Python for Operational Research & Statistics

Handbook

Dr Geraint Palmer

2024

Contents

1 Numerical Variables

Variables are pointers to Python objects. We can think of them as storing some value. There are two main types of numerical variables: integers (`int`), which act like the mathematical integers $\mathbb{Z}$, and floats (`float`), which act like the real numbers $\mathbb{R}$ expressed as decimals.

```
>>> a = 31
>>> type(a)
<class 'int'>
>>> b = 3.14
>>> type(b)
<class 'float'>
```

We can perform mathematical operations on either of these:

```
>>> 4 + 6.5
10.5
```

And combine these with the assignment operator `=` to increment a variable's value:

```
>>> c = 6.5
>>> c += 2.1
>>> c
8.6
```

A few operations are listed in the table below:

Operation	Description	Example

`+`	Addition	`>>> 4 + 6.5`
		`10.5`

`-`	Subtraction	`>>> 20 - 5`
		`15`

`*`	Multiplication	`>>> 3 * 7`
		`21`

`/`	Division	`>>> 63 / 6`
		`10.5`

`**`	Exponentiation	`>>> 5 ** 2`
		`25`

`%`	Modulo	`>>> 13 % 5`
		`3`

`abs`	Absolute value	`>>> abs(-8)`
		`8`

`round`	Rounding	`>>> round(34.78412, 2)`
		`34.78`

There is a special type of float that is useful to know about: `inf`, representing the concept of infinity, that is, a float greater than every other float:

```
>>> d = float('inf')
>>> d
inf
>>> type(d)
<class 'float'>
```

The modulo operator (`%`) can sometimes be confusing. It represents modulo, or clock, arithmetic. That is, what is the remainder when dividing one integer by another.

Floating point arithmetic

Python cannot represent *all* real numbers $\mathbb{R}$, there is an infinitely many of them, and only a finite amount of bits on the computer. For example, consider the number $1/3$, its decimal representation repeats forever, and so would require an infinite amount of bits to represent. Instead, Python approximates this number to 55 decimal places (and only displays 16 decimal places for convenience):

```
>>> 1 / 3
0.3333333333333333
```

This can cause some slight inaccuracies in arithmetic calculations, e.g. try:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Don't panic, this is normal, and doesn't cause too many problems in everyday coding, and there are workarounds for when it does.

2 Strings, Lists & Tuples

Strings

Strings are ordered sequences of characters, defined using quotations marks. Charachters include letters, both lower and upper case, numerals, punctuation marks, and whitespace:

```
>>> my_name = "Geraint"
>>> my_name
'Geraint'
```

Either double (`"`) or single (`'`) quotation marks can be used, but you need to be consistent. Using double quotation marks allows the single quotation mark to be a character in the string, and vice versa:

```
>>> double_quotes = "I'm hungry"
>>> single_quotes = 'The dog said "woof" all night'
```

Strings are *indexable*, meaning we can access specific characters within the string. Python begins counting from 0, so the 0th index is the first character, and so on:

```
>>> my_name = "Geraint"
>>> my_name[0]
'G'
>>> my_name[1]
'e'
>>> my_name[2]
'r'
>>> my_name[3]
'a'
```

We can also count backwards from the end of the string:

```
>>> my_name = "Geraint"
>>> my_name[-1]
't'
>>> my_name[-2]
'n'
```

And we can take slices of the string using `[i:j]`, going from the i-th character, to, but not including, the j-th character. Leaving out either i or j will assume the beginning or end of the string:

```
>>> my_name = "Geraint"
>>> my_name[2:5]
'rai'
>>> my_name[2:]
'raint'
>>> my_name[:5]
'Gerai'
>>> my_name[1:-3]
'era'
```

We can ask how many characters are in the string, that is the length of the string, with the `len` function. Note that white spaces count as distinct characters too:

```
>>> my_name = "Geraint Palmer"
>>> len(my_name)
14
```

Adding strings together concatenate them:

```
>>> first_name = "Geraint"
>>> surname = "Palmer"

>>> full_name = first_name + surname
>>> full_name
'GeraintPalmer'
```

However, we can only do this with two strings. In order to concatenate strings with other types, such as integers or floats, there are other ways to do this. One way is by using f-strings:

```
>>> number_of_cities = 7
>>> sentence = f"There are {number_of_cities} cities in Wales."
>>> sentence
'There are 7 cities in Wales.'
```

And we can format lists with `upper` and `lower`:

```
>>> randomly_capitalised = 'thIS striNG HaS soMe RANdom capITaLiSatioN'
>>> randomly_capitalised.upper()
'THIS STRING HAS SOME RANDOM CAPITALISATION'
>>> randomly_capitalised.lower()
'this string has some random capitalisation'
```

Note that string numerals and numeric variables are not the same:

```
>>> '7' == 7
False
>>> type('7'), type(7)
(<class 'str'>, <class 'int'>)
```

Lists

Lists are another type of Python object. Similar to strings, they are ordered sequences of objects, however these objects can be *anything*, including other lists.

```
>>> my_favourite_things = ["chocolate", 2, 22, 3.14159, "lemongrass"]
>>> my_favourite_things
['chocolate', 2, 22, 3.14159, 'lemongrass']
```

They are indexed the same as strings, and have a length like strings:

```
>>> len(my_favourite_things)
5
>>> my_favourite_things[3]
3.14159
>>> my_favourite_things[2:]
[22, 3.14159, 'lemongrass']
```

A list of lists:

```
>>> a_list_of_lists = [[0], [1, 2], [3, 4, 5, 6]]
>>> a_list_of_lists
[[0], [1, 2], [3, 4, 5, 6]]
>>> len(a_list_of_lists)
3
>>> a_list_of_lists[2]
[3, 4, 5, 6]
>>> len(a_list_of_lists[2])
4
>>> a_list_of_lists[2][-1]
6
```

We can add things to lists, using `append`, which adds one more element to the end of the list:

```
>>> uk_capitals = ['Belfast', 'Cardiff', 'Edinburgh']
>>> uk_capitals.append('London')
>>> uk_capitals
['Belfast', 'Cardiff', 'Edinburgh', 'London']
```

Similarly, we can remove things from lists. We will look at two ways of doing this: `remove` removes a certain element, and `pop` removes a given index. Note that pop also gives that removed element:

```
>>> prime_ministers = ['Cameron', 'May', 'Johnson', 'Truss', 'Sunak']
>>> prime_ministers.remove('Johnson')
>>> prime_ministers
['Cameron', 'May', 'Truss', 'Sunak']
>>> latest_pm = prime_ministers.pop(-1)
>>> prime_ministers
['Cameron', 'May', 'Truss']
>>> latest_pm
'Sunak'
>>> another_pm = prime_ministers.pop(1)
>>> prime_ministers
['Cameron', 'Truss']
>>> another_pm
'May'
```

Adding lists works as expected, and multiplying by an integer repeats a list:

```
>>> some_primes = [2, 3, 5]
>>> more_primes = [7, 11, 13]
>>> some_primes + more_primes
[2, 3, 5, 7, 11, 13]
>>> some_primes * 3
[2, 3, 5, 2, 3, 5, 2, 3, 5]
```

When lists contain all numerical variables, we can sort a list. This is done two ways: `sorted` gives a new list, and does not change the original, while `sort` changes the original:

```
>>> A = [7, 6, 1.5, -3, 5.5]
>>> B = sorted(A)
>>> B
[-3, 1.5, 5.5, 6, 7]
>>> A
[7, 6, 1.5, -3, 5.5]
>>> A.sort()
>>> A
[-3, 1.5, 5.5, 6, 7]
```

And when when lists contain all numerical values, we can find the maximum or minimum of a list:

```
>>> A = [7, 6, 1.5, -3, 5.5]
>>> max(A)
7
>>> min(A)
-3
```

Lists of *mutable*, which means we can change the entries of lists:

```
>>> uk_capitals = ['Belfast', 'Cardiff', 'Edinburgh', 'London']
>>> uk_capitals[1] = 'Swansea'
>>> uk_capitals
['Belfast', 'Swansea', 'Edinburgh', 'London']
```

But be careful, variables simply point to lists, and more than one variable can point to the *same* list, and so changing one list will change all variables pointing to that list:

```
>>> numbers = [1, 2, 3, 4, 5, 6]
>>> numbers_again = numbers

>>> numbers[2] = 999
>>> numbers_again
[1, 2, 999, 4, 5, 6]
```

Empty lists exist:

```
>>> empty_list = []
>>> empty_list
[]
>>> len(empty_list)
0
```

We can make lists from strings by splitting on a given character, or string of characters:

```
>>> kings_full_name = 'Charles Philip Arthur George'
>>> kings_names = kings_full_name.split(' ')
>>> kings_names
['Charles', 'Philip', 'Arthur', 'George']
```

And we can make strings from lists by joining on some character:

```
>>> list_of_foods = ['cheese', 'pork', 'bread', 'grapes']
>>> foods = ', and '.join(list_of_foods)
>>> foods
'cheese, and pork, and bread, and grapes'
```

Tuples

Tuples are similar to lists, but are less flexible, they are *immutable*, which means once created they cannot be changed. This means they can take less memory that lists and might be faster to work with:

```
>>> coordinate = (4.5, -1, 8.5)
>>> coordinate
(4.5, -1, 8.5)
>>> coordinate[2]
8.5
>>> len(coordinate)
3
>>> type(coordinate)
<class 'tuple'>
```

And we can similarly have tuples of tuples:

```
>>> tuple_of_tuples = ((0, 1), (2, 3))
>>> len(tuple_of_tuples)
2
>>> tuple_of_tuples[1]
(2, 3)
>>> tuple_of_tuples[1][1]
3
```

But items cannot be appended or removed.

Tuple unpacking is when we can assign values to multiple variables from a tuple, for example:

```
>>> coordinate = (4.5, -1, 8.5)
>>> a, b, c = coordinate
>>> a
4.5
>>> b
-1
>>> c
8.5
```

3 Booleans, If-Statements & While Loops

Booleans

Booleans are another type of variable, that take one of two values, either `True` or `False`. They are created from logical comparisons:

```
>>> 31 > 9
True
>>> 31 < 9
False
```

Variables can point to Booleans:

```
>>> a = 10 < 7
>>> a
False
>>> type(a)
<class 'bool'>
```

And `isinstance` gives a Boolean if a variable is a certain type:

```
>>> b = 35.1
>>> isinstance(b, float)
True
>>> isinstance(b, int)
False
```

A few operations that create Booleans are listed in the table below:

Operation	Description	Example

`==`	Equal to	`>>> 4 == 10`
		`False`

`!=`	Not equal to	`>>> 4 != 10`
		`True`

`<`	Less than	`>>> 1.2 < 3.14`
		`True`

`<=`	Less than or equal to	`>>> 8.5 <= 2`
		`False`

`>`	Greater than	`>>> 5 > 5`
		`False`

`>=`	Greater than or equal to	`>>> 7 >= 3.1`
		`True`

`in`	Is in	`>>> 2 in [1, 3, 8, 7, 0]`
		`False`

`is`	Is an object	`>>> None is None`
		`True`

Note that `is` is slightly different to `==`, is checks if two variables point to the same object, rather than if they both have the same value.

Booleans can be combined, giving other Booleans, by operating on them with logical operators:

Operation	Description	Example

`not`	Not	`>>> a = False`
		`>>> not a`
		`True`

`and`	And (are both True?)	`>>> b = True`
		`>>> c = False`
		`>>> b and c`
		`False`

`or`	Or (is either True?)	`>>> b = True`
		`>>> c = False`
		`>>> b or c`
		`True`

`all`	Are all True?	`>>> A = [True, False, True]`
		`>>> all(A)`
		`False`

`any`	Is any True	`>>> A = [True, False, True]`
		`>>> any(A)`
		`True`

As Boolean operators give Booleans, and operate on Booleans, they can be combined:

```
>>> unacceptable_ages = [21, 22, 23]
>>> maximum_age = 40
>>> age = 38
>>> (age not in unacceptable_ages) and (age < maximum_age)
True
```

If-Statements

If-statements (`if`) are ways to *only* run a piece of code if a Boolean is True:

```
>>> a = True
>>> b = 0.0
>>> if a:
...     b += 3.5
>>> b
3.5
```

and the code *won't* run if the Boolean is False:

```
>>> a = False
>>> b = 0.0
>>> if a:
...     b += 3.5
>>> b
0.0
```

The colon `:` is *required*. The whole indented is section is run or ignored depending on the Boolean, therefore the indents are also *required*.

If-statements can be extended with `else` and `elif` statements. Else-statements run only if all previous conditions of the if-statement was False:

```
>>> a = False
>>> b = 0.0
>>> if a:
...     b += 3.5
... else:
...     b += 2.1
>>> b
2.1
```

Elif-statements check another Boolean if the first condition was False:

```
>>> a = False
>>> b = True
>>> c = 1000
>>> if a:
...     c += 100
... elif b:
...     c += 10
... else:
...     c += 1
>>> c
1010
```

While Loops

Similar to if-statements, while loops only run a piece of indented code if a condition is True. However, if-statements run that code once. While loops repeatedly run that code until the condition becomes False.

This code continuously halves 1600 until it drops below 90:

```
>>> a = 1600
>>> while a > 90:
...     a /= 2
>>> a
50.0
```

In order to see what is happening here, let's print some information at every loop:

```
>>> a = 1600
>>> while a > 90:
...     a /= 2
...     print(a)
800.0
400.0
200.0
100.0
50.0
>>> a
50.0
```

4 For Loops

A for loop allows us to repeat an indented piece of code once for each element of an iterable. Iterables we have already seen include strings, lists, and tuples:

```
>>> numbers = [10, 4, 333, 1.2, 1.5, 0]
>>> for n in numbers:
...     print('Help me')
Help me
Help me
Help me
Help me
Help me
Help me
```

Above, there are 5 elements in the list `numbers`, so the print statement is repeated exactly 5 times. At each iteration of the loop, the current element is available as a variable:

```
>>> numbers = [10, 4, 333, 1.2, 1.5, 0]
>>> total = 0.0
>>> for n in numbers:
...     total += n
...     print((n, total))
(10, 10.0)
(4, 14.0)
(333, 347.0)
(1.2, 348.2)
(1.5, 349.7)
(0, 349.7)
```

We might not have an iterable at hand, but know we want to repeat something a number of times. in that case, we can use the `range` function, which gives a generator, an object that we can iterate over:

```
>>> for iteration in range(6):
...     square = iteration ** 2
...     print(square)
0
1
4
9
16
25
```

Notice that the first value given by range is 0, and iterates up to, but not including, 6. We can change the start point, end point, and step size. Range can take up to three values: `range(a, b, c)`, beginning at `a`, going up to but not including `b`, every `c` steps:

```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(3, 9))
[3, 4, 5, 6, 7, 8]
>>> list(range(4, 40, 7))
[4, 11, 18, 25, 32, 39]
```

The indented section of a for loop, that is the code to be repeated, can include anything, from defining variables, to if-statements, and other while or for loops. Sometimes these are called nested loops. E.g. the following code calculates

$$\sum_{i=0}^{10} \sum_{j=0}^{i} i^j$$

```
>>> total = 0.0
>>> for i in range(11):
...     for j in range(i + 1):
...         total += i ** j
>>> total
11567154206.0
```

List comprehension

List comprehension is a way to create a new list from another iterable using a for loop. Say we want a list of the first 10 square numbers:

```
>>> first_10_squares = [i ** 2 for i in range(1, 11)]
>>> first_10_squares
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

And of course we can create lists of lists in the same way:

```
>>> multiplication_table = [[i * j for j in range(4)] for i in range(5)]
>>> multiplication_table
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6], [0, 3, 6, 9], [0, 4, 8, 12]]
```

Sum comprehensions are similar, and allow us to sum the list we are creating with a list comprehension, without creating the list:

```
>>> sum_of_first_10_squares = sum(i ** 2 for i in range(1, 11))
>>> sum_of_first_10_squares
385
```

Generators

We have already seen the `range` generator, which takes in integers and gives something that will iterate over regularly spaced integers until a given value is reached. Two other important iterators are `enumerate` and `zip`, both of which give more than one value at each iteration.

The `enumerate` generator iterates over a list, while also giving the index of the current item:

```
>>> people = ['Rose', 'Martha', 'Donna', 'Amy']
>>> for i, person in enumerate(people):
...     print(person, i ** 2)
Rose 0
Martha 1
Donna 4
Amy 9
```

The `zip` generator takes multiple lists, and returns all elements with the same index, for example:

```
>>> monarchs = ['Charles III', 'Elizabeth II', 'George VI']
>>> consorts = ['Camilla', 'Philip', 'Elizabeth']
>>> for monarch, consort in zip(monarchs, consorts):
...     sentence = f"{monarch} married {consort}."
...     print(sentence)
Charles III married Camilla.
Elizabeth II married Philip.
George VI married Elizabeth.
```

Zip can take more than two lists, and if the lists are different sizes then it will only iterate until the shortest list is complete:

```
>>> A = [1, 2, 3, 4, 5]
>>> B = [10, 20, 30, 40, 50]
>>> C = [100, 200, 300]
>>> D = [a + b + c for a, b, c in zip(A, B, C)]
>>> D
[111, 222, 333]
```

5 Functions

Functions are variables that point to executable pieces of code. They are first *defined*, and then *called*. The executable piece of code is not run at the time the function is defined, but is ready to run on demand whenever the function is called.

Consider the function below that prints a sentence, here we define it:

```
>>> def print_proverb():
...     print("Practice makes perfect")
```

The indented lines (tht begin with 4 white spaces) are not run, they are the executable piece of code that will be run when the function is called. To call the function, we use the function name with open and closed round brackets after it:

```
>>> print_proverb()
Practice makes perfect
>>> print_proverb()
Practice makes perfect
```

Functions save repeating code.

In addition to being executable pieces of code, functions behave similarly to mathematical functions, they can take in one or more inputs, and give an output. Consider the function $f(x) = x^2$:

```
>>> def f(x):
...     y = x ** 2
...     return y
```

This function has one input, `x`, and when run, it will create a local variable `y` which takes on the value of the square of x, then will output y. E.g:

```
>>> f(3)
9
>>> f(4)
16
>>> f(-1.5)
2.25
>>> f(0.0)
0.0
```

Functions can take more than one input:

```
>>> def euclidean_distance(x, y):
...     hypotenuse = ((x ** 2) + (y ** 2)) ** 0.5
...     return hypotenuse

>>> euclidean_distance(5, 4)
6.4031242374328485
```

And to avoid confusion, it is good to use the input names when calling the function:

```
>>> euclidean_distance(x=5, y=4)
6.4031242374328485
```

If we do not give a function the inputs it requires, the Python will raise an error. However, if we want to, we can set default values for the inputs of a function, in the case where we do not wish to input all the variables all the time. Below, we tell the function that the default value of `x` is 1, and the default value of `y` is 2:

```
>>> def euclidean_distance(x=1, y=2):
...     hypotenuse = ((x ** 2) + (y ** 2)) ** 0.5
...     return hypotenuse
```

And so:

```
>>> euclidean_distance(x=5)
5.385164807134504

>>> euclidean_distance(y=3)
3.1622776601683795

>>> euclidean_distance()
2.23606797749979
```

Not all functions necessarily need to output, or `return` anything. They might modify, or report something, for example:

```
>>> def append_to_list_number_of_times(a_list, an_element, n_times):
...     for n in range(n_times):
...         a_list.append(an_element)

>>> B = [1, 2, 3]
>>> append_to_list_number_of_times(B, 9, 5)
>>> B
[1, 2, 3, 9, 9, 9, 9, 9]
```

Functions can be called from anywhere, including inside another function:

```
>>> def square(x):
...     return x ** 2

>>> def reciprocal_of_square(x):
...     return 1 / square(x)

>>> reciprocal_of_square(2)
0.25
```

And as functions are themselves variables or objects, they might even be an input to another function:

```
>>> def reciprocal_of_function(x, function):
...     return 1 / function(x)

>>> reciprocal_of_function(x=2, function=square)
0.25
```

Recursive Functions

Recursive functions are functions that call themselves. Mathematically they are equivalent to sequences that are defined recursively, e.g. the Fibonacci sequence:

$$\begin{aligned} a_0 &= 0 \\ a_1 &= 1 \\ a_n &= a_{n-1} + a_{n-2} \end{aligned}$$

That is, the n-th number in the Fibonacci sequence is defined only by knowing the n-1 and n-2-th numbers in the Fibonacci sequence, and so on. In Python:

```
>>> def fibonacci(n):
...     if n == 0:
...         return 0
...     if n == 1:
...         return 1
...     return fibonacci(n - 1) + fibonacci(n - 2)
```

Now calling the function `fibonacci` with $n = 2$ would require calling the same function with $n = 1$ and $n = 0$, giving 1 and 0 respectively, then adding them up:

```
>>> fibonacci(2)
1
```

And if for larger n, the function is called more and more times, recursively, until we simply get sums of the base cases:

```
>>> fibonacci(20)
6765
```

Though we should be careful about the number of times a recursive function is called, as the number of times might be 'hidden'. For example, each time the `fibonacci` function above is called when it isn't the base case, it is then called 2 more times, and so on. It just so happens, that here, when run `fibonacci` with the integer `n`, we actually run the function `fibonacci(n+1)` times!

Lambda Functions

Lambda functions are short, throwaway functions that we can use without needing to assign them to variables. They are usually used when we need to pass a function to another function.

For example, consider that we want a sort a list of numbers by the value given when plugging it into a polynomial. One way to do that is to define a function and use it like so:

```
>>> def polynomial(x):
...     return (x ** 2) - (4 * x) + 7

>>> numbers = [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6]
>>> sorted(numbers, key=polynomial)
[2, 1, 3, 0, 4, -1, 5, -2, 6, -3, -4]
```

Another way, without needing to define the variable `polynomial`, is to use a lambda function, like so:

```
>>> numbers = [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6]
>>> sorted(numbers, key=lambda x: (x ** 2) - (4 * x) + 7)
[2, 1, 3, 0, 4, -1, 5, -2, 6, -3, -4]
```

Here `lambda x:` says that we are creating a function with input `x`, and the expression after the colon is the output of the function.

Docstrings

Functions can be documented by writing *docstrings*, similar to comments, that explain the function's usage. They are placed after defining the function name and inputs, and before the main indented block of the function. We use three double quotes to begin and end docstrings. They should tell us three things:

- a description of what the function does,
- what inputs the functions takes and the expected types,
- what outputs the function gives, and the expected types.

For example:

