Lecture Book
Python for Eng. and Eco.

D. T. McGuiness, PhD

Version: 2024

Topics on Computer Science

(2024, D. T. McGuiness, PhD)

Current version is 2024.

This document includes the contents of Python for Eng. and Eco. taught at MCI. This document is the part of Interdisciplinary Elective.

All relevant code of the document is done using *SageMath* v10.3 and Python v3.12.5.

This document was compiled with LuaTEX and all editing were done using
GNU Emacs using AUCTEX and org-mode package.

This document is based on resources on Python programming, which includes *Defining Your Own Python Function* by J. Sturtz.

The current maintainer of this work along with the primary lecturer
is D. T. McGuiness, PhD (dtm@mci4me.at).

Contents

Part I.

Fundamentals

Chapter 1

Functions

Table of Contents

1.1 Introduction

We may be familiar with the mathematical concept of a function. A function is a relationship or mapping between one or more inputs and a set of outputs. In mathematics, a function is typically represented like this:

$$z = f(x, y),$$

where, f is a function operating on inputs x and y. The output of the function is z. However, programming functions are much more generalised and versatile compared to its mathematical counterpart. In fact, appropriate function definition and use is so critical to proper software development that virtually all modern programming languages support both built-in and user-defined functions.

In programming, a function is defined as:

> *self-contained block of code encapsulating a specific task or a group of tasks.*

Previously, we have worked with some of the *built-in functions* provided by Python.

`id()` Takes one (1) argument and returns the object's unique integer identifier:

```python
s = 'foobar'
id(s)
```

`len()` Returns the length of the argument passed to it:

```python
a = ['foo', 'bar', 'baz', 'qux']
print(len(a))
```

```text
4
```

`any()` Takes an iterable as its argument and returns `True` if any of the items in the iterable are truthy and `False` otherwise:

```python
print(any([False, False, False]))
print(any([False, True, False]))
print(any(['bar' == 'baz', len('foo') == 4, 'qux' in {'foo', 'bar', 'baz'}]))
print(any(['bar' == 'baz', len('foo') == 3, 'qux' in {'foo', 'bar', 'baz'}]))
```

```text
False
True
False
True
```

Each of these built-in functions performs a specific task. The code that accomplishes the task is defined somewhere, but we don't need to know where or even how the code works. All we need to know about is the function's **interface**:

1. What **arguments** (if any) it takes
2. What **values** (if any) it returns

Then we call the function and pass the appropriate arguments. Program execution goes off to the designated body of code and does its useful thing. When the function is finished, execution returns to our code where it left off.

> The function may or may not return data for our code to use.

When we define our **own** function, it works just the same. From somewhere in our code, we'll call our Python function and program execution will transfer to the body of code that makes up the function.

When the function is finished, execution returns to the location where the function was called. Depending on how we designed the function's interface, data may be passed in when the function is called, and return values may be passed back when it finishes.

1.2 Importance of Python Functions

Almost all programming languages used today support a form of **user-defined functions**, although they aren't always called functions. In other languages, we may see them referred to as one of the following:

- Subroutines
- Procedures
- Methods
- Subprograms

We may start to wonder what is the big deal of defining our own function? There are several very good reasons. Let's go over a few now.

1.2.1 Abstraction and Re-usability

Suppose we write some code that does something useful. As we continue development, we find the task performed by that code is one we need often, in many different locations within our application.

What should we do?

Well, we could just replicate the code over and over again, using Ctrl-C & Ctrl-V.

Later on, we'll probably decide that the code in question needs to be modified. We'll either find something wrong with it that needs to be fixed, or we'll want to enhance it in some way. If copies of the code are scattered all over our application, then we'll need to make the necessary changes in every location.

> At first, it may be a reasonable solution, but it's likely to be a maintenance nightmare. While our code editor may help by providing a search-and-replace function (such as Emacs replace-string), this method is error-prone, and we could easily introduce bugs into our code that will be difficult to find.

A better solution is to define a Python function that performs the task. Anywhere in our application that we need to accomplish the task, we simply call the function. Down the line, if we decide to change how it works, then we only need to change the code in one location, which is the place where the function is defined. The changes will automatically be picked up anywhere the function is called.

The abstraction of functionality into a function definition is an example of the **Don't Repeat Yourself** (DRY) Principle of software development.

This is arguably the strongest motivation for using functions.

1.2.2 Modularity

Functions allow complex processes to be broken up into smaller steps. For example, we have a program that reads in a file, processes the file contents, and then writes an output file. Our code could look like this:

```python
# Main program

# Code to read file in
<statement>
<statement>

# Code to process file
<statement>
<statement>

# Code to write file out
<statement>
<statement>
```

In this example, the main program is a bunch of code strung together in a long sequence, with whitespace and comments to help organize it. However, if the code were to get much lengthier and more complex, then we'd have an increasingly difficult time wrapping our head around it.

Or, we could structure the code more like the following:

```python
def read_file():
    # Code to read file in
    <statement>
    <statement>

def process_file():
    # Code to process file
    <statement>
    <statement>

def write_file():
    # Code to write file out
    <statement>
    <statement>

# Main program
read_file()
process_file()
write_file()
```

The above example is modularised. Instead of all the code being strung together, it's broken out into **separate functions**, each of which focuses on a specific task. Those tasks are read, process, and write. The main program now simply needs to call each of these in turn.

The `def` keyword introduces a new Python function definition.

In life, we do this sort of thing all the time, even if we don't explicitly think of it that way. If we wanted to tackle any problem we'd divide the job into manageable steps.

Breaking a large task into smaller, bite-sized sub-tasks helps make the large task easier to think about and manage. As programs become more complicated, it becomes increasingly beneficial to modularize them in this way.

1.2.3 Namespace Separation

A **namespace** is a region of a program in which identifiers have meaning. As we'll see below, when a Python function is called, a new namespace is created for that function, one that is distinct from all other namespaces that already exist.

The practical upshot of this is that variables can be defined and used within a Python function even if they have the same name as variables defined in other functions or in the main program. In these cases, there will be no confusion or interference because they're kept in separate namespaces.

This means that when we write code within a function, we can use variable names and identifiers without worrying about whether they're already used elsewhere outside the function. This helps minimize errors in code considerably.

1.3 Function Calls and Definition

The usual syntax for defining a Python function is as follows:

```python
def <function_name>([<parameters>]):
    <statement(s)>
```

The components of the definition are explained in the table below:

Component	Meaning
`def`	The keyword that informs Python that a function is being defined
`<function_name>`	A valid Python identifier that names the function
`<parameters>`	An optional, comma-separated list of parameters that may be passed to the function
`:`	Punctuation that denotes the end of the Python function header (the name and parameter list)
`<statement(s)>`	A block of valid Python statements

Table 1.1.: The components of what makes a function.

The final item, `<statement(s)>`, is called the **body of the function**. The body is a block of statements that will be executed when the function is called. The body of a Python function is defined by indentation in accordance with the off-side rule.

This is the same as code blocks associated with a control structure, similar to an `if` or `while` statement.

The syntax for calling a Python function is as follows:

```python
<function_name>([<arguments>])
```

`<arguments>` are the values passed into the function. They correspond to the `<parameters>` in the Python function definition. We can define a function that doesn't take any arguments, but the parentheses are still required. Both a function definition and a function call must **ALWAYS** include parentheses, even if they're empty.

As usual, we will start with a small example and add complexity from there. Keeping the mathematical tradition in mind, we will call our first Python function `f()`. Here's a code snippet defining and calling `f()`:

```python
def f():
    s = '-- Inside f()'
    print(s)

print('Before calling f()')
f()
print('After calling f()')
```

```text
Before calling f()
-- Inside f()
After calling f()
```

Here's how this code works:

1. First line uses the `def` keyword to indicate that a function is being defined. Execution of the `def` statement merely creates the definition of `f()`.
2. All the following lines that are indented become part of the body of `f()` and are stored as its definition, but they aren't executed yet.
3. Empty line is a bit of whitespace between the function definition and the first line of the main program. While it isn't syntactically necessary, it is nice to have.
4. This is the first statement that isn't indented because it isn't a part of the definition of `f()`. It's the start of the main program. When the main program executes, this statement is executed first.
5. This is a call to `f()`. Note that empty parentheses are always required in both a function definition and a function call, even when there are no parameters or arguments. Execution proceeds to `f()` and the statements in the body of `f()` are executed.
6. Here execute once the body of `f()` has finished. Execution returns to this `print()` statement.

Occasionally, we may want to define an empty function that does **nothing**. This is referred to as a **stub**, which is usually a temporary placeholder for a Python function that will be fully implemented at a later time. Just as a block in a control structure can't be empty, neither can the body of a function. To define a stub function, use the passstatement:

```python
def f():
    pass

f()
```

As we can see above, a call to a stub function is syntactically valid but doesn't do anything.

Example Sum of Two Number 1

Write a function which takes two values (say a, and b) and returns their sum.

Solution Sum of Two Number

Sum of Two Number

```python
def sum_values(a, b):
    result = a + b
    return result

print(sum_values(1, 2))
```

Example Maximum of Three Numbers 2

Write a Python function to find the maximum of three numbers.

Solution Maximum of Three Numbers

Maximum of Three Numbers

```python
# Define a function that returns the maximum of two numbers
def max_of_two(x, y):
    # Check if x is greater than y
    if x > y:
        # If x is greater, return x
        return x
    # If y is greater or equal to x, return y
    return y

# Define a function that returns the maximum of three numbers
def max_of_three(x, y, z):
    # Call max_of_two function to find the maximum of y and z,
    # then compare it with x to find the overall maximum
    return max_of_two(x, max_of_two(y, z))

# Print the result of calling max_of_three function with arguments 3, 6, and -5
```

```python
print(max_of_three(3, 6, -5))
```

Sum of a list — 3 Example

Write a Python function to sum all the numbers in a list.

- Sample List : (8, 2, 3, 0, 7)
- Expected Output : 20

Sum of a list — Solution

Sum of a list

```python
# Define a function named 'sum' that takes a list of numbers as input
def sum(numbers):
    # Initialize a variable 'total' to store the sum of numbers, starting at 0
    total = 0

    # Iterate through each element 'x' in the 'numbers' list
    for x in numbers:
        # Add the current element 'x' to the 'total'
        total += x

    # Return the final sum stored in the 'total' variable
    return total

# Print the result of calling the 'sum' function with a tuple of numbers (8, 2, 3,
↪  0, 7)
print(sum((8, 2, 3, 0, 7)))
```

Multiply Values in a List — 4 Example

Write a Python function to multiply all the numbers in a list.

- Sample List: (8, 2, 3, -1, 7)
- Expected Output : -336

Multiply Values in a List — Solution

Multiply Values in a List

```python
# Define a function named 'multiply' that takes a list of numbers as input
def multiply(numbers):
    # Initialize a variable 'total' to store the multiplication result, starting at
    ↪  1
    total = 1

    # Iterate through each element 'x' in the 'numbers' list
    for x in numbers:
        # Multiply the current element 'x' with the 'total'
        total *= x

    # Return the final multiplication result stored in the 'total' variable
    return total

# Print the result of calling the 'multiply' function with a tuple of numbers (8,
↪  2, 3, -1, 7)
print(multiply((8, 2, 3, -1, 7)))
```

Example Printing a Sentence Multiple Times 5

Write a Python function that prompts the user to enter a number. Then, given that number, it prints the sentence ”Hello, Python!” that many times.

Solution Printing a Sentence Multiple Times

Printing a Sentence Multiple Times

```python
def print_sentence_multiple_times():
    num = int(input("Enter a number: "))
    for _ in range(num):
        print("Hello, Python!")


print_sentence_multiple_times()
```

Example Function to Make a List 6

Write a function that accepts different values as parameters and returns a list.

Solution Function to Make a List

Function to Make a List

```python
def myFruits(f1,f2,f3,f4):
    FruitsList = [f1,f2,f3,f4]
    return FruitsList

```

```python
output = myFruits("Apple","Bannana","Grapes","Orange")
print(output)
```

Reversing a String — 7 Example

Write a Python program to reverse a string.

- Sample String : ”1234abcd”
- Expected Output : ”dcba4321”

Reversing a String — Solution

Reversing a String

```python
# Define a function named 'string_reverse' that takes a string 'str1' as input
def string_reverse(str1):
    # Initialize an empty string 'rstr1' to store the reversed string
    rstr1 = ''

    # Calculate the length of the input string 'str1'
    index = len(str1)

    # Execute a while loop until 'index' becomes 0
    while index > 0:
        # Concatenate the character at index - 1 of 'str1' to 'rstr1'
        rstr1 += str1[index - 1]

        # Decrement the 'index' by 1 for the next iteration
        index = index - 1

    # Return the reversed string stored in 'rstr1'
    return rstr1

# Print the result of calling the 'string_reverse' function with the input string
↪  '1234abcd'
print(string_reverse('1234abcd'))
```

Counting Cases — 8 Example

Write a Python function that accepts a string and counts the number of upper and lower case letters.

- Sample String : ’The quick Brow Fox’
- Expected Output :

- No. of Upper case characters : 3
- No. of Lower case Characters : 12

Solution Counting Cases

Counting Cases

```python
# Define a function named 'string_test' that counts the number of upper and lower
↪  case characters in a string 's'
def string_test(s):
    # Create a dictionary 'd' to store the count of upper and lower case characters
    d = {"UPPER_CASE": 0, "LOWER_CASE": 0}

    # Iterate through each character 'c' in the string 's'
    for c in s:
        # Check if the character 'c' is in upper case
        if c.isupper():
            # If 'c' is upper case, increment the count of upper case characters in
            ↪  the dictionary
            d["UPPER_CASE"] += 1
        # Check if the character 'c' is in lower case
        elif c.islower():
            # If 'c' is lower case, increment the count of lower case characters in
            ↪  the dictionary
            d["LOWER_CASE"] += 1
        else:
            # If 'c' is neither upper nor lower case (e.g., punctuation, spaces),
            ↪  do nothing
            pass

    # Print the original string 's'
    print("Original String: ", s)

    # Print the count of upper case characters
    print("No. of Upper case characters: ", d["UPPER_CASE"])

    # Print the count of lower case characters
    print("No. of Lower case Characters: ", d["LOWER_CASE"])

# Call the 'string_test' function with the input string 'The quick Brown Fox'
string_test('The quick Brown Fox')
```

1.4 Argument Passing

So far, the functions we have defined haven't taken any arguments. That can sometimes be useful, and we'll occasionally write such functions. More often, though, we'll want to pass data into a function so that its behaviour can vary from one invocation to the next. Let's see how to do that.

1.4.1 Positional Arguments

The most straightforward way to pass arguments to a Python function is with positional arguments (it is also called required arguments). In the function definition, we specify a comma-separated list of parameters inside the parentheses:

```python
def f(qty, item, price):
    print(f'{qty} {item} cost ${price:.2f}')
```

When the function is called, we specify a corresponding list of arguments:

```python
f(6, 'bananas', 1.74)
```

```text
6 bananas cost $1.74
```

The parameters (`qty`, `item`, and `price`) behave like variables that are defined locally to the function. When the function is called, the arguments that are passed are bound to the parameters in order, as though by variable assignment:

Parameter	Argument
qty	6
item	bananas
price	1.74

Table 1.2.: A closer look at the parameters.

> In some programming texts, the parameters given in the function definition are referred to as formal parameters, and the arguments in the function call are referred to as actual parameters

Although positional arguments are the most straightforward way to pass data to a function, they also afford the least flexibility. To start, the order of the arguments in the call must match the order of the parameters in the definition. There's nothing to stop us from specifying positional arguments out of order, of course:

```python
f('bananas', 1.74, 6)
```

The function may even still run, as it did in the example above, but it's very unlikely to produce the correct results. It's the responsibility of the programmer who defines the function to document what the appropriate arguments should be, and it's the responsibility of the user of the function to be aware of that information and abide by it.

With positional arguments, the arguments in the call and the parameters in the definition must agree not only in order but in number as well. That's the reason positional arguments are also referred to as **required arguments**. We can't leave any out when calling the function:

```python
# Too few arguments and would give an error
f(6, 'bananas')
```

We can't also specify extra ones:

```python
# Too many arguments
f(6, 'bananas', 1.74, 'kumquats')
```

Positional arguments are conceptually straightforward to use, but they're not very forgiving.

> We must specify the same number of arguments in the function call as there are parameters in the definition, and in exactly the same order.

1.4.2 Keyword Arguments

When we're calling a function, we can specify arguments in the form `<keyword>=<value>`. In that case, each `<keyword>` must match a parameter in the Python function definition. For example, the previously defined function `f()` may be called with keyword arguments as follows:

```python
f(qty=6, item='bananas', price=1.74)
```

Referencing a keyword that doesn't match any of the declared parameters generates an exception:

```python
f(qty=6, item='bananas', cost=1.74)
```

Using keyword arguments lifts the restriction on argument order.

Each keyword argument explicitly designates a specific parameter by name, so we can specify them in any order and Python will still know which argument goes with which parameter:

```python
f(item='bananas', price=1.74, qty=6)
```

Like with positional arguments, though, the number of arguments and parameters must **still match**:

```python
# Still too few arguments
f(qty=6, item='bananas')
```

So, keyword arguments allow flexibility in the order that function arguments are specified, but the number of arguments is still rigid.

We can call a function using both positional and keyword arguments:

```python
print(f(6, price=1.74, item='bananas'))
print(f(6, 'bananas', price=1.74))
```

When positional and keyword arguments are both present, all the positional arguments must come first:

```python
f(6, item='bananas', 1.74)
```

Once we've specified a keyword argument, there can't be any positional arguments to the right of it.

