analysis*. John Wiley & Sons, Inc., Hoboken, New Jersey, second edition edition, 2013.

# Solving Coupled and Higher Order ODEs

A common way to solve higher order ODEs using numerical methods is to convert this to a system of first order ODEs. We will first look at solving systems of coupled first order differential equations, and then we will focus on reducing higher order ODEs to solve them in a similar manner.

## Coupled Ordinary Differential Equations

Consider a system of first order coupled ODEs in the form:

$$\begin{aligned}\frac{dx}{dt}(t) &= f(t, x(t), y(t)) \\ \frac{dy}{dt}(t) &= g(t, x(t), y(t))\end{aligned}$$

given the initial conditions:

$$\begin{aligned}x(t_0) &= x_0 \\ y(t_0) &= y_0\end{aligned}$$

where there is one independent variable  $t$ , and two dependent variables  $x(t)$  and  $y(t)$ .

Note that we cannot simply solve the ODEs for  $x$  and  $y$  independently, as the ODE functions contain both of these variables. Instead, for our numerical solution, we must solve them simultaneously, step-by-step. Applying Euler's method, defining  $t_{n+1} = t_n + h$ ,  $x_n = x(t_n)$  and  $y_n = y(t_n)$ , the update step is:

$$\begin{aligned}x_{n+1} &= x_n + f(t_n, x_n, y_n) \\ y_{n+1} &= y_n + g(t_n, x_n, y_n)\end{aligned}$$

As you can see, in order to calculate  $x_{n+1}$ , you need to know both the values of  $x_n$  and  $y_n$  (the same goes for  $y_{n+1}$ ).

## Worked Example

Consider the coupled system of first order ODEs:

$$\begin{aligned}\frac{dx}{dt} &= t + xy \\ \frac{dy}{dt} &= t - x\end{aligned}$$

with the initial conditions

$$\begin{aligned}x(0) &= 0 \\ y(0) &= 1\end{aligned}$$

Let's write a script to integrate these ODEs using Euler's method to find  $x(t)$  and  $y(t)$  up to  $t = 10$ . We'll store the values in an array and plot them at the end.

```

import numpy as np
import matplotlib.pyplot as plt

t0, x0, y0 = 0, 0, 1 #initial conditions
h = 0.05 #step size
t_end = 10 #last time step

#The ODE functions
def f(t, x, y):
    return t + x * y

def g(t, x, y):
    return t - x

#Constructing the arrays:
t_arr = np.arange(t0, t_end + h, h) #make sure it goes up to and including t_end

x_arr = np.zeros(t_arr.size)
y_arr = np.zeros(t_arr.size)

#Setting the initial conditions
x_arr[0] = x0
y_arr[0] = y0

#Performing the Euler method, note we don't use the last t values in the update calculation
for i,t in enumerate(t_arr[:-1]):
    x_arr[i + 1] = x_arr[i] + h * f(t, x_arr[i], y_arr[i])
    y_arr[i + 1] = y_arr[i] + h * g(t, x_arr[i], y_arr[i])

##Plotting both curves
fig, ax = plt.subplots(2,1, sharex = True, figsize = (6.4, 5))

ax[0].plot(t_arr, x_arr)

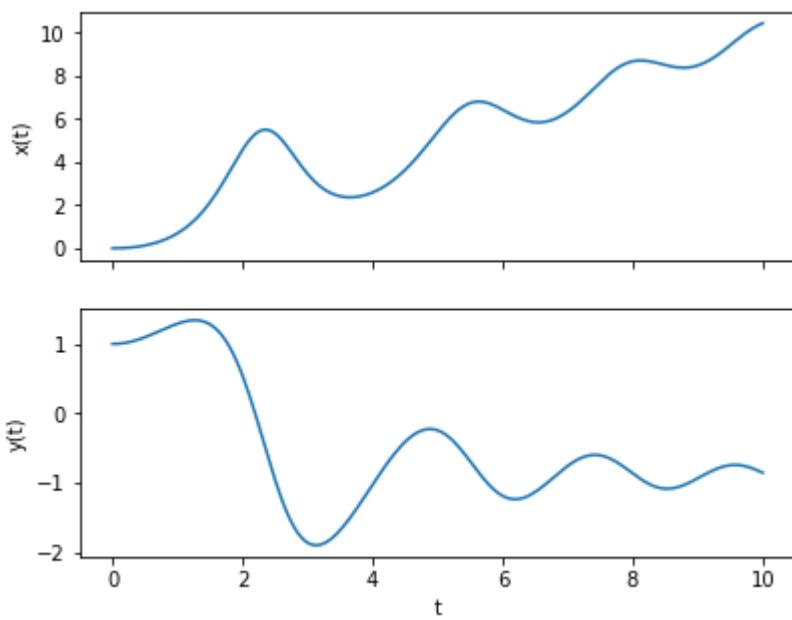
ax[0].set_ylabel('x(t)')

ax[1].plot(t_arr, y_arr)

ax[1].set_xlabel('t')
ax[1].set_ylabel('y(t)')

plt.show()

```



## Arbitrarily Many Coupled Ordinary Differential Equations

Now, let's consider a general solution for arbitrarily many coupled ODEs. Consider the system of coupled first order differential equations:

$$\begin{aligned} \frac{dx_1}{dt} &= f_1(t, x_1, x_2, \dots, x_m) \\ \frac{dx_2}{dt} &= f_2(t, x_1, x_2, \dots, x_m) \\ \frac{dx_3}{dt} &= f_3(t, x_1, x_2, \dots, x_m) \\ &\vdots \\ \frac{dx_m}{dt} &= f_m(t, x_1, x_2, \dots, x_m) \end{aligned}$$

with initial conditions:

$$x_1(t_0) = x_{10}$$

$$x_2(t_0) = x_{20}$$

$$x_3(t_0) = x_{30}$$

$$\vdots$$

$$x_m(t_0) = x_{m0}$$

We can boil down the update step to a single line of code by vectorizing the equations and conditions using NumPy arrays. Let's define the following vector and vector function:

$$\vec{x}(t) = \begin{pmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ \vdots \\ x_m(t) \end{pmatrix} \quad \text{and} \quad \vec{f}(\vec{x}) = \begin{pmatrix} f_1(t, \vec{x}) \\ f_2(t, \vec{x}) \\ f_3(t, \vec{x}) \\ \vdots \\ f_m(t, \vec{x}) \end{pmatrix}$$

The system of ODEs can now be written as:

$$\frac{d\vec{x}}{dt} = \vec{f}(t, \vec{x})$$

Using our usual definitions of  $t_{n+1} = t_n + h$  and  $\vec{x}(t_n) = \vec{x}_n$ , the Euler update step can be written as:

$$\vec{x}_{n+1} = \vec{x}_n + h\vec{f}(t_n, \vec{x}_n)$$

## Worked Example

Consider the system of 3 first order coupled ODEs:

$$\begin{aligned} \frac{dx}{dt} &= t + xy \\ \frac{dy}{dt} &= t - x \\ \frac{dz}{dt} &= y \end{aligned}$$

with the initial conditions

$$\begin{aligned} x(0) &= 0 \\ y(0) &= 1 \\ z(0) &= 0 \end{aligned}$$

Let's adapt the previous script to integrate these coupled ODEs using Euler's method, this time making use of NumPy array's vectorized operations.

```

import numpy as np
import matplotlib.pyplot as plt

t0 = 0
x0 = np.array([0, 1, 0]) #initial conditions
h = 0.05 #step size
t_end = 10 #last time step

#The ODE function
def f(t, x):
    return np.array([
        t + x[0] * x[1],
        t - x[0],
        x[1]
    ])

#Constructing the arrays:
t_arr = np.arange(t0, t_end + h, h) #make sure it goes up to and including t_end
x_arr = np.zeros((t_arr.size, x0.size)) #a 2D array, first dimension for t steps, second for x dimensions

#Setting the initial conditions
x_arr[0, :] = x0

#Performing the Euler method
for i,t in enumerate(t_arr[:-1]):
    x_arr[i + 1, :] = x_arr[i, :] + h * f(t, x_arr[i, :])

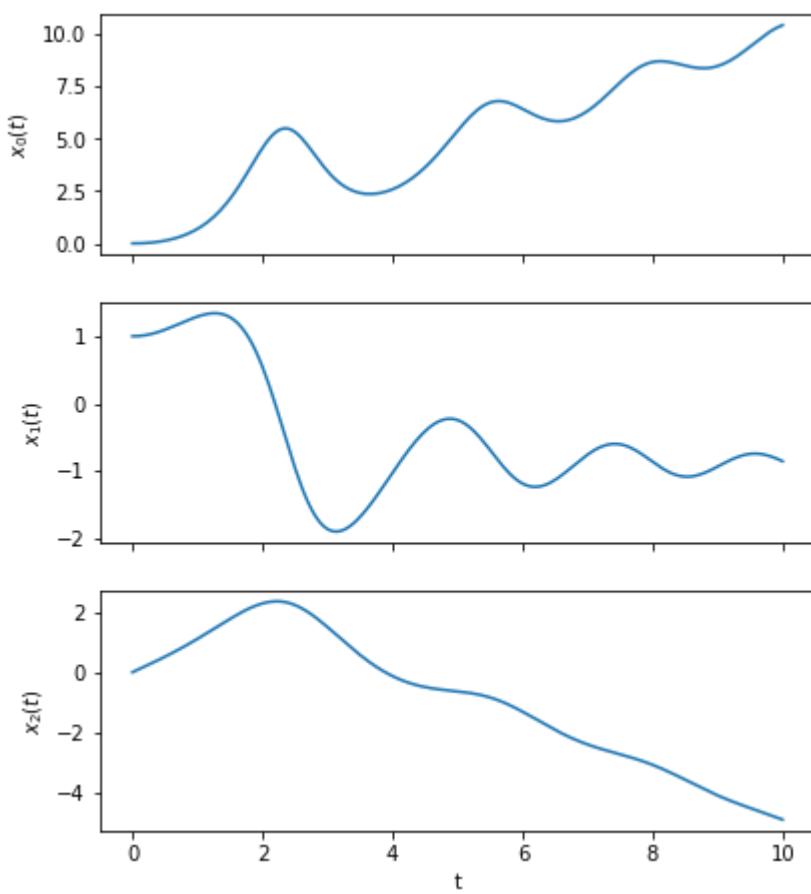
##Plotting all of the curves
fig, ax = plt.subplots(x0.size, 1, sharex = True, figsize = (6.4, 2.5 * x0.size))

for i in range(x0.size):
    ax[i].plot(t_arr, x_arr[:, i])
    ax[i].set_ylabel(f'$x_{i}(t)$')

ax[-1].set_xlabel('t')

plt.show()

```



## 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:

$$\begin{aligned}\frac{dy}{dx} &= v \\ \frac{dv}{dx} &= f(x, y, v)\end{aligned}$$

with the initial conditions  $y(x_0) = y_0$  and  $v(x_0) = y'_0$ .