```
>>> def square_a_number(x):
...     """
...     This function raises a number to the power of 2.
...
...     Parameters
...     x : float or int
...         The number to be squared.
...
...     Returns
...     float or int
...         The square of the input.
...     """
...     y = x ** 2
...     return y
```

Docstrings, along with descriptive function names written in full English, help others, and our future selves, understand more easily what the code is doing, ensuring that the work is clear, understandable, reproducible, and extendible.

6 Classes

In Python, everything is an object, and we have seen many of these previously, for example floats, strings, functions, and lists. For example lists: we can do the following:

```
>>> A = [6, 1, 2, 9]
>>> A.append(8)
>>> A.sort()
```

Here both `append` and `sort` are *methods* of the list object, that is functions that are attached to the list, and operate upon itself. Namely one adds an element, and the other rearranges the entries.

In general objects can have two types of thing associated with it:

- **Attributes**: these are variables associated with the object,
- **Methods**: these are functions associated with the object.

It is useful to define our own types of objects. These are called classes. Classes are a recipe for creating objects. It's in this recipe that we tell it what attributes and methods all objects of that class should have, but we can also add attributes manually.

Let's create an object to keep information about an actor:

```
>>> class Actor:
...     pass
```

At the moment this does nothing. We have not created any object, only the recipe for creating objects, similar to defining functions. So it now allows us to create an object and give it some attributes manually:

```
>>> leading_lady = Actor()
>>> leading_lady.name = "Judi Dench"
>>> leading_lady.date_of_birth = 1934

>>> leading_lady.name
'Judi Dench'
>>> leading_lady.date_of_birth
1934
```

And we can use the same recipe to create many more objects:

```
>>> villain = Actor()
>>> villain.name = "Anthony Hopkins"
>>> villain.name
'Anthony Hopkins'
```

Both are different objects created from the same recipe, the class `Actor`. We can create objects with inputs upon their creation, and initialise attributes using an `__init__` method:

```
>>> class Actor:
...     def __init__(self, name, date_of_birth):
...         self.name = name
...         self.date_of_birth = date_of_birth
```

Now we can create objects with pre-set attributes by giving it inputs:

```
>>> side_character = Actor("Michelle Yeoh", 1962)
>>> side_character.name
'Michelle Yeoh'
>>> side_character.date_of_birth
1962
```

Objects become more powerful when they have methods. Methods are defined like functions, but within the class. Their first argument is always `self`, which is a special argument and refers to the object itself that the method is attached to. In that way the method knows about the object, and can access the object's other attributes and methods. For example, consider a class for creating quadratic objects:

```
>>> class Quadratic:
...     def __init__(self, a, b, c):
...         self.a = a
...         self.b = b
...         self.c = c
...
...     def evaluate(self, x):
...         return (self.a * x ** 2) + (self.b * x) + self.c
```

The method `evluate` has one input, `x`, but can access the objects attributes `a`, `b` and `c` through `self`. A method is called like so:

```
>>> q = Quadratic(3, -3, 1)
>>> q.evaluate(2)
7
```

Like functions, methods do not just have to return things, they could perhaps be used to change the object attributes:

```
>>> class Quadratic:
...     def __init__(self, a, b, c):
...         self.a = a
...         self.b = b
...         self.c = c
...
...     def reciprocate(self):
...         """Swaps the a and c coefficients"""
...         self.a, self.c = self.c, self.a

>>> q = Quadratic(4, 5, 6)
>>> q.a, q.b, q.c
(4, 5, 6)
>>> q.reciprocate()
>>> q.a, q.b, q.c
(6, 5, 4)
```

Magic Methods

Note that `__init__` is itself a method, a special method that is run at the time of object creation:

```
>>> class Quadratic:
...     def __init__(self, a, b, c):
...         print('Hello')
...         self.a = a
...         self.b = b
...         self.c = c

>>> q = Quadratic(-2, 1, 0)
Hello
```

There are other special methods like this, called magic or dunder methods: the `__repr__` methods lets us define a representation of the object for when we wish to print out the object, and the `__add__` method allows us to add two object together with `+`:

```
>>> class Quadratic:
...     def __init__(self, a, b, c):
...         self.a = a
...         self.b = b
...         self.c = c
...
...     def __repr__(self):
...         return f"{self.a}x^2 + {self.b}x + {self.c}"
...
...     def __add__(self, q):
...         return Quadratic(self.a + q.a, self.b + q.b, self.c + q.c)

>>> q1 = Quadratic(1, 1, 1)
>>> q2 = Quadratic(3, 4, 6)
>>> q1
1x^2 + 1x + 1
>>> q2
3x^2 + 4x + 6
>>> q1 + q2
4x^2 + 5x + 7
```

Inheritance

Inheritance is a way of defining classes based off other classes. Usually there is a parent-child relationship between the classes, where the child class has all the methods and attributes of the parent, but with some extra or some modifications. Consider a class for a regular polygon:

```
>>> class RegularPolygon:
...     def __init__(self, side_length, sides):
...         self.side_length = side_length
...         self.sides = sides
...         self.interior_angle = ((self.sides - 2) / self.sides) * 180
...
...     def perimiter(self):
...         return self.side_length * self.sides

>>> pentagon = RegularPolygon(2, 5)
>>> pentagon.interior_angle
108.0
>>> pentagon.perimiter()
10
```

Now say we want another class for a square, with a method to calculate its area. As a square is also a regular polygon, we don't need to write it all again. A square can inherit from the `RegularPolygon` class:

```
>>> class Square(RegularPolygon):
...     def __init__(self, side_length):
...         self.side_length = side_length
...         self.sides = 4
...         self.interior_angle = ((self.sides - 2) / self.sides) * 180
...         self.name = 'square'
...
...     def area(self):
...         return self.side_length ** 2

>>> s = Square(12)
>>> s.area()
144
>>> s.perimiter()
48
```

Notice the Square inherits the `perimeter` method from the `RegularPolygon`, so there was no need to write that again. In fact, except for fixing the number of sides as 4, there is no need to re-write the `__init__` method either. We can just call the parent's `__init__` method instead. We access the parent class with `super()`:

```
>>> class Square(RegularPolygon):
...     def __init__(self, side_length):
...         super().__init__(side_length, 4)
...         self.name = 'square'
...
...     def area(self):
...         return self.side_length ** 2

>>> s = Square(12)
>>> s.sides
4
>>> s.interior_angle
90.0
>>> s.perimiter()
48
>>> s.area()
144
>>> s.name
'square'
```

Now notice that the square has all the attributes set in the parent's `__init__` method, plus any extra attributes set after `super().__init__` has run.

7 Dictionaries & Sets

Dictionaries

Dictionaries are another data structure available in Python, a type of hash table. Recall that lists are *ordered* collections of objects, indexable by their position in the list:

```
>>> capitals = ['Rome', 'Havana', 'Kyoto']
>>> capitals[1]
'Havana'
```

The order itself however might be meaningless, but we might want a more meaningful way to index the elements in our collection. Dictionaries are collections that map keys to values, allowing us to index them in a more meaningful way:

```
>>> capitals = {'Italy': 'Rome', 'Cuba': 'Havana', 'Japan': 'Kyoto'}
>>> capitals['Cuba']
'Havana'
>>> capitals['Italy']
'Rome'
```

Dictionaries are defined with curly brackets, key-value pairs are defined using a colon, and items separated with commas.

If we try to access a key that doesn't exist, Python will give an error. However sometimes this is unavoidable, e.g. if we do not know the contents of the dictionary. Another way of accessing keys is with `.get`, which also returns a default value if the key is not available. Here the key `'France'` is not in the dictionary, but `'Cuba'` is, and the default return will be `'Swansea'`:

```
>>> capitals.get('Cuba', 'Swansea')
'Havana'
>>> capitals.get('France', 'Swansea')
'Swansea'
```

We can explicitly check if a key is in the dictionary with the `in` operator. However, this only checks for keys, not values:

```
>>> 'Italy' in capitals
True
>>> 'Rome' in capitals
False
```

Therefore looping over a list only loops over the keys:

```
>>> for thing in capitals:
...     print(thing)
Italy
Cuba
Japan
```

We can extract just the keys, the values, or the key value pairs like so:

```
>>> capitals.keys()
dict_keys(['Italy', 'Cuba', 'Japan'])
>>> capitals.values()
dict_values(['Rome', 'Havana', 'Kyoto'])
>>> capitals.items()
dict_items([('Italy', 'Rome'), ('Cuba', 'Havana'), ('Japan', 'Kyoto')])
```

And add items to the dictionary:

```
>>> capitals['Spain'] = 'Madrid'
>>> len(capitals)
4
```

While the same operation can re-write the value of a specific key:

```
>>> capitals['Japan'] = 'Tokyo'
>>> capitals
{'Italy': 'Rome', 'Cuba': 'Havana', 'Japan': 'Tokyo', 'Spain': 'Madrid'}
```

Dictionary values can be anything: strings, floats, lists, other dictionaries, functions, objects, and so on. Two keys can even map to the same value. Keys, however, must be unique, and must be of specific, immutable, types: integers, floats, strings, Booleans, and tuples.

In the exact same way as list comprehension, dictionary comprehension allows us to create a dictionary from an iterable:

```
>>> squares = {x: x ** 2 for x in range(5)}
>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

>>> squares[3]
9
```

Sets

Sets are another data structure in Python, and act exactly like mathematical sets: they are unordered, cannot contain duplicates, and we can perform set operations on them. We define sets using curly brackets and commas:

```
>>> primes_under_20 = {2, 3, 5, 7, 11, 13, 17, 19}
>>> 5 in primes_under_20
True
>>> 6 in primes_under_20
False
```

They can be created from lists with `set`, and note that this will remove any duplicates:

```
>>> A = set([6, 7, 7, 1, 2, 2, 1, 1, 5])
>>> B = set([5, 6, 4, 11])
>>> A
{1, 2, 5, 6, 7}
>>> B
{11, 4, 5, 6}
```

We can operate on sets just like we would mathematically:

Operation	Description	Example

`union`	Union, $\cup$	`>>> A.union(B)`
		`{1, 2, 4, 5, 6, 7, 11}`

`intersection`	Intersection, $\cap$	`>>> A.intersection(B)`
		`{5, 6}`

`difference`	Set difference, $\setminus$	`>>> A.difference(B)`
		`{1, 2, 7}`

`issubset`	Checks if is a subset, $\subseteq$	`>>> A.issubset(B)`
		`False`

`add`	Adds an element to a set	`>>> A.add(222)`
		`>>> A`
		`{1, 2, 5, 6, 7, 222}`

`remove`	Removes an element from a set	`>>> A.remove(5)`
		`>>> A`
		`{1, 2, 6, 7, 222}`

8 Libraries

Libraries are external pieces of code, that usually define a number of objects, functions and classes, that can be imported into our code for our use. Some libraries are so common that they form part of the ‘standard library’, that is, they come already installed with Python itself. Others need to be installed manually.

Here we will look at three libraries from the standard library.

The `math` library

The `math` library gives us access to some mathematical constants and functions. To use a library, we must first import it:

```
>>> import math
```

This gives us access to a module object, and we can access various objects with `.`, for example three mathematical constants:

```
>>> math.pi
3.141592653589793

>>> math.e
2.718281828459045

>>> math.inf
inf
```

It also gives access to a number of standard mathematical functions, a few are listed here:

Function	Description	Example

`sin`	Sin (similar for other trig functions)	`>>> math.sin(math.pi / 4)`
		`0.7071067811865475`

`factorial`	Factorial, $n!$	`>>> math.factorial(10)`
		`3628800`

`ceil`	Ceiling, round up to integer (similar for `floor`)	`>>> math.ceil(75.3451)`
		`76`

`comb`	Combinations, ${}^nC_k = \binom{n}{r} = \frac{n!}{r!(n-r)!}$	`>>> math.comb(11, 4)`
		`330`

`exp`	Exponential function	`>>> math.exp(4.5)`
		`90.01713130052181`

`log`	Logarithmic function	`>>> math.log(25, 5)`
		`2.0`

`isinf`	Boolean checking if a variable is infinity	`>>> a = float('inf')`
		`>>> math.isinf(a)`
		`True`

The `random` library

The `random` library is a useful library for working with stochastic processes and simulations. It allows generates pseudorandom numbers of choices for us.

```
>>> import random
```

True randomness is not possible on a computer, but this library allows us to generate pseudorandom numbers. This is a deterministic sequence of numbers that look to us, and behave like random numbers, and the library uses these to generate seemingly random stuff. To begin this sequence, we have to set a seed, this ensures every time we run the code, we get the same sequence of random things, ensuring reproducibility. We can get different sequences by setting different random seeds.

E.g. `random.random` is a function that gives a pseudorandom number drawn from a uniform distribution between 0 and 1:

```
>>> random.seed(1)
>>> random.random()
0.13436424411240122
>>> [random.random() for n in range(3)]
[0.8474337369372327, 0.763774618976614, 0.2550690257394217]

>>> random.seed(1)
>>> random.random()
0.13436424411240122
>>> [random.random() for n in range(3)]
[0.8474337369372327, 0.763774618976614, 0.2550690257394217]

>>> random.seed(2)
>>> random.random()
0.9560342718892494
>>> [random.random() for n in range(3)]
[0.9478274870593494, 0.05655136772680869, 0.08487199515892163]
```

We can sample values from other distributions too:

- `random.expovariate(lambd=5.0)` :
  a random number from an Exponential distribution with rate parameter 5.
- `random.normalvariate(mu=6.5, sigma=1.2)` :
  a random number from a Normal distribution with mean 6.5 and standard deviation 1.2.
- `random.triangular(low=3.1, high=11.7, mode=5.0)` :
  a random number from a Triangular distribution with limits 3.1 and 11.7, and mode 5.
- `random.uniform(a=-4, b=1.9)` :
  a random number from a Uniform distribution with between -4 and 1.9.
- `random.randint(a=5, b=31)` :
  a random integer chosen with equal probability between 5 and 31.

Another useful function the library provides is to randomly choose elements from a list:

```
>>> things = [7, 22, 'ice cream', 4.1121]
>>> random.choice(things)
22
>>> random.choice(things)
'ice cream'
>>> random.choice(things)
'ice cream'
>>> random.choice(things)
22
```

And we can lists can be shuffled:

```
>>> things = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> random.shuffle(things)
>>> things
[8, 1, 4, 7, 6, 5, 3, 2, 0, 9]
>>> random.shuffle(things)
>>> things
[5, 9, 1, 7, 4, 2, 3, 8, 6, 0]
```

The `itertools` library

The `itertools` library provides ways of creating generators that we can iterate over. For example the `combinations` function gives a generator of all combinations of a given size of elements from a list:

```
>>> import itertools
>>> people = ['Judi', 'Anthony', 'Michelle']
>>> for pair in itertools.combinations(people, r=2):
...     print(pair)
('Judi', 'Anthony')
('Judi', 'Michelle')
('Anthony', 'Michelle')
```

Here are some other examples:

Generator | Example

`accumulate`
Running sum of a list

```
>>> A = [10, 4, 2, 1.5, 22]
>>> [i for i in itertools.accumulate(A)]
[10, 14, 16, 17.5, 39.5]
```

`pairwise`
Each consecutive pair

```
>>> B = [1, 2, 3, 4, 5]
>>> [i for i in itertools.pairwise(B)]
[(1, 2), (2, 3), (3, 4), (4, 5)]
```

`product`
All pairings in two lists

```
>>> C = ['A', 'B']
>>> D = [1, 2]
>>> [i for i in itertools.product(C, D)]
[('A', 1), ('A', 2), ('B', 1), ('B', 2)]
```

`combinations`
All combinations size r

```
>>> E = ['A', 2, 'gamma']
>>> [i for i in itertools.combinations(E, r=2)]
[('A', 2), ('A', 'gamma'), (2, 'gamma')]
```

`permutations`
All permutations size r

```
>>> F = '123'
>>> generator = itertools.permutations(F, r=3)
>>> [''.join(i) for i in generator]
['123', '132', '213', '231', '312', '321']
```

Others

There are a number of other libraries in the standard library we can use. Sometimes we might only need one function or object from a library. In that case we can import just that one thing. For example, once useful object is the `Counter` object from the `collections` library: it counts the number of occurrences of an element in a container:

```
>>> from collections import Counter
>>> word = 'mississippi'
>>> Counter(word)
Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```

Finally dates and times can be awkward, and the `datetime` library helps with that. For example, working out when I return from holiday:

```
>>> import datetime
>>> holiday_start = datetime.datetime(year=2022, month=12, day=20)
>>> holiday_duration = datetime.timedelta(days=14)
>>> holiday_start + holiday_duration
datetime.datetime(2023, 1, 3, 0, 0)
```

9 Data Frames

The Pandas library is a tool for conducting data analysis in Python, by giving us data frame objects. These object work similarly to data frames in R. It is not part of the standard library, so may need to be installed using `pip`. By convention, we usually import Pandas using the alias `pd`. It is good to stick to conventions in order for your code to me more accessible to the wider Python using community:

```
>>> import pandas as pd
```

We can create a data frame object from a dictionary. This dictionary will have column names as keys, and equal sized lists as values, representing the columns themselves:

```
>>> data_dictionary = {
...     'Name': ['Anna', 'Barry', 'Carys', 'Dave', 'Elin'],
...     'Age': [62, 78, 59, 66, 68],
...     'Hometown': ['Penarth', 'Penarth', 'Rumney', 'Penarth', 'Rumney']
... }
>>> data = pd.DataFrame(data_dictionary)
>>> data
```

	Name	Age	Hometown
0	Anna	62	Penarth
1	Barry	78	Penarth
2	Carys	59	Rumney
3	Dave	66	Penarth
4	Elin	68	Rumney

We usually create data frames by reading in from a file. In this case, Jupyter might not be able to display the whole file:

```
>>> titanic = pd.read_csv('titanic.csv')
>>> titanic
```

	Name	PClass	Age	Sex	Survived
0	Allen, Miss Elisabeth Walton	1st	29.00	female	1
1	Allison, Miss Helen Loraine	1st	2.00	female	0
2	Allison, Mr Hudson Joshua Creighton	1st	30.00	male	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.00	female	0
4	Allison, Master Hudson Trevor	1st	0.92	male	1
...	...	...	...	...	...
1308	Zakarian, Mr Artun	3rd	27.00	male	0
1309	Zakarian, Mr Maprieder	3rd	26.00	male	0
1310	Zenni, Mr Philip	3rd	22.00	male	0
1311	Lievens, Mr Rene	3rd	24.00	male	0
1312	Zimmerman, Leo	3rd	29.00	male	0

1313 rows × 5 columns

Accessing columns & rows

Columns can be accessed individually using square brackets and the string of the column name:

```
>>> titanic['Age']
0       29.00
1        2.00
2       30.00
3       25.00
4        0.92
        ...
1308    27.00
1309    26.00
1310    22.00
1311    24.00
1312    29.00
Name: Age, Length: 1313, dtype: float64
```

And from these we can get a number of summary statistics:

Method	Description	Example

`.mean()`	Mean	`>>> titanic['Age'].mean()`
		`30.397989417989418`

`.median()`	Median	`>>> titanic['Age'].median()`
		`28.0`

`.var()`	Variance	`>>> titanic['Age'].var()`
		`203.32047012439116`

`.std()`	Standard deviation	`>>> titanic['Age'].std()`
		`14.259048710359018`

`.sum()`	Sum	`>>> titanic['Age'].sum()`
		`22980.88`

`.skew()`	Skewness	`>>> titanic['Age'].skew()`
		`0.36851087371648295`

`.quantile(p)`	A proportion p of the data is less than or equal to this value	`>>> titanic['Age'].quantile(0.25)`
		`21.0`
		`>>> titanic['Age'].quantile(0.9)`
		`50.0`

Rows can also be accessed using `.loc` and square brackets around the index of the row. To get the row at index 777:

```
>>> titanic.loc[777]
Name        Dorkings, Mr Edward Arthur
PClass                             3rd
Age                               19.0
Sex                               male
Survived                             1
Name: 777, dtype: object
```

And similarly, individual entries of the data frame are accessed with in the same way, specifying both row index and column:

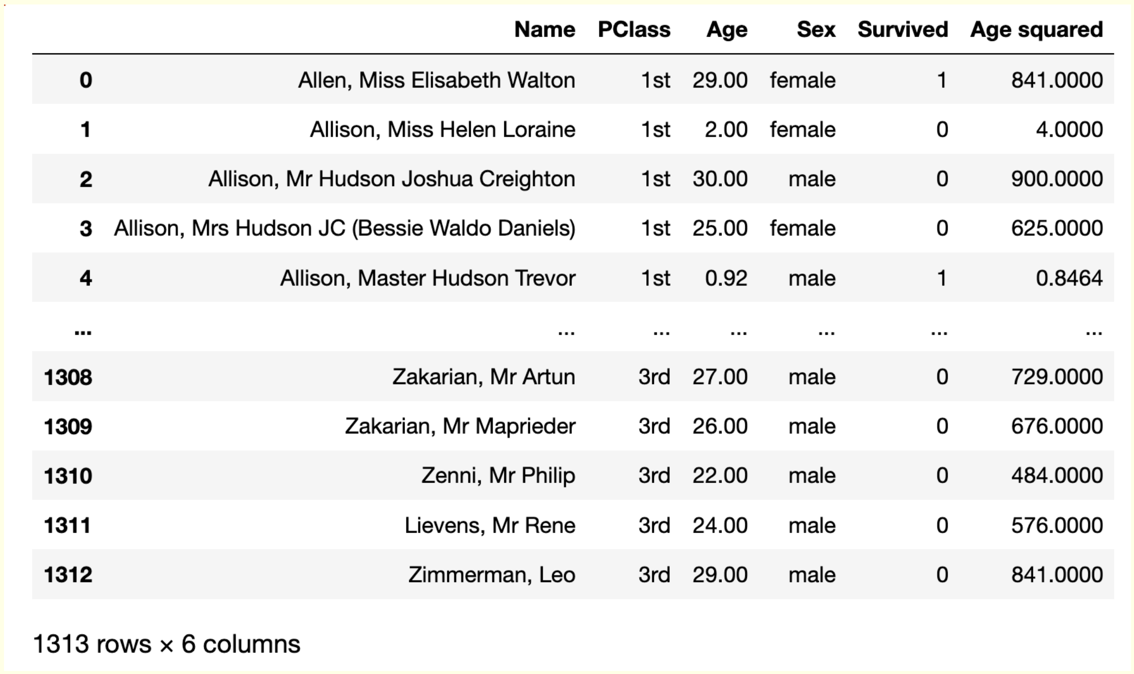
```
>>> titanic.loc[777, 'Sex']
'male'
>>> titanic.loc[30, 'Name']
'Bowen, Miss Grace Scott'
```