1.4.3 Default Parameters

If a parameter specified in a Python function definition has the form `<name>=<value>`, then `<value>` becomes a default value for that parameter. Parameters defined this way are referred to as **default** or **optional** parameters. An example of a function definition with default parameters is shown below:

```python
def f(qty=6, item='bananas', price=1.74):
    print(f'{qty} {item} cost ${price:.2f}')
```

When this version of `f()` is called, any argument that's left out assumes its default value:

```python
print(f(4, 'apples', 2.24))
print(f(4, 'apples'))
print(f(4))
print(f())
print(f(item='kumquats', qty=9))
print(f(price=2.29))
```

1.4.4 Mutable Default Parameter Values

mutable is the ability of objects to change their values

Things can get weird if we specify a default parameter value that is a **mutable** object. Consider this Python function definition:

```python
def f(my_list=[]):
    my_list.append('###')
    return my_list
```

Here, `f()` takes a single list parameter, appends the string '###' to the end of the list, and returns the result:

```python
print(f(['foo', 'bar', 'baz']))
print(f([1, 2, 3, 4, 5]))
```

```text
['foo', 'bar', 'baz', '###']
[1, 2, 3, 4, 5, '###']
```

The default value for parameter `my_list` is the empty list, so if `f()` is called without any arguments, then the return value is a list with the single element '###':

```python
print(f())
```

```text
['###']
```

Everything makes sense so far. Now, what would we expect to happen if `f()` is called without any parameters a second and a third time? Let's see:

```python
print(f())
print(f())
```

```text
['###', '###']
['###', '###', '###']
```

We expected each subsequent call to also return the singleton list ['###'], just like the first. Instead, the return value keeps growing. What happened?

In Python, default parameter values are defined only once when the function is defined (that is, when the def statement is executed). The default value isn't re-defined each time the function is called.

Thus, each time we call `f()` without a parameter, we're performing `.append()` on the same list.
We can demonstrate this with `id()`:

```python
def f(my_list=[]):
    print(id(my_list))
    my_list.append('###')
    return my_list

print(f())
print(f())
print(f())
```

```text
4310647808
['###']
4310647808
['###', '###']
4310647808
['###', '###', '###']
```

The object identifier displayed confirms , when `my_list` is allowed to default, the value is the same object with each call. As lists are mutable, each subsequent `.append()` call causes the list to get longer. This is a common and pretty well-documented pitfall when we're using a mutable object as a parameter's default value. It potentially leads to confusing code behavior, and is probably best avoided.

As a workaround, consider using a default argument value that signals no argument has been specified. Most any value would work, but None is a common choice. When the sentinel value indicates no argument is given, create a new empty list inside the function:

```python
def f(my_list=None):
    if my_list is None:
        my_list = []
    my_list.append('###')
    return my_list

print(f())
print(f())
print(f())
print(f(['foo', 'bar', 'baz']))
print(f([1, 2, 3, 4, 5]))
```

```text
['###']
['###']
['###']
['foo', 'bar', 'baz', '###']
[1, 2, 3, 4, 5, '###']
```

This ensures that my_{list} now truly defaults to an empty list whenever `f()` is called without an argument.

Example Celcius to Fahrenheit 9

Write a program which converts celcius to fahrenheit.

$$F = \left(C\frac{9}{5}\right) + 32$$

Solution Celcius to Fahrenheit

Celcius to Fahrenheit

```python
# creating a function that converts the given celsius degree temperature
# to Fahrenheit degree temperature
def convertCelsiustoFahrenheit(c):
   # converting celsius degree temperature to Fahrenheit degree temperature
   f = (9/5)*c + 32
   # returning Fahrenheit degree temperature of given celsius temperature
   return (f)
# input celsius degree temperature
celsius_temp = 80
print("The input Temperature in Celsius is ",celsius_temp)
# calling convertCelsiustoFahrenheit() function by passing
# the input celsius as an argument
```

```python
fahrenheit_temp = convertCelsiustoFahrenheit(celsius_temp)
# printing the Fahrenheit equivalent of the given celsius degree temperature
print("The Fahrenheit equivalent of input celsius degree = ", fahrenheit_temp)
```

1.5 The return Statement

What's a Python function to do then? After all, in many cases, if a function doesn't cause some change in the calling environment, then there isn't much point in calling it at all. How should a function affect its caller?

As a tradition to let the user know everything worked, functions return 0 and if an error has occurred, they would return 1. This originally comes from UNIX computers.

Well, one possibility is to use function **return** values. A return statement in a Python function serves two purposes:

- It immediately terminates the function and passes execution control back to the caller.
- It provides a mechanism by which the function can pass data back to the caller.

1.5.1 Exiting a Function

Within a function, a return statement causes immediate exit from the Python function and transfer of execution back to the caller:

```python
def f():
    print('foo')
    print('bar')
    return

print(f())
```

```text
foo
bar
None
```

In this example, the return statement is actually superfluous (i.e., unnecessary). A function will return to the caller when it falls off the end—that is, after the last statement of the function body is executed.

So, this function would behave identically without the return statement.

However, return statements don't need to be at the end of a function. They can appear anywhere in a function body, and even multiple times.

Consider the following example:

```python
def f(x):
    if x < 0:
        return
    if x > 100:
        return
    print(x)

print(f(-3))
print(f(105))
print(f(64))
```

```text
None
None
64
None
```

The first two calls to `f()` don't cause any output, because a return statement is executed and the function exits prematurely, before the `print()` statement is reached. This sort of paradigm can be useful for error checking in a function. We can check several error conditions at the start of the function, with return statements that bail out if there's a problem:

```python
def f():
    if error_cond1:
        return
    if error_cond2:
        return
    if error_cond3:
        return

    <normal processing>
```

If none of the error conditions are encountered, then the function can proceed with its normal processing.

1.5.2 Returning Data to the Caller

In addition to exiting a function, the return statement is also used to pass data back to the caller. If a return statement inside a Python function is followed by an expression, then in the calling environment, the function call evaluates to the value of that expression:

```python
def f():
    return 'foo'

s = f()
s
```

Here, the value of expression `f()` is `foo`, which is subsequently assigned to variable `s`.

A function can return any type of object. In Python, that means pretty much anything whatsoever. In the calling environment, the function call can be used syntactically in any way that makes sense for the type of object the function returns.

You might be wondering the use of foo and bar. The terms "foo" and "bar" come from FUBAR, a military acronym meaning Fucked Up Beyond All Repair, a phrase which applies to many pieces of software. Leave it to programmers to have a dry sense of humour!

For example, in the code below, `f()` returns a dictionary. In the calling environment then, the expression `f()` represents a dictionary, and `f()['baz']` is a valid key reference into that dictionary:

```python
def f():
    return dict(foo=1, bar=2, baz=3)

print(f())
print(f()['baz'])
```

```text
{'foo': 1, 'bar': 2, 'baz': 3}
3
```

In the next example, `f()` returns a string that we can slice like any other string:

```python
def f():
    return 'foobar'

print(f()[2:4])
```

```text
ob
```

Here, `f()` returns a list that can be indexed or sliced:

```python
def f():
    return ['foo', 'bar', 'baz', 'qux']


print(f())
print(f()[2])
print(f()[::-1])
```

```text
['foo', 'bar', 'baz', 'qux']
baz
['qux', 'baz', 'bar', 'foo']
```

If multiple comma-separated expressions are specified in a return statement, then they're packed and returned as a `tuple`:

```python
def f():
    return 'foo', 'bar', 'baz', 'qux'

print(type(f()))

t = f()
print(t)

a, b, c, d = f()
print(f'a = {a}, b = {b}, c = {c}, d = {d}')
```

```text
<class 'tuple'>
('foo', 'bar', 'baz', 'qux')
a = foo, b = bar, c = baz, d = qux
```

When no return value is given, a Python function returns the special Python value None:

```python
def f():
    return

print(f())
```

```text
None
```

The same thing happens if the function body doesn't contain a return statement at all and the function falls off the end:

```python
def g():
    pass

print(g())
```

```text
None
```

None is treated as `False` when evaluated in a Boolean context.

Since functions that exit through a bare return statement or fall off the end return `None`, a call to such a function can be used in a Boolean context:

```python
def f():
    return

def g():
    pass

if f() or g():
    print('yes')
else:
    print('no')
```

```text
no
```

In the code above, calls to both `f()` and `g()` are falsy, so `f()` or `g()` is as well, and the `else` clause executes.

1.6 Variable-Length Argument Lists

In some cases, when we're defining a function, we may not know beforehand how many arguments we'll want it to take. Suppose, for example, that we want to write a Python function that computes the average of several values. We could start with something like this:

```python
def avg(a, b, c):
    return (a + b + c) / 3
```

All is well if we want to average just three (3) values:

```python
print(avg(1, 2, 3))
```

```text
2.0
```

However, as we've already seen, when **positional arguments** are used, the number of arguments passed must agree with the number of parameters declared. Clearly then, all isn't well with this implementation of `avg()` for any number of values that is not three:

```python
avg(1, 2, 3, 4)
```

We could try to define `avg()` with optional parameters:

```python
def avg(a, b=0, c=0, d=0, e=0):
    .
    .
    .
```

This allows for a variable number of arguments to be specified. The following calls are at least **syntactically** correct:

```python
avg(1)
avg(1, 2)
avg(1, 2, 3)
avg(1, 2, 3, 4)
avg(1, 2, 3, 4, 5)
```

But this approach still suffers from a couple of problems. For starters, it still only allows up to five arguments, not an arbitrary number. Worse yet, there's no way to distinguish between the arguments that were specified and those that were allowed to default.

The function has no way to know how many arguments were actually passed, so it doesn't know what to divide by:

```python
def avg(a, b=0, c=0, d=0, e=0):
    return (a + b + c + d + e) / # Divided by what???
```

Evidently, this won't do either.
We could write `avg()` to take a single list argument:

```python
def avg(a):
    total = 0
    for v in a:
            total += v
    return total / len(a)


```

```python
print(avg([1, 2, 3]))
print(avg([1, 2, 3, 4, 5]))
```

This works. It allows an arbitrary number of values and produces a correct result. As an added bonus, it works when the argument is a tuple as well:

```python
t = (1, 2, 3, 4, 5)
print(avg(t))
```

The drawback is that the added step of having to group the values into a list or tuple is probably not something the user of the function would expect, and it isn't very elegant. Whenever we find Python code that looks inelegant, there's probably a better option.

In this case, Python provides a way to pass a function a variable number of arguments with argument tuple packing and unpacking using the asterisk (∗) operator.

1.6.1 Argument Tuple Packing

When a parameter name in a Python function definition is preceded by an asterisk (∗), it indicates argument tuple packing. Any corresponding arguments in the function call are packed into a tuple that the function can refer to by the given parameter name.

Here's an example:

```python
def f(*args):
    print(args)
    print(type(args), len(args))
    for x in args:
            print(x)

print(f(1, 2, 3))
print(f('foo', 'bar', 'baz', 'qux', 'quux'))
```

```text
(1, 2, 3)
<class 'tuple'> 3
1
2
3
None
('foo', 'bar', 'baz', 'qux', 'quux')
<class 'tuple'> 5
```

```
foo
bar
baz
qux
quux
None
```

In the definition of `f()`, the parameter specification `*args` indicates tuple packing. In each call to `f()`, the arguments are packed into a tuple that the function can refer to by the name args. Any name can be used, but `args` is so commonly chosen that it's practically a standard.

Using tuple packing, we can clean up `avg()` like this:

```python
def avg(*args):
    total = 0
    for i in args:
        total += i
    return total / len(args)

print(avg(1, 2, 3))
print(avg(1, 2, 3, 4, 5))
```

```python
def avg(*args):
    total = 0
    for i in args:
        total += i
    return total / len(args)

print(avg(1, 2, 3))
print(avg(1, 2, 3, 4, 5))
```

Better still, we can tidy it up even further by replacing the for loop with the built-in Python function `sum()`, which sums the numeric values in any iterable:

```python
def avg(*args):
    return sum(args) / len(args)

print(avg(1, 2, 3))
print(avg(1, 2, 3, 4, 5))
```

```python
def avg(*args):
    return sum(args) / len(args)

print(avg(1, 2, 3))
print(avg(1, 2, 3, 4, 5))
```

Now, avg() is concisely written and works as intended.

Still, depending on how this code will be used, there may still be work to do. As written, avg() will produce a TypeError exception if any arguments are non-numeric:

TypeError is raised whenever an operation is performed on an incorrect/unsupported object type.

```python
avg(1, 'foo', 3)
```

To be as robust as possible, we could add code to check that the arguments are of the proper type.

1.6.2 Argument Tuple Unpacking

An analogous operation is available on the other side of the equation in a Python function call. When an argument in a function call is preceded by an asterisk (*), it indicates that the argument is a tuple that should be unpacked and passed to the function as separate values:

```python
def f(x, y, z):
    print(f'x = {x}')
    print(f'y = {y}')
    print(f'z = {z}')

print(f(1, 2, 3))

t = ('foo', 'bar', 'baz')
print(f(*t))
```

```text
x = 1
y = 2
z = 3
None
x = foo
y = bar
z = baz
None
```

In this example, *t in the function call indicates that t is a tuple that should be un-

packed. The unpacked values `'foo'`, `'bar'`, and `'baz'` are assigned to the parameters `x`, `y`, and `z`, respectively.
Although this type of unpacking is called tuple unpacking, it doesn't only work with tuples. The asterisk (`*`) operator can be applied to any iterable in a Python function call. For example, a list or set can be unpacked as well:

```python
a = ['foo', 'bar', 'baz']

print(type(a))
print(f(*a))

s = {1, 2, 3}
print(type(s))
print(f(*s))
```

```text
<class 'list'>
x = foo
y = bar
z = baz
None
<class 'set'>
x = 1
y = 2
z = 3
None
```

1.6.3 Argument Dictionary Unpacking

Argument dictionary unpacking is analogous to argument tuple unpacking. When the double asterisk (`**`) precedes an argument in a Python function call, it specifies that the argument is a dictionary that should be unpacked, with the resulting items passed to the function as keyword arguments:

```python
def f(a, b, c):
    print(F'a = {a}')
    print(F'b = {b}')
    print(F'c = {c}')

d = {'a': 'foo', 'b': 25, 'c': 'qux'}
f(**d)
```

The items in the dictionary `d` are unpacked and passed to `f()` as keyword arguments. So, `f(**d)` is equivalent to `f(a='foo', b=25, c='qux')`:

```python
f(a='foo', b=25, c='qux')
```

In fact, we can see it in the code below

```python
f(**dict(a='foo', b=25, c='qux'))
```

```text
a = foo
b = 25
c = qux
```

1.6.4 Putting it all together

Think of `*args` as a variable-length positional argument list, and `**kwargs` as a variable-length keyword argument list.

All three—standard positional parameters, `*args`, and `**kwargs`—can be used in one Python function definition. If so, then they should be specified in that order:

```python
def f(a, b, *args, **kwargs):
    print(F'a = {a}')
    print(F'b = {b}')
    print(F'args = {args}')
    print(F'kwargs = {kwargs}')

f(1, 2, 'foo', 'bar', 'baz', 'qux', x=100, y=200, z=300)
```

```text
a = 1
b = 2
args = ('foo', 'bar', 'baz', 'qux')
kwargs = {'x': 100, 'y': 200, 'z': 300}
```

This provides just about as much flexibility as you could ever need in a function interface.

Chapter 2

Object Oriented Programming

Table of Contents

2.1 Introduction

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviours into individual objects. In this chapter, we'll focus the basics of object-oriented programming in Python.

Terminology invoking ”objects” in the modern sense of object-oriented programming made its first appearance at the artificial intelligence group at MIT in the late 1950s and early 1960s. ”Object” referred to LISP atoms with identified properties (attributes)

Conceptually, objects are like the components of a system. Think of a program as a factory assembly line of sorts. At each step of the assembly line, a system component processes some material, ultimately transforming raw material into a finished product.

An object contains data, like the raw or pre-processed materials at each step on an assembly line. In addition, the object contains behaviour, like the action that each assembly line component performs.

2.2 Defining Object Oriented Programming

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviours are bundled into individual objects.

> Up to know what we were writing is what is called **imperative programming**.

For example, an object could represent a person with properties like a name, age, and address and behaviours such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviours like adding attachments and sending.

Put another way, object-oriented programming is an approach for modelling concrete, real-world things, like cars, as well as relations between things, like companies and employees or students and teachers. OOP models real-world entities as software objects that have some data associated with them and can perform certain operations.

> Objects are at the centre of object-oriented programming in Python. In other programming paradigms, objects only represent the data. In OOP, they additionally inform the overall structure of the program.

2.3 Defining a Class

In Python, we define a class by using the `class` keyword followed by a name and a colon (`:`). Then we use `.__init__()` to declare which attributes each instance of the class we would like to have have:

```python
class Employee:
    def __init__(self, name, age):
        self.name =  name
        self.age = age
```

But what does all of that mean? And why do we even need classes in the first place? Lets' take a step back and consider using built-in, primitive data structures as an alternative.

Primitive data structures (numbers, strings, and lists) are designed to represent straightforward pieces of information, such as the cost of an apple, the name of a poem, or your favourite colours, respectively.

What if we want to represent something more complex?