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:

$$\begin{aligned}y_0 &= y \\ y_1 &= \frac{dy}{dt} = \frac{dy_0}{dt}\end{aligned}$$

which form the vector:

$$\vec{y} = \begin{pmatrix} y_0 \\ y_1 \end{pmatrix}$$

in order to reduce the second order ODE to a coupled system of two first order ODEs:

$$\begin{aligned}\frac{dy_0}{dt} &= y_1 \\ \frac{dy_1}{dt} &= 100|\sin(t)| - 10y_1 - 100y_0\end{aligned}$$

which can be vectorized as:

$$\frac{d\vec{y}}{dt} = \begin{pmatrix} y_1 \\ 100|\sin(t)| - 10y_1 - 100y_0 \end{pmatrix}$$

This can be solved by modifying our solutions from the previous worked example:

```

import numpy as np
import matplotlib.pyplot as plt

t0 = 0
y0 = np.array([0.1, -0.5]) #initial conditions
h = 0.05 #step size
t_end = 10

#The ODE function
def f(t, y):
    return np.array([
        y[1],
        100*np.abs(np.sin(t)) - 10 * y[1] - 100 * y[0]
    ])

#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.size, y0.size))

#Setting the initial conditions
y_arr[0, :] = y0

#Performing the Euler method, note we don't use the last x value in the update calculation
for i,t in enumerate(t_arr[:-1]):
    y_arr[i + 1, :] = y_arr[i, :] + h * f(t, y_arr[i, :])

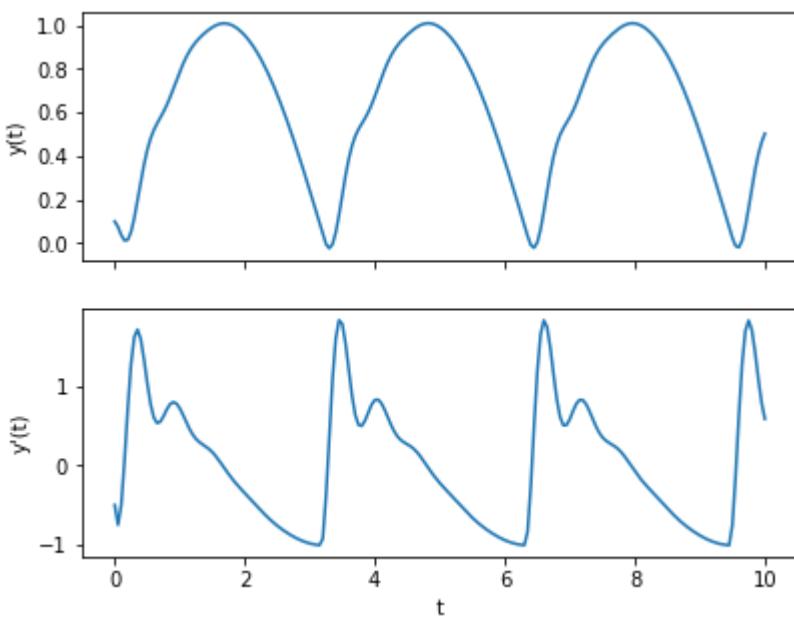
##Plotting both curves
fig, ax = plt.subplots(2,1, sharex = True, figsize = (6.4, 5))

ax[0].plot(t_arr, y_arr[:, 0])
ax[0].set_ylabel('y(t)')

ax[1].plot(t_arr, y_arr[:, 1])
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:

$$\frac{d^m y}{dx^m} = f\left(x, \frac{dy}{dx}, \frac{d^2y}{dx^2}, \frac{d^3y}{dx^3}, \dots, \frac{d^{m-1}y}{dx^{m-1}}\right)$$

with initial conditions

$$y(x = x_0) = y_0 \quad \frac{dy}{dx}(x = x_0) = y'_0 \quad \frac{d^2y}{dx^2}(x = x_0) = y''_0 \quad \dots \quad \frac{d^{m-1}y}{dx^{m-1}}(x = x_0) = y_0^{(m)}$$

We start by introducing the variables:

$$y_0 = y \quad y_1 = \frac{dy}{dx} \quad y_2 = \frac{d^2y}{dx^2} \quad \dots \quad y_{m-1} = \frac{d^{m-1}y}{dx^{m-1}}$$

we can transform the order  $m$  ODE to a set of  $m$  first order coupled differential equations:

$$\begin{aligned}
\frac{dy_0}{dx} &= y_1 \\
\frac{dy_1}{dx} &= y_2 \\
\frac{dy_2}{dx} &= y_3 \\
&\vdots \\
\frac{dy_{m-2}}{dx} &= y_{m-1} \\
\frac{dy_{m-1}}{dx} &= f(x, y_0, y_1, y_2, y_3, \dots, y_{m-2}, y_{m-1})
\end{aligned}$$

Again, we can vectorize this in order to use a solution similar to those above:

$$\vec{y} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{m-1} \end{pmatrix}$$

# 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 (RK) 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 [RK1] is not a full derivation of the method, as this requires theorems outside the scope of this course.

## Second Order Runge-Kutta (RK2)

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  $\alpha$ .

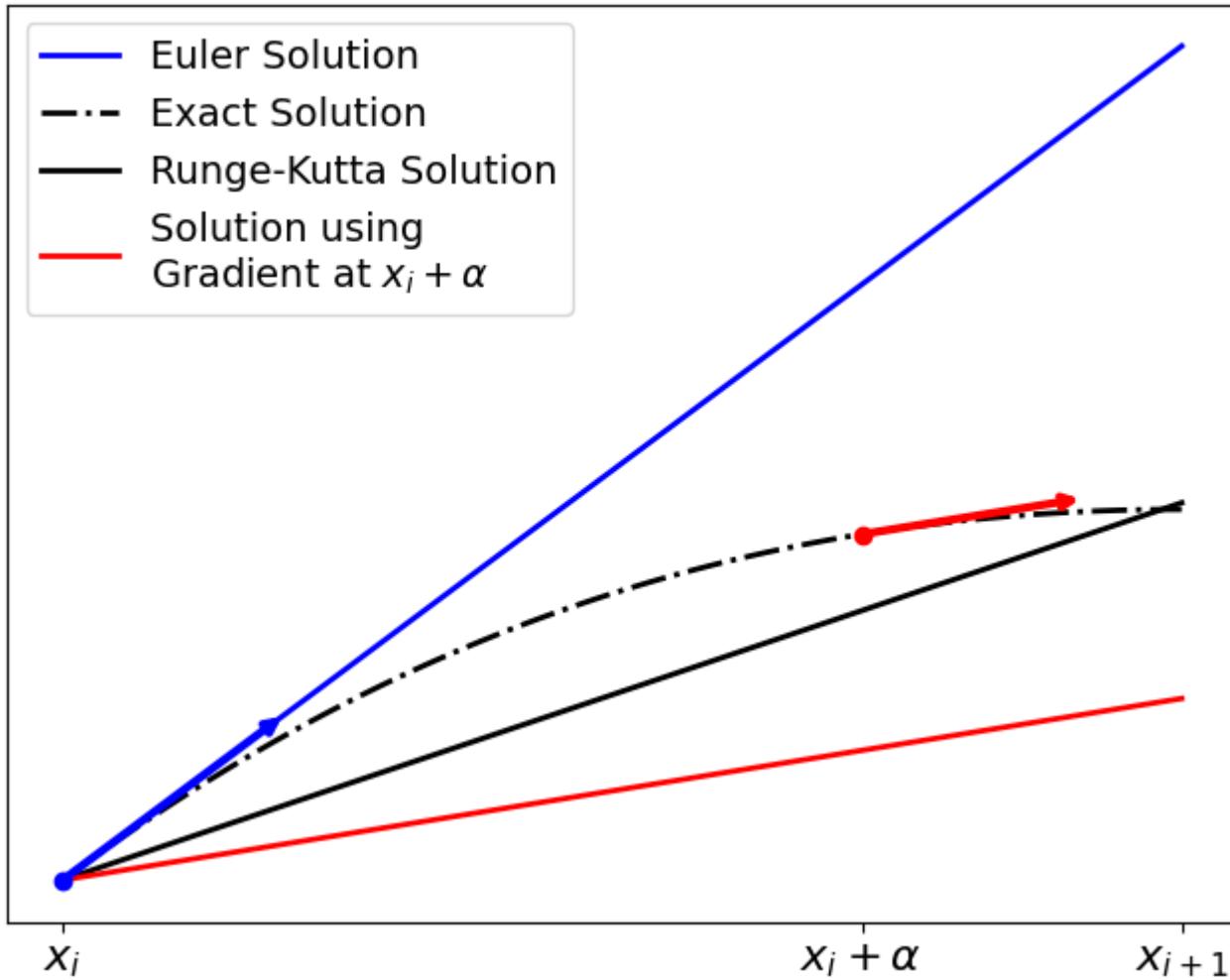
We then approximate the value of  $y_{i+1}$  using Euler's method with each of these gradients:

- $y_{i+1} \approx y_i + h f(x_i, y_i)$
- $y_{i+1} \approx y_i + h f(x_i + \alpha h, y_i + \alpha h f(x_i, y_i))$

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  $\alpha$ ? 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:**  $c_1 = c_2 = \frac{1}{2}$  and  $\alpha = 1$ , which yields:

$$y_{i+1} = y_i + \frac{1}{2}h [f(x_i, y_i) + f(x_i + h, y_i + hf(x_i, y_i))]$$

**The midpoint rule:**  $c1 = 0$ ,  $c2 = 1$  and  $\alpha = \frac{1}{2}$ , which yields:

$$y_{i+1} = y_i + hf\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hf(x_i, y_i)\right)$$

Both of these methods have an accumulated/global truncated error of  $O(h^2)$ , as opposed to the Euler method's  $O(h)$  [RK1] (remember that an  $O(h)$  trend dominates  $O(h^2)$  for  $0 < h < 1$ , i.e.  $O(h)$  will have higher error in general).

## 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 our results to.

Let's solve this ODE up to  $x = 1$  for both the trapezoidal and the midpoint rule RK2 methods.

```

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_trapz = np.zeros(x_arr.shape) #trapezoidal rule solution
y_trapz[0] = y0

y_mid = np.zeros(x_arr.shape) #midpoint rule solution
y_mid[0] = y0

#Runge-Kutta method
for i,x in enumerate(x_arr[:-1]):
    #Trapezoidal update step
    f_trapz = f(x, y_trapz[i])
    y_trapz[i + 1] = y_trapz[i] + 0.5 * h * ( f_trapz + f(x + h, y_trapz[i] + h * f_tr

    #Midpoint update step
    y_mid[i + 1] = y_mid[i] + h * f(x + 0.5 * h, y_mid[i] + 0.5 * h * f(x, y_mid[i]))