Columns act as vectors, and so we can operate on them as we would variables:

```
>>> titanic['Age'] ** 2
0       841.0000
1         4.0000
2       900.0000
3       625.0000
4         0.8464
          ...
1308    729.0000
1309    676.0000
1310    484.0000
1311    576.0000
1312    841.0000
Name: Age, Length: 1313, dtype: float64
```

And we can create new columns from these, just like adding items to a dictionary:

```
>>> titanic['Age squared'] = titanic['Age'] ** 2
>>> titanic
```



	Name	PClass	Age	Sex	Survived	Age squared
0	Allen, Miss Elisabeth Walton	1st	29.00	female	1	841.0000
1	Allison, Miss Helen Loraine	1st	2.00	female	0	4.0000
2	Allison, Mr Hudson Joshua Creighton	1st	30.00	male	0	900.0000
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.00	female	0	625.0000
4	Allison, Master Hudson Trevor	1st	0.92	male	1	0.8464
...	...	...	...	...	...	...
1308	Zakarian, Mr Artun	3rd	27.00	male	0	729.0000
1309	Zakarian, Mr Maprieder	3rd	26.00	male	0	676.0000
1310	Zenni, Mr Philip	3rd	22.00	male	0	484.0000
1311	Lievens, Mr Rene	3rd	24.00	male	0	576.0000
1312	Zimmerman, Leo	3rd	29.00	male	0	841.0000

1313 rows × 6 columns

Filtering

We can create columns of Booleans by using conditional operators:

```
>>> titanic['Age'] > 25
0        True
1       False
2        True
3       False
4       False
         ...
1308     True
1309     True
1310    False
1311    False
1312     True
Name: Age, Length: 1313, dtype: bool
```

Another way to create an array of Booleans is to check if the elements are in some collection:

```
>>> titanic['PClass'].isin(['2nd', '3rd'])
0       False
1       False
2       False
3       False
4       False
        ...
1308     True
1309     True
1310     True
1311     True
1312     True
Name: PClass, Length: 1313, dtype: bool
```

We can filter, or take slices of a data frame, based on these arrays of Booleans. For example, to only get the rows with an age greater than 70:

```
>>> titanic[titanic['Age'] > 70]
```

	Name	PClass	Age	Sex	Survived	Age squared
9	Artagaveytia, Mr Ramon	1st	71.0	male	0	5041.0
119	Goldschmidt, Mr George B	1st	71.0	male	0	5041.0
505	Mitchell, Mr Henry Michael	2nd	71.0	male	0	5041.0

And this is itself a data frame, so we can use the same operations as before:

```
>>> titanic[titanic['Age'] > 70]['Age'].mean()
71.0

>>> titanic[titanic['Sex'] == 'male']['Age'].median()
29.0
```

Thinking of arrays of Booleans as ways of filtering a data frame, it is useful to combine conditional statements with logical operators. However, logical operators on arrays of Booleans do not use the same syntax Boolean variables. For demonstration, consider a data frame of two Boolean columns:

```
>>> demo = pd.DataFrame({
...     'A': [True, True, False, False],
...     'B': [True, False, True, False]
... })
```

The Pandas equivalents of `and`, `or` and `not` are `&`, `|` and `~` respectively:

Operator	Description	Example

`&`	And	`>>> demo['A'] & demo['B']`
		`0     True`
		`1    False`
		`2    False`
		`3    False`
		`dtype: bool`

`|`	Or	`>>> demo['A'] | demo['B']`
		`0     True`
		`1     True`
		`2     True`
		`3    False`
		`dtype: bool`

`~`	Not	`>>> ~demo['A']`
		`0    False`
		`1    False`
		`2     True`
		`3     True`
		`Name: A, dtype: bool`

Further analysis

Panda data frames have a large number of very useful methods that help with our data analyses.

The `value_counts` method counts the number of times a value appears in the data frame:

```
>>> titanic['PClass'].value_counts()
PClass
3rd    711
1st    322
2nd    279
*        1
Name: count, dtype: int64
```

The `groupby` methods greats a useful object, that allows analysis to be split by rows with common values in a specific column. Alone this does not do much, but we can perform analysis on this object as we would a data frame:

```
>>> titanic.groupby('PClass')['Age'].mean()
PClass
*            NaN
1st    39.667788
2nd    28.300142
3rd    25.208585
Name: Age, dtype: float64
```

We can use most of the same methods on `groupby` objects as with data frame objects. We can group by more than one column too:

```
>>> titanic.groupby(['PClass', 'Sex'])['Age'].median()
PClass  Sex
*       male       NaN
1st     female    38.0
        male      42.0
2nd     female    28.0
        male      28.0
3rd     female    22.0
        male      25.5
Name: Age, dtype: float64
```

Combining data frames

Combining data frames can be tricky. We might want to add rows, or add columns. We need different methods for each of these. Consider three data frames we might want to merge together:

```
>>> data1 = pd.DataFrame({
...     'species': ['Dog', 'Ant', 'Octopus'],
...     'n_limbs': [4, 6, 8]
... })
>>> data1
```

	species	n_limbs
0	Dog	4
1	Ant	6
2	Octopus	8

```
>>> data2 = pd.DataFrame({
...     'species': ['Centipede', 'Cat', 'Mouse', 'Starfish'],
...     'n_limbs': [100, 4, 4, 5]
... })
>>> data2
```

	species	n_limbs
0	Centipede	100
1	Cat	4
2	Mouse	4
3	Starfish	5

```
>>> data3 = pd.DataFrame({
...     'species': ['Dog', 'Ant', 'Octopus'],
...     'diet': ['omnivore', 'omnivore', 'carnivore']
... })
>>> data3
```

	species	diet
0	Dog	omnivore
1	Ant	omnivore
2	Octopus	carnivore

Data frames `data1` and `data2` contain the same columns, but different rows, so we would want to combine these by stacking them on top of one another. We can do that with `concat`:

```
>>> pd.concat([data1, data2])
```

	species	**n_limbs**
0	Dog	4
1	Ant	6
2	Octopus	8
0	Centipede	100
1	Cat	4
2	Mouse	4
3	Starfish	5

There's a problem here though: the index has been repeated, making it difficult to select rows. To rectify this we would need to `reset_index`, this creates a new index column, giving each row a new, unique, index, and retaining the old indices as a new column:

```
>>> pd.concat([data1, data2]).reset_index()
```

	index	**species**	**n_limbs**
0	0	Dog	4
1	1	Ant	6
2	2	Octopus	8
3	0	Centipede	100
4	1	Cat	4
5	2	Mouse	4
6	3	Starfish	5

Data frames `data1` and `data3` contain the same rows, but with different columns, so we would want to combine these by stacking them side by side. We can do that with `merge`:

```
>>> pd.merge(data1, data3)
```

	species	n_limbs	diet
0	Dog	4	omnivore
1	Ant	6	omnivore
2	Octopus	8	carnivore

10 Statistics

Summary Statistics

The `statistics` library, part of the standard library, has some basic functions for performing summary statistics. It is useful for quick calculations when not dealing with data frames with Pandas. For example:

```
>>> import statistics as st
>>> X = [1.66, 1.53, 1.69, 1.69, 1.79, 1.7, 1.61, 1.55, 1.63, 1.76, 1.82]
>>> Y = [72.3, 69, 72.1, 74.0, 77.5, 70.8, 68.0, 70.0, 69.8, 73.2, 77.8]
```

Then it has the following functions for summary statistics:

Function	Description	Example
`mean`	Mean (Similarly `median`)	`>>> st.mean(X)` `1.6754545454545455`
`variance`	Sample variance (Similarly `stdev`)	`>>> st.variance(X)` `0.008567272727272727`
`covariance`	Covariance	`>>> st.covariance(X, Y)` `0.2615363636363636`
`correlation`	Pearson's correlation coefficient	`>>> st.correlation(X, Y)` `0.8755847867646794`

We can also find a line of best fit with the `linear_regression` function:

```
>>> st.linear_regression(X, Y)
LinearRegression(slope=30.527376910016976, intercept=21.08004032258065)
```

Here giving a model of $Y = 30.5X + 21.1$.

Probability distributions

The `scipy.stats` library has a large number of probability distribution objects. This lets us calculate specific values of their pdf, cdf, and inverse cdfs, as well as summary statistics.

To create an object representing the Normal distribution with mean 5 and standard deviation 2, use:

```
>>> import scipy.stats
>>> NormalDist = scipy.stats.norm(loc=5, scale=2)
```

Then to get it's measures of centrality, spread, and shape:

```
>>> mean, variance, skew, kurtosis = NormalDist.stats(moments='mvsk')
>>> mean
5.0
>>> variance
4.0
>>> skew
0.0
>>> kurtosis
0.0
```

We can evaluate specific values of its probability density function:

```
>>> NormalDist.pdf(2.1)
0.06971528322268014
>>> NormalDist.pdf(5.5)
0.19333405840142465
```

And, more usefully, specific values of its cumulative distribution function:

```
>>> NormalDist.cdf(2.1)
0.07352925960964836
>>> NormalDist.cdf(5.5)
0.5987063256829237
>>> NormalDist.cdf(13)
0.9999683287581669
```

That is the probability of a value being 5.5 smaller is 0.5987. Or, if it is known that some observations follow this distribution, then 59.87% of all values are less than or equal to 5.5.

It is also useful to know the opposite, what value are 59.87% of all observations less than? We can use the inverse cdf, or `ppf` to find that:

```
>>> NormalDist.ppf(0.0735)
2.0995802346196806
>>> NormalDist.ppf(0.5987)
5.499967281140009
>>> NormalDist.ppf(0.9999)
12.438032970911419
```

Some useful probability distributions for operational research that are included in the library include:

- `scipy.stats.expon`
- `scipy.stats.gamma`
- `scipy.stats.lognorm`
- `scipy.stats.norm`
- `scipy.stats.t`
- `scipy.stats.binom`
- `scipy.stats.geom`
- `scipy.stats.poisson`

Statistical testing

The `scipy.stats` library can be used for conducting hypothesis tests too. In order to demonstrate this, let's make three sample data sets `A`, `B` and `C`. Here `B` and `C` are drawn from the same populations and `A` from another, and `A` and `C` are correlated, but `A` and `B` not:

```
>>> A = [2, 9, 6, 5, 7, 14, 11, 6, 6, 9, 3, 9, 4, 1, 3, 3, 4, 7, 1, 6, 3]
>>> B = [1, 4, 1, 1, 6, 2, 4, 3, 4, 1, 1, 4, 1, 5, 2, 5, 3, 0, 2, 4, 4]
>>> C = [1, 5, 2, 1, 2, 6, 6, 3, 3, 1, 0, 7, 3, 2, 0, 5, 2, 3, 3, 5, 1]
```

- *Testing centrality against a value:*

  For a sample drawn from a Normally distributed population, we can use a one-sample t-test. The null hypothesis is that the population mean is equal to a given value, and the alternative is that it isn't equal.

  Testing if the mean of the population that `A` was drawn from is equal to $\mu = 5$ and $\mu = 3$:

```
>>> test5 = scipy.stats.ttest_1samp(A, 5)
>>> test5.pvalue
0.3749553181816273

>>> test3 = scipy.stats.ttest_1samp(A, 3)
>>> test3.pvalue
0.0016681456447453214
```

  For $\mu = 5$ there is not enough evidence to reject the null hypothesis at the 5% level (as the p-value is greater than 0.05). For $\mu = 3$ we reject the null hypothesis at the 5% level (as the p-value is less than 0.05) and conclude that $\mu \neq 3$

- *Comparing centralities, 2 samples:*

  For two samples drawn from Normally distributed populations, we can use a two-sample t-test. The null hypothesis is that the population means are equal, and the alternative is that they are not equal.

  Testing if the means of the populations that `A` and `B` were drawn from, and that `B` and `C` were drawn from, are equal:

```
>>> testAB = scipy.stats.ttest_ind(A, B)
>>> testAB.pvalue
0.001063756284957039

>>> testAC = scipy.stats.ttest_ind(B, C)
>>> testAC.pvalue
0.8069268481224856
```

  For `A` and `B`, we reject the null hypothesis at the 5% level and conclude their populations do not have equal means. For `B` and `C`, we cannot reject the null hypothesis at the 5% level.

If the samples were not drawn from Normally distributed population, then a non-parametric test is required. The equivalent non-parametric test is the Mann-Whitney U test. The null hypothesis is that the population *medians* are equal, and the alternative is that they are not equal:

```
>>> testAB = scipy.stats.mannwhitneyu(A, B)
>>> testAB.pvalue
0.0028338040832376872

>>> testAC = scipy.stats.mannwhitneyu(B, C)
>>> testAC.pvalue
0.9187262121270814
```

- *Comparing centralities, more than 2 samples:*

For more than two samples drawn from Normally distributed populations, we can use a one-way ANOVA test. The null hypothesis is that all the population means are equal, and the alternative is that at least one is not equal to at least on of the others.

Testing if the means of the populations that `A`, `B` and `C` were drawn from are equal:

```
>>> testABC = scipy.stats.f_oneway(A, B, C)
>>> testABC.pvalue
0.0003343519884107473
```

We reject the null hypothesis at the 5% level and conclude at least one of the populations do not have equal to at least one other population. We saw that this was true as a two-sample t-test showed that `A` and `B` were drawn from populations with different means.

If the samples were not drawn from Normally distributed population, then a non-parametric test is required. The equivalent non-parametric test is the Kruskal-Wallis test. The null hypothesis is that all the population *medians* are equal, and the alternative is that there is at least one median not equal to another:

```
>>> testABC = scipy.stats.kruskal(A, B, C)
>>> testABC.pvalue
0.0029207603981587934
```

- *Testing correlations:*

  For a sample drawn from a Normally distributed population, and the relationship we want to measure is linear, then we can measure and test the correlation between the populations by using the Pearson correlation. The correlation coefficient is a number ranging from -1 to +1, with -1 indicating a strong negative correlation, +1 indicating a strong positive correlation, and 0 indicating no correlation. The hypothesis test roughly corresponds to: the null hypothesis being that the correlation coefficient of the population is equal to 0, and the alternative is that it isn't equal to 0.

  Measuring and testing the Pearson correlation between `A` and `B`, and `A` and `C`:

```
>>> testAB = scipy.stats.pearsonr(A, B)
>>> testAB.statistic
0.03785546389774479
>>> testAB.pvalue
0.8705886352818722

>>> testAC = scipy.stats.pearsonr(A, C)
>>> testAC.statistic
0.611869423305717
>>> testAC.pvalue
0.003201123015985214
```

  So `A` and `B` have a small correlation, but we cannot reject the null hypothesis that it is 0. `A` and `C` have a relatively positive correlation, and we reject the null hypothesis at the 5% level, concluding that this correlation is not equal to 0 in the population.

  If the samples were not drawn from Normally distributed population, or if the relationship we are measuring is not linear, then a non-parametric test is required. The equivalent non-parametric test is the Spearman correlation test:

```
>>> testAB = scipy.stats.spearmanr(A, B)
>>> testAB.statistic
0.03964258894405297
>>> testAB.pvalue
0.8645322686132907

>>> testAC = scipy.stats.spearmanr(A, C)
>>> testAC.statistic
0.5437421480946801
>>> testAC.pvalue
0.010839218779198412
```

- *Testing independence:* We can test if two categorical variables are independent of one another using the χ^2 test of independence. The null hypothesis is that the variables are independent, while the alternative is that they are not. Look at the two contingency tables below: in one, smoking and cancer are clearly not independent, and in the other eye colour and sex are independent:

	Smokes	Doesn't Smoke
Cancer	45	18
No Cancer	610	580

	Male	Female
Brown Eyes	71	76
Blue Eyes	58	56
Green Eyes	12	12

With Python, we can represent these as lists of lists, and test for independence:

```
>>> smoking = [
...     [45, 18],
...     [610, 580]
... ]
>>> test = scipy.stats.chi2_contingency(smoking)
>>> test.pvalue
0.002754578438069624

>>> eyes = [
...     [71, 76],
...     [58, 56],
...     [12, 12]
... ]
>>> test = scipy.stats.chi2_contingency(eyes)
>>> test.pvalue
0.9168622674505268
```

This library contains a very large number of hypothesis tests including testing of normality (`normaltest`), the binomial test (`binomtest`), and tests associated with linear regression (`linregress`).

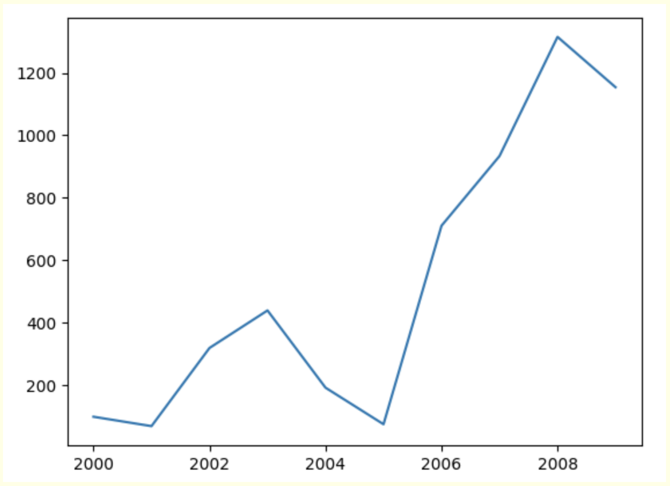
11 Data Visualisation

The library for producing plots in Python is `matplotlib`. It is a large library with numerous options for creating lots of different kinds of plots, including plot customisation. To create a plot we fist need to import the library, create a then figure and axis object. Then the plotting can begin, and afterwards we must show or save the plot:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1)
>>>
>>> plt.show()
>>> fig.savefig('my_plot.pdf')
```

Before showing the figure, we edit the axis to include any data visualisations we want, using methods on the axis. These usually take in lists or arrays of data. For example, a line plot:

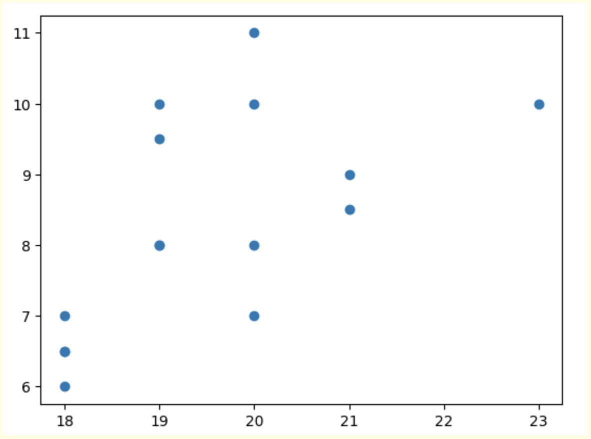
```
>>> years = [2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009]
>>> measles = [100, 70, 320, 440, 193, 76, 711, 934, 1315, 1154]
>>> fig, ax = plt.subplots(1)
>>> ax.plot(years, measels)
>>> plt.show()
```



In line plots, the order of the lists matter: first as the x and y coordinates of the line are paired via the list index; and also because line will join two consecutive points in the order of the list. It plots ratio or interval data (x-axis) against ratio data (y-axis), when there is a one-to-one relationship.

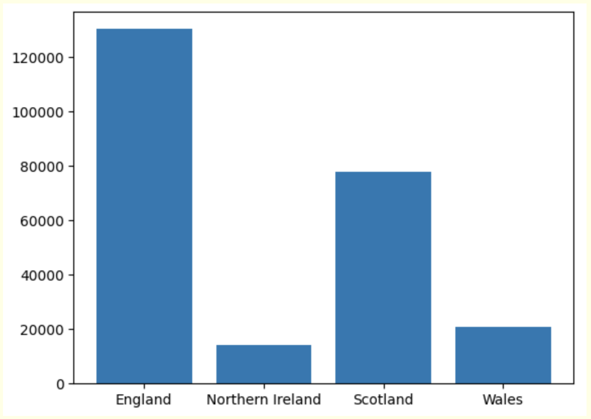
When order doesn't matter, but coordinates pairing does, we can use a scatter plot. This plots ratio data (x-axis) against ratio data (y-axis).