For example, we might want to track employees in an organisation. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a list:

```python
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

There are a number of issues with this approach.

- it can make larger code files more difficult to manage. If we reference `kirk[0]` several lines away from where we declared the kirk list, will we remember that the element with index 0 is the employee's name?
- it can introduce errors if employees don't have the same number of elements in their respective lists. In the mccoy list above, the age is missing, so `mccoy[1]` will return "Chief Medical Officer" instead of Dr. McCoy's age.

A great way to make this type of code more manageable and more maintainable is to use classes.

2.4 Classes v. Instances

Classes allow we to create **user-defined data structures**. Classes define functions called methods, which identify the behaviours and actions that an object created from the class can perform with its data.

In this chapter, we'll create a Dog class that stores some information about the characteristics and behaviours that an individual dog can have.

A class is a **blueprint** for how to define something. It doesn't actually contain any data. The Dog class specifies that a name and an age are necessary for defining a dog, but it doesn't contain the name or age of any specific dog.

While the class is the blueprint, an instance is an object that's built from a class and contains real data. An instance of the Dog class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

a class is like a form or questionnaire. An instance is like a form that we've filled out with information. Just like many people can fill out the same form with their own unique information, we can create many instances from a single class.

2.5 Class Definition

We start all class definitions with the `class` keyword, then add the name of the class and a colon (`:`). Python will consider any code that we indent below the class definition as part of the class's body.

Here's an example of a Dog class:

```python
class Dog:
    pass
```

The body of the Dog class consists of a single statement:

the `pass` keyword.

Python programmers often use pass as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

Python class names are written in **CapitalizedWords** notation by convention. For example, a class for a specific breed of dog, like the Jack Russell Terrier, would be written as `JackRussellTerrier`.

The Dog class isn't very interesting right now, so we'll spruce it up a bit by defining some properties that all Dog objects should have. There are several properties that you can choose from, including:

- name,
- age,
- coat color,
- breed, ...

To keep the example small in scope, we'll just use **name** and **age**.

We define the properties all Dog objects must have in a method called `.__init__()`. Every time we create a new Dog object, `.__init__()` sets the initial state of the object by assigning the values of the object's properties.

`.__init__()` initializes each new instance of the class.

The Role of `.__init__()`

`.__init__()` doesn't initialize a class, it initializes an instance of a class or an object. Each dog has colour, but dogs as a class don't. Each dog has four or fewer feet, but the class of dogs doesn't. The class is a concept of an object. When you see Fido and Spot, you recognise their similarity, their doghood. That's the class.

The `__init__()` function is called a constructor, or initializer, and is automatically called when you create a new instance of a class. Within that function, the newly created object is assigned to the parameter `self`. The notation `self.name` is an attribute called name of the object in the variable self. Attributes are kind of like variables, but they describe the state of an object, or particular actions (functions) available to the object.

You can give `.__init__()` any number of parameters, but the first parameter will **always** be a variable called `self`. When you create a new class instance, then Python automatically passes the instance to the self parameter in `.__init__()` so that Python can define the new attributes on the object.

The role of `self`

The reason you need to use `self.` is because Python does not use special syntax to refer to instance attributes. Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on. That makes methods entirely the same as functions, and leaves the actual name to use up to the programmer (although self is the convention, and people will generally frown at you when you use something else.) self is not special to the code, it's just another object.

We update the Dog class with an `.__init__()` method that creates `.name` and `.age` attributes:

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Make sure that you indent the `.__init__()` method's signature by four spaces, and the body of the method by eight spaces.

This indentation is vitally important. It tells Python that the `.__init__()` method belongs to the Dog class.

In the body of `.__init__()`, there are two (2) statements using the `self` variable:

1. `self.name = name` creates an attribute called name and assigns the value of the name parameter to it.
2. `self.age = age` creates an attribute called age and assigns the value of the age parameter to it.

Attributes created in `.__init__()` are called **instance attributes**. An instance attribute's value is specific to a particular instance of the class. All Dog objects have a name and an age, but the values for the name and age attributes will vary depending on the Dog instance.

On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `.__init__()`.

For example, the following Dog class has a class attribute called species with the value *Canis familiaris*:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

You define class attributes directly beneath the first line of the class name and indent them by four spaces. You always need to assign them an initial value. When you create an instance of the class, then Python automatically creates and assigns class attributes to their initial values.

Use class attributes to define properties that should have the same value for every class instance. Use instance attributes for properties that vary from one instance to another.

Now that you have a Dog class, it's time to create some dogs!

Creating a Class with no Attributes 10 **Example**

Create a Vehicle class without any variables and methods.

Creating a Class with no Attributes **Solution**

Creating a Class with no Attributes

```python
class Vehicle:
    pass
```

2.6 To Instance a Class

Creating a new object from a class is called **instantiating a class**. You can create a new object by typing the name of the class, followed by opening and closing parentheses:

```python
class Dog:
    pass

print(Dog())
```

We first create a new Dog class with **NO** attributes or methods, and then we instantiate the Dog class to create a Dog object.

```text
<__main__.Dog object at 0x100dda240>
```

In the output above, you can see that you now have a new Dog object at `0x100dda240`. This funny-looking string of letters and numbers is a memory address that indicates where Python stores the Dog object in your computer's memory.

> The address on your screen will be different.

Now instantiate the Dog class a second time to create another Dog object:

```python
print(Dog())
```

```text
<__main__.Dog object at 0x100cdbe60>
```

The new Dog instance is located at a **different memory address**. That's because it's an entirely new instance and is completely unique from the first Dog object that we created.

To see this another way, type the following:

```python
a = Dog()
b = Dog()
print(a == b)
```

```text
False
```

In this code, we create two (2) new Dog objects and assign them to the variables a and b. When you compare a and b using the == operator, the result is False. Even though a and b are both instances of the Dog class, they represent two distinct objects in memory.

Example Creating a Class 11

Write a Python program to create a `Vehicle` class with `max_speed` and mileage instance attributes.

Solution Creating a Class

Creating a Class

```python
class Vehicle:
    def __init__(self, max_speed, mileage):
        self.max_speed = max_speed
        self.mileage = mileage

modelX = Vehicle(240, 18)
print(modelX.max_speed, modelX.mileage)
```

2.7 Class and Instance Attributes

```python
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

To instantiate this Dog class, we need to provide values for name and age.

If we don't do it, then Python raises a `TypeError`.

To pass arguments to the name and age parameters, put values into the parentheses after the class name:

```python
miles = Dog("Miles", 4)
buddy = Dog("Buddy", 9)
```

This creates two (2) new Dog instances:

1. four-year-old dog named Miles
2. nine-year-old dog named Buddy

The Dog class's `.__init__()` method has three (3) parameters, so why are you only passing two arguments to it in the example?

When you instantiate the Dog class, Python creates a new instance of Dog and passes it to the first parameter of `.__init__()`. This essentially removes the self parameter, so you only need to worry about the name and age parameters.

Behind the scenes, Python both creates and initializes a new object when you use this syntax.

After we create the Dog instances, you can access their instance attributes using dot (`.`) notation:

```python
print(miles.name)
print(miles.age)
print(buddy.name)
print(buddy.age)
```

```text
Miles
4
Buddy
9
```

You can access class attributes the same way:

```python
print(buddy.species)
```

```text
Canis familiaris
```

A great advantage of using classes to organise data is that instances are guaranteed to have the attributes you expect. All Dog instances have:

- `.species`
- `.name`
- `.age`

attributes, so you can use those attributes with confidence, knowing that they'll always return a value. Although the attributes are guaranteed to exist, their values can change dynamically:

```python
buddy.age = 10
print(buddy.age)

miles.species = "Felis silvestris"
print(miles.species)
```

```text
10
Felis silvestris
```

In this example, we changed the `.age` attribute of the buddy object to `10`. Then you change the `.species` attribute of the miles object to *Felis silvestris*, which is a species of cat.

The key takeaway here is that custom objects are **mutable by default**. An object is mutable if you can alter it dynamically. For example, lists and dictionaries are mutable, but strings and tuples are immutable.

2.7.1 Instance Methods

Instance methods are functions that you define inside a class and can only call on an instance of that class. Similar to `.__init__()`, an instance method always takes self as its first parameter. Let's start our code by typing in the following Dog class:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

This Dog class has two (2) instance methods:

1. `.description()` returns a string displaying the name and age of the dog.
2. `.speak()` has one parameter called sound and returns a string containing the dog's name and the sound that the dog makes.

Then we type the following to see our instance methods in action:

```python
miles = Dog("Miles", 4)

print(miles.description())
print(miles.speak("Woof Woof"))
print(miles.speak("Bow Wow"))
```

```text
Miles is 4 years old
Miles says Woof Woof
Miles says Bow Wow
```

In the above Dog class, `.description()` returns a string containing information about the Dog instance miles. When writing your own classes, it's a good idea to have a method that returns a string containing useful information about an instance of the class.

However, `.description()` isn't the most Pythonic way of doing this.
When you create a list object, you can use `print()` to display a string that looks like the list:

```python
names = ["Miles", "Buddy", "Jack"]
print(names)
```

```text
['Miles', 'Buddy', 'Jack']
```

Go ahead and print the miles object to see what output you get:

```python
print(miles)
```

```text
<__main__.Dog object at 0x100cdbe60>
```

When you print miles, you get a cryptic-looking message telling you that miles is a Dog object at the memory address 0x00aeff70. This message isn't very helpful. You can change what gets printed by defining a special instance method called `.__str()__`. Let's change the name of the Dog class's `.description()` method to `.__str()__`:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
         return f"{self.name} says {sound}"
```

Now, when we print miles, we get a much friendlier output:

```python
miles = Dog("Miles", 4)
print(miles)
```

```text
Miles is 4 years old
```

Methods like `.__init__()` and `.__str__()` are called **dunder methods** as they begin and end with double underscores. There are many dunder methods that you can use to customise classes in Python. Understanding dunder methods is an important part of mastering object-oriented programming in Python, but for our first exploration of the topic, you'll stick with these two dunder methods.

2.8 Inheritance

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called child classes, and the classes that you derive child classes from are called parent classes.

We inherit from a parent class by creating a new class and putting the name of the parent class into parentheses:

```python
class Parent:
    hair_color = "brown"

class Child(Parent):
    pass
```

In this minimal example, the child class `Child` inherits from the parent class Parent. Because child classes take on the attributes and methods of parent classes, `Child.hair_color` is also brown without your explicitly defining that.

Child classes can override or extend the attributes and methods of parent classes. In other words:

> Child classes inherit all of the parent's attributes and methods but can also specify attributes and methods that are unique to themselves.

Although the analogy isn't perfect, you can think of object inheritance sort of like genetic inheritance.

You may have inherited your hair color from your parents. It's an attribute that you were born with. But maybe you decide to color your hair purple. Assuming that your parents don't have purple hair, you've just overridden the hair color attribute that you inherited from your parents:

```python
class Parent:
    hair_color = "brown"

class Child(Parent):
    hair_color = "purple"
```

If you change the code example like this, then `Child.hair_color` will be "purple". You also inherit, in a sense, your language from your parents. If your parents speak English, then you'll also speak English. Now imagine you decide to learn a second language, like German. In this case, you've extended your attributes because you've added an attribute that your parents don't have:

```python
class Parent:
    speaks = ["English"]

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.speaks.append("German")
```

You'll learn more about how the code above works in the sections below. But before you dive deeper into inheritance in Python, we'll take a walk to a dog park to better understand why we might want to use inheritance in our own code.

The `super()` Function

Used to refer to the parent class or superclass. It allows you to call methods defined in the superclass from the subclass, enabling you to extend and customize the functionality inherited from the parent class.

Example Class Inheritance 12

Create a Bus class that inherits from the Vehicle class. Give the capacity argument of `Bus.seating_capacity()` a default value of `50`.
Use the following code for your parent Vehicle class.

```python
class Vehicle:
    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

    def seating_capacity(self, capacity):
        return f"The seating capacity of a {self.name} is {capacity} passengers"
```

Solution Class Inheritance

Class Inheritance

```python
class Vehicle:
    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

    def seating_capacity(self, capacity):
        return f"The seating capacity of a {self.name} is {capacity} passengers"
```

2.9 Parent Classes v. Child Classes

In this section, we'll create a child class for each of the three (3) breeds mentioned above:

- Jack Russell terrier,
- dachshund,
- and bulldog.

For reference, here's the full definition of the Dog class that we're currently working with:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

To create a child class, you create a new class with its own name and then put the name of the parent class in parentheses. Add the following lines to create three (3) new child classes of the Dog class:

```python
class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

With the child classes defined, you can now create some dogs of specific breeds:

```python
miles = JackRussellTerrier("Miles", 4)
buddy = Dachshund("Buddy", 9)
jack = Bulldog("Jack", 3)
jim = Bulldog("Jim", 5)
```

Instances of child classes inherit all of the attributes and methods of the parent class:

```python
miles.species
buddy.name
print(jack)
jim.speak("Woof")
```

To determine which class a given object belongs to, you can use the built-in `type()`:

```python
type(miles)
```

What if you want to determine if miles is also an instance of the Dog class? You can do this with the built-in `isinstance()`:

```python
print(isinstance(miles, Dog))
```

Notice that `isinstance()` takes two (2) arguments, an object and a class. In the example above, isinstance() checks if miles is an instance of the Dog class and returns True.

```text
True
```

The miles, buddy, jack, and jim objects are all Dog instances, but miles isn't a Bulldog instance, and jack isn't a Dachshund instance:

```python
print(isinstance(miles, Bulldog))
print(isinstance(jack, Dachshund))
```

More generally, all objects created from a child class are instances of the parent class, although they may not be instances of other child classes.

Now that you've created child classes for some different breeds of dogs, we can give each breed its own sound.

Example Creating a Child Class 13

Create a child class Bus that will inherit all of the variables and methods of the Vehicle class. Use the following code as example.

```python
class Vehicle:

    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage
```

Create a Bus object that will inherit all of the variables and methods of the parent Vehicle class and display it.
`Vehicle Name: School Volvo Speed: 180 Mileage: 12`

Creating a Child Class **Solution**

Creating a Child Class

```python
class Vehicle:

    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

class Bus(Vehicle):
    pass

School_bus = Bus("School Volvo", 180, 12)
print("Vehicle Name:", School_bus.name, "Speed:", School_bus.max_speed, "Mileage:",
↪   School_bus.mileage)
```

2.9.1 Extending Parent Class Functionality

As different breeds of dogs have slightly different barks, we want to provide a default value for the sound argument of their respective `.speak()` methods. To do this, you want to override `.speak()` in the class definition for each breed.

To override a method defined on the parent class, we define a method with the same name on the child class. Here's what that looks like for the JackRussellTerrier class:

```python
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return f"{self.name} says {sound}"
```

Now `.speak()` is defined on the JackRussellTerrier class with the default argument for sound set to `"Arf"`. Let's update our previous code with the new `JackRussellTerrier` class. You can now call `.speak()` on a `JackRussellTerrier` instance without passing an argument to sound:

```python
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Sometimes dogs make **different** noises, so if Miles gets angry and growls, you can still call .speak() with a different sound:

```
miles.speak("Grrr")
```

One thing to keep in mind about class inheritance is that changes to the parent class automatically propagate to child classes. This occurs as long as the attribute or method being changed isn't overridden in the child class.
For example, in the editor window, change the string returned by `.speak()` in the Dog class

```
# dog.py

class Dog:
    # ...

    def speak(self, sound):
        return f"{self.name} barks: {sound}"

# ...
```

Save the file and press F5. Now, when you create a new Bulldog instance named jim, jim.speak() returns the new string:

```
jim = Bulldog("Jim", 5)
jim.speak("Woof")
```

However, calling `.speak()` on a JackRussellTerrier instance won't show the new style of output:

```
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Sometimes it makes sense to completely override a method from a parent class. But in this case, you don't want the JackRussellTerrier class to lose any changes that you might make to the formatting of the Dog.speak() output string.
To do this, you still need to define a `.speak()` method on the child JackRussellTerrier class. But instead of explicitly defining the output string, you need to call the Dog class's .speak() from inside the child class's `.speak()` using the same arguments that you passed to JackRussellTerrier.speak().

You can access the parent class from inside a method of a child class by using `super()`:

```python
# dog.py

# ...

class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return super().speak(sound)

# ...
```

When you call super().speak(sound) inside JackRussellTerrier, Python searches the parent class, Dog, for a `.speak()` method and calls it with the variable sound. Update dog.py with the new JackRussellTerrier class. Save the file and press F5 so you can test it in the interactive window:

```python
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Now when you call miles.speak(), you'll see output reflecting the new formatting in the Dog class.

> In the above examples, the class hierarchy is very straightforward. The Jack-RussellTerrier class has a single parent class, Dog. In real-world examples, the class hierarchy can get quite complicated.

2.10 The classmethod() function

The classmethod() is an inbuilt function in Python, which returns a class method for a given function. This means that `classmethod()` is a built-in Python function that transforms a regular method into a class method.

When a method is defined using the `@classmethod` decorator (which internally calls classmethod()), the method is **bound to the class and not to an instance of the class**. As a result, the method receives the class (`cls`) as its first argument, rather than an instance (`self`).

2.10.1 Class v. Static Method

There are some differences between a class method and a static method which is worth mention:

- A class method takes class as the first parameter while a static method needs no specific parameters.
- A class method can access or modify the class state while a static method can't access or modify it.
- In general, static methods know nothing about the class state. They are utility-type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as a parameter.
- We use @classmethod decorator in Python to create a class method and we use @staticmethod decorator to create a static method in Python.

Example An Example of @classmethod 14

Create a simple classmethod

In this example, we are going to see how to create a class method in Python. For this, we created a class named "Geeks" with a member variable "course" and created a function named "purchase" which prints the object. Now, we passed the method Geeks.purchase into a class method using the @classmethod decorator, which converts the method to a class method. With the class method in place, we can call the function "purchase" without creating a function object, directly using the class name "Geeks."