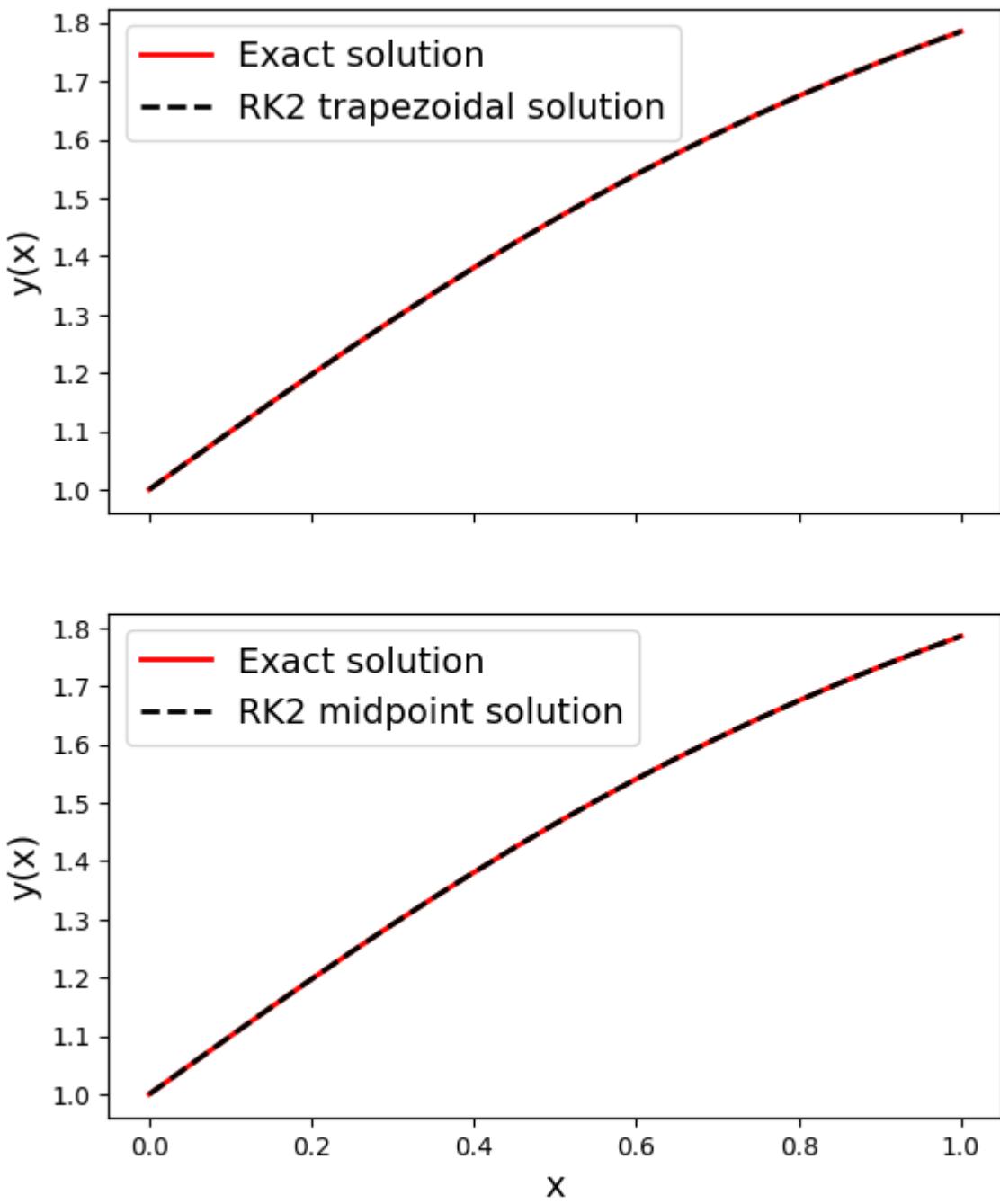
#Plotting the solutions
fig, ax = plt.subplots(2, 1, sharex = True, figsize = (6.4, 8))

##Plotting trapezoidal
ax[0].plot(x_arr, y_exact(x_arr), color = 'red', label = 'Exact solution', linewidth =
ax[0].plot(x_arr, y_trapz, '--k', label = 'RK2 trapezoidal solution', linewidth = 2)
ax[0].set_ylabel('y(x)', fontsize = 14)
ax[0].legend(fontsize = 14)

##Plotting midpoint
ax[1].plot(x_arr, y_exact(x_arr), color = 'red', label = 'Exact solution', linewidth =
ax[1].plot(x_arr, y_mid, '--k', label = 'RK2 midpoint solution', linewidth = 2)
ax[1].set_ylabel('y(x)', fontsize = 14)
ax[1].set_xlabel('x', fontsize = 14)
ax[1].legend(fontsize = 14)

plt.show()

```



## 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:

$$\begin{aligned}k_1 &= f(x_i, y_i) \\k_2 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}h k_1\right) \\k_3 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}h k_2\right) \\k_4 &= f(x_i + h, y_i + h k_3)\end{aligned}$$

$k_1$  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  $O(h^4)$

## Worked Example

Again, consider the ordinary differential equation:

$$\frac{dy}{dx} = \frac{1}{1+x^2}$$

with initial conditions  $y = 1$  at  $x = 0$ , and the exact solution:

$$y = 1 + \arctan(x)$$

which we can compare our results to. Let's solve this ODE up to  $x = 1$  using the RK4 method.

```

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)

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] + h*k3)

    #update
    y_arr[i+1] = y_arr[i] + h * (k1 + 2*k2 + 2*k3 + k4) / 6.0

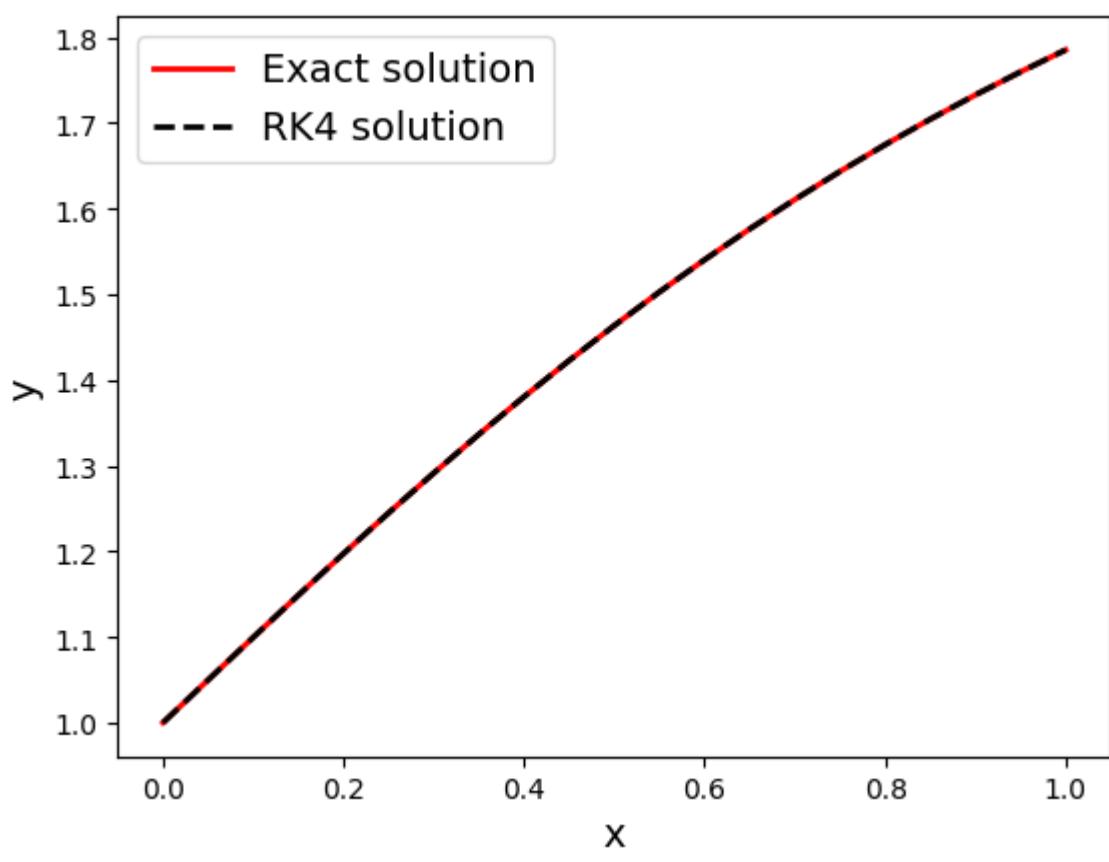
#Plotting the solution
fig, ax = plt.subplots()

ax.plot(x_arr, y_exact(x_arr), color = 'red', 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()

```



## Coupled and Higher 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:

$$\begin{aligned}
 \frac{dy_1}{dx} &= f_1(x, y_1, y_2, \dots, y_m) \\
 \frac{dy_2}{dx} &= f_2(x, y_1, y_2, \dots, y_m) \\
 &\vdots \\
 \frac{dy_m}{dx} &= f_m(x, y_1, y_2, \dots, y_m)
 \end{aligned} \tag{1}$$

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.

For the step from  $x_n$  to  $x_{n+1}$ :

$$y_{j,n+1} = y_{j,n} + \frac{h}{6}(k_{1,j} + 2k_{2,j} + 2k_{3,j} + k_{4,j})$$

Note that the  $y_j$  variables are not explicitly dependent on each other, but on the independent 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 simultaneous 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  $y_l$ .

$$\begin{aligned} k_{1,j} &= f_j(x_n, y_{1,n}, \dots, y_{j,n}, \dots, y_{n-1,n}) \\ k_{2,j} &= f_j(x_n + \frac{1}{2}h, y_{1,n} + \frac{1}{2}k_{1,0}, \dots, y_{j,n} + \frac{1}{2}k_{1,j}, \dots, y_{m,n} + \frac{1}{2}k_{1,m}) \\ k_{3,j} &= f_j(x_n + \frac{1}{2}h, y_{1,n} + \frac{1}{2}k_{2,0}, \dots, y_{j,n} + \frac{1}{2}k_{2,j}, \dots, y_{m,n} + \frac{1}{2}k_{2,m}) \\ k_{4,j} &= f_j(x_n + h, y_{1,n} + k_{3,1}, \dots, y_{j,n} + k_{3,j}, \dots, y_{m,n} + k_{3,m}) \end{aligned}$$

This looks more complicated than it is to apply in practice. All we need to do is vectorize the solution, as in the previous section. We can represent all the  $y_j$  as a vector:

$$\vec{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

the ODE can thus be represented as:

$$\frac{d\vec{y}}{dx} = \vec{f}(x, \vec{y}) = \begin{pmatrix} f_1(x, \vec{y}) \\ f_2(x, \vec{y}) \\ \vdots \\ f_m(x, \vec{y}) \end{pmatrix}$$

and an update step as:

$$\vec{y}_{n+1} = \vec{y}_n + \frac{1}{6}h(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4)$$

where:

$$\vec{k}_i = \begin{pmatrix} k_{i,1} \\ k_{i,2} \\ \vdots \\ k_{i,m} \end{pmatrix}$$

Note that we can write:

$$\begin{pmatrix} y_{1,n} + \frac{1}{2}hk_{1,1} \\ \vdots \\ y_{j,n} + \frac{1}{2}hk_{1,j} \\ \vdots \\ y_{m,n} + \frac{1}{2}hk_{1,m} \end{pmatrix} = \vec{y}_n + \frac{1}{2}h\vec{k}_1$$

with this in mind, we can simply write the  $k$  values as:

$$\begin{aligned} \vec{k}_1 &= \vec{f}(x_n, \vec{y}_n) \\ \vec{k}_2 &= \vec{f}\left(x_n + \frac{1}{2}h, \vec{y}_n + \frac{1}{2}h\vec{k}_1\right) \\ \vec{k}_3 &= \vec{f}\left(x_n + \frac{1}{2}h, \vec{y}_n + \frac{1}{2}h\vec{k}_2\right) \\ \vec{k}_4 &= \vec{f}\left(x_n + h, \vec{y}_n + h\vec{k}_3\right) \end{aligned}$$

## 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  $y_2(x) = y''(x)$ :

$$\begin{aligned}\frac{dy_1}{dx} &= y_2 \\ \frac{dy_2}{dx} &= y_3 \\ \frac{dy_3}{dx} &= -12xy_1 - 4x^2y_2\end{aligned}$$

```

import numpy as np
import matplotlib.pyplot as plt

x0 = 0 #initial conditions
y0 = np.array([0, 0, 2]) #initial conditions
h = 0.1
x_end = 5

def f(x, y):
    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)

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 * (k1 + 2*k2 + 2*k3 + k4) / 6.0