```
>>> months = [18, 19, 18, 21, 18, 21, 19, 18, 23, 20, 19, 20, 20, 19, 20]
>>> shoe_size = [6, 8, 6.5, 8.5, 6.5, 9, 8, 7, 10, 8, 9.5, 10, 7, 10, 11]
>>> fig, ax = plt.subplots(1)
>>> ax.scatter(months, shoe_size)
>>> plt.show()
```



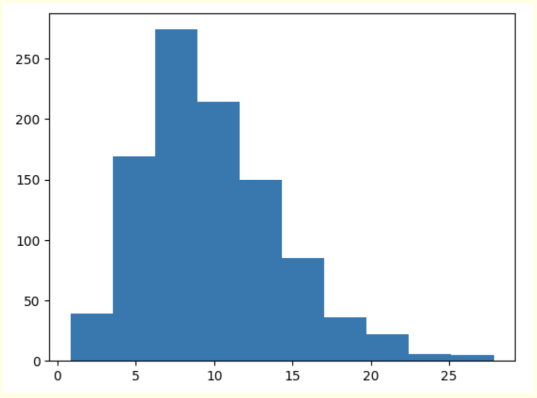
Bar charts plots categorical data (x-axis) against ratio data (y-axis):

```
>>> countries = ['England', 'Northern Ireland', 'Scotland', 'Wales']
>>> areas = [130279, 14130, 77933, 20779]
>>> fig, ax = plt.subplots(1)
>>> ax.bar(countries, areas)
>>> plt.show()
```



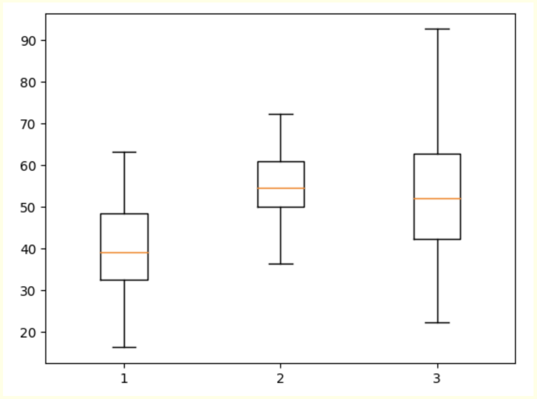
A histogram is for displaying the distribution of on ratio variable:

```
>>> import random
>>> gamma_dist = [random.gammavariate(5, 2) for _ in range(1000)]
>>> fig, ax = plt.subplots(1)
>>> ax.hist(gamma_dist)
>>> plt.show()
```



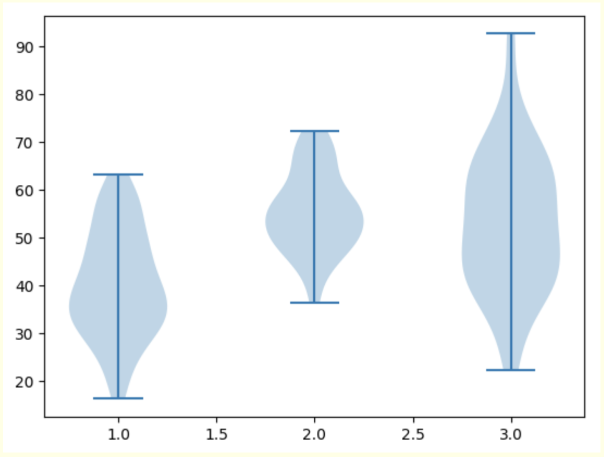
Comparing distributions with histograms is difficult, so we instead use box plots. A box plot is a visual representation of a distributions summary statistics, with the box's lower and upper limits extending from the first quartile to the third quartile, it's whiskers extending from its minimum to maximum value, and a horizontal line representing the median. Sometimes outliers are not included and plotted as fliers. They can be plotted side-by-side for comparison:

```
>>> distributions = [
...     [random.normalvariate(40, 10) for _ in range(100)],
...     [random.normalvariate(55, 8) for _ in range(60)],
...     [random.normalvariate(50, 14) for _ in range(75)]
... ]
>>> fig, ax = plt.subplots(1)
>>> ax.boxplot(distributions)
>>> plt.show()
```



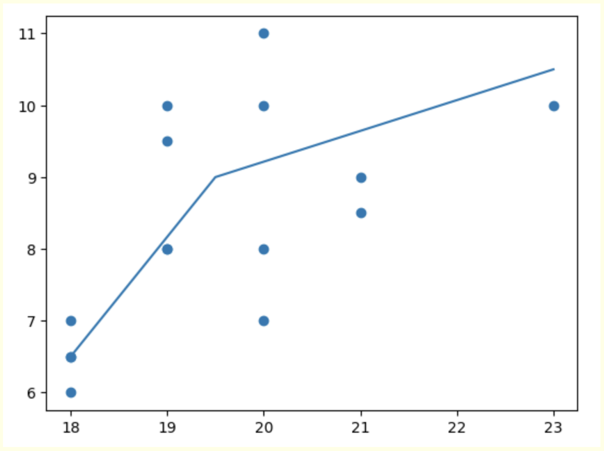
Similarly, violin plots visualise and compare the shape of the distributions:

```
>>> fig, ax = plt.subplots(1)
>>> ax.violinplot(distributions)
>>> plt.show()
```



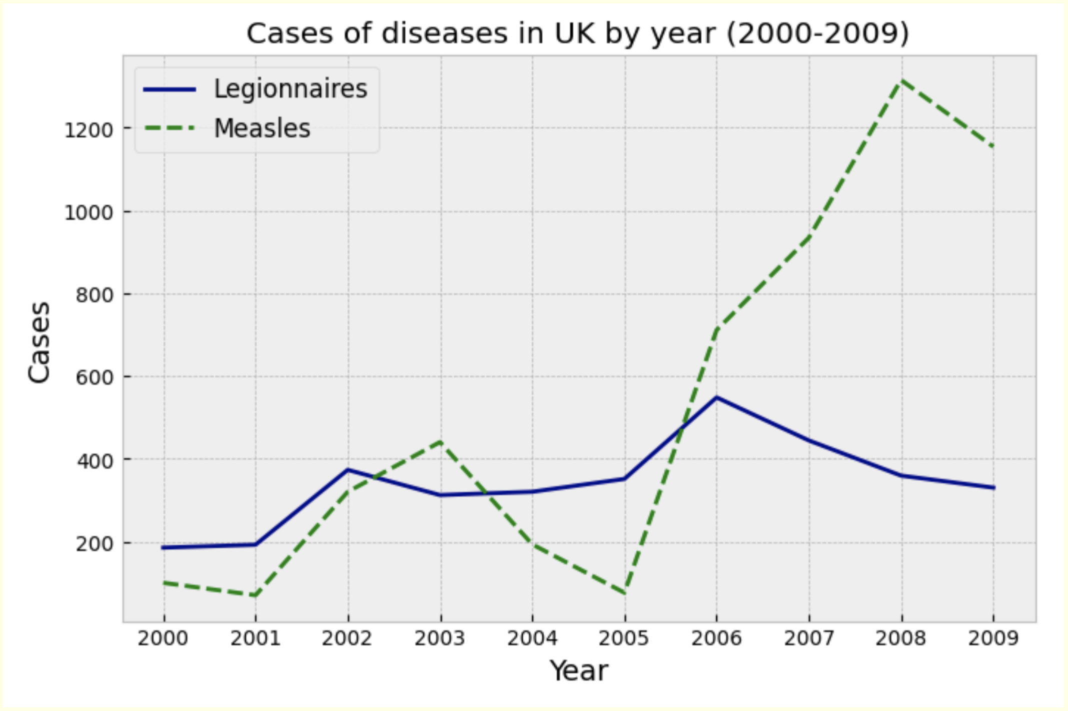
Plot elements can be overlaid on top of one another, to produce multiple plots on the same axis:

```
>>> months = [18, 19, 18, 21, 18, 21, 19, 18, 23, 20, 19, 20, 20, 19, 20]
>>> shoe_size = [6, 8, 6.5, 8.5, 6.5, 9, 8, 7, 10, 8, 9.5, 10, 7, 10, 11]
>>> guideline_x = [18, 19.5, 23]
>>> guideline_y = [6.5, 9, 10.5]
>>> fig, ax = plt.subplots(1)
>>> ax.scatter(months, shoe_size)
>>> ax.plot(guideline_x, guideline_y)
>>> plt.show()
```



Matplotlib allows for an incredible amount of customisation on plots. Consider the plot below that has axis labels, custom ticks, and colours:

```
>>> years = [2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009]
>>> measles = [100, 70, 320, 440, 193, 76, 711, 934, 1315, 1154]
>>> legionnairs = [185, 192, 373, 312, 320, 351, 548, 444, 359, 330]

>>> plt.style.use('bmh')

>>> fig, ax = plt.subplots(1, figsize=(8, 5))
>>> ax.plot(years, legionnairs, label='Legionnaires', c='darkblue')
>>> ax.plot(years, measles, label='Measles', c='green', linestyle='--')
>>> ax.set_xlabel('Year', fontsize=14)
>>> ax.set_ylabel('Cases', fontsize=14)
>>> ax.set_xticks(years)
>>> ax.set_title('Cases of diseases in UK by year (2000-2009)')
>>> ax.legend(fontsize=12)
>>> plt.show()
```



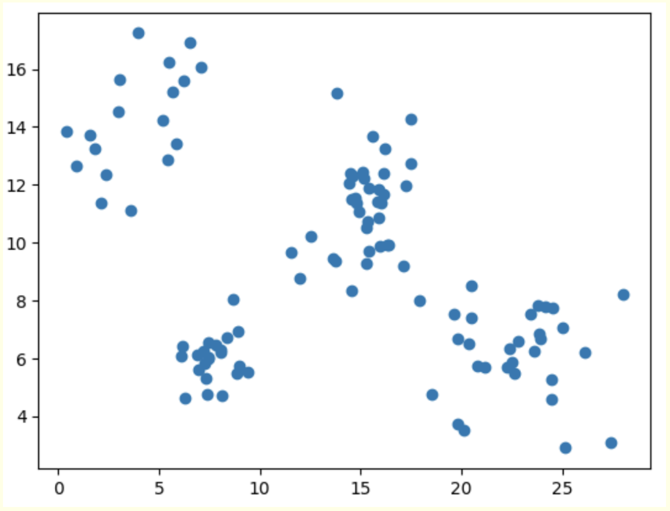
12 Machine Learning

There are a number of very good quality and very powerful libraries for building and using machine learning models. One particularly versatile library is Scikit-Learn, allowing us to do unsupervised learning, supervised learning, data pre-processing, and calculate performance metrics

Unsupervised learning: clustering

First import some data and look at it:

```
>>> plants = pd.read_csv('plants.csv')
>>> plants.columns
Index(['Height', 'Weight'], dtype='object')

>>> fig, ax = plt.subplots(1)
>>> ax.scatter(plants['Height'], plants['Weight'])
>>> plt.show()
```

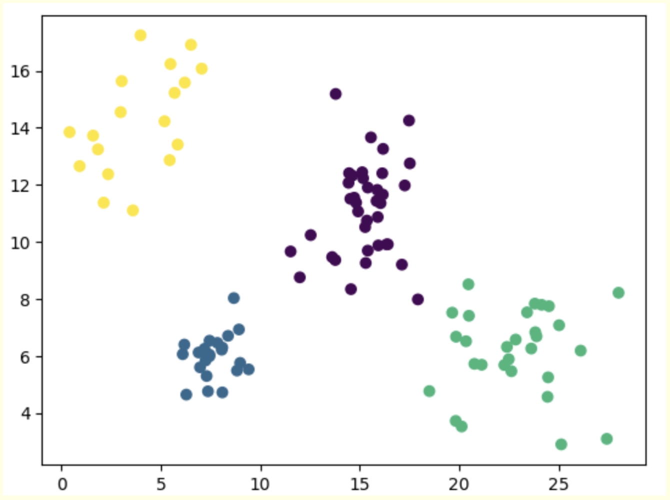


It looks as if there might be 4 clusters of points here. To use k-means clustering, we can use Scikit-Learn to make a clustering object for 4 clusters, then use it on the plants data:

```
>>> import sklearn.cluster
>>> kmeans = sklearn.cluster.KMeans(
...     n_clusters=4,
...     random_state=0,
...     n_init="auto"
... ).fit(plants)
>>> kmeans.predict(plants)
array([1, 2, 2, 1, 1, 3, 2, 2, 2, 3, 2, 2, 1, 3, 2, 1, 2, 0, 1, 1, 3, 0,
       1, 0, 2, 3, 3, 3, 2, 0, 0, 1, 1, 3, 2, 3, 1, 3, 1, 0, 0, 2, 3, 1,
       1, 2, 1, 3, 2, 2, 0, 1, 1, 2, 2, 3, 0, 2, 0, 0, 0, 1, 3, 3, 1, 1,
       1, 2, 2, 3, 0, 3, 1, 1, 1, 1, 2, 1, 3, 3, 2, 1, 3, 2, 0, 0, 3, 1,
       1, 1, 2, 1, 1, 1, 1, 1, 3, 0, 1, 2, 0, 2, 1, 0, 2, 2], dtype=int32)
```

This is an array of labels, indicating which cluster each row should belong to. We can add this as a new column of the data frame and plot:

```
>>> plants['Cluster'] = kmeans.predict(plants)

>>> fig, ax = plt.subplots(1)
>>> ax.scatter(plants['Height'], plants['Weight'], c=plants['Cluster'])
>>> plt.show()
```



In general this is how machine learning works with Scikit-Learn, an model object is created by training on some data with `.fit`, and the `.predict` method is used to use the model.

Data pre-processing

Before undertaking machine learning tasks, some data pre-processing might be required. We can complete these tasks with both Scikit-Learn, and also with Pandas if dealing with data frames.

One useful data transform we can perform is *one-hot* encoding, transforming one column of categorical data into multiple binary columns representing each category. The easiest way of doing this would be with Pandas' `get_dummies` function. Consider the cabin class column in the Titanic data set:

```
>>> data = pd.read_csv('titanic.csv')
>>> data['PClass']
0       1st
1       1st
2       1st
3       1st
4       1st
       ...
1308    3rd
1309    3rd
1310    3rd
1311    3rd
1312    3rd
Name: PClass, Length: 1313, dtype: object

>>> pd.get_dummies(data['PClass'], dtype=int)
```

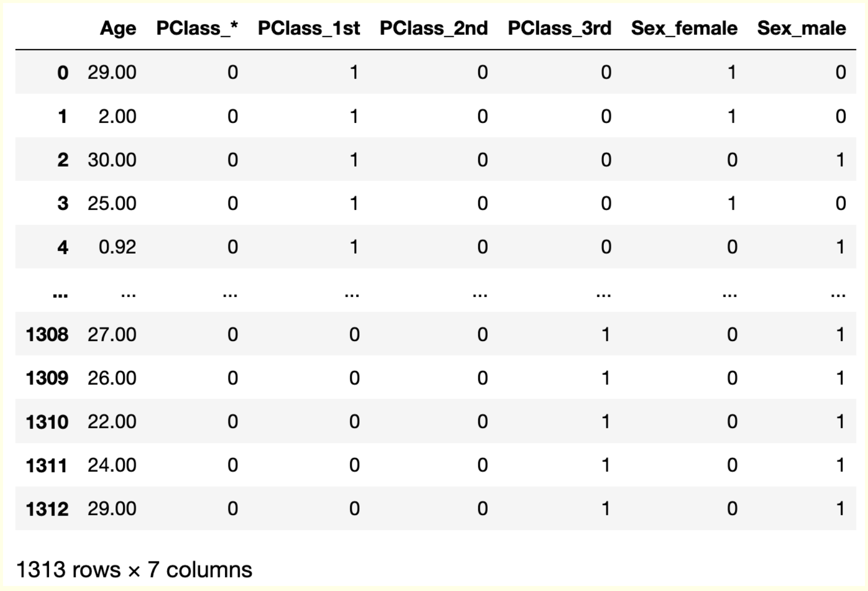
	*	1st	2nd	3rd
0	0	1	0	0
1	0	1	0	0
2	0	1	0	0
3	0	1	0	0
4	0	1	0	0
...	...	...	...	...
1308	0	0	0	1
1309	0	0	0	1
1310	0	0	0	1
1311	0	0	0	1
1312	0	0	0	1

1313 rows × 4 columns

There were four categories in the `'PClass'` column, which has been converted into a data frame with 4 columns, one for each category, containing 1s and 0s indicating if that row belongs to that category or not. In fact, we would only need three of these columns to uniquely determine the categories.

We can apply `get_dummies` to all columns, returning the one-hot encoding for categorical variables, and keeping numerical variables the same:

```
>>> pd.get_dummies(data[['Age', 'PClass', 'Sex']], dtype=int)
```



	Age	PClass_*	PClass_1st	PClass_2nd	PClass_3rd	Sex_female	Sex_male
0	29.00	0	1	0	0	1	0
1	2.00	0	1	0	0	1	0
2	30.00	0	1	0	0	0	1
3	25.00	0	1	0	0	1	0
4	0.92	0	1	0	0	0	1
...	...	...	...	...	...	...	...
1308	27.00	0	0	0	1	0	1
1309	26.00	0	0	0	1	0	1
1310	22.00	0	0	0	1	0	1
1311	24.00	0	0	0	1	0	1
1312	29.00	0	0	0	1	0	1

1313 rows × 7 columns

Another important bit of data pre-processing required for machine learning is splitting the data into training and testing datasets. Data consists of input columns X, and the column we are trying to predict or classify, y. Training data is the data that the model will be trained and parameterised on, containing both input columns X_{train} and target columns y_{train}. Testing data is a small subset of the data that will be kept behind and not used in the training: we can then use the trained model on the testing input data X_{test}, and see if the predicted column matches the target test data y_{test}

Usually we randomly select rows to be in the training or testing data, according to a proportion. Below we use Scikit-Learn to split the Titanic data into training and testing data, where the input columns are Age, cabin class, and Sex, the target column is passenger survival, and where 80% of the data is for training and 20% of the data is for testing:

```
>>> X = pd.get_dummies(data[['PClass', 'Sex', 'Age']], dtype=int)
>>> y = data['Survived']

>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.2, random_state=1
... )
```

We can see that X_train and y_train have the same indices, X_test and y_test have the same indices, and their proportions are roughly 80%-20%:

```
>>> all(X_train.index == y_train.index)
True
>>> all(X_test.index == y_test.index)
True
>>> len(X_test) / (len(X_train) + len(X_test))
0.2003046458492003
```

Supervised learning: classification

Classification involves predicting the categorical value of a target column. Continuing with the Titanic example, let's use Age, cabin class, and Sex to predict the survival of the passengers. We have already split the data into training and testing data, and transformed the categorical variables using one-hot encoding. Now we need to choose a classification algorithm. Here we will use a Decision Tree:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> decision_tree = DecisionTreeClassifier(
...     max_depth=5,
...     random_state=1
... ).fit(X_train, y_train)
```

This works in a similar way to the clustering algorithm before: a decision_tree object is created, using the decision tree classifier, and trained on X_train and y_train. Then, we can use this to predict survival for a given observation (a 31 year old male in a second class cabin):

```
>>> observation = pd.DataFrame({
...     'Age': [31],
...     'PClass_*': [0],
...     'PClass_1st': [0],
...     'PClass_2nd': [1],
...     'PClass_3rd': [0],
...     'Sex_female': [0],
...     'Sex_male': [1],
... })
>>> decision_tree.predict(observation)
array([0])
```

This has predicted that this individual will not have survived. It is common practice to predict all observations from the testing data set, in order to compare the predictions with the true observed values:

```
>>> y_pred = decision_tree.predict(X_test)
>>> y_pred
array([0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0,
       0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1,
       0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0,
       0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0,
       0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1,
       0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0,
       0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0])
```

This gives, for each row in `X_test`, the predicted value of the target column. In order to compare `y_pred` with `y_test`, there are performance metric functions available.

A number of other classifier algorithms are available in Scikit-Learn, that work identically, albeit with different parameters:

- **Multinomial Naive Bayes**: `from sklearn.naive_bayes import GaussianNB`
- **Gaussian Naive Bayes**: `from sklearn.naive_bayes import MultinomialNB`
- **k-Nearest Neighbours**: `from sklearn.neighbors import KNeighborsClassifier`
- **Decision Tree**: `from sklearn.tree import DecisionTreeClassifier`
- **Random Forest**: `from sklearn.ensemble import RandomForestClassifier`
- **Support Vector Machines**: `from sklearn.svm import SVC`

And there are some Regression algorithms too, for when the target variable is a continuous numerical value, rather than a categorical value:

- **Linear Regression**: `from sklearn.linear_model import LinearRegression`
- **Logistic Regression**: `from sklearn.linear_model import LogisticRegression`
- **Stochastic Gradient Descent**: `from sklearn.linear_model import SGDRegressor`

Performance metrics

For classification models we assess their quality on their performance on a testing data set, that is by comparing `y_pred` with `y_test`. There are a number of standard metrics used for this, implemented in Scikit-Learn.

For binary classification, such as predicting survival on the Titanic, a confusion matrix is useful. Here we categorise each prediction as either a True Positive (TP), True Negative (TN), False Positive (FP) or False Negative (FN), as so:

	Predicted 0	Predicted 1
Observed 0	TN	FP
Observed 1	FN	TP

A table giving the number of TPs, TNs, FPs, FNs for a given model and test is called a confusion matrix, given by:

```
>>> import sklearn.metrics
>>> sklearn.metrics.confusion_matrix(y_test, y_pred)
array([[151,   5],
       [ 48,  59]])
```

Giving 151 True Negatives, 5 False Positives, 48 False Negatives, and 59 True Positives.