```python
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

    def seating_capacity(self, capacity):
        return f"The seating capacity of a {self.name} is {capacity} passengers"

class Bus(Vehicle):
    # assign default value to capacity
    def seating_capacity(self, capacity=50):
        return super().seating_capacity(capacity=50)

School_bus = Bus("School Volvo", 180, 12)
print(School_bus.seating_capacity())

#+end_src

** Practicals

#+NAME: PR-1
#+begin_src python :session :results output
class Geeks:
    course = 'DSA'
    list_of_instances = []

    def __init__(self, name):
        self.name = name
        Geeks.list_of_instances.append(self)

    @classmethod
```

An Example of @classmethod **Solution**

An Example of @classmethod

Part II.

Python For Scientific Applications

Chapter 3

Numpy

Table of Contents

Figure 3.1.: The logo of numpy.

3.1 Introduction

NumPy is a Python library that provides a simple yet powerful data structure:

the n-dimensional array.

This is the foundation on which almost all the power of Python's data science toolkit is built, and learning NumPy is the first step on any Python data scientist's journey. This tutorial will provide you with the knowledge you need to use NumPy and the higher-level libraries that rely on it.

3.2 Why use Numpy

Since you already know Python, you may be asking yourself if you really have to learn a whole new paradigm to do data science.
Here are the top four benefits that NumPy can bring to your code:

- **More speed:** Algorithms written in C that complete in nanoseconds rather than seconds.
- **Fewer loops:** Reduces loops and keep from getting tangled up in iteration indices.
- **Clearer code:** Without loops, your code will look more like the equations you're trying to calculate.
- **Better quality:** Thousands of contributors working to keep NumPy fast, friendly, and bug free.

Because of these benefits, NumPy is the de facto[1] standard for multidimensional arrays in Python data science, and many of the most popular libraries are built on top of it.

[1]De facto means it is an unofficial standard, while *de jure* means it is a standard with a legal backing

Learning NumPy is a great way to set down a solid foundation as we expand our knowledge into more specific areas of data science.

3.3 An Introductionary Example - Calculating Grades

This first example introduces a few core concepts in NumPy that we'll use throughout the rest of the tutorial:

- Creating arrays using `numpy.array()`
- Treating complete arrays like individual values to make vectorised calculations more readable
- Using built-in NumPy functions to modify and aggregate the data

These concepts are the core of using NumPy effectively.
Imagine a scenario: You're a teacher who has just graded your students on a recent test. Unfortunately, you may have made the test too challenging, and most of the

students did worse than expected. To help everybody out, you're going to curve everyone's grades[2] .

[2]Grading on a curve is a term that describes a variety of different methods that a teacher uses to adjust the scores their students received on a test in some way. Most of the time, grading on a curve boosts the students' grades by moving their actual scores up a few notches, perhaps increasing the letter grade

It'll be a relatively rudimentary curve, though. You'll take whatever the average score is and declare that a **C**. Additionally, you'll make sure that the curve doesn't accidentally hurt your students' grades or help so much that the student does better than 100%.

python

```
1  import numpy as np
2  CURVE_CENTER = 80
3  grades = np.array([72, 35, 64, 88, 51, 90, 74, 12])
4  def curve(grades):
5      average = grades.mean()
6      change = CURVE_CENTER - average
7      new_grades = grades + change
8      return np.clip(new_grades, grades, 100)
9
10 curve(grades)
```

The original scores have been increased based on where they were in the pack, but none of them were pushed over 100%.

Here are the important highlights:

- Line 1 imports NumPy using the `np` alias, which is a common convention[3] that saves you a few keystrokes.
- Line 3 creates your first NumPy array, which is one-dimensional and has a shape of (8,) and a data type of `int64`. Don't worry too much about these details yet. You'll explore them in more detail later in the tutorial.
- Line 5 takes the average of all the scores using `.mean()`. Arrays have a lot of methods.

[3]It is a convention and not a rule, but all documentations use np so you might as well

On line 7, you take advantage of two (2) important concepts at once:

1. **Vectorization:** process of performing the same operation in the same way for each element in an array. This removes for loops from your code but achieves the same result.
2. **Broadcasting:** process of extending two arrays of different shapes and figuring out how to perform a vectorized calculation between them

Remember, grades is an array of numbers of shape (8,) and change is a scalar, or single number, essentially with shape (1,). In this case, NumPy adds the scalar to each item in the array and returns a new array with the results.

Finally, on line 8, you limit, or clip, the values to a set of minimums and maximums. In addition to array methods, NumPy also has a large number of built-in functions. You don't need to memorize them all—that's what documentation is for. Anytime you get stuck or feel like there should be an easier way to do something, take a peek at the documentation and see if there isn't already a routine that does exactly what you need.

In this case, you need a function that takes an array and makes sure the values don't exceed a given minimum or maximum. `clip()` does exactly that.
Line 8 also provides another example of broadcasting. For the second argument to `clip()`, you pass grades, ensuring that each newly curved grade doesn't go lower than the original grade. But for the third argument, you pass a single value: 100.

NumPy takes that value and broadcasts it against every element in `new_grades`, ensuring that none of the newly curved grades exceeds a perfect score.

3.4 Getting Into Shape: Array Shapes and Axes

Now that you've seen some of what NumPy can do, it's time to firm up that foundation with some important theory. There are a few concepts that are important to keep in mind, especially as you work with arrays in higher dimensions.
Vectors, which are one-dimensional arrays of numbers, are the least complicated to keep track of. Two dimensions aren't too bad, either, because they're similar to spreadsheets.

But things start to get tricky at three dimensions, and visualizing four? At this point geometrical thinking of arrays becomes unfeasible and its better to think them **ONLY** as data points.

3.4.1 Understanding Shapes

Shape is an important concept when you're using multidimensional arrays. At a certain point, it's easier to forget about visualizing the shape of your data and to instead follow some mental rules and trust NumPy to tell you the correct shape.

All arrays have a property called `.shape` which returns a tuple of the size in each dimension. It's less important which dimension is which, but it's critical that the arrays you pass to functions are in the shape that the functions expect. A common way to confirm that your data has the proper shape is to print the data and its shape until you're sure everything is working like you expect.

This next example will show this process. You'll create an array with a complex shape, check it, and reorder it to look like it's supposed to:

```python
import numpy as np

temperatures = np.array([
    29.3, 42.1, 18.8, 16.1, 38.0, 12.5,
    12.6, 49.9, 38.6, 31.3, 9.2, 22.2
]).reshape(2, 2, 3)

print(temperatures.shape)
print(temperatures)
print(np.swapaxes(temperatures, 1, 2))
```

```text
(2, 2, 3)

[[[29.3 42.1 18.8]
  [16.1 38.  12.5]]
 [[12.6 49.9 38.6]
  [31.3  9.2 22.2]]]

[[[29.3 16.1]
  [42.1 38. ]
  [18.8 12.5]]
 [[12.6 31.3]
  [49.9  9.2]
  [38.6 22.2]]]
```

Here, you use a `numpy.ndarray` method called `.reshape()` to form a 2-by-2-by-3 block of data. When you check the shape of your array in input 3, it's exactly what you told it to be. However, you can see how printed arrays quickly become hard to visualize in three or more dimensions. After you swap axes with `.swapaxes()`, it becomes little clearer which dimension is which. You'll see more about axes in the next section.

Shape will come up again in the section on broadcasting. For now, just keep in mind that these little checks don't cost anything. You can always delete the cells or get rid of the code once things are running smoothly.

3.4.2 Understanding Axes

The example above shows how important it is to know not only what shape your data is in but also which data is in which axis. In NumPy arrays, axes are zero-indexed and identify which dimension is which. For example, a two-dimensional array has a vertical axis (axis 0) and a horizontal axis (axis 1).

Lots of functions and commands in NumPy change their behavior based on which axis you tell them to process.x

This example will show how .max() behaves by default, with no axis argument, and how it changes functionality depending on which axis you specify when you do supply an argument:

```python
import numpy as np

table = np.array([
    [5, 3, 7, 1],
    [2, 6, 7 ,9],
    [1, 1, 1, 1],
    [4, 3, 2, 0],
])

print(table.max())

print(table.max(axis=0))

print(table.max(axis=1))
```

```text
9
[5 6 7 9]
[7 9 1 4]
```

By default, `.max()` returns the largest value in the entire array, no matter how many dimensions there are. However, once you specify an axis, it performs that calculation for each set of values along that **particular axis**.

For example, with an argument of `axis=0`, `.max()` selects the maximum value in each of the four vertical sets of values in table and returns an array that has been flattened, or aggregated into a one-dimensional array.
In fact, many of NumPy's functions behave this way:

> If no axis is specified, then they perform an operation on the entire dataset. Otherwise, they perform the operation in an axis-wise fashion.

3.4.3 Broadcasting

So far, you've seen a couple of smaller examples of broadcasting, but the topic will start to make more sense the more examples you see. Fundamentally, it functions around one (1) rule:

> arrays can be broadcast against each other if their dimensions match or if one of the arrays has a size of 1.

If the arrays match in size along an axis, then elements will be operated on element-by-element, similar to how the built-in Python function `zip()`[4] works.

[4]an iterator that will aggregate elements from two or more iterables. You can use the resulting iterator to quickly and consistently solve common programming problems, like creating dictionaries.

If one of the arrays has a size of 1 in an axis, then that value will be broadcast along that axis, or duplicated as many times as necessary to match the number of elements along that axis in the other array.

Here's a quick example. Array `A` has the shape `(4, 1, 8)`, and array `B` has the shape `(1, 6, 8)`. Based on the rules above, you can operate on these arrays together:

- In axis 0, A has a 4 and B has a 1, so B can be broadcast along that axis.
- In axis 1, A has a 1 and B has a 6, so A can be broadcast along that axis.
- In axis 2, the two arrays have matching sizes, so they can operate successfully.

All three (3) axes successfully follow the rule. You can set up the arrays like this:

```python
import numpy as np

A = np.arange(32).reshape(4, 1, 8)

print(A)

B = np.arange(48).reshape(1, 6, 8)

print(B)
```

```text
[[[ 0  1  2  3  4  5  6  7]]
 [[ 8  9 10 11 12 13 14 15]]
 [[16 17 18 19 20 21 22 23]]
 [[24 25 26 27 28 29 30 31]]]
[[[ 0  1  2  3  4  5  6  7]
  [ 8  9 10 11 12 13 14 15]
  [16 17 18 19 20 21 22 23]
  [24 25 26 27 28 29 30 31]
  [32 33 34 35 36 37 38 39]
  [40 41 42 43 44 45 46 47]]]
```

A has 4 planes, each with 1 row and 8 columns. B has only 1 plane with 6 rows and 8 columns. Watch what NumPy does for you when you try to do a calculation between them.

Add the two (2) arrays together:

```python
print(A + B)
```

```text
[[[ 0  2  4  6  8 10 12 14]
  [ 8 10 12 14 16 18 20 22]
  [16 18 20 22 24 26 28 30]
  [24 26 28 30 32 34 36 38]
  [32 34 36 38 40 42 44 46]
  [40 42 44 46 48 50 52 54]]
 [[ 8 10 12 14 16 18 20 22]
  [16 18 20 22 24 26 28 30]
  [24 26 28 30 32 34 36 38]
  [32 34 36 38 40 42 44 46]
  [40 42 44 46 48 50 52 54]
  [48 50 52 54 56 58 60 62]]
 [[16 18 20 22 24 26 28 30]
  [24 26 28 30 32 34 36 38]
  [32 34 36 38 40 42 44 46]
  [40 42 44 46 48 50 52 54]
  [48 50 52 54 56 58 60 62]
  [56 58 60 62 64 66 68 70]]
 [[24 26 28 30 32 34 36 38]
  [32 34 36 38 40 42 44 46]
  [40 42 44 46 48 50 52 54]
  [48 50 52 54 56 58 60 62]
  [56 58 60 62 64 66 68 70]
  [64 66 68 70 72 74 76 78]]]
```

The way broadcasting works is that NumPy duplicates the plane in B three times so that you have a total of four, matching the number of planes in A. It also duplicates the single row in A five times for a total of six, matching the number of rows in B. Then it adds each element in the newly expanded A array to its counterpart in the same location in B. The result of each calculation shows up in the corresponding location of the output[5] .

[5] It also provides a means of vectorizing array operations so that looping occurs in C instead of Python

> This is a good way to create an array from a range using arange()!

Once again, even though you can use words like "plane," "row," and "column" to describe how the shapes in this example are broadcast to create matching three-dimensional shapes, things get more complicated at higher dimensions. A lot of times, you'll have to simply follow the broadcasting rules and do lots of print-outs to make sure things

are working as planned.
Understanding broadcasting is an important part of mastering vectorized calculations, and vectorized calculations are the way to write clean, idiomatic NumPy code.

3.5 Data Science Operations: Filter, Order, Aggregate

That wraps up a section that was heavy in theory but a little light on practical, real-world examples. In this section, you'll work through some examples of real, useful data science operations:

filtering, sorting, and aggregating data.

3.5.1 Indexing

Indexing uses many of the same idioms that normal Python code uses. You can use positive or negative indices to index from the front or back of the array. You can use a colon (`:`) to specify "the rest" or "all," and you can even use two colons to skip elements as with regular Python lists.

Here's the difference: NumPy arrays use commas between axes, so you can index multiple axes in one set of square brackets. An example is the easiest way to show this off. It's time to confirm Dürer's magic square!

Dürer's magic square

Dürer's magic square is a magic square with magic constant 34 used in an engraving entitled *Melencolia I* by *Albrecht Dürer*. The engraving shows a disorganized jumble of scientific equipment lying unused while an intellectual sits absorbed in thought.

Dürer's magic square is located in the upper right-hand corner of the engraving. The numbers 15 and 14 appear in the middle of the bottom row, indicating the date of the engraving, 1514.

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

If you add up any of the rows, columns, or diagonals, then you'll get the same number, 34. That's also what you'll get if you add up each of the four quadrants, the center four squares, the four corner squares, or the four corner squares of any of the contained 3-by-3 grids.

Let's prove this statement:

```python
import numpy as np

square = np.array([
    [16, 3, 2, 13],
    [5, 10, 11, 8],
    [9, 6, 7, 12],
    [4, 15, 14, 1]
])

for i in range(4):
    assert square[:, i].sum() == 34
    assert square[i, :].sum() == 34


assert square[:2, :2].sum() == 34
assert square[2:, :2].sum() == 34
assert square[:2, 2:].sum() == 34
assert square[2:, 2:].sum() == 34
```

Inside the `for` loop, you verify that all the rows and all the columns add up to 34. After that, using selective indexing, you verify that each of the quadrants also adds up to 34.

The keyword assert

The assert statement exists in almost every programming language. It has two (2) main uses:

- It helps detect problems early in your program, where the cause is clear, rather than later when some other operation fails. A type error in Python, for example, can go through several layers of code before actually raising an Exception if not caught early on.
- It works as documentation for other developers reading the code, who see the assert and can confidently say that its condition holds from now on.

One last thing to note is that you're able to take the sum of any array to add up all of its elements globally with `square.sum()`. This method can also take an axis argument to do an axis-wise summing instead.

3.6 Masking and Filtering

Index-based selection is great, but what if you want to filter your data based on more complicated non-uniform or non-sequential criteria? This is where the concept of a mask comes into play.

A mask is an array that has the exact same shape as your data, but instead of your values, it holds Boolean values: either `True` or `False`. You can use this mask array to index into your data array in nonlinear and complex ways. It will return all of the elements where the Boolean array has a `True` value.

Here's an example showing the process, first in slow motion and then how it's typically done, all in one line:

```python
1  import numpy as np
2
3  numbers = np.linspace(5, 50, 24, dtype=int).reshape(4, -1)
4
5  print(numbers)
6
7  mask = numbers % 4 == 0
8
9  print(mask)
10
11 numbers[mask]
12
13 by_four = numbers[numbers % 4 == 0]
14
15 print(by_four)
```

```text
[[ 5  6  8 10 12 14]
 [16 18 20 22 24 26]
 [28 30 32 34 36 38]
 [40 42 44 46 48 50]]

[[False False  True False  True False]
 [ True False  True False  True False]
 [ True False  True False  True False]
 [ True False  True False  True False]]

 [ 8 12 16 20 24 28 32 36 40 44 48]
```

You'll see an explanation of the new array creation in line 3 in a moment, but for now, focus on the meat of the example. These are the important parts:

- Line 7 creates the mask by performing a vectorized Boolean computation, taking each element and checking to see if it divides evenly by four. This returns a mask array of the same shape with the element-wise results of the computation.

- Line 13 uses this mask to index into the original numbers array. This causes the array to lose its original shape, reducing it to one dimension, but you still get the data you're looking for.
- Line 13 provides a more traditional, idiomatic masked selection that you might see in the wild, with an anonymous filtering array created inline, inside the selection brackets.

> This syntax is similar to usage in the R programming language. A language used predominantly in data science.

Coming back to line 3, you encounter three (3) new concepts:

- Using `np.linspace()` to generate an evenly spaced array
- Setting the `dtype` of an output
- Reshaping an array with `-1`

np.linspace() generates n numbers evenly distributed between a minimum and a maximum, which is useful for evenly distributed sampling in scientific plotting.

Because of the particular calculation in this example, it makes life easier to have integers in the numbers array. But because the space between 5 and 50 doesn't divide evenly by 24, the resulting numbers would be floating-point numbers. You specify a dtype of int to force the function to round down and give you whole integers. You'll see a more detailed discussion of data types later on.

Finally, array.reshape() can take -1 as one of its dimension sizes. That signifies that NumPy should just figure out how big that particular axis needs to be based on the size of the other axes. In this case, with 24 values and a size of 4 in axis 0, axis 1 ends up with a size of 6.

Example A Simple Filter Exercise 15

Write a NumPy program to extract all numbers from a given array less and greater than a specified number.

Solution A Simple Filter Exercise

```python
#+begin_src python :session :results output
# Importing the NumPy library with an alias 'np'
import numpy as np

# Creating a NumPy array 'nums' containing values in a 3x3 matrix
```

```python
nums = np.array([[5.54, 3.38, 7.99],
                 [3.54, 4.38, 6.99],
                 [1.54, 2.39, 9.29]])

# Printing a message indicating the original array
print("Original array:")
print(nums)

# Assigning value 5 to the variable 'n' and printing elements greater than 'n' in
↪ the array
n = 5
print("\nElements of the said array greater than", n)
print(nums[nums > n])

# Assigning value 6 to the variable 'n' and printing elements less than 'n' in the
↪ array
n = 6
print("\nElements of the said array less than", n)
```

```text
#+begin_example
Original array:
[[5.54 3.38 7.99]
 [3.54 4.38 6.99]
 [1.54 2.39 9.29]]
Elements of the said array greater than 5
[5.54 7.99 6.99 9.29]
Elements of the said array less than 6
```

Here's one more example to show off the power of masked filtering. The normal distribution is a probability distribution in which roughly 95.45% of values occur within two standard deviations of the mean.

You can verify that with a little help from NumPy's random module for generating random values:

```python
import numpy as np

from numpy.random import default_rng

rng = default_rng()

values = rng.standard_normal(10000)

```

```python
print(values[:5])

std = values.std()

print(std)

filtered = values[(values > -2 * std) & (values < 2 * std)]

print(filtered.size)

print(values.size)

filtered.size / values.size
```

```text
[-0.7239233   1.71462297 -1.21091617  0.22540724  0.89326428]
1.005225298547061
9545
10000
```

Here you use a potentially strange-looking syntax to combine filter conditions: a binary & operator. Why would that be the case? It's because NumPy designates & and | as the vectorized, element-wise operators to combine Booleans. If you try to do A and B, then you'll get a warning about how the truth value for an array is weird, because the and is operating on the truth value of the whole array, not element by element.

Example Creating a Numpy Array 16

Write a NumPy program to create an array of integers from 30 to 70.

Solution Creating a Numpy Array

```python
# Importing the NumPy library with an alias 'np'
import numpy as np

# Creating an array of integers from 30 to 70 using np.arange()
array = np.arange(30, 71)

# Printing a message indicating an array of integers from 30 to 70
print("Array of the integers from 30 to 70")

# Printing the array of integers from 30 to 70
print(array)
```

```text
Array of the integers from 30 to 70

```

```
[30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70]
```

An Identity Matrix 17 **Example**

Write a 3-by-3 identity matrix.

An Identity Matrix **Solution**

```python
# Importing the NumPy library with an alias 'np'
import numpy as np

# Creating a 3x3 identity matrix using np.identity()
array_2D = np.identity(3)

# Printing a message indicating a 3x3 matrix
print('3x3 matrix:')

# Printing the 3x3 identity matrix
print(array_2D)

```

```text
3x3 matrix:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

A Random Value 18 **Example**

Write a NumPy program to generate a random number between 0 and 1.