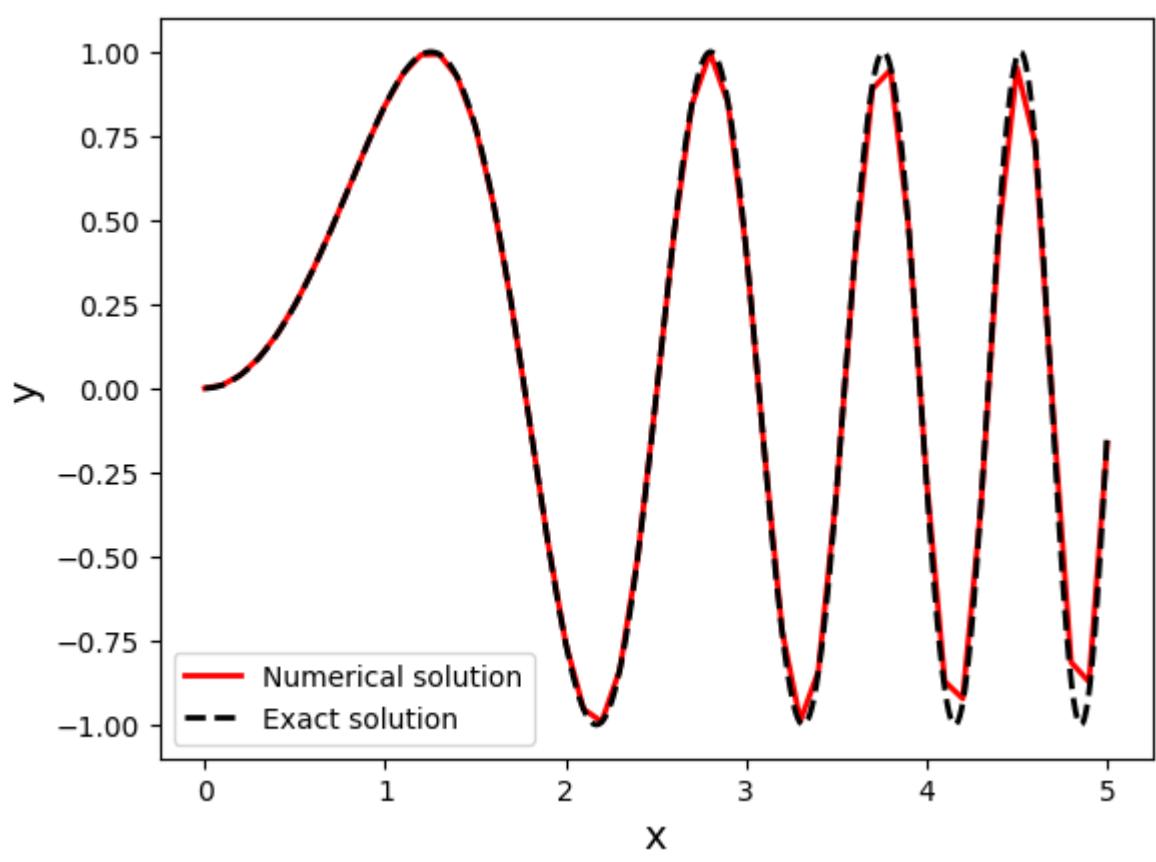
#Plotting the solution
fig, ax = plt.subplots()

ax.plot(x_arr, y_arr[:, 0], color = 'red', linewidth = 2, label = 'Numerical solution')
x_exact = np.linspace(x_arr[0], x_arr[-1], 1000) #x_arr is too sparse to properly plot
ax.plot(x_exact, y_exact(x_exact), 'k--', linewidth = 2, label = 'Exact solution')

ax.set_xlabel('x', fontsize = 14)
ax.set_ylabel('y', fontsize = 14)