In general, the more True Positives and True Negatives the better, however under different scenarios some can be more valuable than others. Therefore there are three important metrics we can obtain from this:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

All of which can be found using Scikit-Learn:

```
>>> sklearn.metrics.accuracy_score(y_test, y_pred)
0.7984790874524715

>>> sklearn.metrics.precision_score(y_test, y_pred)
0.921875

>>> sklearn.metrics.recall_score(y_test, y_pred)
0.5514018691588785
```

For regression problems, useful metrics include:

- R^2 **score**: `sklearn.metrics.r2_score`
- **Mean squared error**: `sklearn.metrics.mean_squared_error`
- **Mean absolute error**: `sklearn.metrics.mean_absolute_error`

13 Symbolic Mathematics

Usually when we do programming we are doing numerical computations. But when we do algebra and calculus, we deal with symbolic mathematics. Sympy is a library that allows symbolic mathematics in Python. One consequence of this is that we are no longer confined to floating point arithmetic. Consider one third, or the square root of two:

```
>>> 1 / 3
0.3333333333333333

>>> 2 ** (1 / 2)
1.4142135623730951
```

These are not exact values due to floating point arithmetic. Now, using Sympy, we can treat these numbers as symbols and get an exact representation:

```
>>> import sympy as sym
>>> sym.S(1) / 3
```
$\frac{1}{3}$

```
>>> 2 ** (sym.S(1) / 2)
```
$\sqrt{2}$

Similarly to the `math` library, Sympy has come constants and mathematical functions available to us:

```
>>> sym.pi
```
π

```
>>> sym.tan(sym.pi / 3)
```
$\sqrt{3}$

Algebra

Sympy allows us to do algebra and calculus by defining our own symbolic variables:

```
>>> x = sym.Symbol('x')
>>> x
```
x

We can treat this symbolic variable as we would any other Python variable, and create new variables from it. These new variables will be Sympy objects that have their own methods for doing mathematical manipulation. For example we can expand and factorise expressions:

```
>>> expression = (x - 4) ** 3 + (x - 2) ** 2
>>> expression
```
$(x-4)^3 + (x-2)^2$

```
>>> expression.expand()
```
$x^3 - 11x^2 + 44x - 60$

```
>>> expression.factor()
```
$(x-3)\left(x^2 - 8x + 20\right)$

And we can substitute in values to the function using a dictionary that maps variables to values:

```
>>> substitution_dict = {x: sym.sqrt(11)}
>>> evaluated_expression = expression.subs(substitution_dict)
>>> evaluated_expression
```
$\left(-4+\sqrt(11)\right)^3 + \left(-2+\sqrt{11}\right)^2$

```
>>> expression.simplify()
```
$-181 + 55\sqrt{11}$

And we can get the float value of this with the `float`:

```
>>> float(evaluated_expression)
1.4143634695469918
```

Sympy can find all solutions to equations of the form

$$f(x) = 0$$

Consider a quadratic with two roots:

```
>>> quadratic = x ** 2 + (5 * x) - 14
>>> sym.solveset(quadratic, x)
{-7, 2}
```

If we have more than one variable, the second argument of solveset indicates the variable that we are solving for:

```
>>> expression = (a ** 2) - (3 * x) - 8
>>> sym.solveset(expression, a)
```

$$\left\{-\sqrt{3x+8}, \sqrt{3x+8}\right\}$$

```
>>> sym.solveset(expression, x)
```

$$\left\{\frac{a^2}{3} - \frac{8}{3}\right\}$$

Sympy can even represent infinite sets when there are infinite solutions:

```
>>> sym.solveset(sym.cos(x), x)
```

$$\left\{2n\pi + \frac{\pi}{2} \mid n \in \mathbb{Z}\right\} \cup \left\{2n\pi + \frac{3\pi}{2} \mid n \in \mathbb{Z}\right\}$$

Calculus

We can take limits, derivatives, and compute integrals with Sympy. First, recognise that we can obtain an expression for the concept of infinity:

```
>>> sym.oo
```

$$\infty$$

We can take limits. Consider computing the following limit:

$$\lim_{n \to \infty} \frac{3n^2 + 5n}{2n^2 + 8}$$

```
>>> n = sym.Symbol('n')
>>> expression = ((3 * n ** 2) + (5 * n)) / ((2 * n ** 2) + 8)
>>> sym.limit(expression, n, sym.oo)
```

$\frac{3}{2}$

We can differentiate using `sym.diff`, allowing us to take first derivatives, second derivatives, third derivatives, and so on. Take:

$$f(x) = x^3 + e^{5x} + \sin(x)\cos(x)$$

we can find $\frac{d}{dx}f(x)$, $\frac{d^2}{dx^2}f(x)$ and $\frac{d^3}{dx^3}f(x)$:

```
>>> expression = (x ** 3) + sym.exp(5 * x) - (sym.sin(x) * sym.cos(x))
>>> expression
```

$x^3 + e^{5x} + \sin(x)\cos(x)$

```
>>> sym.diff(expression, x)
```

$3x^2 + 5e^{5x} + \sin^2(x) - \cos^2(x)$

```
>>> sym.diff(expression, x, 2)
```

$6x + 25e^{5x} + 4\sin(x)\cos(x)$

```
>>> sym.diff(expression, x, 3)
```

$125e^{5x} - 4\sin^2(x) + 4\cos^2(x) + 6$

We can integrate using `sym.integrate`, which also allows us to find definite integrals. For example, for the same function $f(x)$ we can find

$$\int f(x)dx \quad \text{and} \quad \int_{-\pi}^{\pi} f(x)dx$$

```
>>> sym.integrate(expression, x)
```

$\frac{x^4}{4} + \frac{e^{5x}}{5} - \frac{\sin^2(x)}{2}$

```
>>> sym.integrate(expression, (x, -sym.pi, sym.pi))
```

$-\frac{1}{5e^{5\pi}} + \frac{e^{5\pi}}{5}$

Differential Equations

Differential equations can be solved with `sym.dsolve`. In order to do thing, we need to define a symbolic function, as well as a symbolic variable, and create an Equation object to solve.

Find the general solution of the differential equation

$$\frac{dy}{dx} = 3y + 12$$

```
>>> x = sym.Symbol('x')
>>> y = sym.Function('y')
>>> dy = sym.diff(y(x), x)
>>> differential_equation = sym.Eq(dy, (3 * y(x)) + 12)
>>> sym.dsolve(differential_equation, y(x))
```

$$y(x) = C_1 e^{3x} - 4$$

Given the initial condition $y(0) = 3$, we can find the particular solution. We do this by defining a dictionary of initial conditions as using the keyword argument `ics`:

```
>>> conditions = {y(0): 3}
>>> sym.dsolve(differential_equation, y(x), ics=conditions)
```

$$y(x) = 7e^{3x} - 4$$

And second order differential equations can be solved. Finding the general solution of

$$\frac{d^2y}{dx^2} = 2\frac{dy}{dx} - 3y$$

```
>>> d2y = sym.diff(y(x), x, 2)
>>> differential_equation = sym.Eq(d2y, (2 * dy) - (3 * y(x)))
>>> sym.dsolve(differential_equation, y(x))
```

$$y(x) = \left(C_1 \sin\left(\sqrt{2}x\right) + C_2 \cos\left(\sqrt{2}x\right)\right) e^{x}$$

14 Linear Algebra & Markov Chains

Matrices & Vectors

The Numpy library allows for fast and efficient linear algebra, including the creation and manipulation of matrices and arrays. We create matrices from lists of lists:

```
>>> import numpy as np
>>> A = np.matrix(
...     [
...         [1, 2, -3],
...         [1, 1, 0],
...         [-2, 2, 4]
...     ]
... )
>>> A
matrix([[ 1,  2, -3],
        [ 1,  1,  0],
        [-2,  2,  4]])
```

We can access elements, rows or columns of the matrix with tuple or integer indexing:

```
>>> A[2]     # 3rd row
matrix([[-2,  2,  4]])

>>> A[:,0]   # 1st column
matrix([[ 1],
        [ 1],
        [-2]])

>>> A[2,0]   # The element in the 3rd row, 1st column
-2
```

The transpose of a matrix can be taken, that is when rows and columns are exchanged:

```
>>> A.T
matrix([[ 1,  1, -2],
        [ 2,  1,  2],
        [-3,  0,  4]])
```

The matrix multiplication operator in Python is `@`. Notice that mathematically:

$$AB \neq BA$$

that is, the order of matrix multpication matters, it isn't commutative. For example:

```
>>> B = np.matrix(
...     [
...         [3, 3, 7],
...         [-2, 0, 0],
...         [0, 0, 1]
...     ]
... )

>>> A @ B
matrix([[ -1,   3,   4],
        [  1,   3,   7],
        [-10,  -6, -10]])

>>> B @ A
matrix([[-8, 23, 19],
        [-2, -4,  6],
        [-2,  2,  4]])
```

Scalar multiplication of matrices works as normal multiplication of numerical variables:

```
>>> 555 * A
matrix([[  555,  1110, -1665],
        [  555,   555,     0],
        [-1110,  1110,  2220]])
```

Vectors, or arrays, are simply matrices with only one column or one row:

```
>>> C = np.matrix(
...     [
...         [80, 70, 60, 70, 90, 30, 20, 10]
...     ]
... )
>>> C
matrix([[80, 70, 60, 70, 90, 30, 20, 10]])
```

And we can change row vectors to column vectors, and back, by transposing it:

```
>>> C.T
matrix([[80],
        [70],
        [60],
        [70],
        [90],
        [30],
        [20],
        [10]])
```

With vectors, we have some additional methods, such as finding the maximum and minimum values of the array:

```
>>> C.max()
90
>>> C.min()
10
```

And finding the index of the maximum and minimum values of the array:

```
>>> C.argmax()
4
>>> C.argmin()
7
```

Matrix multiplication works between vectors and matrices. Remember that the order of multiplication is dependant on the vector shape. If A is a square matrix, then:

- dA is defined if d is a row vector, Ad is undefined,
- Ad is defined if d is a column vector, dA is undefined:

```
>>> d = np.matrix([[2, 0, 1]])
>>> d @ A
matrix([[ 0,  6, -2]])
```

And then:

- $dA = (A^T d^T)^T$ if d is a row vector,
- $Ad = (d^T A^T)^T$ if d is a column vector:

```
>>> d = np.matrix([[2, 0, 1]])
>>> A.T @ d.T
matrix([[ 0],
        [ 6],
        [-2]])
```

We can also add rows or columns to matrices using `np.vstack` (adding a row vector as a row) and `np.hstack` (adding a column vector as a column):

```
>>> np.vstack([A, d])
matrix([[ 1,  2, -3],
        [ 1,  1,  0],
        [-2,  2,  4],
        [ 2,  0,  1]])

>>> np.hstack([A, d.T])
matrix([[ 1,  2, -3,  2],
        [ 1,  1,  0,  0],
        [-2,  2,  4,  1]])
```

We can find the determinant of a matrix using the `np.linalg` module. Note that in this example, the determinant is subject to floating point arithmetic errors, however it does not effect the numerical mathematics much:

```
>>> np.linalg.det(A)
-15.999999999999998
```

And if the determinant is not zero, then we can find the inverse of a matrix:

```
>>> A.I
matrix([[-0.25  ,  0.875 , -0.1875],
        [ 0.25  ,  0.125 ,  0.1875],
        [-0.25  ,  0.375 ,  0.0625]])
```

And we can check that the inverse is correct, as $AA^{-1} = \mathbb{I}$, where $\mathbb{I}$ is the identity matrix, with ones along the main diagonal, and zeros elsewhere. We can check this:

```
>>> A @ A.I
matrix([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

Finally we can raise a matrix to am integer power in the same way as numerical variables:

```
>>> A ** 3
matrix([[ 37, -14, -87],
        [ 11,   1, -18],
        [-46,  34, 112]])

>>> A @ A @ A
matrix([[ 37, -14, -87],
        [ 11,   1, -18],
        [-46,  34, 112]])
```

Solving Systems of Linear Equations

Set of linear equations can be solved using linear algebra. Consider the following set of equations:

$$\begin{aligned} x + y + z &= 2 \\ 6x - 4y + 5z &= 31 \\ 5x + 2y + 2z &= 13 \end{aligned}$$

This is equivalent to the matrix equation:

$$\begin{pmatrix} 1 & 1 & 1 \\ 6 & -4 & 5 \\ 5 & 2 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 \\ 31 \\ 13 \end{pmatrix}$$

And multiplying through by the inverse gives:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 6 & -4 & 5 \\ 5 & 2 & 2 \end{pmatrix}^{-1} \begin{pmatrix} 2 \\ 31 \\ 13 \end{pmatrix}$$

And so can be solves with Numpy:

```
>>> coefficients = np.matrix(
...     [
...         [1, 1, 1],
...         [6, -4, 5],
...         [5, 2, 2]
...     ]
... )
>>> rhs = np.matrix(
...     [
...         [2],
...         [31],
...         [13]
...     ]
... )
>>> coefficients.I @ rhs
matrix([[ 3.],
        [-2.],
        [ 1.]])
```

Giving the solution of $x = 3$, $y = -2$, and $z = 1$.

As we are using the inverse of a matrix to solve this equation, we require that there is an equal number of equations to unknowns (in this case 3), in order to take the inverse of a square matrix. Sometimes we have more equations than unknowns, and do not know which are superfluous, in that case a different method is required. We can use another library, the Scipy library, for this:

```
>>> import scipy
>>> x, _, _, _ = scipy.linalg.lstsq(coefficients, rhs)
>>> x
array([[ 3.],
       [-2.],
       [ 1.]])
```

Giving the same solution.

Symbolic Matrices

The symbolic mathematics of Sympy can also be applied to matrices, for example the entries of a matrix might be mathematical unknowns:

```
>>> import sympy as sym
>>> a = sym.Symbol('a')
>>> A = sym.Matrix(
...     [
...         [1, 1, 1],
...         [6, -4, a],
...         [5, 2, 2]
...     ]
... )
>>> A
```

$$\begin{bmatrix} 1 & 1 & 1 \\ 6 & -4 & a \\ 5 & 2 & 2 \end{bmatrix}$$

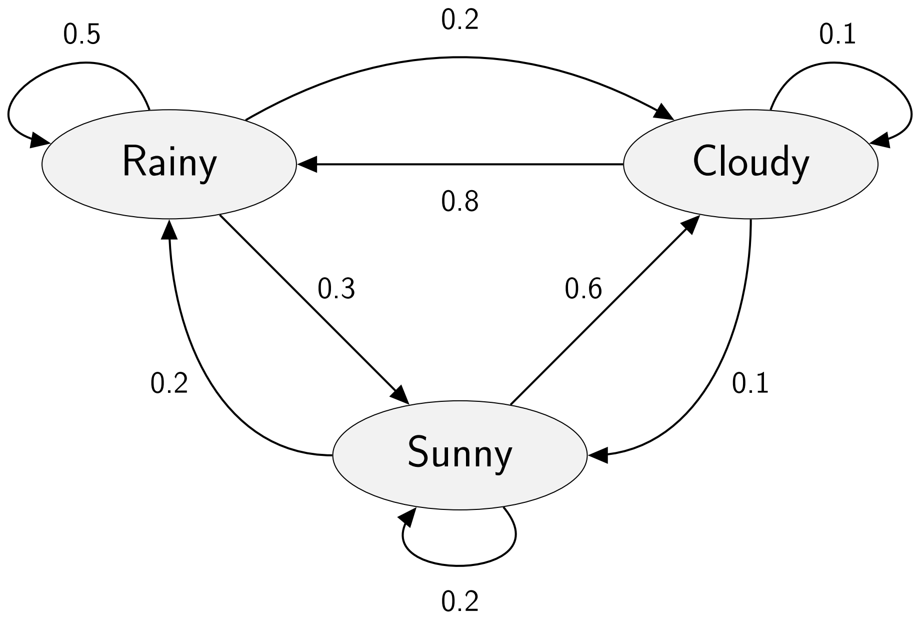
Sympy also recognises the `@` operator for matrix multiplication, and we can still simplify, expand, and factorise symbolic matrix expressions. In Sympy however, we use the `.inv()` method for the inverse matrix:

```
>>> b = sym.Matrix(
...     [
...         [a],
...         [31],
...         [13]
...     ]
... )

>>> x = A.inv() @ b
>>> x.simplify()
>>> x
```

$$\begin{bmatrix} \frac{13}{3} - \frac{2a}{3} \\ \frac{5\left(a^2-5a-3\right)}{3(a+4)} \\ \frac{32a-37}{3(a+4)} \end{bmatrix}$$

Markov Chains - Discrete Time

Markov chains are represented by matrices, and are analysed using linear algebra. Consider a discrete time Markov chain describing the weather from day to day. There are three states, “Rainy”, “Cloudy”, and “Sunny”. Transitions between states represent the probability of having that weather tomorrow, given the weather today. The Markov chain is depicted below:



For example, the probability of it being rainy tomorrow, given that it is sunny today, is 0.2. This can be represented by a transition matrix P:

$$P = \begin{pmatrix} 0.5 & 0.2 & 0.3 \\ 0.8 & 0.1 & 0.1 \\ 0.2 & 0.6 & 0.2 \end{pmatrix}$$

A vector v represents today's state. For example let $v = (0, 1, 0)$, denoting that is is cloudy today. The probability of being in each state tomorrow is given by vP. This can be calculated with Python:

```
>>> P = np.matrix(
...     [
...         [0.5, 0.2, 0.3],
...         [0.8, 0.1, 0.1],
...         [0.2, 0.6, 0.2]
...     ]
... )
>>> state_0 = np.matrix([0, 1, 0])

>>> state_1 = state_0 @ P
>>> state_1
matrix([[0.8, 0.1, 0.1]])
```

So a probability of 0.8 of being rainy, 0.1 of being cloudy, and 0.1 of being sunny. And in two days time:

```
>>> state_2 = state_1 @ P
>>> state_2
matrix([[0.5 , 0.23, 0.27]])
```

And three days time:

```
>>> state_3 = state_2 @ P
>>> state_3
matrix([[0.488, 0.285, 0.227]])
```

Which can be calculated directly from today's state using matrix powers:

```
>>> state_3 = state_0 @ (P ** 3)
>>> state_3
matrix([[0.488, 0.285, 0.227]])
```

In the long run, a Markov chain might reach a steady state, that is when the probability of being in each state does not change from day to day. That is the state v when $v = vP$. We can find the steady state probabilities by solving:

$$v(P - \mathbb{I}) = \underline{0}$$

Along with with the fact that v is a probability vector, so $\sum v = 1$:

```
>>> coefficients = np.vstack([(P - np.eye(3)).T, np.ones(3)])
>>> lhs = np.array([0, 0, 0, 1])
>>> steady_state, _, _, _ = scipy.linalg.lstsq(coefficients, lhs)

>>> steady_state
array([0.51162791, 0.26356589, 0.2248062 ])
```

And we can double check that this indeed gives a vector such that $v = vP$:

```
>>> steady_state @ P
matrix([[0.51162791, 0.26356589, 0.2248062 ]])
```

Markov Chains - Continuous Time

Continuous time Markov chains are similar, but instead of probabilities, each entry in the matrix is now a rate. They are commonly used to model queueing systems. Consider a queue with two servers, customers arrive at a rate of 3 per time unit, and are served at a rate of 4 per time unit, and there is only enough room for 5 customers to wait in line before customers are turned away. This system is depicted as follows:

$$\begin{tikzcd} (0) \arrow[r, bend left, "3"] & (1) \arrow[l, bend left, "4"] \arrow[r, bend left, "3"] & (2) \arrow[l, bend left, "8"] \arrow[r, bend left, "3"] & (3) \arrow[l, bend left, "8"] \arrow[r, bend left, "3"] & (4) \arrow[l, bend left, "8"] \arrow[r, bend left, "3"] & (5) \arrow[l, bend left, "8"] \end{tikzcd}$$

where states represent the number of customers present in the system. This can be represented by a transition rate matrix Q:

$$Q = \begin{pmatrix} -3 & 3 & 0 & 0 & 0 & 0 \\ 4 & -7 & 3 & 0 & 0 & 0 \\ 0 & 8 & -11 & 3 & 0 & 0 \\ 0 & 0 & 8 & -11 & 3 & 0 \\ 0 & 0 & 0 & 8 & -11 & 3 \\ 0 & 0 & 0 & 0 & 8 & -8 \end{pmatrix}$$

Now when v is a vector representing the probability of being in a given state, then the probability of being in each state at time t is given by ve^{Qt}. In Python, we can use Scipy to find the matrix exponential with `scipy.linalg.expm`.

So in this queue, starting from an empty system $v = (1, 0, 0, 0, 0, 0)$, the probability of being in each state at time $t = 0.05$ is given by:

```
>>> Q = np.matrix(
...     [
...         [-3, 3, 0, 0, 0, 0],
...         [4, -7, 3, 0, 0, 0],
...         [0, 8, -11, 3, 0, 0],
...         [0, 0, 8, -11, 3, 0],
...         [0, 0, 0, 8, -11, 3],
...         [0, 0, 0, 0, 8, -8]
...     ]
... )
>>> state_0 = np.array([1, 0, 0, 0, 0, 0])
>>> state_t = state_0 @ scipy.linalg.expm(Q * 0.05)
>>> state_t.round(3)
array([0.873, 0.119, 0.008, 0.   , 0.   , 0.   ])
```

Notice here the use of the `round` method on the Numpy array, to display the probabilities rounded to 3 decimal places.

In the long run the system will reach a steady state, where advancing the time does not change the state probabilities:

```
>>> state_0 @ scipy.linalg.expm(Q * 10000).round(4)
array([0.4564, 0.3423, 0.1284, 0.0481, 0.0181, 0.0068])

>>> state_0 @ scipy.linalg.expm(Q * 10001).round(4)
array([0.4564, 0.3423, 0.1284, 0.0481, 0.0181, 0.0068])
```

We can find this analytically by solving $vQ = 0$:

```
>>> A = np.vstack([Q.T, np.ones(6)])
>>> b = np.array([0, 0, 0, 0, 0, 0, 1])
>>> steady_state, _, _, _ = scipy.linalg.lstsq(A, b)
>>> steady_state.round(4)
array([0.4564, 0.3423, 0.1284, 0.0481, 0.0181, 0.0068])
```

As the states represent the number of customers in the system, we can use the steady state probability vector to find the expected number of customers in the system, by taking the dot product of the steady state probability vector, and the vector representing the number of customers present in each state:

```
>>> customers = np.array([0, 1, 2, 3, 4, 5])
>>> expected_number_customers = np.dot(steady_state, customers)
>>> expected_number_customers
0.8494665589570755
```

So on average there are 0.85 customers in the system.

15 Simulation

Simulation involved generating pseudo-random numbers according to some probability distribution, manipulating them, and observing their behaviour.

Pseudo-random numbers

We have seen the `random` library earlier in Chapter 8. To recap, we can sample Unifromly distributed pseudo-random numbers with `random.random()`, or sample numbers from given distributions:

```
>>> import random
>>> random.seed(0)
>>> random.random()
0.8444218515250481

>>> random.normalvariate(mu=500, sigma=10)
507.63731855355195
```

Remember that it is important to set a seed for the pseudo-random number generator so that we get the same sequence of random things, ensuring reproducibility.

Monte Carlo Simulation

Monte Carlo simulation allows us to generate many instances of random events, and estimate probabilities of outcomes occurring. This works because of the Law of Large Numbers. A common way to do this is to define a function for an exporiment, or the random event of interest, and then repeat that experiment a number of times by calling the function many times.

Let's see with some examples:

Example 1: Falling Fridge Magnets

You have a fridge with fidge magnets on that spell the words **HELLO WORLD***. An earthquake strikes and two letters fall off the fridge at random. What is the probability that one of those letters is a vowel?*

First we will write a function that will perform this random event: two letters randomly fall off, and we check if either is a vowel or not:

```
>>> def run_experiment(fridge_magnets):
...     first_two = random.sample(fridge_magnets, 2)
...     one_is_vowel = any(letter in "AEIOU" for letter in first_two)
...     return one_is_vowel
```

Testing how this works:

```
>>> fridge_magnets = ["H", "E", "L", "L", "O", "W", "O", "R", "L", "D"]
>>> random.seed(5)
>>> run_experiment(fridge_magnets)
True

>>> run_experiment(fridge_magnets)
False
```

Now to estimate the probability of a `True`, that is the probability of the random event occurring, we repreat this a number of times and count the outcomes:

```
>>> def run_experiments(num_experiments, fridge_magnets):
...     successes = []
...     random.seed(0)
...     for trial in range(num_experiments):
...         outcome = run_experiment(fridge_magnets)
...         successes.append(outcome)
...     prob_of_success = sum(successes) / num_experiments
...     return prob_of_success

>>> run_experiments(num_experiments=10, fridge_magnets=fridge_magnets)
0.5
```

So in 10 earthquakes, a vowel fell off the fridge in half of them. This is just an estimate. Due to the Law of Large Numbers, as we increase the number of experiments, or trials, the proportion of times the event occurred will approach the true probability of that event occurring:

```
>>> run_experiments(num_experiments=50000, fridge_magnets=fridge_magnets)
0.5326
```

In this case, the true probability of this event ocurring would be:

$$\begin{aligned}\mathbb{P}(\text{Two vowels}) &= \mathbb{P}(\text{First is vowel}) + \mathbb{P}(\text{First isn't a vowel})\mathbb{P}(\text{Second is vowel})\\ &= \frac{3}{10} + \left(1 - \frac{7}{10}\right)\left(\frac{3}{9}\right) = \frac{8}{15} = 0.5\overline{3}\end{aligned}$$

Example 2: Weight of a Shopping Bag

Sometimes we know the probability of an event occurring and we want to investigate the effects of that event. In this case, to sample the event with probability q, we can sample a uniformly distributed pseudo-random number and see if it is below q or not. Another use of Monte Carlo simulation is to investigate how probability distributions combine. For example:

When grocery shopping I roll buy apples with probability 0.4 and melons with probability 0.2. I roll a die to see how many of each fruit I buy. The weight of an apple is Uniformly distributed between 70 and 100 grams. The weight of a melon is Normally distributed with mean 1200g and standard deviation 250g. What is the average weight of my shopping bag? What is the probability of exceeding 6kg? What does its weight distribution look like?

First a function to choose a type of fruit:

```
>>> def choose_type_of_fruit():
...     if random.random() < 0.4:
...         return "Apple"
...     return "Melon"
```

Then a function to randomly fill the shopping bag and total its weight:

```
>>> def fill_shopping_bag():
...     fruit = choose_type_of_fruit()
...     number = random.choice([1, 2, 3, 4, 5, 6])
...     bag_weight = 0
...     if fruit == "Apple":
...         for apple in range(number):
...             bag_weight += random.uniform(70, 100)
...     if fruit == "Melon":
...         for melon in range(number):
...             bag_weight += random.normalvariate(1200, 250)
...     return bag_weight

>>> random.seed(0)
>>> fill_shopping_bag()
4071.3242946541422
```

Then a function to repeat the experiment:

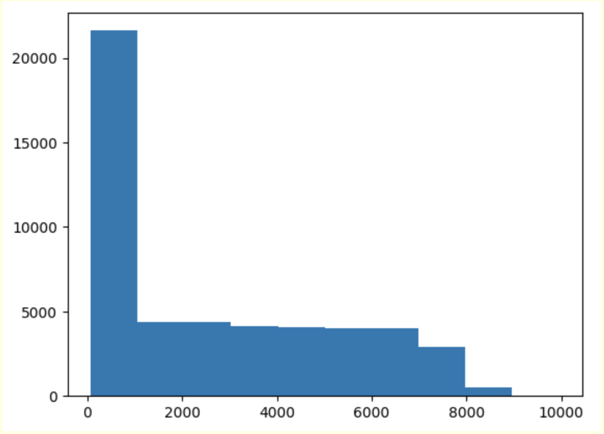
```
>>> def run_experiments(num_experiments):
...     outcomes = []
...     random.seed(0)
...     for experiment in range(num_experiments):
...         outcomes.append(fill_shopping_bag())
...     return outcomes
```

Running this many times allows us to take an average of the weights of the shopping bags over the trials, and the probability of exceeding 6kg:

```
>>> bags = run_experiments(num_experiments=50000)
>>> expected_bag_weight = sum(bags) / len(bags)
>>> expected_bag_weight
2625.1377470632874

>>> prob_over_6kg = sum(b > 6000 for b in bags) / len(bags)
>>> prob_over_6kg
0.14796
```

And to plot a histogram to look at it's weight distribution:

```
>>> plt.hist(bags);
```



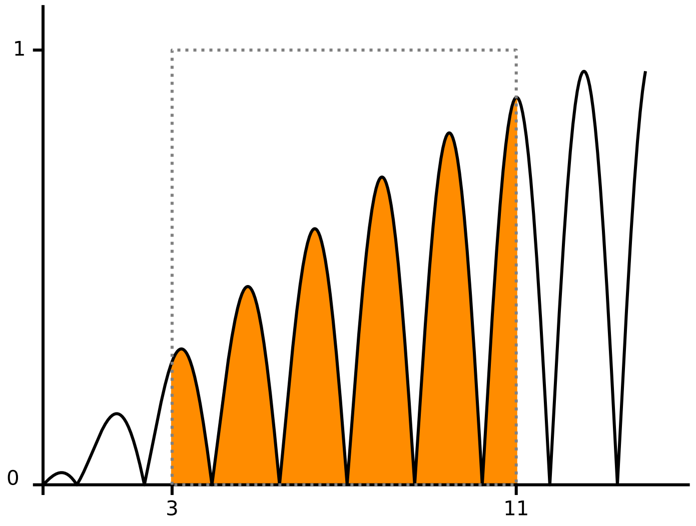
Example 3: Evaluating an Integral

Another common demonstration of Monte Carlo simulation is finding the value of a definite integral. One interpretation of an integral is the area under the curve of the integrand. If randomly sampling points within a rectangle, whose horizontal extent is the domain of integration, and whose vertical extend entirely contains the integrand curve at those points, then the value of the integral would be the area of the rectangle multiplied by the proportion of points that lie under the curve of the integrand. Checking is a point is under the curve of the integrand is a straightforward task. This can be useful when estimating the value of an integral that is difficult to obtain analytically.

Estimate the following integral:

$$I = \int_3^{11} \left| \sin\left(\frac{x}{10}\right) \cos\left(2x\right) \right| dx$$

The entire integrand is bounded by 0 below and 1 above. Therefore we can consider the rectangle with horizontal extend $[3, 11]$ and vertical extent $[0, 1]$ (they grey dotted box below). The value of the definite integral (the orange shaded area) can be estimated by sampling points in this rectangle and finding the proportion that lie below the integrand.



First we define the integrand curve:

```
>>> import math
>>> def integrand(x):
...     return abs(math.sin(x / 10) * math.cos(2 * x))
```

Then we generate a given number of points, find the proportion that lie under the curve, and so find the value of the integral:

```
>>> n_points = 100000
>>> lower, upper = 3, 11
>>> random.seed(0)
>>> random_points = [
...     (
...         random.uniform(lower, upper),
...         random.uniform(0, 1)
...     ) for point in range(n_points)
... ]

>>> under_curve = [point[1] <= integrand(point[0]) for point in random_points]
>>> proportion_under_curve = sum(under_curve) / n_points
>>> rectangle_area = (upper - lower)
>>> estimated_value_of_integral = rectangle_area * proportion_under_curve
>>> estimated_value_of_integral
3.21976
```

Discrete Event Simulation

Discrete event simulation (DES) is a more formalised way of using Monte Carlo simulation to describe more complex processes, usually queueing systems. In Python there are a number of libraries available for doing DES, including `simpy`, which uses a process-based approach. Here we will use the Ciw library, which uses an event-scheduling approach.

In Ciw, there are two main objects to consider: the Network object, which describes the system being modelled; and the Simulation object, which contains information about the simulation run itself. We will look at an example:

Example 1: An M/M/c queue

Assume you are a bank manager and would like to know how long customers wait in your bank. Customers arrive randomly, roughly 12 per hour with Exponentially distributed inter-arrival times. Service times are random, also Exponentially distributed, lasting on average roughly 10 minutes. The bank is open 24 hours a day, 7 days a week, and has three servers who are always on duty. If all servers are busy, newly arriving customers form a queue and wait for service. On average how long do customers wait?

First we need to understand the system we are trying to model. There is one queue involved, with three key components:

- The **inter-arrival time distribution** describes the probability distribution from which the times between consecutive customer arrivals are drawn.
- The **service time distribution** describes the probability distribution from which the service times are drawn.
- The **number of servers** is the number of members of staff ready to serve the waiting customers. That is, the maximum number of customers who can be served at the same time.

With these things, we can build a Network object in Ciw that describes the bank:

```
>>> import ciw
>>> N = ciw.create_network(
...     arrival_distributions=[ciw.dists.Exponential(rate=12)],
...     service_distributions=[ciw.dists.Exponential(rate=6)],
...     number_of_servers=[3]
... )
```

When creating the Network object, `N`, we give the relevant information by using the `ciw.create_network` function the keywords `arrival_distributions`, `service_distributions` and `number_of_servers`. These keywords take lists, with an entry for every queue, or node, in the system. Here there is only one queue, and so we only need lists of length one. Note the time units, they are unitless, so we can use whichever unit is most relevant to the problem, as long as we are consistent throughout. Here, we have used hours.

Now we can create a Simulation object, which will contain information and methods for running the model:

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
```

The Simulation object takes in the Network object, so it knows which model to run. We also set a seed here, as the initialisation of the Simulation object may be random. Let's run this for a year, that is 24×7 hours:

```
>>> Q.simulate_until_max_time(24 * 7)
```

Once the simulation has run, we can obtain all information about the run with the `get_all_records` method. This returns a list of named tuples, representing every completed service from the simulation run. However, it is a lot easier to analyse these results by converting this into a Pandas data frame:

```
>> recs = Q.get_all_records()
>> recs_df = pd.DataFrame(recs)
>> recs_df

      id_number  customer_class  original_customer_class  node  arrival_date  waiting_time  service_start_date  service_time
   0          2        Customer                 Customer     1      0.132272      0.000000            0.132272      0.049077
   1          3        Customer                 Customer     1      0.189277      0.000000            0.189277      0.099485
   2          1        Customer                 Customer     1      0.012024      0.000000            0.012024      0.313359
   3          5        Customer                 Customer     1      0.285355      0.003407            0.288762      0.094498
   4          7        Customer                 Customer     1      0.438290      0.000000            0.438290      0.000351
 ...        ...             ...                      ...   ...           ...           ...                 ...           ...
2012       2014        Customer                 Customer     1    167.478280      0.000000          167.478280      0.150870
2013       2015        Customer                 Customer     1    167.558105      0.027590          167.585695      0.077298
2014       2016        Customer                 Customer     1    167.644168      0.000000          167.644168      0.037939
2015       2010        Customer                 Customer     1    167.198880      0.000000          167.198880      0.576728
2016       2017        Customer                 Customer     1    167.845139      0.000000          167.845139      0.050081

2017 rows × 16 columns
```

This data frame contains a row for each completed service. Its many columns represent different information about those services, including which customer was served, when they arrived at the queue, when they began service, ended service, and how long they waited. We could use data frame manipulations in order to find the average waiting time:

```
>>> recs_df['waiting_time'].mean()
0.07271120446749425
```

Remembering our time units are in hours, this shows that for this run, customers waited on average 0.727 hours.

Good Simulation Practice

Finding some key performance indicator (KPI) in a discrete event simulation, such as the mean waiting time of customers, is essentially the same as finding the outcome of a random event in Monte Carlo simulation. In order to have an accurate estimate, we should generate many of these random events and take an average. This is good simulation practice, ensuring our KPIs are as accurately estimated as possible. There are two key concepts:

- **Running trials:** Running trials means running the same simulation multiple times, with different random number seeds, and collecting the desired KPIs many times. These KPIs will themselves be random variables, and we can take an average over the outcomes of the trials to find a more accurate estimate of the KPI. This is sometimes called smoothing out the randomness.
- **Warm-up time:** In general, a complex simulation such as a queueing system can have many states. The beginning state of the simulation can have drastic effects on the KPIs recorded. To overcome this, we usually begin the system from an arbitrary state (in queueing systems this is usually an empty system with no customers), and let the simulation run for a period of time before data collection begins. This ensures a random, yet common enough state to begin the simulation data collection.

In Python, we can implement trials with a for-loop, and implement a warm-up time by filtering the final records before calculating the KPI. For example, let's repeat the above with a warm-up time of one day, and repeat for 10 trials:

```
>>> waiting_times = []
>>> for trial in range(10):
...     ciw.seed(trial)
...     Q = ciw.Simulation(N)
...     Q.simulate_until_max_time(24 * 8)
...     recs = Q.get_all_records()
...     recs_df = pd.DataFrame(recs)
...     warm_recs = recs_df[recs_df['arrival_date'] >= 24]
...     wait = warm_recs['waiting_time'].mean()
...     waiting_times.append(wait)

>>> sum(waiting_times) / len(waiting_times)
0.07398067446773089
```

Example 2: Networks of Queues

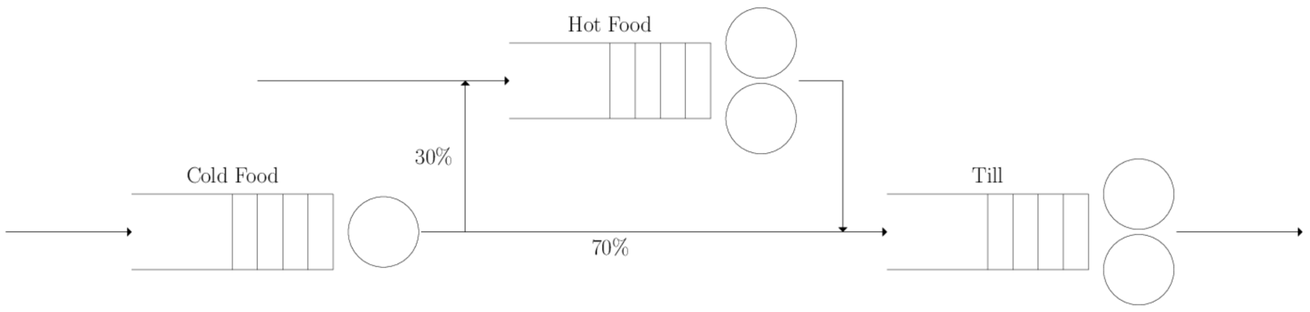
Ciw can handle networks of queues, that is queues where once a customer finishes service, there is a probability of joining another queue. In order to do so, we need to enforce an arbitrary ordering of the nodes.

Imagine a café that sells both hot and cold food. Customers arrive and can take a few routes:

- *Customers only wanting cold food must queue at the cold food counter, and then take their food to the till to pay.*
- *Customers only wanting hot food must queue at the hot food counter, and then take their food to the till to pay.*
- *Customers wanting both hot and cold food must first queue for cold food, then hot food, and then take both to the till and pay.*

In this system there are three nodes: Cold food counter (Node 1), Hot food counter (Node 2), and the till (Node 3): Customers wanting hot food only arrive at a rate of 12 per hour; customers wanting cold food arrive at a rate of 18 per hour; 30% of all customer who buy cold food also want to buy hot food. On average it takes 1 minute to be served cold food, 2 and a half minutes to be served hot food, and 2 minutes to pay. There is 1 server at the cold food counter, 2 servers at the hot food counter, and 2 servers at the till.

A diagram of the system is shown below:



In Ciw, inter-arrival distributions, service distributions, and number of servers are given to the Network object with a list. Therefore the corresponding attributes for each node in the network are given as entries in these lists. The transitions between nodes are given my a routing matrix (`routing`), with entry (ij) corresponding to the probability of joining node j after finishing service at node i. In the case of the café, that is:

$$\begin{pmatrix} 0 & 0.3 & 0.7 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

The Network object is created with:

```
>>> N = ciw.create_network(
...     arrival_distributions=[ciw.dists.Exponential(rate=0.3),
...                            ciw.dists.Exponential(rate=0.2),
...                            None],
...     service_distributions=[ciw.dists.Exponential(rate=1.0),
...                            ciw.dists.Exponential(rate=0.4),
...                            ciw.dists.Exponential(rate=0.5)],
...     routing=[[0.0, 0.3, 0.7],
...              [0.0, 0.0, 1.0],
...              [0.0, 0.0, 0.0]],
...     number_of_servers=[1, 2, 2]
... )
```

We will simulate one shift of lunchtime of 3 hours (180 mins). At the beginning of lunchtime the café opens, and so it begins from an empty system, and no warm-up time is required. We'll run 10 trials, to get an estimate for the average number of customers that have their service completed within the lunchtime (that is, how many customers pass through the final Till node):

```
>>> completed_custs = []
>>> for trial in range(10):
...     ciw.seed(trial)
...     Q = ciw.Simulation(N)
...     Q.simulate_until_max_time(180)
...     recs = Q.get_all_records()
...     num_completed = len([r for r in recs if r.node==3])
...     completed_custs.append(num_completed)

>>> sum(completed_custs) / len(completed_custs)
80.4
```

Ciw's Features

Ciw is able to simulate queueing systems with a number of other features not discussed in detail here. A list is given:

- **Type I blocking:** Customers not advancing to the next node if the next queue is full, remaining with the server blocking other customers from beginning service there.
- **A range of sampling distributions:** From standard distributions such as Normal and Triangular, to phase-type distributions, time-dependent and state-dependent distributions.
- **Batch arrivals:** More than one customer arriving at the same time.
- **Baulking customers:** Customers deciding not to join a queue.
- **Reneging customers:** Customers deciding to leave the queue.
- **Processor sharing:** All customers served simultaneously, sharing a proportion of the server's time.
- **Multiple customer classes:** Different types of customer behaving differently in the same system.
- **Customers changing classes:** Customers changing classes or priorities while waiting or after service.
- **Service disciplines:** The way in which customers are chosen from the queue to begin service, including FIFO, SIRO, LIFO, and priorities.
- **Server schedules:** Servers that only work at certain times of the day.
- **State tracking:** A way of recording information about the system's state over time.

16 System Dynamics

We saw in Chapter 13 that we can solve some differential equations algebraically using the Sympy library. Sometimes there are differential equations, or systems of differential equations that are too difficult to solve algebraically. In these cases we can solve them numerically. We can do this with the `scipy` library.

Fist consider the differential equation:

$$\frac{dy}{dt} = \frac{1}{50}y(240 - y)$$

with initial condition $y(0) = 1$. We would like to describe the curve $y(t)$ over the interval $t \in [0, 3]$.

First, we can write a function that returns the instantaneous derivative. This function needs to take y and t as inputs:

```
>>> def dy(y, t):
...     return (1/50) * y * (240 - y)
```

Now we can find the numerical values of the curve $y(t)$ in the given interval using `scipy`. First we define the interval we are interested in, we will do that using Numpy, to get an array of many (1001) equally spaced points between 0 and 3:

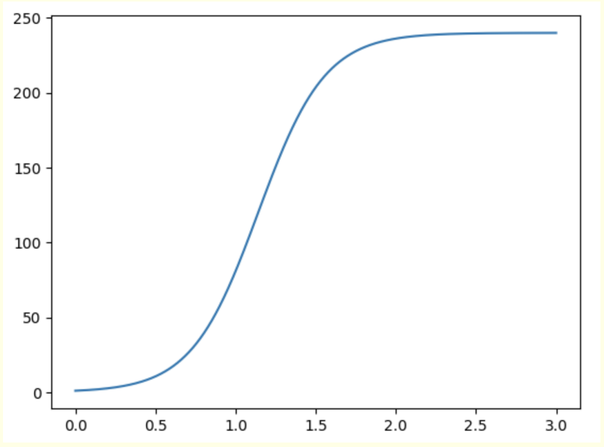
```
>>> import numpy as np
>>> t = np.linspace(0, 3, 1001)
>>> t
array([0.   , 0.003, 0.006, ..., 2.994, 2.997, 3.   ])
```

and, with the initial condition of $y(0) = 1$, we can find the numerical solution to the ordinary differential equation with `scipy.integrate.odeint`, which takes in the derivative function, initial conditions, and time domain, and returns an array of the numerical curve $y(t)$.

```
>>> y0 = [1]
>>> sol = scipy.integrate.odeint(dy, y0, t)
>>> sol
array([[  1.        ],
       [  1.0144429 ],
       [  1.02909343],
       ...,
       [239.96709861],
       [239.96756894],
       [239.96803254]])
```

To visualise this, we can plot, showing a logistic function:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(t, sol)
```

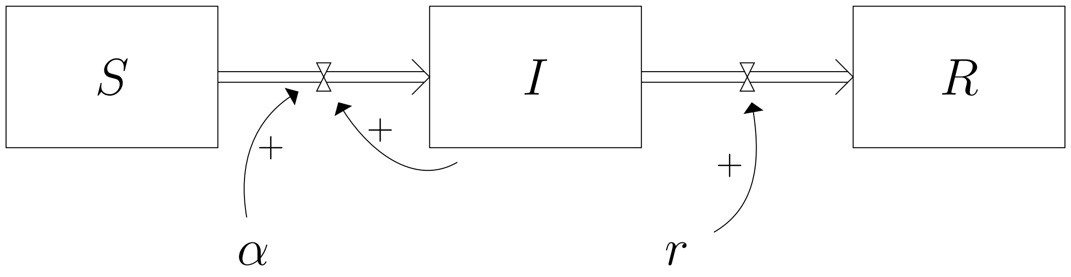


If necessary, the derivative function can take additional arguments, and the `scipy.integrate.odeint` function can pass these to the function whenever it is called. For example:

```
>>> def dy(y, t, r, m):
...     return r * y * (m - y)
>>> sol = scipy.integrate.odeint(dy, y0, t, args=(1/50, 240))
```

Systems of Equations

The same function can be used to solve systems of differential equations, sometimes referred to as system dynamics. As an example we will use the classic SIR model of disease spread, depicted in the stock-and-flow diagram below:



There are three stocks, $S(t)$, $I(t)$ and $R(t)$, representing the numbers of people susceptible to a disease, people infected with the disease, and recovered from the disease. These will be the functions we are trying to estimate. α is the infection rate, how many individuals does one other individual infect in one time unit; and r is the recovery rate, where $\frac{1}{r}$ is the average recovery time. The system of differential equations to solve gives the relationships between these:

$$\frac{dS}{dt} = -\frac{\alpha I S}{N}$$
$$\frac{dI}{dt} = \frac{\alpha I S}{N} - rI$$
$$\frac{dR}{dt} = rI$$

where N is the constant, $N = S(t) + I(t) + R(t)$, representing the total number of individuals in the system. These equations come directly from the stock-and-flow diagrams. The rate of change of a stock is the sum of all its input flows minus the sum of all its output flows; while a flow represents the product of all its influencing factors multiplied by the stock at the beginning of the flow.

Writing this system of differential equations in Python is similar to before, however now the input `y` is an array of the three stocks, while the output is an array of the three derivatives:

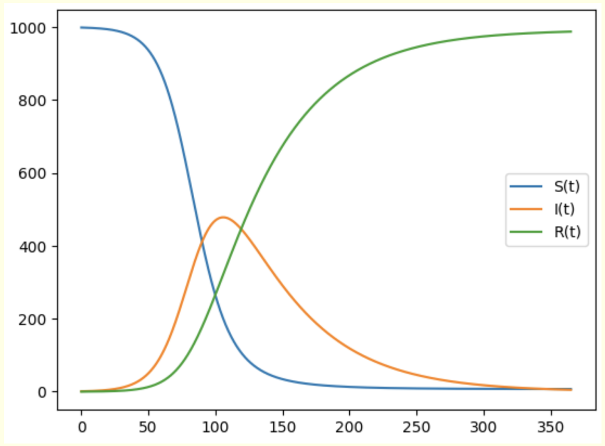
```
>>> def sir_differential_equations(y, t, infection_rate, recovery_rate):
...     S, I, R = y
...     N = S + I + R
...     dS = -(infection_rate * I * S) / N
...     dI = (infection_rate * I * S) / N - (recovery_rate * I)
...     dR = recovery_rate * I
...     return dS, dI, dR
```

Solving the system is the same as before. Here we use initial conditions $S(0) = 999$, $I(0) = 1$ and $R(0) = 0$, that is, in a population of 1000, one person is infected:

```
>>> t = np.linspace(0, 365, 10001)
>>> y0 = [999, 1, 0]
>>> infection_rate = 0.1
>>> recovery_rate = 0.02
>>> sol = scipy.integrate.odeint(
...     sir_differential_equations,
...     y0,
...     t,
...     args=(infection_rate, recovery_rate,)
... )
```

And plotting gives:

```
>>> plt.plot(t, sol, label=['S(t)', 'I(t)', 'R(t)'])
>>> plt.legend()
```



17 Linear Programming

The `pulp` library can be used to access solvers that can solve linear programming problems. It is a way of representing linear programming problems in Python, and then passing them to a solver to perform the relevant algorithms, and then obtaining the optimal solution as Python variables.

In linear programming problems there are three key components: *decision variables*, *constraints*, and an *objective function*. The latter two must be written as linear combinations of the decision variables. For example, the following is a linear programming problem:

Maximise:

$$3x_1 + 5x_2 \tag{17.1}$$

subject to

$$x_1 - x_2 \leq 6 \tag{17.2}$$
$$4x_2 + x_1 \leq 10 \tag{17.3}$$
$$3x_1 - x_2 \geq 2 \tag{17.4}$$
$$x_1, x_2 \geq 0 \tag{17.5}$$

Where

- x_1 and x_2 are the decision variables;
- (17.1) is the objective function, a linear combination of the decision variables, which in this case we want to maximise;
- (17.2), (17.3) and (17.4) are the constraints, linear combinations of the decision variables that must satisfy a relationship ($=$, $\leq$, $\geq$) with a scalar;
- (17.5) is the non-negativity constraint. Although we usually take this for granted, when using PuLP, we must explicitly define this.

In the language of linear programming, a *feasible solution* is a pair of values for (x_1, x_2) that satisfy the constraints. For example $(x_1 = 1, x_2 = 1)$ is a solution, as the all the constraints are satisfied. So is $(x_1 = 2, x_2 = 0)$. But $(x_1 = 10, x_2 = 2)$ does not satisfy all the constraints, and it called *infeasible*. The *optimal solution* is a feasible solution that gives the maximum value of the objective. We will use PuLP to find it.

First, we import the library. Then we define the problem itself, where we need to give the problem a name, say `"My_LP_Problem"`, and note that it is maximisation problem, that it we wish to maximise the objective function (we could also minimise the objective function):

```
>>> import pulp
>>> prob = pulp.LpProblem("My_LP_Problem", pulp.LpMaximize)
```

The first component we need to define for our problem is the decision variables. These take the form of a dictionary, defined by giving PuLP a label and an iterable for the keys:

```
>>> x = pulp.LpVariable.dicts("x", [1, 2])
```

Here the label is `"x"`, and the iterable is a list containing the numbers 1 and 2. These act as keys for accessing each decision variable:

```
>>> x[1]
x_1
>>> x[2]
x_2
```

The next component we need to define for our problem is the objective function. We do this my adding a linear combination of the decision variables to the problem object:

```
>>> objective_function = (3 * x[1]) + (2 * x[2])
>>> prob += objective_function
```

Finally, the next component we need to define for our problem is the constraints. This is done in the same way, by adding inequalities to the problem object, defined by linear combinations of the decision variables. Remember, non-negativity constraints must be defined explicitly:

```
>>> prob += x[1] - x[2] <= 6
>>> prob += (4 * x[2]) + x[1] <= 10
>>> prob += (3 * x[1]) - x[2] >= 2
>>> prob += x[1] >= 0
>>> prob += x[2] >= 0
```

This fully defined the linear programming problem in Python. We can summarise the problem by printing it:

```
>>> print(prob)
My_LP_Problem:
MAXIMIZE
3*x_1 + 2*x_2 + 0
SUBJECT TO
_C1: x_1 - x_2 <= 6

_C2: x_1 + 4 x_2 <= 10

_C3: 3 x_1 - x_2 >= 2

_C4: x_1 >= 0

_C5: x_2 >= 0

VARIABLES
x_1 free Continuous
x_2 free Continuous
```

PuLP itself is a library that is used to define linear programming problems in Python, and to access solvers. So in order to find the optimal solution, we need to state which solver we wish to use. We can list any available solvers with `pulp.listSolvers(onlyAvailable=True)`, and more solvers can be added by installing them externally. Some powerful solvers include GLPK, Gurobi, and Coin-OR. A solver that comes ready installed with PuLP is `"PULP_CBC_CMD"`, and we will use this, asking the solver to suppress all output messages (such as solve statistics and runtime performances). Then, we solve the problem with the `.solve` method, indicating which solver to use:

```
>>> solver = pulp.getSolver('PULP_CBC_CMD', msg=False)
>>> prob.solve(solver)
1
```

This does output one important piece of information, a status code of `1`, indicating that the problem was feasible and solved to optimality. The solve method assigns values to the decision variables corresponding to the optimal solution. To see these values we use:

```
>>> x[1].value()
6.8
>>> x[2].value()
0.8
```

Indicating that the optimal solution to this problem is $(x_1 = 6.8, x_2 = 0.8)$.

Other options

The objective function can either be maximised or minimised:

Objective	Code
Maximise	`pulp.LpProblem("MyProblem", pulp.LpMaximize)`
Minimise	`pulp.LpProblem("MyProblem", pulp.LpMinimize)`

Decision variables can be restricted to from particular categories:

Category	Code
Real numbers ($\mathbb{R}$)	`pulp.LpVariable.dicts("x", [1, 2], cat='Continuous')`
Integers ($\mathbb{Z}$)	`pulp.LpVariable.dicts("x", [1, 2], cat='Integer')`
Binary ($\{0, 1\}$)	`pulp.LpVariable.dicts("x", [1, 2], cat='Binary')`

The status code that is returned by the solve method can be recalled with:

```
>>> prob.status
1
```

This can take a number of values depending on the status of the problem:

Status code	Explanation
1	The problem has been solved to optimality
0	The problem has not been solved yet
-1	The problem is infeasible
-2	The problem is unbounded
-3	The problem is undefined

Exam scheduling example

A university department teaches 15 modules, and each module requires an examination be be scheduled on a day during the exam period. The exam period is 15 days long, but the department would like to ensure that the number of days used for the exam is minimised. Some modules cannot have their exams scheduled on the same day, as they share students. These clashed are given by the clashes matrix C below, where rows and columns correspond to modules, the matrix contains a 1 if they share students, and a zero otherwise.

Which exams should be scheduled on which days, to minimise the number of days used?

$$C = \begin{pmatrix}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0
\end{pmatrix}$$

Formulation: Let M be the set of 15 modules, indexed by m, and let D be the set of 15 possible days, indexed by d, and let C be the clashed matrix above. Define two sets of decision variables: let x_{md}, for all $m \in M$ for all $d \in D$, be a binary variable indicating if exam m is scheduled of day d; and let y_d, for all $d \in D$, be a binary variable representing if day d is used. Then the formulation becomes:

Minimise:

$$\sum_{d \in D} y_d \tag{17.6}$$

subject to

$$\sum_{d \in D} x_{md} = 1 \qquad \forall \quad m \in M \tag{17.7}$$

$$x_{m_1 d} + x_{m_2 d} \leq 2 - C_{m_1 m_2} \qquad \forall \quad m_1, m_2 \in M, d \in D \tag{17.8}$$

$$|M| y_d \geq \sum_{m \in M} x_{md} \qquad \forall \quad d \in D \tag{17.9}$$

$$\tag{17.10}$$

where the objective (17.6) is minimising the days used; 17.7 ensures every exam is scheduled once and only once; 17.8 avoids clashes; and 17.9 defines y_d, ensuring it takes a value of 1 if the day is used by a module.

Now to solve in PuLP, first define C, M and D as `clashes`, `modules`, and `days`.

```
>>> modules = range(15)
>>> days = range(15)
>>> clashes = [
...     [0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
...     [1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
...     [1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
...     [1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
...     [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
...     [1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0],
...     [1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0],
...     [1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0],
...     [1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0],
...     [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0],
...     [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0],
...     [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
...     [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],
...     [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1],
...     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0]
... ]
```

Now to define the problem, decision variables, objective, and constraints. Here, the translation into Python is straightforward, was all summations ($\sum$) can be coded with `sum` operators, and all $\forall$ repetitions can be placed inside a for loop:

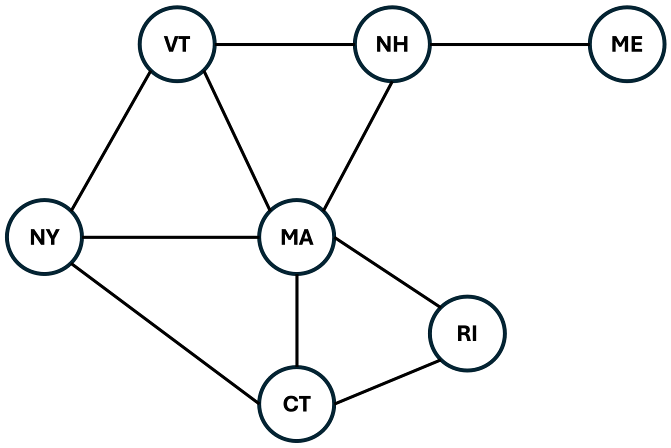
```
>>> prob = pulp.LpProblem("ExamScheduling", pulp.LpMinimize)
>>> x = pulp.LpVariable.dicts("x", (modules, days), cat='Binary')
>>> y = pulp.LpVariable.dicts("y", days, cat='Binary')

>>> # Objective
>>> prob += sum(y[d] for d in days)

>>> # Constraints
>>> for m in modules:
...     prob += sum(x[m][d] for d in days) == 1

>>> for m1 in modules:
...     for m2 in modules:
...         for d in days:
...             prob += x[m1][d] + x[m2][d] <= 2 - clashes[m1][m2]

>>> for d in days:
...     prob += (15 * y[d]) >= sum(x[m][d] for m in modules)
```

Now to solve, and print off the solution:

```
>>> prob.solve(solver)
1
>>> for d in days:
...     exams = [m for m in modules if x[m][d].value() == 1]
...     if len(exams) != 0:
...         print(f'Day {d}', exams)
Day 0 [2, 8]
Day 1 [0, 13]
Day 2 [1, 11]
Day 4 [4, 5, 9, 14]
Day 5 [3, 6, 10]
Day 6 [12]
Day 12 [7]
```

18 Graph Theory

The NetworkX library allows us to represent, analyse, and perform algorithms on graphs. A graph is a tuple of sets, $G = (V, E)$, where V is the set of vertices, and E is the set of edges that connect them. Consider the graph below:



This is a simple graph, where edges are simply connections, they do not have directions. It has set of vertices

$$V = \{\text{CT}, \text{MA}, \text{ME}, \text{NH}, \text{NY}, \text{RI}, \text{VT}\},$$

and set of edges

$$\begin{aligned} E = \{&(\text{CT}, \text{MA}),\\ &(\text{CT}, \text{NY}),\\ &(\text{CT}, \text{RI}),\\ &(\text{MA}, \text{NH}),\\ &(\text{MA}, \text{NY}),\\ &(\text{MA}, \text{VT}),\\ &(\text{ME}, \text{NH}),\\ &(\text{NH}, \text{VT}),\\ &(\text{NY}, \text{VT})\}. \end{aligned}$$

Creating graphs

There are a number of ways of creating graphs with NetworkX. We will look at two main methods, from an edge list, and from an adjacency matrix. First, it is convention to import NetworkX with the alias `nx`. We can then create a graph from a list of edges:

```
>>> import networkx as nx
>>> edge_list = [
...     ("CT", "MA"),
...     ("CT", "NY"),
...     ("CT", "RI"),
...     ("MA", "NH"),
...     ("MA", "NY"),
...     ("MA", "RI"),
...     ("MA", "VT"),
...     ("ME", "NH"),
...     ("NH", "VT"),
...     ("NY", "VT")
... ]

>>> G = nx.Graph()
>>> G.add_edges_from(edge_list)
```

This gives the graph nodes (or vertices) and edges. Notice here, we are using string as nodes. We can use any hashable type as a node, that is, anything that can be a dictionary key can be a node of the graph.

We can view the nodes and edges of this graph:

```
>>> G.nodes
NodeView(('CT', 'MA', 'NY', 'RI', 'NH', 'VT', 'ME'))

>>> for e in G.edges:
...     print(e)
('CT', 'MA')
('CT', 'NY')
('CT', 'RI')
('MA', 'NH')
('MA', 'NY')
('MA', 'RI')
('MA', 'VT')
('NY', 'VT')
('NH', 'ME')
('NH', 'VT')
```

Another common way of creating graphs is by defining an adjacency matrix. This is a matrix where rows and columns represent nodes, and entries are binary, a one or a zero, indicating if there is an edge between the nodes or not. In our case:

	CT	MA	ME	NH	NY	RI	VT
CT	0	1	0	0	1	1	0
MA	1	0	0	1	1	1	1
ME	0	0	0	1	0	0	0
NH	0	1	1	0	0	0	1
NY	1	1	0	0	0	0	1
RI	1	1	0	0	0	0	0
VT	0	1	0	1	1	0	0

We can define this adjacency matrix with a numpy array, and create the graph from that array. Doing so assumes that the node labels are the row and column indices. So we can relabel the nodes with a mapping from indices to strings:

```
>>> adj_matrix = np.array([
...     [0, 1, 0, 0, 1, 1, 0],
...     [1, 0, 0, 1, 1, 1, 1],
...     [0, 0, 0, 1, 0, 0, 0],
...     [0, 1, 1, 0, 0, 0, 1],
...     [1, 1, 0, 0, 0, 0, 1],
...     [1, 1, 0, 0, 0, 0, 0],
...     [0, 1, 0, 1, 1, 0, 0]
... ])
>>> mapping = {
...     0: 'CT',
...     1: 'MA',
...     2: 'ME',
...     3: 'NH',
...     4: 'NY',
...     5: 'RI',
...     6: 'VT'
... }
>>> H = nx.from_numpy_array(adj_matrix)
>>> H = nx.relabel_nodes(H, mapping)

>>> G.nodes == H.nodes
True
>>> G.edges == H.edges
True
```

Subgraphs

A graph $\tilde{G} = (\tilde{V}, \tilde{E})$ is a subgraph of a graph $G = (V, E)$, if $\tilde{V} \subseteq V$ and $\tilde{E} \subseteq E$. We can check these with Python's set data structures that we saw in Chapter 7:

```
>>> H = nx.Graph()
>>> H.add_edges_from([
...     ("CT", "MA"),
...     ("CT", "NY"),
...     ("CT", "RI")
... ])
>>> set(H.nodes).issubset(set(G.nodes))
True
>>> set(H.edges).issubset(set(G.edges))
True
```

A subgraph can be *induced*, that is created from, another graph, and a subset of its nodes. This means that it will contain only that subset of nodes, and only edges that appear in the original graph and connect those nodes:

```
>>> H = G.subgraph(['MA', 'ME', 'NH', 'VT'])
>>> H.nodes
NodeView(('MA', 'NH', 'VT', 'ME'))
>>> H.edges
EdgeView([('MA', 'NH'), ('MA', 'VT'), ('NH', 'ME'), ('NH', 'VT')])
```

Attributes

NetworkX allows us to look at the structure of these graphs and their attributes. For example, we can list the neighbours of a node:

```
>>> for neighbour in nx.all_neighbors(G, 'MA'):
...     print(neighbour)
CT
NH
NY
RI
VT
```

A node's *degree* is the number of edges incident to it:

```
>>> G.degree("MA")
5
>>> G.degree("ME")
1
```

Once a graph has been created, edges can be added:

```
>>> G.degree("RI")
2
>>> G.add_edge('NY', 'RI')
>>> G.degree("RI")
3
```

and removed:

```
>>> G.degree("RI")
3
>>> G.remove_edge('NY', 'RI')
>>> G.degree("RI")
2
```

A path is a set of edges that lead from one node to another, through other nodes that are connected by edges. For example VT-NH-ME is a path, as the edge (VT, NH) exists, and so does the edge (NH, ME). We can find all possible paths between two nodes (that do not enter a node more than once) with `all_simple_paths`. For all paths between NY and RI:

```
>>> for path in nx.all_simple_paths(G, 'NY', 'RI'):
...     print(path)
['NY', 'CT', 'MA', 'RI']
['NY', 'CT', 'RI']
['NY', 'MA', 'CT', 'RI']
['NY', 'MA', 'RI']
['NY', 'VT', 'MA', 'CT', 'RI']
['NY', 'VT', 'MA', 'RI']
['NY', 'VT', 'NH', 'MA', 'CT', 'RI']
['NY', 'VT', 'NH', 'MA', 'RI']
```

A cycle is a path that begins and ends in the same node. Similarly, all cycles can be found with `simple_cycles`. We can restrict the length of these cycles with the keyword `length_bound`. For all cycles of length 3 or less in the graph:

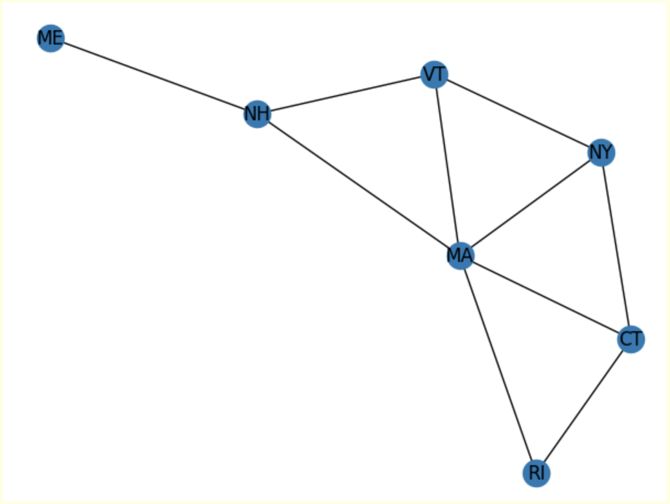
```
>>> for cycle in sorted(nx.simple_cycles(G, length_bound=3)):
...     print(cycle)
['CT', 'MA', 'NY']
['CT', 'MA', 'RI']
['MA', 'NH', 'VT']
['MA', 'NY', 'VT']
```

A bridge is an edge which, if deleted, would separate the graph into two components. That is, if a bridge is deleted, there would be two node from which it would be impossible to travel from one to the other. That is, there are two node for which every path between them contains the bridge. We can find all bridges with `nx.bridges`:

```
>>> for b in nx.bridges(G):
...     print(b)
('NH', 'ME')
```

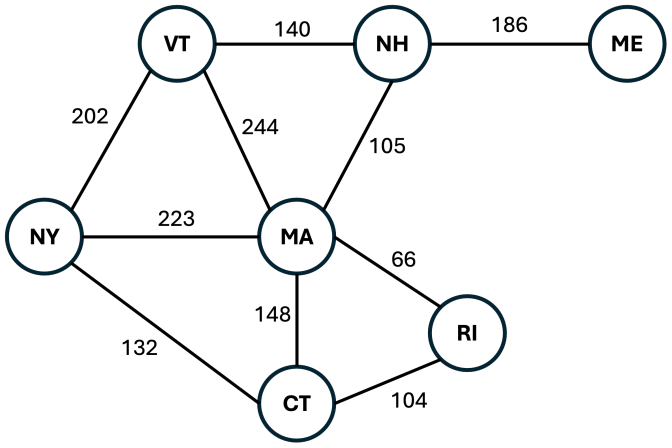
We can also visualise the graph with `draw`. However, be careful with this, as graphs are inherently an abstract mathematical concept, so the visualisation can be random and unhelpful a lot of the time, especially with larger graphs.