A Random Value **Solution**

```python
# Importing the NumPy library with an alias 'np'
import numpy as np

# Generating a random number from a normal distribution with mean 0 and standard
↪  deviation 1 using np.random.normal()
rand_num = np.random.normal(0, 1, 1)

# Printing a message indicating a random number between 0 and 1
print("Random number between 0 and 1:")
```

```python

# Printing the generated random number
print(rand_num)

```

```text
Random number between 0 and 1:
[0.805393]
```

Example A Random Array 19

Write a NumPy program to generate an array of 15 random numbers from a standard normal distribution.

Solution A Random Array

```python
# Importing the NumPy library with an alias 'np'
import numpy as np

# Generating an array of 15 random numbers from a standard normal distribution
↪  using np.random.normal()
rand_num = np.random.normal(0, 1, 15)

# Printing a message indicating 15 random numbers from a standard normal
↪  distribution
print("15 random numbers from a standard normal distribution:")

# Printing the array of 15 random numbers
print(rand_num)

```

```text
15 random numbers from a standard normal distribution:

[ 0.83730584  0.0962134   0.77050723  1.03140118 -0.67267708  1.84883332
  0.70835831 -2.30154701 -0.00770623 -0.7585212  -1.3338401  -0.47316322
  0.20031869 -0.63787389  1.25788111]
```

3.7 Transposing, Sorting, and Concatenating

Other manipulations, while not quite as common as indexing or filtering, can also be very handy depending on the situation you’re in. You’ll see a few examples in this section.

Here's transposing an array:

```python
import numpy as np

a = np.array([
    [1, 2],
    [3, 4],
    [5, 6],
])

print(a.T)

print(a.transpose())
```

```text
[[1 3 5]
 [2 4 6]]

[[1 3 5]
 [2 4 6]]
```

When you calculate the transpose of an array, the row and column indices of every element are switched. Item [0, 2], for example, becomes item [2, 0]. You can also use a.T as an alias for a.transpose().

The following code block shows sorting, but you'll also see a more powerful sorting technique in the coming section on structured data:

```python
import numpy as np

data = np.array([
    [7, 1, 4],
    [8, 6, 5],
    [1, 2, 3]
])

print(np.sort(data))

print(np.sort(data, axis=None))

print(np.sort(data, axis=0))
```

```text
[[1 4 7]
 [5 6 8]
 [1 2 3]]
[1 1 2 3 4 5 6 7 8]
[[1 1 3]
 [7 2 4]
 [8 6 5]]
```

Omitting the axis argument automatically selects the last and innermost dimension, which is the rows in this example. Using None flattens the array and performs a global sort. Otherwise, you can specify which axis you want. In output 5, each column of the array still has all of its elements but they have been sorted low-to-high inside that column.

Finally, here's an example of concatenation. While there's a np.concatenate() function, there are also a number of helper functions that are sometimes easier to read.
Here are some examples:

```python
import numpy as np

a = np.array([
    [4, 8],
    [6, 1]
])

b = np.array([
    [3, 5],
    [7, 2],
])

print(np.hstack((a, b)))

print(np.vstack((b, a)))

print(np.concatenate((a, b)))

print(np.concatenate((a, b), axis=None))
```

```text
[[4 8 3 5]
 [6 1 7 2]]
[[3 5]
 [7 2]
 [4 8]
 [6 1]]
```

```
[[4 8]
 [6 1]
 [3 5]
 [7 2]]
[4 8 6 1 3 5 7 2]
```

Inputs 4 and 5 show the slightly more intuitive functions hstack() and vstack(). Inputs 6 and 7 show the more generic concatenate(), first without an axis argument and then with axis=None. This flattening behavior is similar in form to what you just saw with sort().
One important stumbling block to note is that all these functions take a tuple of arrays as their first argument rather than a variable number of arguments as you might expect. You can tell because there's an extra pair of parentheses.

Sum of an Array 20 **Example**

Write a NumPy program to compute the sum of all elements, the sum of each column and the sum of each row in a given array.

Sum of an Array **Solution**

```python
# Importing the NumPy library with an alias 'np'
import numpy as np

# Creating a NumPy array 'x' with a 2x2 shape
x = np.array([[0, 1], [2, 3]])

# Printing a message indicating the original array 'x'
print("Original array:")
print(x)

# Calculating and printing the sum of all elements in the array 'x' using np.sum()
print("Sum of all elements:")
print(np.sum(x))

# Calculating and printing the sum of each column in the array 'x' using np.sum()
↪  with axis=0
print("Sum of each column:")
print(np.sum(x, axis=0))

# Calculating and printing the sum of each row in the array 'x' using np.sum() with
↪  axis=1
print("Sum of each row:")
print(np.sum(x, axis=1))

```

```text
Original array:
[[0 1]
 [2 3]]
Sum of all elements:
6
Sum of each column:
[2 4]
Sum of each row:
```

3.8 Aggregating

Your last stop on this tour of functionality before diving into some more advanced topics and examples is aggregation. You've already seen quite a few aggregating methods, including .sum(), .max(), .mean(), and .std(). You can reference NumPy's larger library of functions to see more. Many of the mathematical, financial, and statistical functions use aggregation to help you reduce the number of dimensions in your data.

3.9 Practical Exercise Implementing Maclauring Series

Now it's time to see a realistic use case for the skills introduced in the sections above: implementing an equation.

One of the hardest things about converting mathematical equations to code without NumPy is that many of the visual similarities are missing, which makes it hard to tell what portion of the equation you're looking at as you read the code. Summations are converted to more verbose for loops, and limit optimizations end up looking like while loops.

Using NumPy allows you to keep closer to a one-to-one representation from equation to code.

In this next example, you'll encode the Maclaurin series[6] for ex. Maclaurin series are a way of approximating more complicated functions with an infinite series of summed terms centered about zero.

[6]Computers actually use infinite sums like these to approximate functions like sin, cos

For e^{x}, the Maclaurin series is the following summation:

$$e^{\mathrm{x}} = \sum_{n=0}^{\infty} = \frac{x^{\mathrm{n}}}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$$

You add up terms starting at zero and going theoretically to infinity. Each nth term will be x raised to n and divided by n!, which is the notation for the factorial operation.

Now it's time for you to put that into NumPy code.

```python
from math import e, factorial

import numpy as np

fac = np.vectorize(factorial)

def e_x(x, terms=10):
    """Approximates e^x using a given number of terms of
    the Maclaurin series
    """
    n = np.arange(terms)
    return np.sum((x ** n) / fac(n))

if __name__ == "__main__":
    print("Actual:", e ** 3)  # Using e from the standard library

    print("N (terms)\tMaclaurin\tError")

    for n in range(1, 14):
        maclaurin = e_x(3, terms=n)
        print(f"{n}\t\t{maclaurin:.03f}\t\t{e**3 - maclaurin:.03f}")
```

When you run this, you should see the following result:

As you increase the number of terms, your Maclaurin value gets closer and closer to the actual value, and your error shrinks smaller and smaller.

The calculation of each term involves taking x to the n power and dividing by n!, or the factorial of n. Adding, summing, and raising to powers are all operations that NumPy can vectorize automatically and quickly, but not so for factorial().

To use factorial() in a vectorized calculation, you have to use np.vectorize() to create a vectorized version. The documentation for np.vectorize() states that it's little more than a thin wrapper that applies a for loop to a given function. There are no real performance benefits from using it instead of normal Python code, and there are potentially some overhead penalties. However, as you'll see in a moment, the readability benefits are huge.

Once your vectorized factorial is in place, the actual code to calculate the entire Maclaurin series is shockingly short. It's also readable. Most importantly, it's almost exactly one-to-one with how the mathematical equation looks:

```python
n = np.arange(terms)
return np.sum((x ** n) / fac(n))
```

This is such an important idea that it deserves to be repeated. With the exception of the extra line to initialize n, the code reads almost exactly the same as the original math equation. No for loops, no temporary i, j, k variables. Just plain, clear, math.

1 Just like that, you're using NumPy for mathematical programming! For extra practice, try picking one of the other Maclaurin series and implementing it in a similar way.

3.10 Optimizing Storage: Data Types

Now that you have a bit more practical experience, it's time to go back to theory and look at **data types**. Data types don't play a central role in a lot of Python code. Numbers work like they're supposed to, strings do other things, Booleans are true or false, and other than that, you make your own objects and collections.

In NumPy, though, there's a little more detail that needs to be covered. NumPy uses C code under the hood to optimize performance, and it can't do that unless all the items in an array are of the same type. That doesn't just mean the same Python type. They have to be the same underlying C type, with the same shape and size in bits!

3.11 Numerical Types

Since most of your data science and numerical calculations will tend to involve numbers, they seem like the best place to start. There are essentially four numerical types in NumPy code, and each one can take a few different sizes.
The table below breaks down the details of these types:

Name	Number of Bites	Python Type	Numpy Types
Integer	64	`int`	`np.int_`
Booleans	8	`bool`	`np.bool_`
Float	64	`float`	`np.float_`
Complex	128	`complex`	`np.complex_`

Table 3.1.: The data types supported by Python and Numpy.

These are just the types that map to existing Python types. NumPy also has types for the smaller-sized versions of each, like 8-, 16-, and 32-bit integers, 32-bit single-precision floating-point numbers, and 64-bit single-precision complex numbers. The documentation lists them in their entirety.
To specify the type when creating an array, you can provide a dtype argument

Example Find the Missing Values 21

Write a NumPy program to find missing data in a given array.

Solution Find the Missing Values

```python
#+begin_src python :session :results output
# Importing the NumPy library with an alias 'np'
import numpy as np

# Creating a NumPy array 'nums' with provided values, including NaN (Not a Number)
nums = np.array([[3, 2, np.nan, 1],
                 [10, 12, 10, 9],
                 [5, np.nan, 1, np.nan]])

# Printing a message indicating the original array 'nums'
print("Original array:")
print(nums)

# Printing a message indicating finding the missing data (NaN) in the array using
↪  np.isnan()
# This function returns a boolean array of the same shape as 'nums', where True
↪  represents NaN values
print("\nFind the missing data of the said array:")
```

```text
#+begin_example
Original array:
[[ 3.  2. nan  1.]
 [10. 12. 10.  9.]
 [ 5. nan  1. nan]]
Find the missing data of the said array:
[[False False  True False]
 [False False False False]
```

Array Value Parity 22 **Example**

Write a NumPy program to check whether two arrays are equal (element wise) or not.

Array Value Parity **Solution**

```python
#+begin_src python :session :results output
# Importing the NumPy library with an alias 'np'
import numpy as np

# Creating NumPy arrays 'nums1' and 'nums2' with floating-point values
nums1 = np.array([0.5, 1.5, 0.2])
nums2 = np.array([0.4999999999, 1.500000000, 0.2])

# Setting print options to display floating-point precision up to 15 decimal places
np.set_printoptions(precision=15)

```

```python
# Printing a message indicating the original arrays 'nums1' and 'nums2'
print("Original arrays:")
print(nums1)
print(nums2)

# Printing a message asking whether the two arrays are equal element-wise or not
print("\nTest said two arrays are equal (element wise) or not:?")
print(nums1 == nums2)

# Reassigning new values to arrays 'nums1' and 'nums2'
nums1 = np.array([0.5, 1.5, 0.23])
nums2 = np.array([0.4999999999, 1.5000000001, 0.23])

# Printing a message indicating the original arrays 'nums1' and 'nums2'
print("\nOriginal arrays:")
np.set_printoptions(precision=15)
print(nums1)
print(nums2)

# Printing a message asking whether the two arrays are equal element-wise or not
print("\nTest said two arrays are equal (element wise) or not:?")
```

```text
#+begin_example
Original arrays:
[0.5 1.5 0.2]
[0.4999999999 1.5          0.2         ]
Test said two arrays are equal (element wise) or not:?
[False  True  True]
Original arrays:
[0.5   1.5   0.23]
[0.4999999999 1.5000000001 0.23        ]
Test said two arrays are equal (element wise) or not:?
```

Chapter 4

Matplotlib

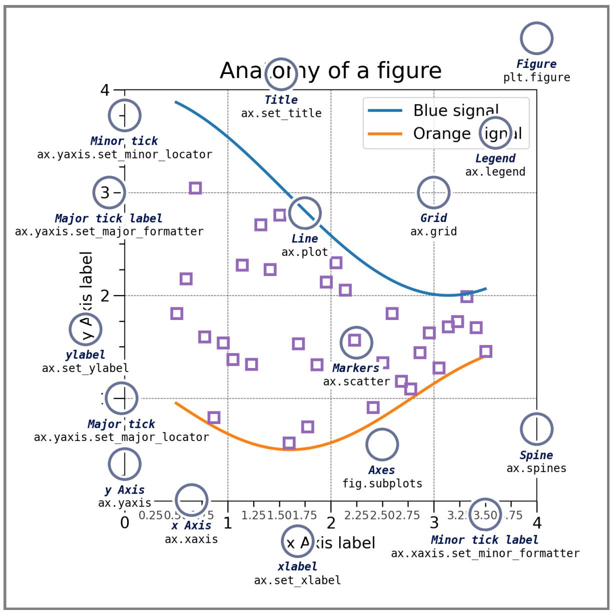
Table of Contents

4.1 Introduction

Matplotlib is an excellent 2D and 3D graphics library for generating scientific figures. Some of the many advantages of this library include:

- Easy to get started
- Support for LaTeX formatted labels and text
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS, and PGF.
- GUI for interactively exploring figures and support for headless generation of figure files (useful for batch jobs)

One of the key features of matplotlib that I would like to emphasize, and that I think makes matplotlib highly suitable for generating figures for scientific publications is that all aspects of the figure can be controlled programmatically. This is important for reproducibility and convenient when one needs to regenerate the figure with updated data or change its appearance.



More information at the Matplotlib web page: `http://matplotlib.org/`
To get started using Matplotlib in a Python program, either include the symbols from the pylab module (the easy way):

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

We also need to import our numpy module to use later on.

```python
import numpy as np
```

Let's start with a simple figure, but before we begin with generating some visuals, we need to create some data-points.

```python
x = np.linspace(0, 5, 10)
y = x ** 2
```

Once we have the data of both (x, y) we can now create some plots.
The main idea with OOP is to have objects that one can apply functions and actions on, and no object or program states should be global.

The real advantage of this approach becomes apparent when more than one figure is created, or when a figure contains more than one subplot.

To use the object-oriented API we store our plot on a reference to the newly created `figure` instance in the `fig` variable, and from it we create a new axis instance axes using the `add_axes` method in the Figure class instance fig:

```python
plt.style.use('bmh')
plt.figure()
plt.plot(x, y, 'r')
plt.xlabel('x')
plt.ylabel('y')
plt.title('title')
plt.savefig("images/Matplotlib/example-figure.pdf")
plt.close()
```

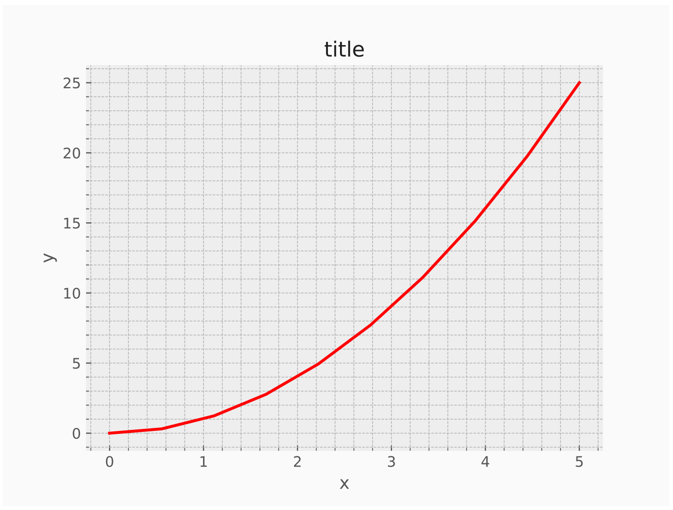


Figure 4.1.