ax.legend()
plt.show()

```



## References

[RK1](1,2) James F. Epperson. *An Introduction to Numerical Methods and Analysis*. John Wiley & Sons, Inc., Hoboken, New Jersey, second edition edition, 2013.

# Numerical Integration Techniques

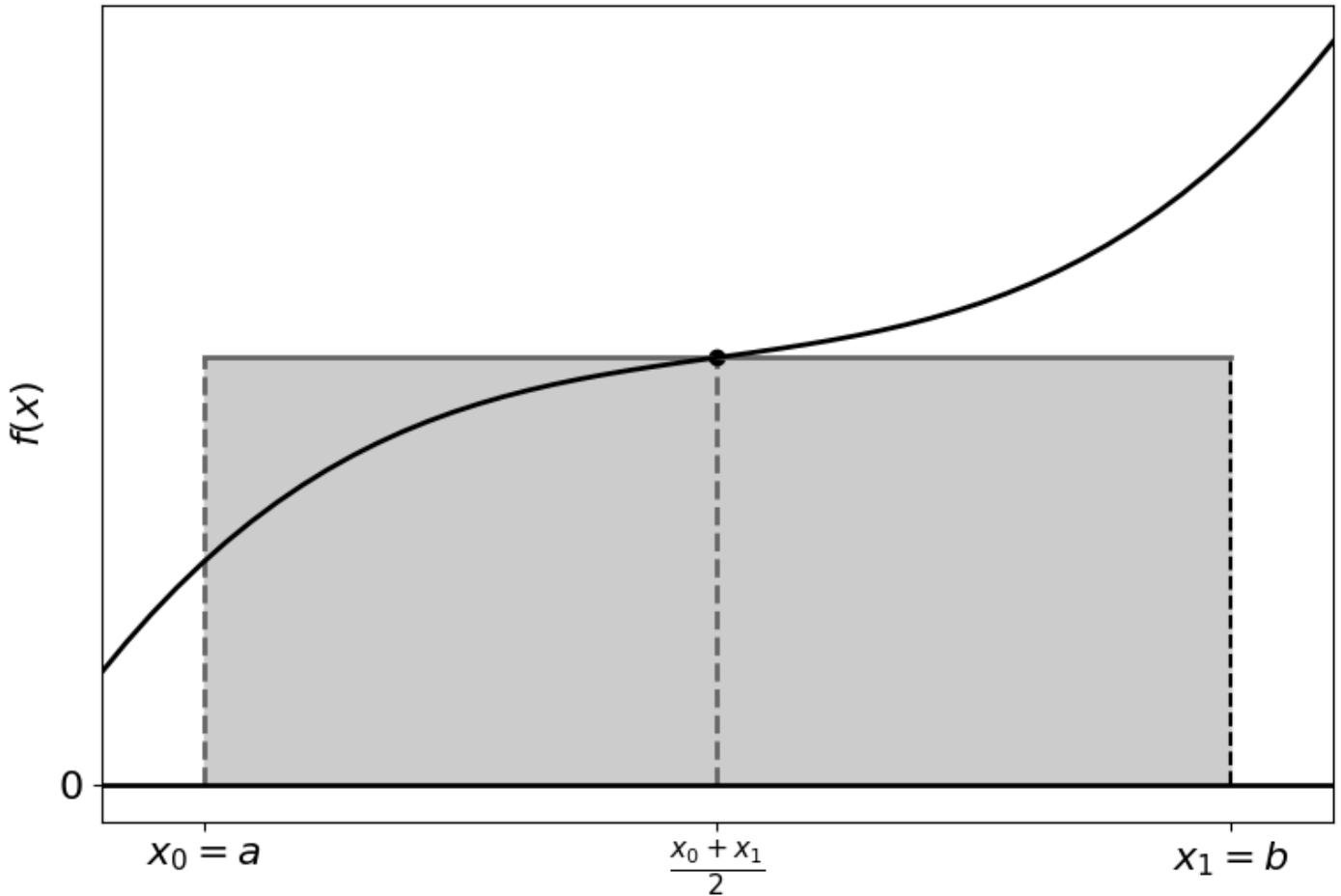
In this chapter we shall discuss three numerical methods that can be used to solve one dimensional integrals of the form

$$\int_a^b f(x) dx$$

These methods are the midpoint, trapezoidal and Simpson's rule.

# Midpoint Rule

In the midpoint rule you approximate the area under the curve as a rectangle with the height as the function value at the midpoint of the interval:

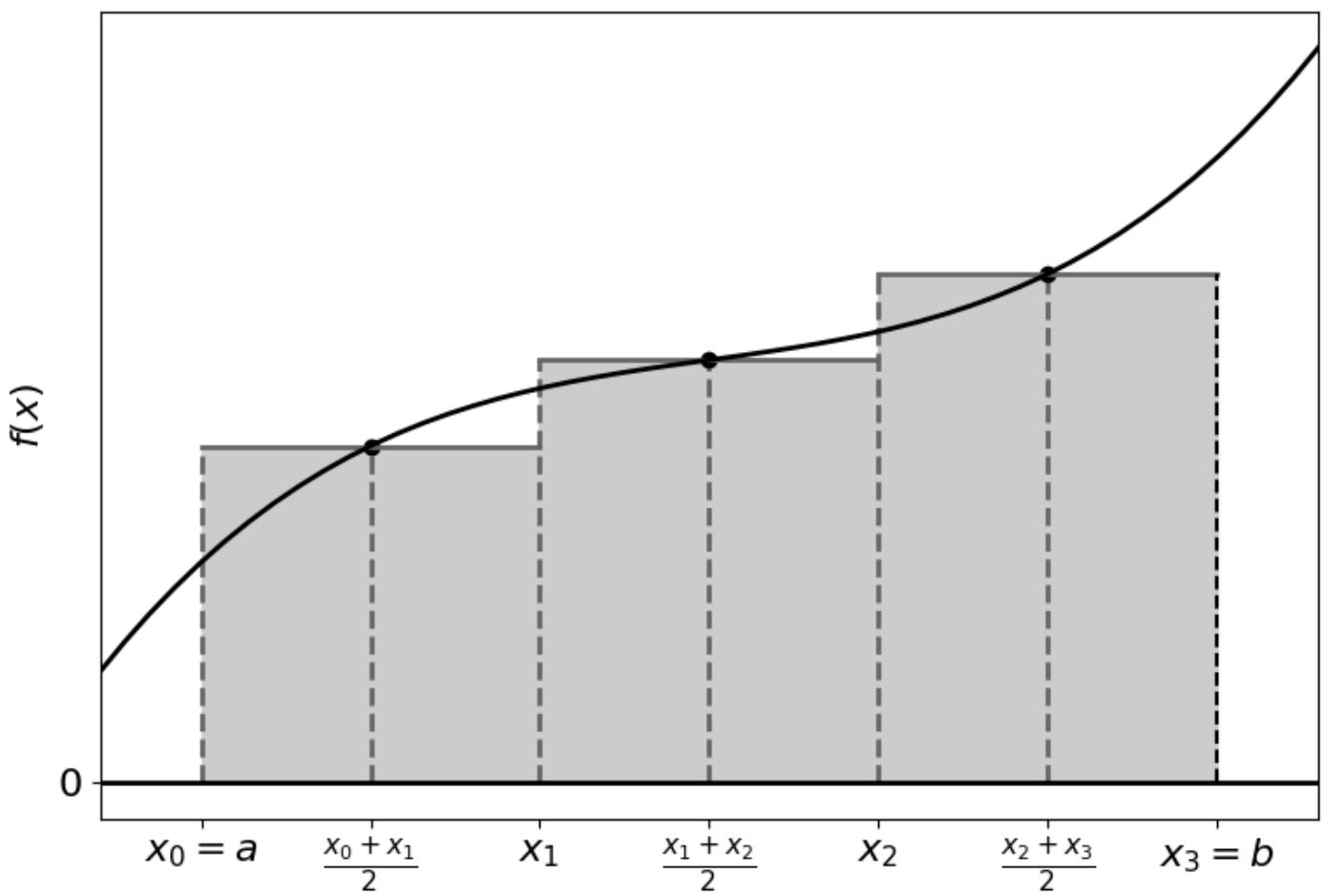
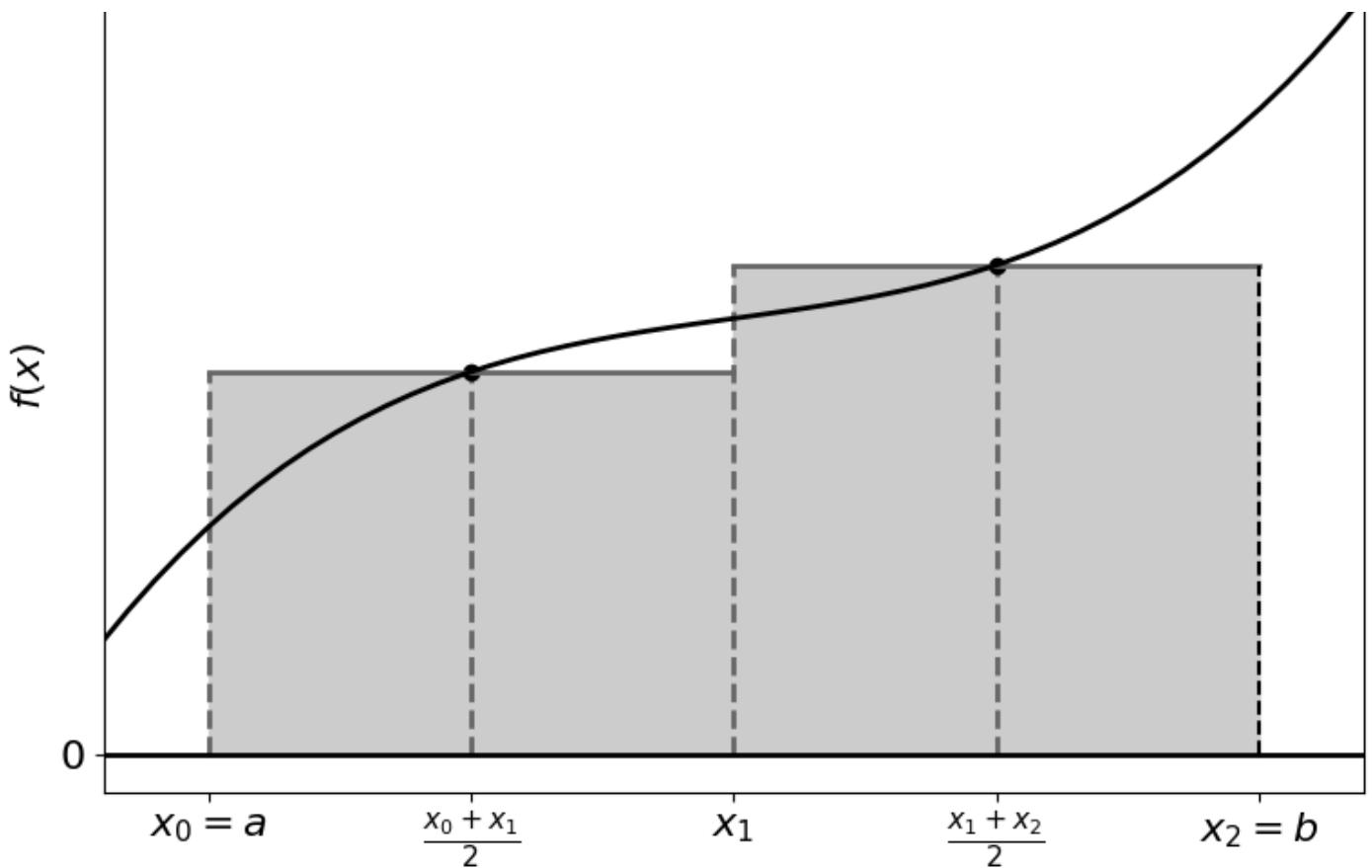


$$\int_a^b f(x) dx \approx f\left(\frac{a+b}{2}\right)(b-a)$$

# Composite Midpoint Rule

For a more accurate solution we can subdivide the interval further, constructing rectangles for each subinterval, with the function value of the midpoint used as the height:





For  $n$  subdivisions:

$$\int_a^b f(x) dx \approx \sum_{i=1}^n (x_i - x_{i-1}) f\left(\frac{x_i + x_{i-1}}{2}\right)$$

If these divisions are equal, then

$$x_i - x_{i-1} = \frac{b-a}{n}$$

which makes the approximation:

$$\int_a^b f(x) dx \approx \frac{b-a}{n} \sum_{i=1}^n f\left(\frac{x_i + x_{i-1}}{2}\right)$$

Assuming that  $n$  is chosen so that  $0 < \frac{b-a}{n} < 1$ , the error for this method is  $O\left(\frac{1}{n^3}\right)$  [IntMid1].

## Composite Midpoint Rule with a Discrete Data Set

Let's consider the case where we have a discrete set of data points  $(x_i, y_i)$  for  $i = 0, \dots, n$ , where:

$$y_i = f(x_i)$$

We want to approximate the integral of  $f(x)$  using this data and the midpoint rule. We can treat each  $x_i$  as the midpoint (except for  $x_0$  and  $x_n$  at the boundaries) and determine the size of the interval around it using the adjacent values.

For equally spaced data points, where  $\Delta x = x_i - x_{i-1}$  is constant, we can approximate the integral as:

$$\int_{x_0}^{x_n} f(x) dx \approx \Delta x \left[ \frac{1}{2}y_0 + \left\{ \sum_{i=1}^{n-1} y_i \right\} + \frac{1}{2}y_n \right]$$

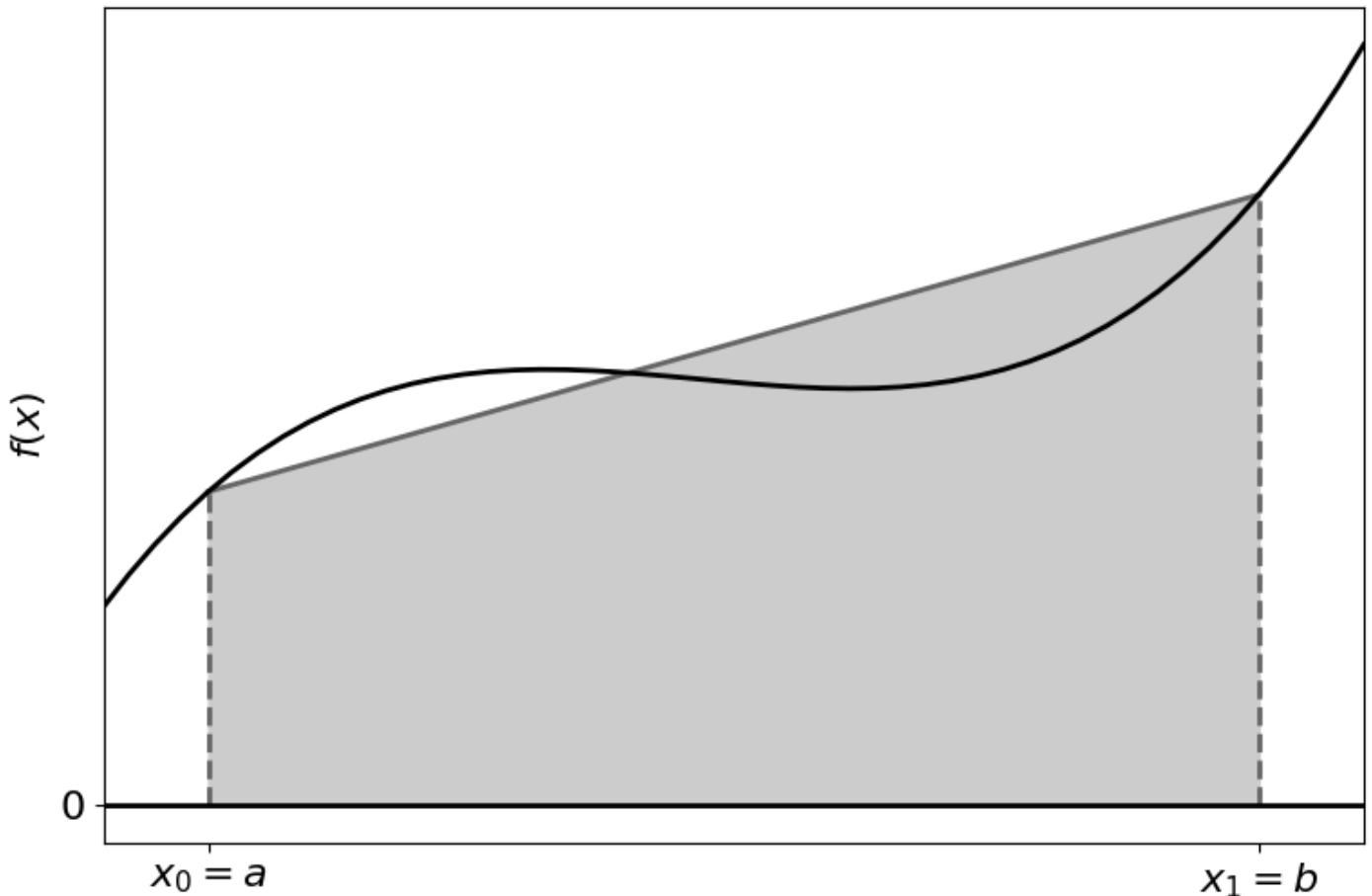
where the first and final contributions are halved as the intervals they represent are halved (note that these aren't at the midpoints of their intervals, rather at the boundaries).

# References

- [IntMid1] James F. Epperson. *An Introduction to Numerical Methods and Analysis*. John Wiley & Sons, Inc., Hoboken, New Jersey, second edition edition, 2013.