As can be seen, we have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

```python
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
axes2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# main figure
axes1.plot(x, y, 'r')
axes1.set_xlabel('x')
```

```python
axes1.set_ylabel('y')
axes1.set_title('Primary Plot')

# insert
axes2.plot(y, x, 'g')
axes2.set_xlabel('y')
axes2.set_ylabel('x')
axes2.set_title('Secondary Plot');

# save figure and close
plt.savefig("images/Matplotlib/figure-in-a-figure.pdf")
plt.close()
```

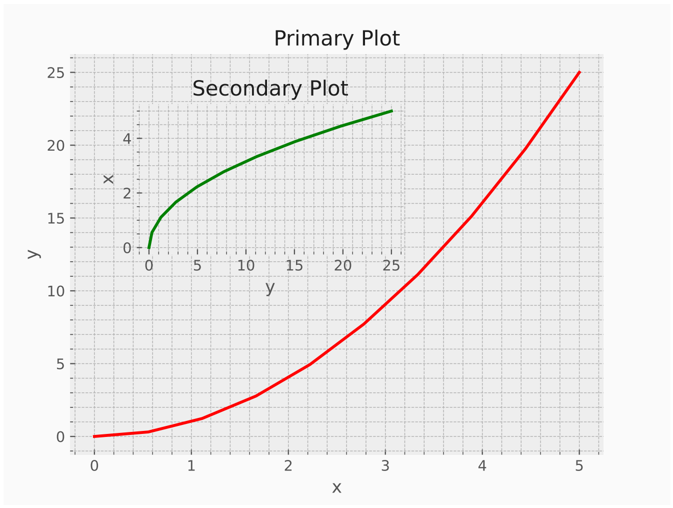


Figure 4.2.

If we don't care about being explicit about where our plot axes are placed in the figure canvas, then we can use one of the many axis layout managers in matplotlib.
A good option is **subplots**, which can be used like this:

```python
fig, axes = plt.subplots()

axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');

# save figure and close
plt.savefig("images/Matplotlib/a-simple-subplot-figure.pdf")
plt.close()
```

And because it is a subplot, we can add anothe subplot into the same frame, such as

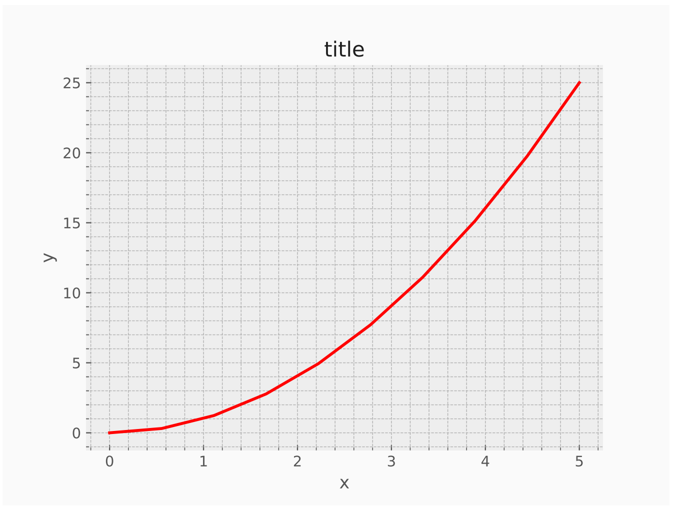


Figure 4.3.

the following:

```python
fig, axes = plt.subplots(nrows=1, ncols=2)

for ax in axes:
    ax.plot(x, y, 'r')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

# save figure and close
plt.savefig("images/Matplotlib/two-subplots-figure.pdf")
plt.close()
```

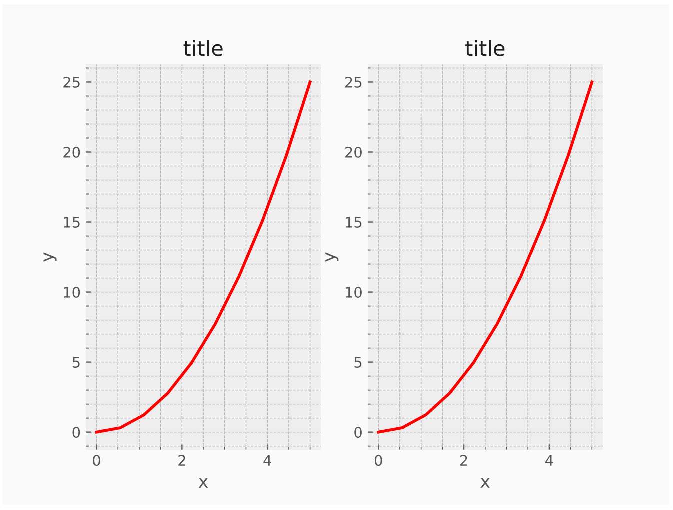


Figure 4.4.

That was (relatively) easy, but it isn't so pretty with overlapping figure axes and labels, right?

We can deal with that by using the `fig.tight_layout` method, which automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content:

```python
fig, axes = plt.subplots(nrows=1, ncols=2)

for ax in axes:
    ax.plot(x, y, 'r')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

fig.tight_layout()

# save figure and close
plt.savefig("images/Matplotlib/tight-subplots-figure.pdf")
plt.close()
```

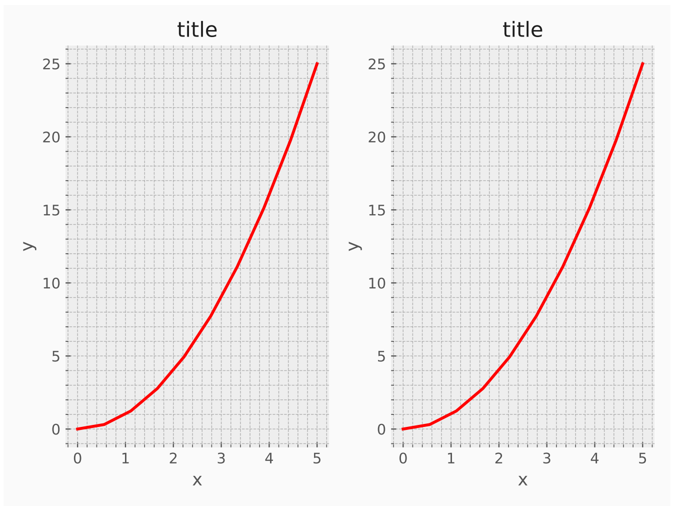


Figure 4.5.

4.1.1 Figure size, aspect ratio and DPI

Matplotlib allows the aspect ratio, DPI and figure size to be specified when the Figure object is created, using the figsize and dpi keyword arguments. figsize is a tuple of the width and height of the figure in inches, and dpi is the dots-per-inch (pixel per inch). To create an 800x400 pixel, 100 dots-per-inch figure, we can do:

```python
fig = plt.figure(figsize=(8,4), dpi=100)
```

The same arguments can also be passed to layout managers, such as the subplots function:

```python
fig, axes = plt.subplots(figsize=(12,3))

axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');

# save figure and close
plt.savefig("images/Matplotlib/long-figure.pdf")
plt.close()
```

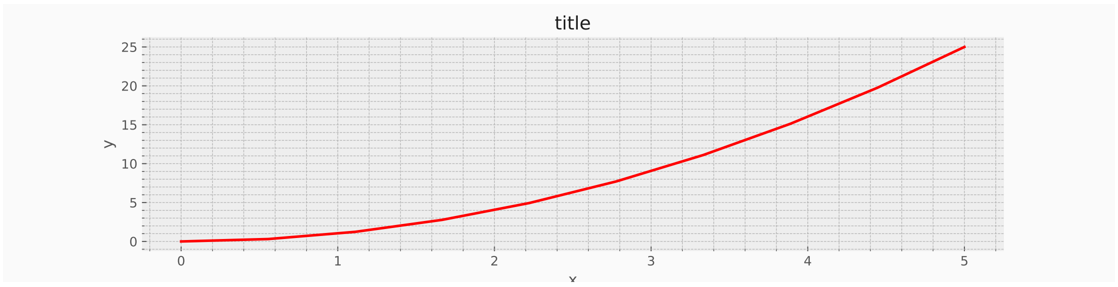


Figure 4.6.

Saving Figures

As you have seen, the method `savefig` method is used quite frequently. This method allows you to save any figure generated by matplotlib. As it stands you can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF. If you are using any non-vector image formats, you can also set the DPI settings, for example

```python
fig.savefig("filename.png", dpi=200)
```

4.1.2 Legends, labels and titles

Now that we have covered the basics of how to create a figure canvas and add axes instances to the canvas, let's look at how decorate a figure with titles, axis labels, and legends.

Figure Titles

A title can be added to each axis instance in a figure. To set the title, use the `set_title` method in the axes instance:

```python
ax.set_title("title");
```

Axis Labels

Similarly, with the methods `set_xlabel` and `set_ylabel`, we can set the labels of the X and Y axes:

```python
ax.set_xlabel("x")
ax.set_ylabel("y")
```

Legends

Legends for curves in a figure can be added in two (2) ways. One method is to use the legend method of the axis object and pass a list/tuple of legend texts for the previously defined curves:

```python
ax.legend(["curve1", "curve2", "curve3"])
```

[1] A software used by engineers for calculations

The method described above follows a similar way to MATLAB[1]. It is somewhat prone to errors and un-flexible if curves are added to or removed from the figure (resulting in a wrongly labelled curve).

A better method is to use the `label="label text"` keyword argument when plots or other objects are added to the figure, and then using the legend method without arguments to add the `legend` to the figure:

```python
ax.plot(x, x**2, label="curve1")
ax.plot(x, x**3, label="curve2")
ax.legend();
```

The advantage with this method is that if curves are added or removed from the figure, the legend is **automatically updated** accordingly.

The `legend` function takes an optional keyword argument `loc` that can be used to specify where in the figure the legend is to be drawn. The allowed values of `loc` are numerical codes for the various places the legend can be drawn. See `http://`

matplotlib.org/users/legend_guide.html#legend-location for details. Some of the most common `loc` values are:

```python
ax.legend(loc=0) # let matplotlib decide the optimal location
ax.legend(loc=1) # upper right corner
ax.legend(loc=2) # upper left corner
ax.legend(loc=3) # lower left corner
ax.legend(loc=4) # lower right corner
# .. many more options are available
```

The following figure shows how to use the figure title, axis labels and legends described above:

```python
fig, ax = plt.subplots()

ax.plot(x, x**2, label="y = x**2")
ax.plot(x, x**3, label="y = x**3")
ax.legend(loc=2); # upper left corner
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('title')

# save figure and close
plt.savefig("images/Matplotlib/legend-figure.pdf")
plt.close()
```

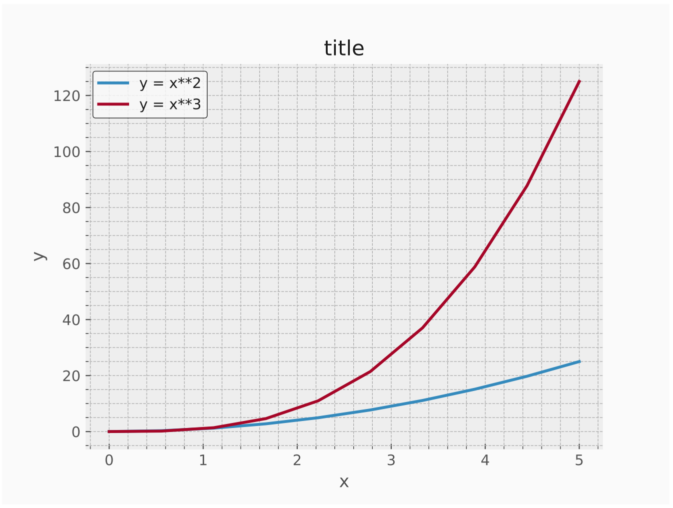


Figure 4.7.

4.1.3 Formatting text: LaTeX, fontsize, font family

The figure above is functional, but it does not (yet) satisfy the criteria for a figure used in a publication.

- we need to have LaTeX formatted text,
- we need to be able to adjust the font size to appear right in a publication.

Matplotlib has great support for LaTeX. All we need to do is to use dollar signs encapsulate LaTeX in any text (legend, title, label, etc.).

For example, "$y = x^3$".

But here we can run into a slightly subtle problem with LaTeX code and Python text strings. In LaTeX, we frequently use the backslash in commands, for example `\alpha` to produce the symbol $\alpha$. But the backslash already has a meaning in Python strings (the escape code character). To avoid Python messing up our latex code, we need to use "raw" text strings. Raw text strings are prepended with an 'r', like `r"\alpha"` or `r'\alpha'` instead of `"\alpha"` or `'\alpha'`:

```python
fig, ax = plt.subplots()

ax.plot(x, x**2, label=r"$y = \alpha^2$")
ax.plot(x, x**3, label=r"$y = \alpha^3$")
ax.legend(loc=2) # upper left corner
ax.set_xlabel(r'$\alpha$', fontsize=18)
ax.set_ylabel(r'$y$', fontsize=18)
ax.set_title('title')

# save figure and close
plt.savefig("images/Matplotlib/latex-figure.pdf")
plt.close()
```

We can also change the global font size and font family, which applies to all text elements in a figure (tick labels, axis labels and titles, legends, etc.):

```python
matplotlib.rcParams.update({'font.size': 18, 'font.family': 'serif'})
```

Or, alternatively, we can request that matplotlib uses LaTeX to render the text elements in the figure:

```
matplotlib.rcParams.update({'font.size': 18, 'text.usetex': True})
```

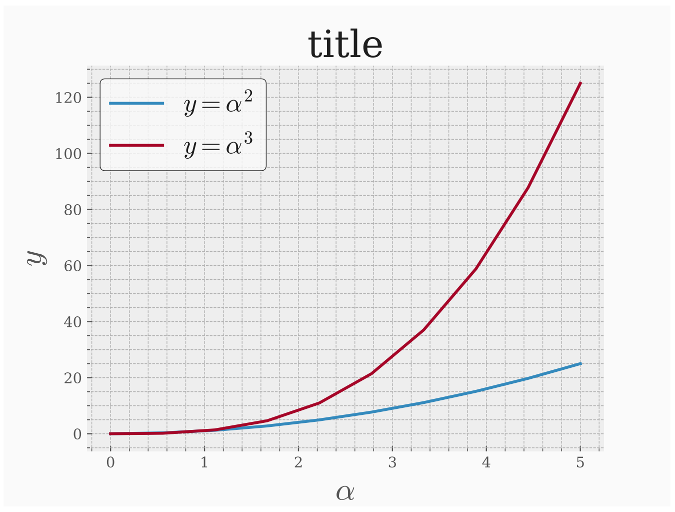


Figure 4.8.

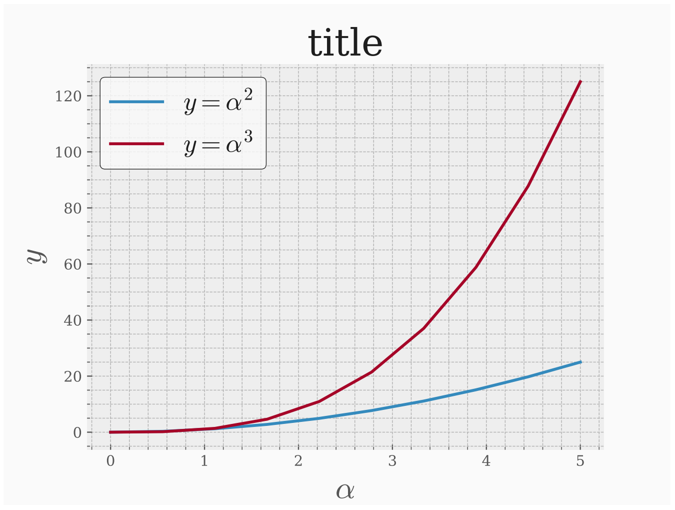


Figure 4.9.

```
fig, ax = plt.subplots()

ax.plot(x, x**2, label=r"$y = \alpha^2$")
ax.plot(x, x**3, label=r"$y = \alpha^3$")
ax.legend(loc=2) # upper left corner
ax.set_xlabel(r'$\alpha$')
ax.set_ylabel(r'$y$')
ax.set_title('title');
```

```
matplotlib.rcParams.update({'font.size': 12, 'font.family': 'sans',
↪  'text.usetex': False})
```

4.1.4 Setting colors, linewidths, linetypes

Colours

With matplotlib, we can define the colors of lines and other graphical elements in a number of ways. First of all, we can use the MATLAB-like syntax where ’b’ means blue, ’g’ means green, etc. The MATLAB API for selecting line styles are also supported: where, for example, ’b.-’ means a blue line with dots:

```
# MATLAB style line color and style
ax.plot(x, x**2, 'b.-') # blue line with dots
ax.plot(x, x**3, 'g--') # green dashed line
```

We can also define colors by their names or RGB hex codes and optionally provide an alpha value using the color and alpha keyword arguments:

```
fig, ax = plt.subplots()

ax.plot(x, x+1, color="red", alpha=0.5) # half-transparent red
ax.plot(x, x+2, color="#1155dd")        # RGB hex code for a bluish color
ax.plot(x, x+3, color="#15cc55")        # RGB hex code for a greenish color
```

Line and marker styles

To change the line width, we can use the linewidth or lw keyword argument. The line style can be selected using the linestyle or ls keyword arguments:

```
fig, ax = plt.subplots(figsize=(12,6))

ax.plot(x, x+1, color="blue", linewidth=0.25)
ax.plot(x, x+2, color="blue", linewidth=0.50)
ax.plot(x, x+3, color="blue", linewidth=1.00)
ax.plot(x, x+4, color="blue", linewidth=2.00)

# possible linestype options ‘-‘, ‘--', ‘-.', ‘:', ‘steps'
ax.plot(x, x+5, color="red", lw=2, linestyle='-')
ax.plot(x, x+6, color="red", lw=2, ls='-.')
ax.plot(x, x+7, color="red", lw=2, ls=':')

# custom dash
line, = ax.plot(x, x+8, color="black", lw=1.50)
line.set_dashes([5, 10, 15, 10]) # format: line length, space length, ...

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2',
↪  '3', '4', ...
```

```
ax.plot(x, x+ 9, color="green", lw=2, ls='--', marker='+')
ax.plot(x, x+10, color="green", lw=2, ls='--', marker='o')
ax.plot(x, x+11, color="green", lw=2, ls='--', marker='s')
ax.plot(x, x+12, color="green", lw=2, ls='--', marker='1')

# marker size and color
ax.plot(x, x+13, color="purple", lw=1, ls='-', marker='o', markersize=2)
ax.plot(x, x+14, color="purple", lw=1, ls='-', marker='o', markersize=4)
ax.plot(x, x+15, color="purple", lw=1, ls='-', marker='o', markersize=8,
↪  markerfacecolor="red")
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=2, markeredgecolor="blue");
```

4.1.5 Control over axis appearance

The appearance of the axes is an important aspect of a figure that we often need to modify to make a publication quality graphics. We need to be able to control where the ticks and labels are placed, modify the font size and possibly the labels used on the axes. In this section we will look at controlling those properties in a matplotlib figure.

Plot range

The first thing we might want to configure is the ranges of the axes. We can do this using the set$_{ylim}$ and set$_{xlim}$ methods in the axis object, or axis('tight') for automatically getting "tightly fitted" axes ranges:

```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

axes[0].plot(x, x**2, x, x**3)
axes[0].set_title("default axes ranges")

axes[1].plot(x, x**2, x, x**3)
axes[1].axis('tight')
axes[1].set_title("tight axes")

axes[2].plot(x, x**2, x, x**3)
axes[2].set_ylim([0, 60])
axes[2].set_xlim([2, 5])
axes[2].set_title("custom axes range");
```

Logarithmic scale

It is also possible to set a logarithmic scale for one or both axes. This functionality is in fact only one application of a more general transformation system in Matplotlib. Each of the axes' scales are set seperately using set$_{xscale}$ and set$_{yscale}$ methods which accept one parameter (with the value "log" in this case):

```
fig, axes = plt.subplots(1, 2, figsize=(10,4))

axes[0].plot(x, x**2, x, np.exp(x))
axes[0].set_title("Normal scale")

axes[1].plot(x, x**2, x, np.exp(x))
axes[1].set_yscale("log")
axes[1].set_title("Logarithmic scale (y)");
```

4.1.6 Placement of ticks and custom tick labels

We can explicitly determine where we want the axis ticks with set$_{xticks}$ and set$_{yticks}$, which both take a list of values for where on the axis the ticks are to be placed. We can also use the set$_{xticklabels}$ and set$_{yticklabels}$ methods to provide a list of custom text labels for each tick location:

```
fig, ax = plt.subplots(figsize=(10, 4))

ax.plot(x, x**2, x, x**3, lw=2)

ax.set_xticks([1, 2, 3, 4, 5])
ax.set_xticklabels([r'$\alpha$', r'$\beta$', r'$\gamma$', r'$\delta$',
↪  r'$\epsilon$'], fontsize=18)

yticks = [0, 50, 100, 150]
ax.set_yticks(yticks)
ax.set_yticklabels(["$%.1f$" % y for y in yticks], fontsize=18); # use LaTeX
↪  formatted labels
```

There are a number of more advanced methods for controlling major and minor tick placement in matplotlib figures, such as automatic placement according to different policies. See `http://matplotlib.org/api/ticker_api.html` for details.

Scientific notation

With large numbers on axes, it is often better use scientific notation:

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

ax.plot(x, x**2, x, np.exp(x))
ax.set_title("scientific notation")

ax.set_yticks([0, 50, 100, 150])

from matplotlib import ticker
formatter = ticker.ScalarFormatter(useMathText=True)
formatter.set_scientific(True)
formatter.set_powerlimits((-1,1))
ax.yaxis.set_major_formatter(formatter)
```

4.1.7 Axis number and axis label spacing

```
# distance between x and y axis and the numbers on the axes
matplotlib.rcParams['xtick.major.pad'] = 5
matplotlib.rcParams['ytick.major.pad'] = 5

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

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("label and axis spacing")

# padding between axis label and axis numbers
ax.xaxis.labelpad = 5
ax.yaxis.labelpad = 5

ax.set_xlabel("x")
ax.set_ylabel("y");
```

```
# restore defaults
matplotlib.rcParams['xtick.major.pad'] = 3
matplotlib.rcParams['ytick.major.pad'] = 3
```

Axis Position Adjustement

Unfortunately, when saving figures the labels are sometimes clipped, and it can be necessary to adjust the positions of axes a little bit. This can be done using $\text{subplots}_{\text{adjust}}$:

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

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("title")
ax.set_xlabel("x")
ax.set_ylabel("y")

fig.subplots_adjust(left=0.15, right=.9, bottom=0.1, top=0.9);
```

4.1.8 Axis grid

With the grid method in the axis object, we can turn on and off grid lines. We can also customize the appearance of the grid lines using the same keyword arguments as the plot function:

```
fig, axes = plt.subplots(1, 2, figsize=(10,3))

# default grid appearance
axes[0].plot(x, x**2, x, x**3, lw=2)
axes[0].grid(True)

# custom grid appearance
axes[1].plot(x, x**2, x, x**3, lw=2)
axes[1].grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
```

4.1.9 Axis Spines

We can also change the properties of axis spines:

```
fig, ax = plt.subplots(figsize=(6,2))

ax.spines['bottom'].set_color('blue')
ax.spines['top'].set_color('blue')

ax.spines['left'].set_color('red')
ax.spines['left'].set_linewidth(2)

# turn off axis spine to the right
ax.spines['right'].set_color("none")
ax.yaxis.tick_left() # only ticks on the left side
```

4.1.10 Twin Axes

Sometimes it is useful to have dual x or y axes in a figure; for example, when plotting curves with different units together. Matplotlib supports this with the twinx and twiny functions:

```
fig, ax1 = plt.subplots()

ax1.plot(x, x**2, lw=2, color="blue")
ax1.set_ylabel(r"area $(m^2)$", fontsize=18, color="blue")
for label in ax1.get_yticklabels():
    label.set_color("blue")

ax2 = ax1.twinx()
ax2.plot(x, x**3, lw=2, color="red")
ax2.set_ylabel(r"volume $(m^3)$", fontsize=18, color="red")
for label in ax2.get_yticklabels():
    label.set_color("red")
```

4.1.11 Axes where x and y is zero

```
fig, ax = plt.subplots()

ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0)) # set position of x spine to x=0

ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))   # set position of y spine to y=0

xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);
```

4.1.12 Other 2D plot styles

In addition to the regular plot method, there are a number of other functions for generating different kind of plots. See the matplotlib plot gallery for a complete list of available plot types: http://matplotlib.org/gallery.html. Some of the more useful ones are show below:

```
n = np.array([0,1,2,3,4,5])
```

```
fig, axes = plt.subplots(1, 4, figsize=(12,3))

axes[0].scatter(xx, xx + 0.25*np.random.randn(len(xx)))
axes[0].set_title("scatter")

axes[1].step(n, n**2, lw=2)
axes[1].set_title("step")

axes[2].bar(n, n**2, align="center", width=0.5, alpha=0.5)
axes[2].set_title("bar")

axes[3].fill_between(x, x**2, x**3, color="green", alpha=0.5);
axes[3].set_title("fill_between");
```

```
# polar plot using add_axes and polar projection
fig = plt.figure()
ax = fig.add_axes([0.0, 0.0, .6, .6], polar=True)
t = np.linspace(0, 2 * np.pi, 100)
ax.plot(t, t, color='blue', lw=3);
```

```
# A histogram
n = np.random.randn(100000)
fig, axes = plt.subplots(1, 2, figsize=(12,4))

axes[0].hist(n)
axes[0].set_title("Default histogram")
axes[0].set_xlim((min(n), max(n)))

axes[1].hist(n, cumulative=True, bins=50)
axes[1].set_title("Cumulative detailed histogram")
axes[1].set_xlim((min(n), max(n)));
```

4.1.13 Text annotation

Annotating text in matplotlib figures can be done using the text function. It supports LaTeX formatting just like axis label texts and titles:

```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)

ax.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="blue")
ax.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="green");
```

4.1.14 Figures with multiple subplots and insets

Axes can be added to a matplotlib Figure canvas manually using fig.add$_{\text{axes}}$ or using a sub-figure layout manager such as subplots, subplot2grid, or gridspec:

subplots

```
fig, ax = plt.subplots(2, 3)
fig.tight_layout()
```

subplot2grid

```
fig = plt.figure()
ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
ax2 = plt.subplot2grid((3,3), (1,0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1,2), rowspan=2)
ax4 = plt.subplot2grid((3,3), (2,0))
ax5 = plt.subplot2grid((3,3), (2,1))
fig.tight_layout()
```

gridspec

```
import matplotlib.gridspec as gridspec
```

```
fig = plt.figure()

gs = gridspec.GridSpec(2, 3, height_ratios=[2,1], width_ratios=[1,2,1])
for g in gs:
    ax = fig.add_subplot(g)
```

```
fig.tight_layout()
```

add_axes

Manually adding axes with add$_{axes}$ is useful for adding insets to figures:

```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)
fig.tight_layout()

# inset
inset_ax = fig.add_axes([0.2, 0.55, 0.35, 0.35]) # X, Y, width, height

inset_ax.plot(xx, xx**2, xx, xx**3)
inset_ax.set_title('zoom near origin')

# set axis range
inset_ax.set_xlim(-.2, .2)
inset_ax.set_ylim(-.005, .01)

# set axis tick locations
inset_ax.set_yticks([0, 0.005, 0.01])
inset_ax.set_xticks([-0.1,0,.1]);
```

4.2 Colormap and contour figures

Colormaps and contour figures are useful for plotting functions of two (2) variables. In most of these functions we will use a colormap to encode one dimension of the data. There are a number of predefined colormaps. It is relatively straightforward to define custom colormaps.

For a list of pre-defined colormaps, see: `http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps`

```python
alpha = 0.7
phi_ext = 2 * np.pi * 0.5

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * np.cos(phi_p) * np.cos(phi_m) \
        - alpha * np.cos(phi_ext - 2*phi_p)
```

```python
phi_m = np.linspace(0, 2*np.pi, 100)
phi_p = np.linspace(0, 2*np.pi, 100)
X,Y = np.meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
```

pcolor

```python
fig, ax = plt.subplots()

p = ax.pcolor(X/(2*np.pi), Y/(2*np.pi), Z, cmap=matplotlib.cm.RdBu,
↪  vmin=abs(Z).min(), vmax=abs(Z).max())
cb = fig.colorbar(p, ax=ax)

# save figure and close
plt.savefig("images/Matplotlib/pcolor-plot.pdf")
plt.close()
```

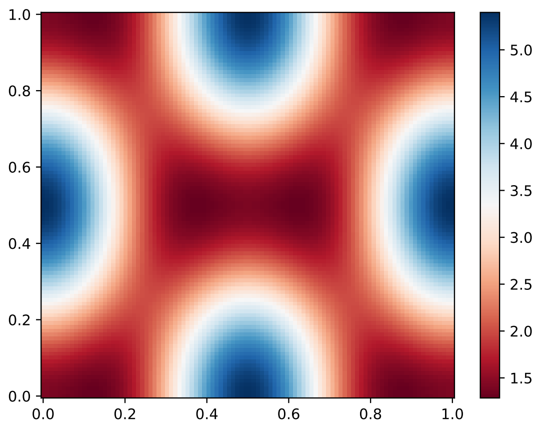


Figure 4.10.

imshow

```python
fig, ax = plt.subplots()

im = ax.imshow(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(),
↪  extent=[0, 1, 0, 1])
im.set_interpolation('bilinear')

cb = fig.colorbar(im, ax=ax)

# save figure and close
plt.savefig("images/Matplotlib/imshow-plot.pdf")
plt.close()
```

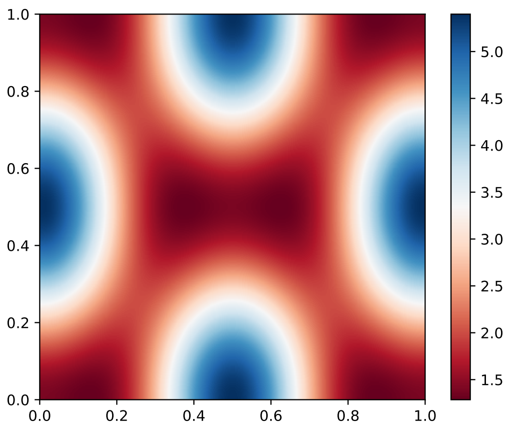


Figure 4.11.

contour

```python
fig, ax = plt.subplots()

cnt = ax.contour(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(),
↪  extent=[0, 1, 0, 1])

# save figure and close
plt.savefig("images/Matplotlib/contour-plot.pdf")
plt.close()
```

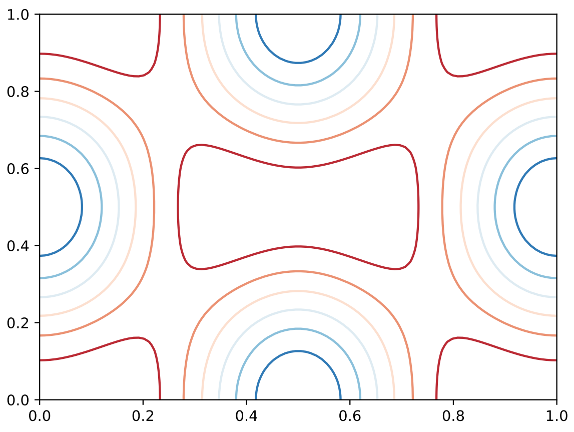


Figure 4.12.

4.2.1 3D Figures

To use 3D graphics in matplotlib, we first need to create an instance of the Axes3D class. 3D axes can be added to a matplotlib figure canvas in exactly the same way as 2D axes; or, more conveniently, by passing a `projection='3d'` keyword argument to the `add_axes` or `add_subplot` methods.

```python
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

Surface Plots

```python
fig = plt.figure(figsize=(14,6))

# `ax` is a 3D-aware axis instance because of the projection='3d' keyword argument
↪  to add_subplot
ax = fig.add_subplot(1, 2, 1, projection='3d')

p = ax.plot_surface(X, Y, Z, rstride=4, cstride=4, linewidth=0)

# surface_plot with color grading and color bar
ax = fig.add_subplot(1, 2, 2, projection='3d')
p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm,
↪  linewidth=0, antialiased=False)
cb = fig.colorbar(p, shrink=0.5)

# save figure and close
plt.savefig("images/Matplotlib/3d-surface-plot.pdf")
plt.close()
```

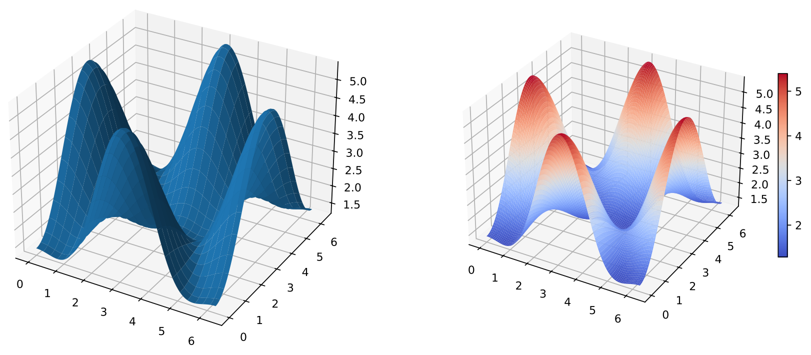


Figure 4.13.

Wire-Frame Plots

```python
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1, 1, 1, projection='3d')

p = ax.plot_wireframe(X, Y, Z, rstride=4, cstride=4)

# save figure and close
plt.savefig("images/Matplotlib/3d-wire-plot.pdf")
plt.close()
```

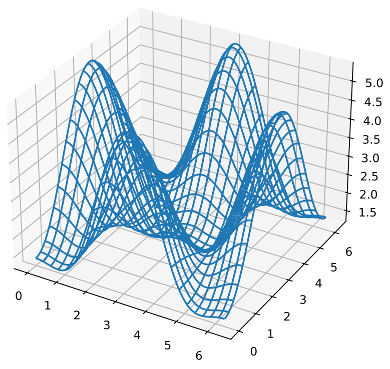


Figure 4.14.

Contour Plots with Projections

```python
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*np.pi, cmap=matplotlib.cm.coolwarm)

ax.set_xlim3d(-np.pi, 2*np.pi);
ax.set_ylim3d(0, 3*np.pi);
ax.set_zlim3d(-np.pi, 2*np.pi);

# save figure and close
plt.savefig("images/Matplotlib/contour-plot-with-projections.pdf")
plt.close()
```

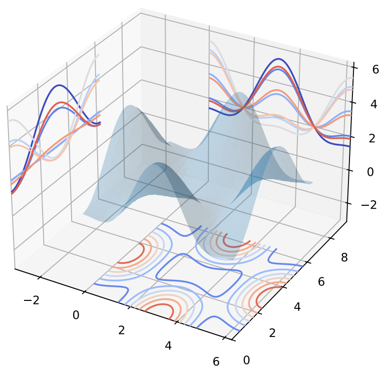


Figure 4.15.

Change the view angle

We can change 3D plot perspective using the method `view_init`, which takes two (2) arguments:

- elevation angle,
- azimuth angle.

> Both these values are in degrees and not radians.