# Trapezoidal Rule

For the trapezoidal rule we approximate the integral of  $f(x)$  on the interval  $[a - b]$  by constructing the trapezium below:



and calculating it's area.

The area of the trapezium is given by:

$$A_{\text{trapezoid}} = A_{\text{rectangle}} + A_{\text{triangle}}$$

In the case where  $f(a) < f(b)$ , this area is given by:

$$\begin{aligned} A_{\text{trapezoid}} &= (b - a)f(a) + \frac{1}{2}(b - a)[f(b) - f(a)] \\ &= \frac{1}{2}(b - a)[f(a) + f(b)] \end{aligned}$$

In the case where  $f(a) > f(b)$ , the area is give by:

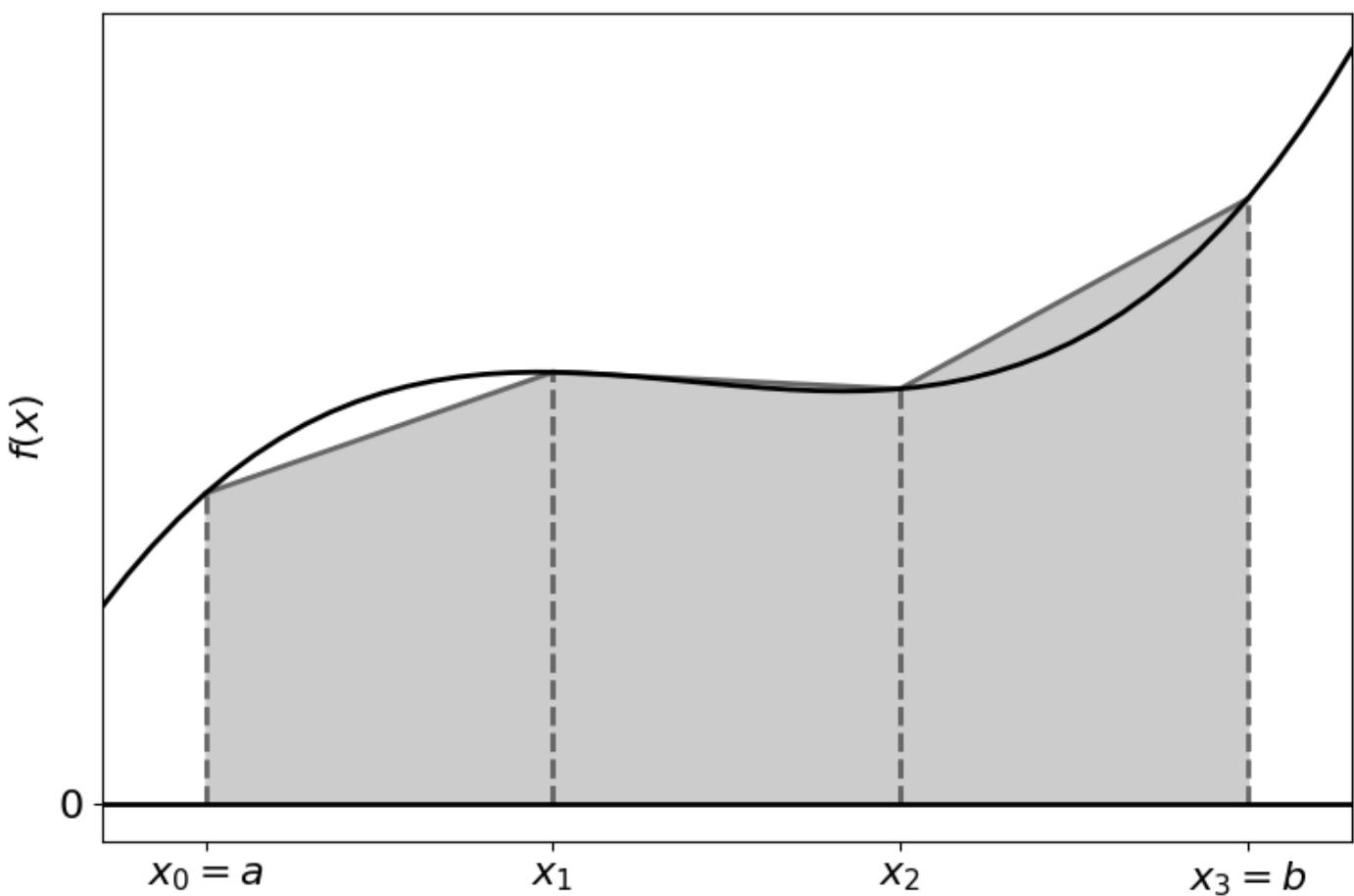
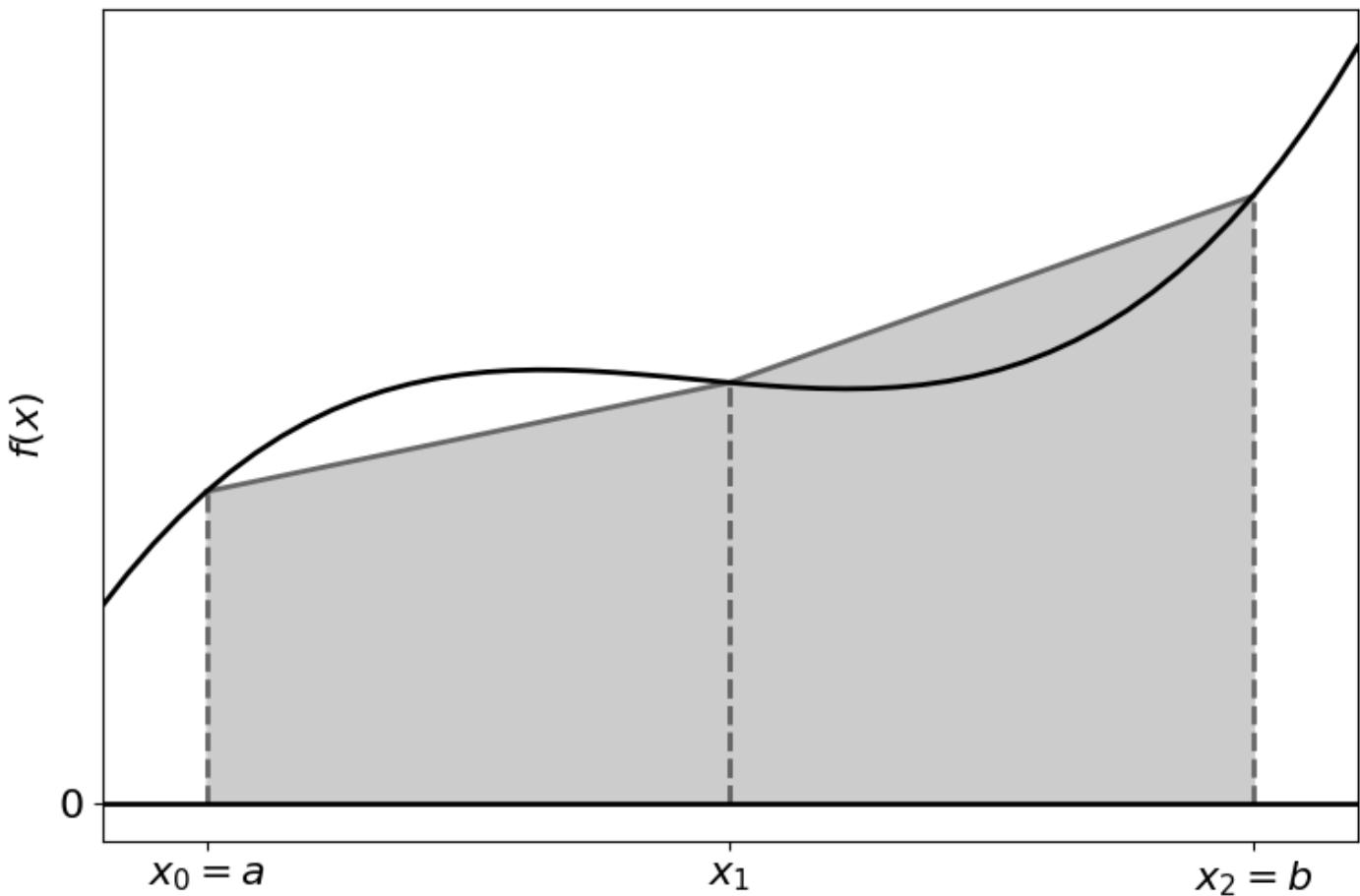
$$\begin{aligned} A_{\text{trapezoid}} &= (b - a)f(b) + \frac{1}{2}(b - a)[f(a) - f(b)] \\ &= \frac{1}{2}(b - a)[f(a) + f(b)] \end{aligned}$$

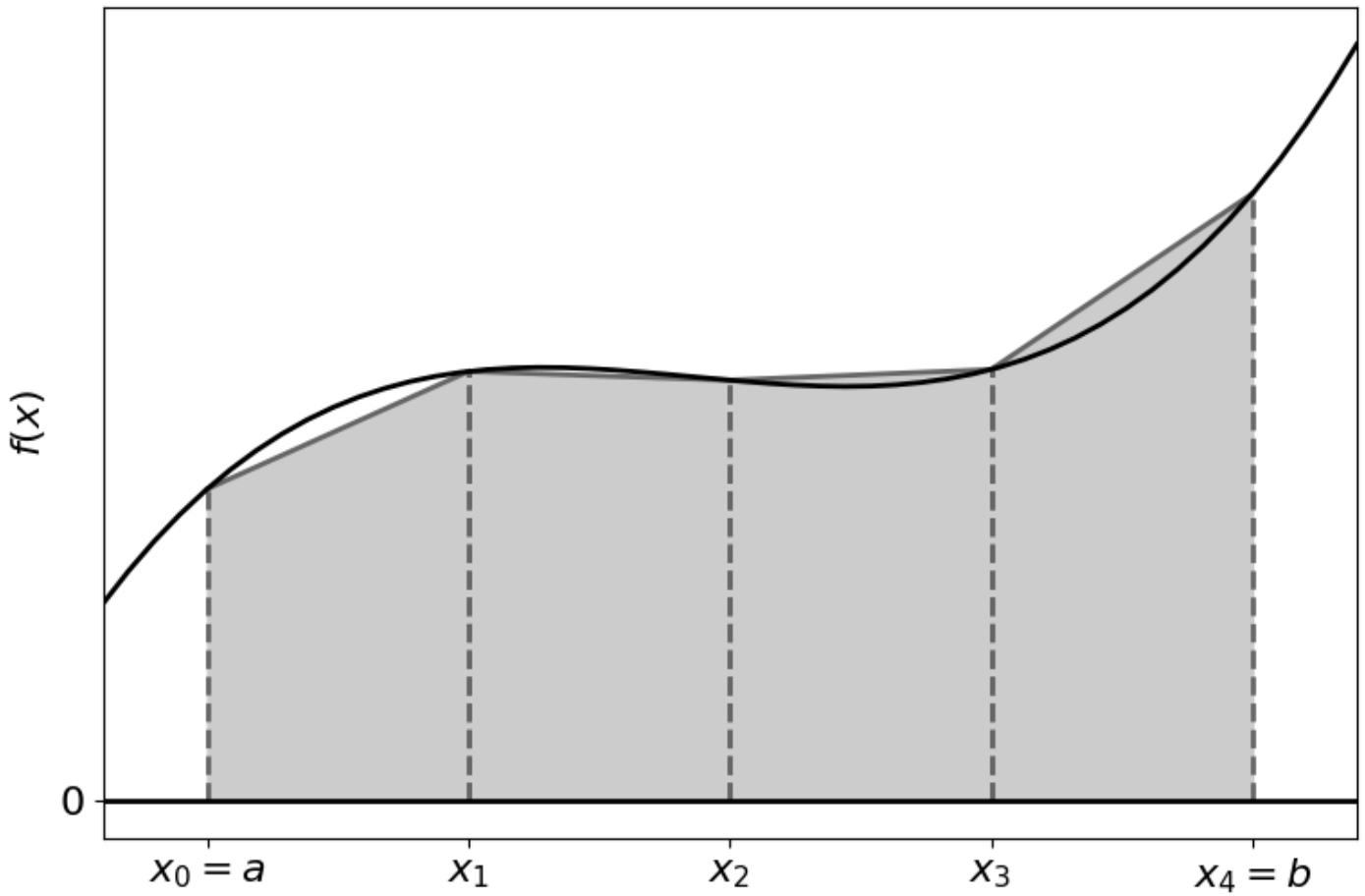
which is the same as above, so in general we can approximate the integral as:

$$\int_a^b f(x) dx \approx \frac{1}{2}(b - a)[f(a) + f(b)]$$

## Composite Trapezoid Rule

Now, if we were to break up this interval into  $n$  equal sub-intervals, and approximate the integral on each of these, we arrive at the composite trapezoidal rule (illustrated in the diagrams that follow).





To calculate this we use the sum:

$$\int_a^b f(x) dx \approx \sum_{i=1}^n \frac{1}{2}(x_i - x_{i-1}) [f(x_{i-1}) + f(x_i)]$$

where  $x_0 = a$  and  $x_n = b$ . As we have specified that each of the  $n$  subintervals are of equal sizes, we have that:

$$x_i - x_{i-1} = \frac{b - a}{n}$$

we can therefore write the approximation as:

$$\int_a^b f(x) dx \approx \frac{b - a}{2n} \sum_{i=1}^n [f(x_{i-1}) + f(x_i)]$$

note how each  $f(x_i)$  in the sum above is repeated twice, except for  $f(x_0)$  and  $f(x_n)$ , which only feature once each. We can now write the approximation as:

$$\int_a^b f(x) dx \approx \frac{b-a}{2n} \left[ f(a) + 2 \left\{ \sum_{i=1}^{n-1} f(x_i) \right\} + f(b) \right]$$

For a choice of  $n$  such that  $0 < \frac{b-a}{n} < 1$ , the error for this method is  $O\left(\frac{1}{n^2}\right)$  [IntTrap1].

## Composite Trapezoidal Rule with a Discrete Data Set

Again, consider the data set  $(x_i, y_i)$  for  $i = 0, \dots, n$ , where

$$f(x_i) = y_i$$

If we wanted to approximate the integral of this data set using the trapezoidal rule, we can apply this to each interval individually:

$$\int_{x_0}^{x_n} f(x) dx \approx \frac{1}{2} \sum_{i=1}^n (x_i - x_{i-1}) [y_{i-1} + y_i]$$

If the  $x_i$  values are evenly spaced, with  $x_i - x_{i-1} = \Delta x$  constant, then we can use the composite formula from the section above:

$$\int_{x_0}^{x_n} f(x) dx \approx \frac{\Delta x}{2} \left[ y_0 + 2 \left\{ \sum_{i=1}^{n-1} y_i \right\} + y_n \right]$$

## References

**[IntTrap1]** James F. Epperson. *An Introduction to Numerical Methods and Analysis*. John Wiley & Sons, Inc., Hoboken, New Jersey, second edition edition, 2013.

# Simpson's Rule

A way to interpret the trapezoidal method is that we approximate the integrand curve as a straight line, and then integrate that directly. The Simpson's method does something similar, but instead of only using the integrand values at the boundaries it uses a third point (the midpoint) to approximate the integrand as a parabola.

The method approximates the integral as:

$$\int_a^b f(x) dx \approx \frac{b-a}{6} \left[ f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right]$$

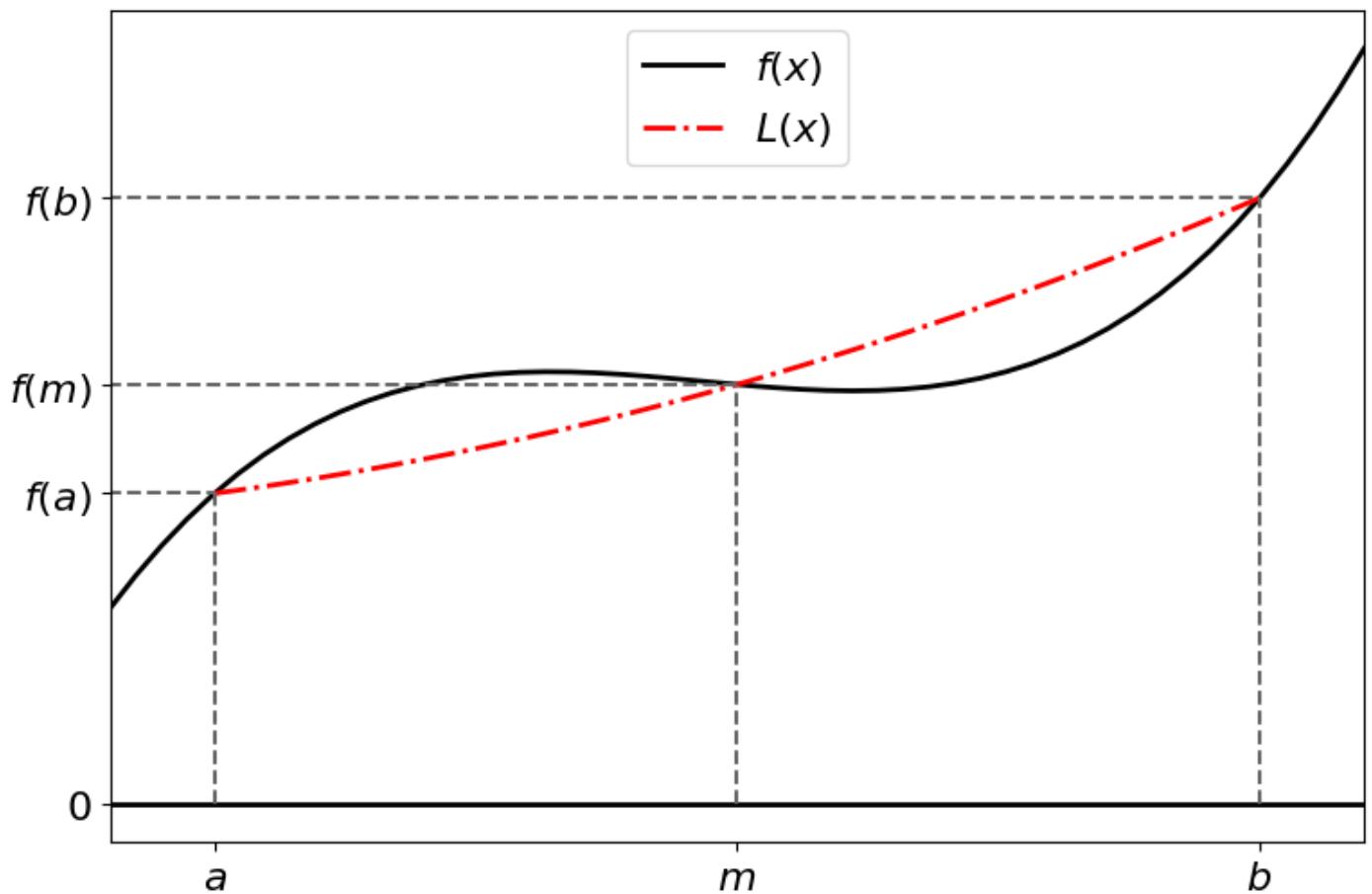
with the derivation of this in the following section.

## Derivation

### Approximating $f(x)$ as a Quadratic Polynomial

In order to approximate  $f(x)$  as a quadratic polynomial we can use a second order Lagrange polynomial. We construct this polynomial by using 3 data points  $(a, f(a))$ ,  $(m, f(m))$  and  $(b, f(b))$ .

$$L(x) = f(a) \frac{(x-m)(x-b)}{(a-m)(a-b)} + f(m) \frac{(x-a)(x-b)}{(m-a)(m-b)} + f(b) \frac{(x-a)(x-m)}{(b-a)(b-m)}$$



If we use the midpoint of  $[a, b]$  for  $m$ , i.e:

$$m = \frac{1}{2}(a + b)$$

then we have

$$m - a = b - m = \frac{1}{2}(b - a)$$

and the Lagrange polynomial becomes:

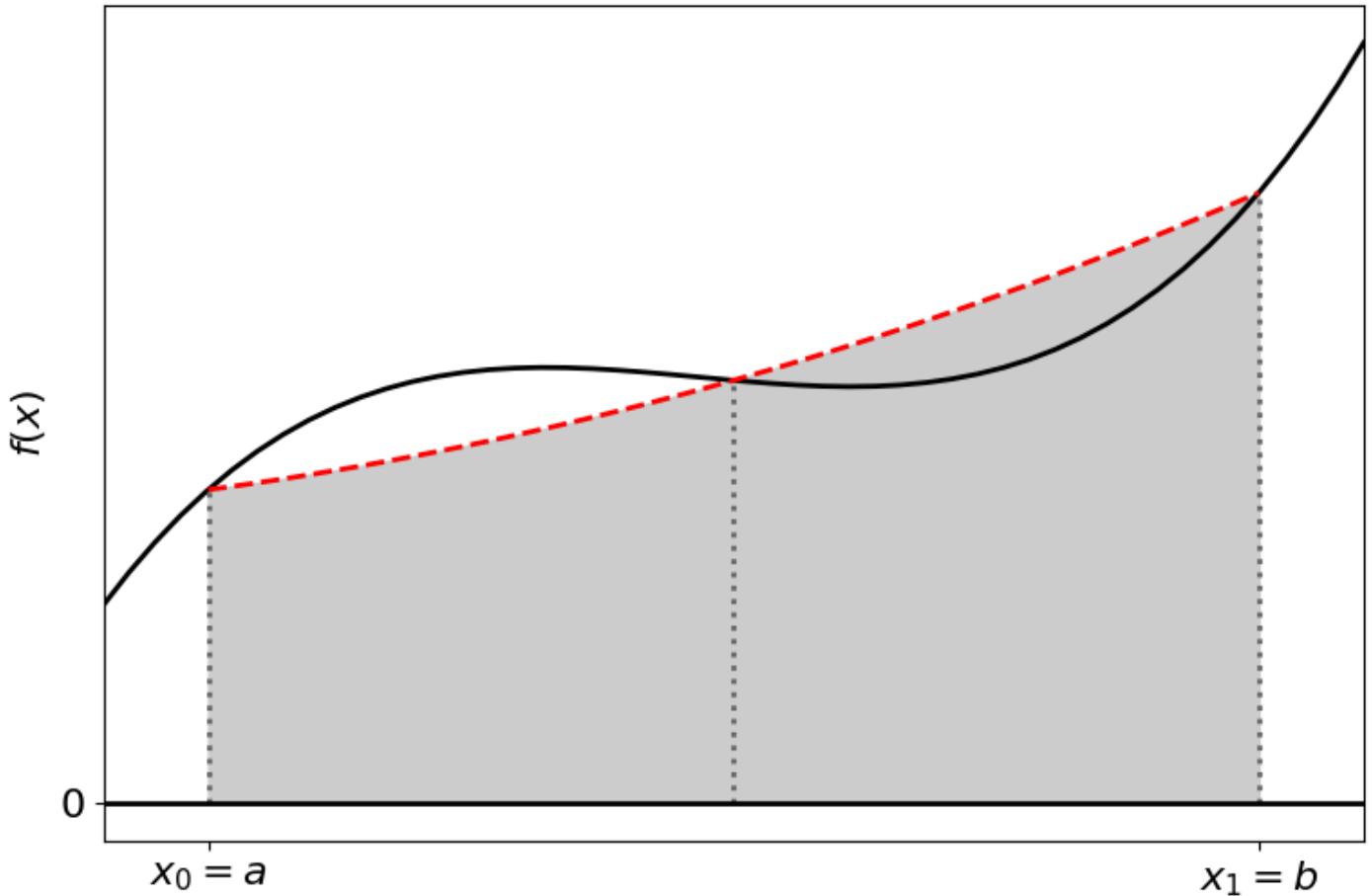
$$L(x) = \frac{2}{(b-a)^2} [f(a)(x-m)(x-b) - 2f(m)(x-a)(x-b) + f(b)(x-a)(x-m)]$$

## Integrating $L(x)$

Now, we wish to approximate the integral of  $f(x)$  as the integral of our Lagrange polynomial:

$$\int_a^b f(x) dx \approx \int_a^b L(x) dx$$

illustrated in the figure below:



by substituting the variable:

$$u = \frac{x - m}{b - m} = 2 \frac{x - \frac{1}{2}(a + b)}{b - a}$$

into the integral of  $L(x)$  we find that:

$$\int_a^b L(x) dx = \frac{b - a}{6} \left[ f(a) + 4f\left(\frac{a + b}{2}\right) + f(b) \right]$$

thus, we approximate the integral of  $f(x)$  as:

$$\int_a^b f(x) dx \approx \frac{b-a}{6} \left[ f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right]$$

## Alternative Derivation

Another way to see the Simpson's rule is as a weighted average of the midpoint and trapezoidal rules:

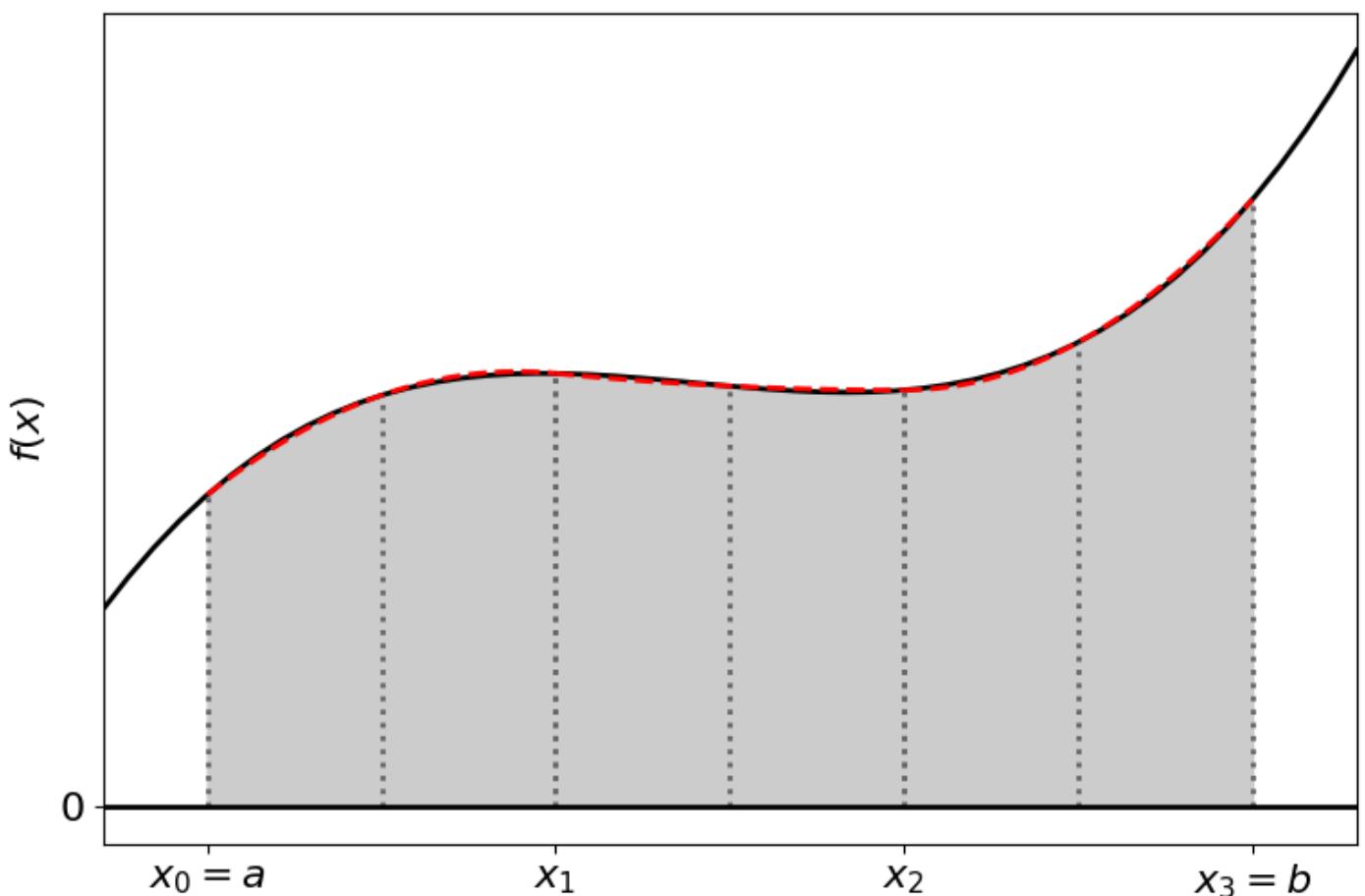
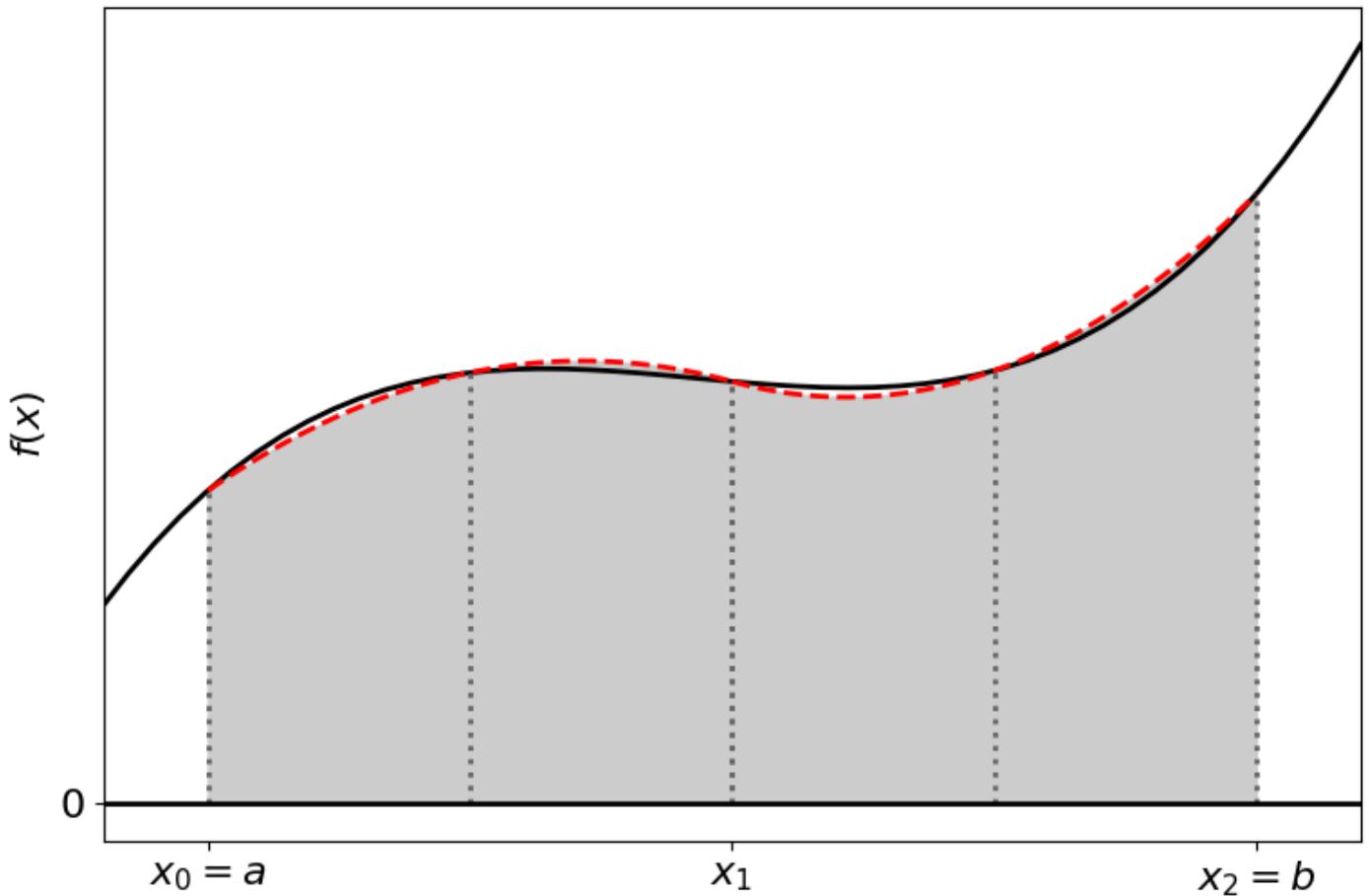
2/3 midpoint + 1/3 trapezoidal

Mathematically:

$$\begin{aligned} \int_a^b f(x) dx &\approx \frac{2}{3} f\left(\frac{a+b}{2}\right)(b-a) + \frac{1}{3} \times \frac{1}{2}(b-a)[f(a) + f(b)] \\ &\approx \frac{b-a}{6} \left[ f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right] \end{aligned}$$

## Composite Simpson's Rule

As before we can improve the accuracy of our solution by subdividing the interval and calculating the integral using Simpson's rule for each subinterval.



The composite Simpson's rule is given by:

$$\int_a^b f(x) dx \approx \sum_{i=1}^n \frac{x_i - x_{i-1}}{6} \left[ f(x_{i-1}) + 4f\left(\frac{x_{i-1} + x_i}{2}\right) + f(x_i) \right]$$

If we use equal sized subintervals, then we have that:

$$x_i - x_{i-1} = \frac{b-a}{n}$$

taking this into account and the values of  $f(x_i)$  which are repeated in the sum, the composite Simpson's rule can be simplified to:

$$\int_a^b f(x) dx \approx \frac{b-a}{6n} \left[ f(a) + 2 \left\{ \sum_{i=1}^{n-1} f(x_i) \right\} + 4 \left\{ \sum_{i=1}^n f\left(\frac{x_{i-1} + x_i}{2}\right) \right\} + f(b) \right]$$

Assuming that  $0 < \frac{b-a}{6n} < 1$ , then the error for this method is  $O\left(\frac{1}{n^4}\right)$  [IntSimp1]

## Composite Simpson's Rule with a Discrete Data Set

Again, consider the data set  $(x_i, y_i)$  for  $i = 0, \dots, n$ , where

$$f(x_i) = y_i$$

Approximating the integral of the data using Simpson's rule is a lot less straight forward than the Midpoint and Trapezoidal rule as we can't calculate  $f((x_{i-1} + x_i)/2)$  directly, and the function values at the boundaries of the intervals are also required.

If the  $x_i$  values are **uniformly spaced** (with  $x_i - x_{i-1} = \Delta x$ ) and there is an odd number of data points, we can pair up the intervals between the points. Treating each pair of subintervals as a single subinterval, we can use the middle  $x$  values as the midpoints. In other words, for even  $i = 2j$ , the  $x_{2j}$  are used as the boundaries of the sub-intervals; for odd  $i = 2k - 1$ , the  $x_{2k-1}$  are used as the midpoints of the subinterval. Thus the integral is approximated as:

$$\int_{x_0}^{x_n} f(x) dx \approx \frac{\Delta x}{3} \left[ y_0 + 2 \left\{ \sum_{i=1}^{n/2-1} y_{2i} \right\} + 4 \left\{ \sum_{i=1}^{n/2} y_{2i-1} \right\} + y_n \right]$$

Note that

$$\frac{b-a}{6n} = \frac{2\Delta x}{6} = \frac{\Delta x}{3}$$

as each subinterval is made up of two intervals of length  $\Delta x$  each.

## References

**[IntSimp1]** James F. Epperson. *An Introduction to Numerical Methods and Analysis*. John Wiley & Sons, Inc., Hoboken, New Jersey, second edition edition, 2013.