```python
fig = plt.figure(figsize=(12,6))

ax = fig.add_subplot(1,2,1, projection='3d')
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
ax.view_init(30, 45)

ax = fig.add_subplot(1,2,2, projection='3d')
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
ax.view_init(70, 30)

fig.tight_layout()


# save figure and close
plt.savefig("images/Matplotlib/view-angle.pdf")
plt.close()
```

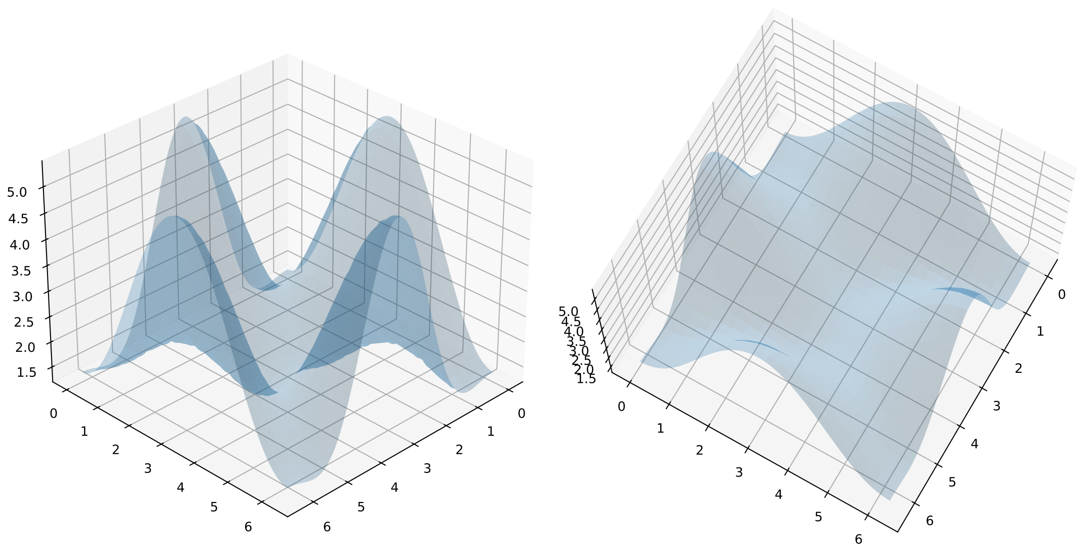


Figure 4.16.

Chapter 5

Computer Algebra System - SymPy

There are two (2) notable Computer Algebra Systems (CAS) for Python:

1. **SymPy** A python module that can be used in any Python program, or in an IPython session, that provides powerful CAS features.

2. **Sage** A full-featured and very powerful CAS environment that aims to provide an open source system that competes with Mathematica and Maple.

> Sage is not a regular Python module, but rather a CAS environment that uses Python as its programming language.

> Sage is in some aspects more powerful than SymPy, but both offer very comprehensive CAS functionality. The advantage of SymPy is that it is a regular Python module and integrates well with the IPython notebook.

In this lecture we will therefore look at how to use SymPy with IPython notebooks. If you are interested in an open source CAS environment I also recommend to read more about Sage as it is a great tool to study linear algebra and differential equations.

To get started using SymPy in a Python program or notebook, import the module `sympy`:

```python
import sympy as sp # for symbolic calculations
import matplotlib.pyplot as plt # for plotting applications
```

To print them in LaTeX format, we can invoke the following command:

```python
sp.init_printing(use_latex=True)
```

5.1 Symbolic Variables

In SymPy we need to create symbols for the variables we want to work with. We can create a new symbol using the `Symbol` class:

```
x = sp.Symbol('x')
```

```
print(sp.latex((sp.pi + x)**2))
```

$$\left(x + \pi\right)^{2}$$

```
# alternative way of defining symbols
a, b, c = sp.symbols("a, b, c")
```

And if we want to look at the type of a data in Python, remember to just use `type()` which will give you the data type of the variable in question.

```
print(type(a))
```

```
<class 'sympy.core.symbol.Symbol'>
```

We can add assumptions to symbols when we create them just write it in the paran-thesis as so:

```
x = sp.Symbol('x', real=True)
```

And if we assume a false assumption, we should get `False`.

```
print(x.is_imaginary)
```

```
False
```

We can also do other assumptions:

```
x = sp.Symbol('x', positive=True)
```

And testing this with a `True` statement, we find:

```
print(x > 0)
```

```
True
```

5.1.1 Complex Numbers

The imaginary unit is denoted `I` in SymPy.

Imaginary Numbers

An imaginary number is the product of a real number and the imaginary unit i, which is defined by its property $i^2 = -1$. The square of an imaginary number bi is $-b^2$. For example, $5i$ is an imaginary number, and its square is -25. The number zero is considered to be both real and imaginary.

Originally coined in the 17th century by René Descartes as a derogatory term and regarded as fictitious or useless, the concept gained wide acceptance following the work of Leonhard Euler (in the 18th century) and Augustin-Louis Cauchy and Carl Friedrich Gauss (in the early 19th century).

An imaginary number bi can be added to a real number a to form a complex number of the form $a + bi$, where the real numbers a and b are called, respectively, the real part and the imaginary part of the complex number.

```python
print(sp.latex(1+1*sp.I))
```

$$1 + i$$

```python
print(sp.latex(sp.I**2))
```

$$-1$$

```python
print(sp.latex((x * sp.I + 1)**2))
```

$$(ix + 1)^2$$

5.1.2 Rational Numbers

There are three different numerical types in SymPy: `Real`, `Rational`, `Integer`:

Real - Rational - Integer

Real Numbers: A number that can be used to measure a continuous one-dimensional quantity such as a distance, duration or temperature. Here, continuous means that pairs of values can have arbitrarily small differences.

Rational Number: a rational number is a number that can be expressed as the quotient or fraction $\frac{p}{q}$ of two integers, a numerator p and a non-zero denominator q.

Integer: is the number zero (0), a positive natural number (1, 2, 3, . . .), or the negation of a positive natural number (-1, -2, -3, . . .).

```
r1 = sp.Rational(4,5)
r2 = sp.Rational(5,4)
```

```
print(sp.latex(r1))
```

$$\frac{4}{5}$$

```
print(sp.latex(r1 + r2))
```

$$\frac{41}{20}$$

```
print(sp.latex(r1 / r2))
```

$$\frac{16}{25}$$

5.2 Numerical Evaluation

SymPy uses a library for arbitrary precision as numerical backend, and has predefined SymPy expressions for a number of mathematical constants, such as: `pi`, `e`, `oo` for infinity.
To evaluate an expression numerically we can use the evalf function (or `N`). It takes an argument `n` which specifies the number of significant digits.

```
print(sp.latex(sp.pi.evalf(n=50)))
```

$$3.1415926535897932384626433832795028841971693993751$$

```
y = (x + sp.pi)**2
```

```
print(sp.latex(sp.N(y, 5))) # same as evalf
```

$$9.8696\,(0.31831x + 1)^2$$

When we numerically evaluate algebraic expressions we often want to substitute a symbol with a numerical value. In SymPy we do that using the subs function

```
print(sp.latex(y.subs(x, 1.5)))
```

$$(1.5 + \pi)^2$$

```
print(sp.latex(sp.N(y.subs(x, 1.5))))
```

$$21.5443823618587$$

The `subs` function can of course also be used to substitute Symbols and expressions:

```
print(sp.latex(y.subs(x, a+sp.pi)))
```

$$(a + 2\pi)^2$$

We can also combine numerical evaluation of expressions with NumPy arrays:

```
import numpy
```

```
x_vec = numpy.arange(0, 10, 0.1)
```

```
y_vec = numpy.array([sp.N(((x + sp.pi)**2).subs(x, xx)) for xx in x_vec])
```

```
fig, ax = plt.subplots()
ax.plot(x_vec, y_vec);
plt.savefig("images/sympy/fplot.pdf")
```

However, this kind of numerical evaluation can be very slow, and there is a much more efficient way to do it: Use the function `lambdify` to ”compile” a SymPy expression into a function that is much more efficient to evaluate numerically:

```
f = sp.lambdify([x], (x + sp.pi)**2, 'numpy')
# the first argument is a list of variables that
# f will be a function of: in this case only x -> f(x)
```

```
y_vec = f(x_vec)  # now we can directly pass a numpy array and f(x) is
↪  efficiently evaluated
```

5.3 Algebraic manipulations

One of the main uses of an CAS is to perform algebraic manipulations of expressions. For example, we might want to expand a product, factor an expression, or simplify an expression. The functions for doing these basic operations in SymPy are demonstrated in this section.

5.3.1 Expand and Factor

The first steps in an algebraic manipulation

```
print(sp.latex((x+1)*(x+2)*(x+3)))
```

$$\left(x + 1\right) \left(x + 2\right) \left(x + 3\right)$$

```
print(sp.latex(sp.expand((x+1)*(x+2)*(x+3))))
```

$$x^{3} + 6 x^{2} + 11 x + 6$$

The `expand` function takes a number of keywords arguments which we can tell the functions what kind of expansions we want to have performed. For example, to expand trigonometric expressions, use the `trig=True` keyword argument:

```
print(sp.latex(sp.sin(a+b)))
```

$$\sin{\left(a + b \right)}$$

```
print(sp.latex(sp.expand(sp.sin(a+b), trig=True)))
```

$$\sin{\left(a \right)} \cos{\left(b \right)} + \sin{\left(b \right)} \cos{\left(a \right)}$$

See `help(expand)` for a detailed explanation of the various types of expansions the expand functions can perform.
The opposite of product expansion is of course factoring. To factor an expression in SymPy use the `factor` function:

```
print(sp.latex(sp.factor(x**3 + 6 * x**2 + 11*x + 6)))
```

$$(x+1)(x+2)(x+3)$$

5.3.2 Simplify

The `simplify` tries to simplify an expression into a nice looking expression, using various techniques. More specific alternatives to the `simplify` functions also exists: `trigsimp`, `powsimp`, `logcombine`, etc. The basic usages of these functions are as follows:

```
# simplify (sometimes) expands a product
print(sp.latex(sp.simplify((x+1)*(x+2)*(x+3))))
```

$$(x+1)(x+2)(x+3)$$

```
# simplify uses trigonometric identities
print(sp.latex(sp.simplify(sp.sin(a)**2 +sp. cos(a)**2)))
```

$$1$$

```
print(sp.latex(sp.simplify(sp.cos(x)/sp.sin(x))))
```

$$\frac{1}{\tan(x)}$$

5.3.3 Apart and Together

```
f1 = 1/((a+1)*(a+2))
```

```
print(sp.latex(f1))
```

$$\frac{1}{(a+1)(a+2)}$$

```
print(sp.latex(sp.apart(f1)))
```

$$-\frac{1}{a+2}+\frac{1}{a+1}$$

```
f2 = 1/(a+2) + 1/(a+3)
```

```
print(sp.latex(f2))
```

$$\frac{1}{a+3} + \frac{1}{a+2}$$

```
print(sp.latex(sp.together(f2)))
```

$$\frac{2a+5}{(a+2)(a+3)}$$

Simplify usually combines fractions but **does not factor**:

```
print(sp.latex(sp.simplify(f2)))
```

$$\frac{2a+5}{(a+2)(a+3)}$$

5.4 Calculus

In addition to algebraic manipulations, the other main use of CAS is to do calculus, like derivatives and integrals of algebraic expressions.

5.4.1 Differentiation

Differentiation is usually simple. Use the `diff` function. The first argument is the expression to take the derivative of, and the second argument is the symbol by which to take the derivative:

```
print(sp.latex(y))
```

$$(x+\pi)^2$$

```
print(sp.latex(sp.diff(y**2, x)))
```

$$4(x+\pi)^3$$

For higher order derivatives we can do:

```
print(sp.latex(sp.diff(y**2, x, x)))
```

$$12(x+\pi)^2$$

```
print(sp.latex(sp.diff(y**2, x, 2))) # same as above
```

$$12\left(x+\pi\right)^{2}$$

To calculate the derivative of a multivariate expression, we can do:

```
x, y, z = sp.symbols("x,y,z")
```

```
f = sp.sin(x*y) + sp.cos(y*z)
```

```
print(sp.latex(sp.diff(f, x, 1, y, 2)))
```

$$-x\left(xy\cos\left(xy\right)+2\sin\left(xy\right)\right)$$

5.4.2 Integration

Integration is done in a similar fashion:

```
print(sp.latex(f))
```

$$\sin\left(xy\right)+\cos\left(yz\right)$$

```
print(sp.latex(sp.integrate(f, x)))
```

$$x\cos\left(yz\right)+\begin{cases}-\frac{\cos\left(xy\right)}{y} & \text{for}\: y\neq 0\\ 0 & \text{otherwise}\end{cases}$$

By providing limits for the integration variable we can evaluate definite integrals:

```
print(sp.latex(sp.integrate(f, (x, -1, 1))))
```

$$2\cos\left(yz\right)$$

and also improper integrals

```
print(sp.latex(sp.integrate(sp.exp(-x**2), (x, -sp.oo, sp.oo))))
```

$$\sqrt{\pi}$$

Remember, oo is the SymPy notation for inifinity.

5.4.3 Sums and Products

We can evaluate sums using the function Sum:

```
n = sp.Symbol("n")
```

```
print(sp.latex(sp.Sum(1/n**2, (n, 1, 10))))
```

$$\sum_{n=1}^{10} \frac{1}{n^2}$$

```
print(sp.latex(sp.Sum(1/n**2, (n,1, 10)).evalf()))
```

$$1.54976773116654$$

```
print(sp.latex(sp.Sum(1/n**2, (n, 1, sp.oo)).evalf()))
```

$$1.64493406684823$$

Products work much the same way:

```
print(sp.latex(sp.Product(n, (n, 1, 10)))) # 10!
```

$$\prod_{n=1}^{10} n$$

5.4.4 Limits

Limits can be evaluated using the `limit` function. For example,

```
print(sp.latex(sp.limit(sp.sin(x)/x, x, 0)))
```

$$1$$

We can use 'limit' to check the result of derivation using the `diff` function:

```
print(sp.latex(f))
```

$$\sin\left(xy\right) + \cos\left(yz\right)$$

```
print(sp.latex(sp.diff(f, x)))
```

$$y \cos\left(xy\right)$$

```
h = sp.Symbol("h")
```

```
print(sp.latex(sp.limit((f.subs(x, x+h) - f)/h, h, 0)))
```

$$y\cos(xy)$$

We can change the direction from which we approach the limiting point using the `dir` keyword argument:

```
print(sp.latex(sp.limit(1/x, x, 0, dir="+")))
```

$$\infty$$

```
print(sp.latex(sp.limit(1/x, x, 0, dir="-")))
```

$$-\infty$$

5.4.5 Series

Series expansion is also one of the most useful features of a CAS. In SymPy we can perform a series expansion of an expression using the series function:

```
print(sp.latex(sp.series(sp.exp(x), x)))
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + O\left(x^6\right)$$

By default it expands the expression around $x = 0$, but we can expand around any value of x by explicitly include a value in the function call:

```
print(sp.latex(sp.series(sp.exp(x), x, 1)))
```

$$e + e\left(x-1\right) + \frac{e\left(x-1\right)^2}{2} + \frac{e\left(x-1\right)^3}{6} + \frac{e\left(x-1\right)^4}{24} + \frac{e\left(x-1\right)^5}{120} + O\left(\left(x-1\right)^6 ; x \to 1\right)$$

And we can explicitly define to which order the series expansion should be carried out:

```
print(sp.latex(sp.series(sp.exp(x), x, 1, 5)))
```

$$e + e\left(x-1\right) + \frac{e\left(x-1\right)^2}{2} + \frac{e\left(x-1\right)^3}{6} + \frac{e\left(x-1\right)^4}{24} + O\left(\left(x-1\right)^5 ; x \to 1\right)$$

The series expansion includes the order of the approximation, which is very useful for keeping track of the order of validity when we do calculations with series expansions of different order:

```
s1 = sp.cos(x).series(x, 0, 5)
print(sp.latex(s1))
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} + O\left(x^5\right)$$

```
s2 = sp.sin(x).series(x, 0, 2)
print(sp.latex(s2))
```

$$x + O\left(x^2\right)$$

```
print(sp.latex(sp.expand(s1 * s2)))
```

$$x + O\left(x^2\right)$$

If we want to get rid of the order information we can use the removeO method:

```
print(sp.latex(sp.expand(s1.removeO() * s2.removeO())))
```

$$\frac{x^5}{24} - \frac{x^3}{2} + x$$

But note that this is not the correct expansion $\cos x \sin x$ of to 5th order:

```
(cos(x)*sin(x)).series(x, 0, 6)
```

5.5 Linear Algebra

Matrices are defined using the Matrix class:

```
m11, m12, m21, m22 = sp.symbols("m11, m12, m21, m22")
b1, b2 = sp.symbols("b1, b2")
```

```
A = sp.Matrix([[m11, m12],[m21, m22]])
print(sp.latex(A))
```

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$$

```
b = sp.Matrix([[b1], [b2]])
print(sp.latex(b))
```

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

With `Matrix` class instances we can do the usual matrix algebra operations:

```
print(sp.latex(A**2))
```

$$\begin{bmatrix} m_{11}^2 + m_{12}m_{21} & m_{11}m_{12} + m_{12}m_{22} \\ m_{11}m_{21} + m_{21}m_{22} & m_{12}m_{21} + m_{22}^2 \end{bmatrix}$$

```
print(sp.latex(A * b))
```

$$\begin{bmatrix} b_1 m_{11} + b_2 m_{12} \\ b_1 m_{21} + b_2 m_{22} \end{bmatrix}$$

And calculate determinants and inverses, and the like:

```
print(sp.latex(A.det()))
```

$$m_{11}m_{22} - m_{12}m_{21}$$

```
print(sp.latex(A.inv()))
```

$$\begin{bmatrix} \frac{m_{22}}{m_{11}m_{22}-m_{12}m_{21}} & -\frac{m_{12}}{m_{11}m_{22}-m_{12}m_{21}} \\ -\frac{m_{21}}{m_{11}m_{22}-m_{12}m_{21}} & \frac{m_{11}}{m_{11}m_{22}-m_{12}m_{21}} \end{bmatrix}$$

5.6 Solving equations

For solving equations and systems of equations we can use the `solve` function:

```
print(sp.latex(sp.solve(x**2 - 1, x)))
```

$$[-1,\ 1]$$

```
print(sp.latex(sp.solve(x**4 - x**2 - 1, x)))
```

$$\left[-i\sqrt{-\frac{1}{2} + \frac{\sqrt{5}}{2}},\ i\sqrt{-\frac{1}{2} + \frac{\sqrt{5}}{2}},\ -\sqrt{\frac{1}{2} + \frac{\sqrt{5}}{2}},\ \sqrt{\frac{1}{2} + \frac{\sqrt{5}}{2}}\right]$$

System of equations:

```
print(sp.latex(sp.solve([x + y - 1, x - y - 1], [x,y])))
```

$$\{x : 1,\ y : 0\}$$

In terms of other symbolic expressions:

```
print(sp.latex(sp.solve([x + y - a, x - y - c], [x,y])))
```

$$\left\{x : \frac{a}{2} + \frac{c}{2},\ y : \frac{a}{2} - \frac{c}{2}\right\}$$