```
>>> nx.draw(G, with_labels=True)
```



Weighted & directed edges

Edges can have attributes, such as a distance or cost measure. This is called a weighted edge. For example, consider the graph below:



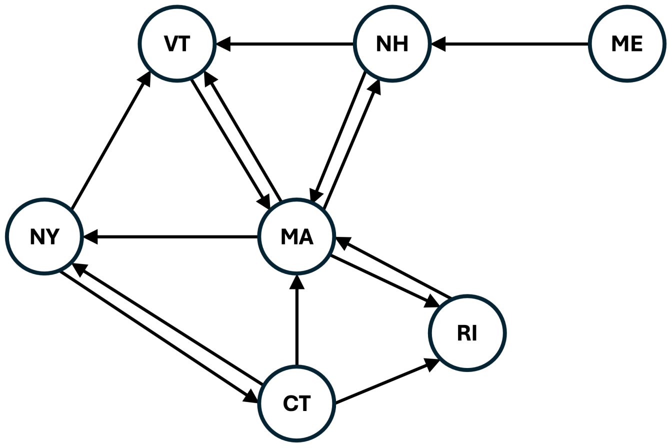
We can create this by replacing `add_edges_from` with `add_weighted_edges_from`:

```
>>> edge_list = [
...     ("CT", "MA", 148),
...     ("CT", "NY", 132),
...     ("CT", "RI", 104),
...     ("MA", "NH", 105),
...     ("MA", "NY", 223),
...     ("MA", "RI", 66),
...     ("MA", "VT", 244),
...     ("ME", "NH", 186),
...     ("NH", "VT", 140),
...     ("NY", "VT", 202)
... ]

>>> W = nx.Graph()
>>> W.add_weighted_edges_from(edge_list)
```

If edges have weights, then so do paths and cycles, corresponding to the sum of the edge weights over all the edges in the path. We can calculate this:

```
>>> path = ['NY', 'CT', 'MA', 'RI']
>>> nx.path_weight(W, path, weight='weight')
346
```

Similarly edges can have directions, called a directed graph, or digraph. Consider the graph below:



We can create this by replacing `nx.Graph()` with `nx.DiGraph()`, and noting the order of the vertices in each tuple within the edge list:

```
>>> edge_list = [
...     ("CT", "MA"),
...     ("CT", "NY"),
...     ("CT", "RI"),
...     ("MA", "NH"),
...     ("MA", "NY"),
...     ("MA", "RI"),
...     ("MA", "VT"),
...     ("ME", "NH"),
...     ("NH", "MA"),
...     ("NH", "VT"),
...     ("NY", "CT"),
...     ("NY", "VT"),
...     ("RI", "MA"),
...     ("VT", "MA"),
... ]
>>> D = nx.DiGraph()
>>> D.add_edges_from(edge_list)
```

This now means that some paths only exist in one direction. Consider the node ME, there is an edge out of ME but not edge into it. We can verify this with the function `nx.has_path`:

```
>>> nx.has_path(D, 'ME', 'CT')
True
>>> nx.has_path(D, 'CT', 'ME')
False
```

With digraphs, the concept of a degree is extended. An *in-degree* of a node is the number of edges into that node, while an *out-degree* is the number of edges out of that node:

```
>>> D.in_degree('CT')
1
>>> D.out_degree('CT')
3
```

Shortest Paths

NetworkX can be used to find the shortest path from one vertex to another, by default using Dijkstra's algorithm. Note that this would behave differently for simple, weighted, and directed graphs, as the concept of 'shortest' for each of these correspond to number of edges, total edge weight, and number of directed edges in the required direction.

For example, consider the shortest path from VT to CT in each of the graphs:

```
>>> nx.shortest_path(G, 'VT', 'CT')
['VT', 'MA', 'CT']
>>> nx.shortest_path(W, 'VT', 'CT', weight='weight')
['VT', 'NY', 'CT']
>>> nx.shortest_path(D, 'VT', 'CT')
['VT', 'MA', 'NY', 'CT']
```

In the simple graph `G` there are two shortest paths from VT to CT, both consisting of two edges, and this algorithm has chosen one arbitrarily. In the weighted graph `W`, both those paths have different edge weights, and this algorithm has determined which of the two has the least total weight. In the directed graph `D`, two of the edges previously considered are not possible due to the edge directions, (VT, NY) and (MA CT), therefore the shortest path now is three directed edges.

The average shortest path length can tell us something about the spread of the graph:

```
>>> nx.average_shortest_path_length(W, weight='weight')
243.28571428571428
```

Graph centralities

Graph centralities are measures on each node indicating how 'central', or 'important', or 'connected' that node is to the rest of the graph. There are a number of ways of defining this, and dozens are implemented in NetworkX. Here we will show three main ones:

Centrality measure | Example

Degree Centrality
The fraction of the nodes each node is connected to.

```
>>> for n in G.nodes:
...     print(n, nx.degree_centrality(G)[n])
CT 0.5
MA 0.8333333333333333
NY 0.5
RI 0.3333333333333333
NH 0.5
VT 0.5
ME 0.16666666666666666
```

Closeness Centrality
The reciprocal of the average shortest path between each node and every other node.

```
>>> for n in G.nodes:
...     print(n, nx.closeness_centrality(G)[n])
CT 0.6
MA 0.8571428571428571
NY 0.6
RI 0.5454545454545454
NH 0.6666666666666666
VT 0.6666666666666666
ME 0.42857142857142855
```

Betweenness Centrality
The fraction of all possible shortest paths that pass through each node.

```
>>> for n in G.nodes:
...     print(n, nx.betweenness_centrality(G)[n])
CT 0.03333333333333333
MA 0.4666666666666667
NY 0.03333333333333333
RI 0.0
NH 0.3333333333333333
VT 0.06666666666666667
ME 0.0
```

Graph algorithms

There are a number of useful graph algorithms commonly used in operational research that are implemented in NetworkX. A dominating set is a subset of the nodes such that all other nodes in the graph are connected to at least one of them; while a minimum edge covering is a minimal set of edges such that all nodes in the graph are incident to at least on edge in the set:

```
>>> nx.dominating_set(G)
{'CT', 'NH'}
>>> nx.min_edge_cover(G)
{('VT', 'MA'), ('RI', 'CT'), ('ME', 'NH'), ('CT', 'NY')}
```

A spanning tree of a graph is a subgraph that contains all nodes, a set of edges that censure a path between each node, but contains no cycles. A minimum spanning tree is a spanning tree whose total edge weights are minimised:

```
>>> MST = nx.minimum_spanning_tree(W, weight='weight')
>>> for edge in MST.edges:
...     print(edge)
('CT', 'RI')
('CT', 'NY')
('MA', 'RI')
('MA', 'NH')
('NH', 'VT')
('NH', 'ME')
>>> [c for c in nx.simple_cycles(MST)]
[]
```

A minimum edge cut gives the minimum number of edges required to disconnect two nodes, that is, the minimum number of edges to cut so that there does not exist a path between two nodes:

```
>>> nx.minimum_edge_cut(G, 'NY', 'RI')
{('CT', 'RI'), ('MA', 'RI')}
```

Finally, a graph colouring is a labelling of the nodes such that no two nodes that are connected by an edge have the same label. A minimum colouring, a colouring that uses the least amount of colours, is difficult, however greedy algorithms exist to find a general colouring:

```
>>> nx.coloring.greedy_color(G)
{'MA': 0, 'CT': 1, 'NY': 2, 'NH': 1, 'VT': 3, 'RI': 2, 'ME': 0}
```

19 Markov Decision Processes

In a similar way to how Markov chains describe the probabilities of transitioning between a set of states, Markov decision processes, or MDPs, describe the probabilities of transitioning between states and gaining rewards in situations where decisions can inpact the outcome.

Formally, an MDP is a 4-tuple (S, A, P, R), where:

- S is a set of states;
- A is a set of actions, and $A_s \subseteq A$ a subset of the actions that can be taken when in state $s \in S$;
- P is a probability matrix such that $p_{sas'}$ is the probability of transitioning to state $s' \in S$ after taking action $a \in A_s$ when in state $s \in S$;
- R is a matrix of rewards such that r_{sa} is the reward obtained for taking action $a \in A_s$ when in state $s \in S$.

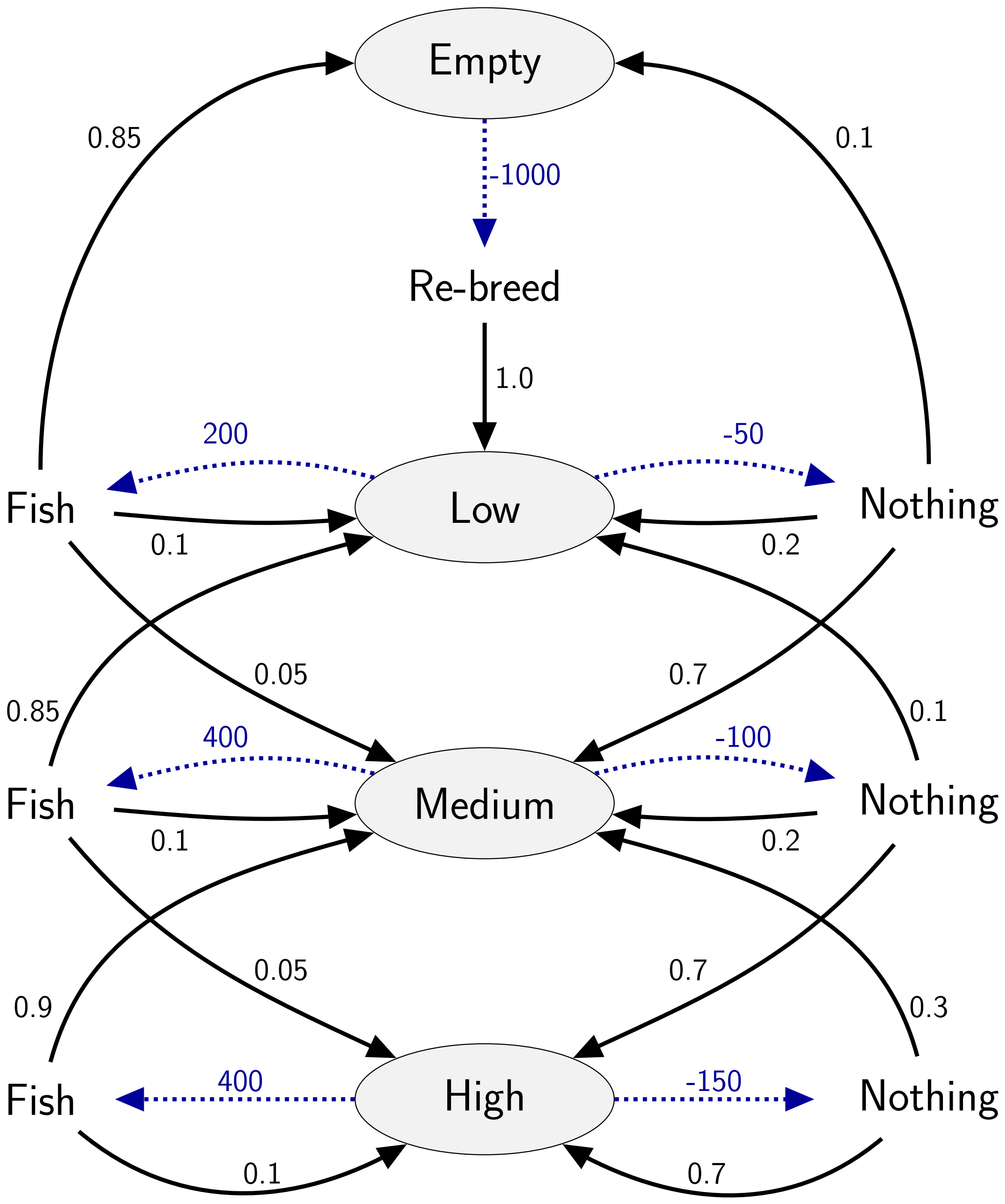
In general we wich to find a policy $\pi(s)$ for all $s \in S$, that is a mapping from states to actions, that will maximise the rewards.

Example

As an example, consider a fishery. The fish stocks can be in four states: Empty, Low, Medium, or High. When the fishery stocks are not empty, each day we can choose to fish or do nothing:

- we gain $£200$ if we fish when stocks are low;
- we gain $£400$ is we fish when stocks are medium or high;
- if we do nothing we pay $£50$ in maintenance when stocks are low, $£100$ when stocks are medium, and $£150$ when stocks are high;
- if stocks run out, we are forced to re-breed the fish, which costs $£1000$.

The probabilities of fish stocks transitioning between low, medium, high, and empty depend on the actions taken. The states, actions, rewards and probabilities are shown in the following diagram:


Empty
-1000
Re-breed
1.0
0.85
0.1
200
-50
Fish
Low
Nothing
0.1
0.2
0.05
0.7
0.85
0.1
400
-100
Fish
Medium
Nothing
0.1
0.2
0.05
0.7
0.9
0.3
Fish
400
High
-150
Nothing
0.1
0.7

We can store the MDP in lists and dictionaries:

```
>>> states = ['Empty', 'Low', 'Medium', 'High']

>>> actions = {
...     'Empty': ['Re-breed'],
...     'Low': ['Fish', 'Nothing'],
...     'Medium': ['Fish', 'Nothing'],
...     'High': ['Fish', 'Nothing']
... }

>>> rewards = {
...     'Empty': {'Re-breed': -1000},
...     'Low': {'Fish': 200}, 'Nothing': -50,
...     'Medium': {'Fish': 400, 'Nothing': -100},
...     'High': {'Fish': 400, 'Nothing': -150}
... }

>>> probs = {
...     'Empty': {
...         'Re-breed': {'Low': 1.0},
...     },
...     'Low': {
...         'Fish': {'Empty': 0.85, 'Low': 0.1, 'Medium': 0.05},
...         'Nothing': {'Empty': 0.1, 'Low': 0.2, 'Medium': 0.7},
...     },
...     'Medium': {
...         'Fish': {'Low': 0.85, 'Medium': 0.1, 'High': 0.05},
...         'Nothing': {'Low': 0.1, 'Medium': 0.2, 'High': 0.7},
...     },
...     'High': {
...         'Fish': {'Medium': 0.9, 'High': 0.1},
...         'Nothing': {'Medium': 0.3, 'High': 0.7}
...     }
... }

>>> MDP = {
...     'states': states,
...     'actions': actions,
...     'rewards': rewards,
...     'probs': probs
... }
```

Value Iteration

One way to find an optimal policy is to assign an expected long-term reward to each state, $V(s)$. That is, given we are in state $s \in S$, and assuming that we will follow the optimal policy forevermore, what will be out expected long-term reward? The Bellman equation below gives that:

$$V(s) = \max_{a \in A_s} \left(r_{sa} + \gamma \sum_{s' \in S} p_{sas'} V(s') \right)$$

that is, the maximum value we get out of all the actions (as we are assuming we are folloing an optimal policy), of the current reward, plus the expected value of any state we end up in by taking that action, representing expected future rewards. This future reward is *discounted* with a discouting factor $0 < \gamma < 1$, ensuring convergence - this could for example represent interest rate, that tomorrow's rewards are worth less than today's rewards.

We can solve this by iterating the values until convergence, called value iteration:

Algorithm 1: Value Iteration

```
$\Delta \leftarrow \infty$;
while $\Delta \geq \epsilon$ do
    $\tilde{V}(s) = \max_{a \in A_s} \left( r_{sa} + \gamma \sum_{s' \in S} p_{sas'} V(s') \right)$;
    $\Delta = \max_s |\tilde{V}(s) - V(s)|$;
    $V(s) \leftarrow \tilde{V}(s)$;
end
```

In Python, we can do this with a loop:

```
>>> def value_iteration(MDP, discount_factor, epsilon):
...     Vs = {s: 0.0 for s in MDP['states']}
...     delta = float('inf')
...     while delta > epsilon:
...         next_Vs = {}
...         for s in states:
...             next_Vs[s] = max(
...                 MDP['rewards'][s].get(a, 0.0) + (
...                     discount_factor * sum(
...                         MDP['probs'][s][a].get(sdash, 0.0) * Vs[sdash]
...                         for sdash in MDP['states']
...                     )
...                 ) for a in MDP['actions'][s]
...             )
...         delta = max(abs(next_Vs[s] - Vs[s]) for s in MDP['states'])
...         Vs = next_Vs
...     return Vs
```

Recall the dictionary method `.get`, which allows us to loop over states that may not exist in the `probs` or `rewards` dictionaries, and still return a sensible value, in this case a 0.

Once these values have been found, we can obtain the policy easily with the following one step lookup:

$$\pi(s) = \arg\max_{a \in A_s} \left(r_{sa} + \gamma \sum_{s' \in S} p_{sas'} V(s') \right)$$

And in Python:

```
>>> def get_policy_from_values(MDP, discount_factor, Vs):
...     policy = {}
...     for s in MDP['states']:
...         action_values = {
...             a: MDP['rewards'][s].get(a, 0.0) + (
...                 discount_factor + sum(
...                     MDP['probs'][s][a].get(sdash, 0.0) * Vs[sdash]
...                     for sdash in MDP['states']
...                 )
...             ) for a in MDP['actions'][s]
...         }
...         policy[s] = max(action_values, key=action_values.get)
...     return policy
```

Applying these functions to the MDP defined above gives:

```
>>> Vs = value_iteration(
...     MDP=MDP,
...     discount_factor=0.99,
...     epsilon=0.00001
... )
>>> policy = get_policy_from_values(
...     MDP=MDP,
...     discount_factor=0.99,
...     Vs=Vs
... )
>>> policy
{'Empty': 'Re-breed', 'Low': 'Nothing', 'Medium': 'Fish', 'High': 'Fish'}
```

This tells us that we should only fish when the stocks are medium or high, and do nothing when stocks are low.

Q-Learning

In value iteration we have nearly a whole model of the system defined, by states and probabilities. In some cases this is not possible, and we must rely on simulation. In such cases, we can still perform find an optimal policy using Q-learning, a model-free algorithm. The simulation of the system will visit sets of states $s \in S$, and choose actions $a \in A_s$ to perform; will observe some reward r_{sa}, and reach a new state $s' \in S$.

Now instead of updating the value of each state with the expected long term reward, we update state-action pairs with the expected long term reward for taking that action in that state, and following the optimal policy from that point onwards, $Q(s, a)$. Updates happen as and when state-action pairs are visited according to the Q-learning update rule:

$$Q(s,a) \leftarrow \alpha \left(r_{sa} + \gamma \max_{a' \in A_{s'}} Q(s', a') \right) + (1 - \alpha) Q(s, a)$$

where again γ is a discount factor, and now α is a learning rate, representing how much we trust the new information we receive, due to stochasticity. The update in Python is:

```
>>> def update_qs(Qs, s, a, sdash, r, MDP, discount_factor, learning_rate):
...     new_Q = r + (
...         discount_factor * max(
...             Qs[sdash][adash] for adash in MDP['actions'][sdash]
...             )
...         )
...     Qs[s][a] = (learning_rate * new_Q) + ((1 - learning_rate) * Qs[s][a])
```

A policy can then be found by considering, for each state, which action gives the best value of $Q(s, a)$:

```
>>> def get_policy_from_Qs(MDP, Qs):
...     policy = {}
...     for s in MDP['states']:
...         policy[s] = max(Qs[s], key=Qs[s].get)
...     return policy
```

The Q-values will converge under repeated sampling such as a simulation. However, detecting convergence within a stochastic simulation can be difficult. Therefore we should perform this update for an arbitrary number of iterations in order to get an approximation of an optimal policy.

As an illustration, let's solve the fisheries MDP using Q-learning. Instead of probabilities, we now need a simulation of the system, and so we will need to sample a next state from the probabilities:

```
>>> import random
>>> def next_state(s, a, MDP):
...     return random.choices(
...         list(MDP['probs'][s][a].keys()),
...         list(MDP['probs'][s][a].values())
...     )[0]
```

The simulation will begin in randomly chosen state. Then from that state an action will be chosen randomly, the next state sampled, and the Q-values updates. This will then repeat, using the sampled state as the next starting state each time:

```
>>> def Q_learning(MDP, discount_factor, learning_rate, num_iterations):
...     Qs = {s: {a: 0.0 for a in actions[s]} for s in MDP['states']}
...     s = random.choice(MDP['states'])
...     for iteration in range(num_iterations):
...         a = random.choice(list(MDP['actions'][s]))
...         sdash = next_state(s, a, MDP)
...         r = MDP['rewards'][s].get(a, 0.0)
...         update_qs(Qs, s, a, sdash, r, MDP, discount_factor, learning_rate)
...         s = sdash
...     return Qs
```

Now performing this on the fisheries MDP:

```
>>> Qs = Q_learning(
...     MDP=MDP,
...     discount_factor=0.99,
...     learning_rate=0.5,
...     num_iterations=10000
... )
>>> policy = get_policy_from_Qs(MDP=MDP, Qs=Qs)
>>> policy
{'Empty': 'Re-breed', 'Low': 'Nothing', 'Medium': 'Fish', 'High': 'Fish'}
```

20 Metaheuristics

Metaheuristics are algorithms that cleverly search for a solution space for an optimal solution to some problem. They are approximate methods, and cannot guarantee an optimal solution, but can get close. They take advantage of randomly generating or randomly changing solutions, and comparing to previously checked solutions.

There are numerous variants of metaheuristics, and each part, such as evaluating a solution, or changing a solution, can be swapped out for another method. This makes Python classes and inheritance ideal implementation strategies for them.

Random Descent Methods

The idea behind random descent methods is to iterate on a solution, making small changes via a *neighbourhood operator*, and evaluating the new solution. If the new solution is better than the previous solution, keep it and iterate on this new solution. Otherwise disregard that change and try again. In general:

Algorithm 2: Random Descent

```
Randomly generate a solution $s$;
Evaluate that solution $v_{\text{best}} \leftarrow v(s)$;
for $n$ iterations do
    Apply neighbourhood operator to get new solution: $s' \leftarrow N(s)$;
    Evaluate $s'$: $v(s')$;
    if $v(s') < v_{best}$ then
        $s \leftarrow s'$;
        $v_{\text{best}} \leftarrow v(s')$;
    end
end
Output: $s$
```

The evaluation function v and form of the solution s will depend on the problem itself. A choice neighbourhood operator N will determine how effective the metaheuristic is in finding a good solution.

Let's consider an example of finding a solution to the travelling salesman problem (TSP). Given a set of cities, and distances between each city, what order of cities will minimise the total route around the cities, returning to the original location?

Let's look at an example of 100 Spanish cities, first load the data:

```
>> cities = pd.read_csv('spain.csv', names=['City', 'Lat', 'Long'])
>> distances = np.genfromtxt('spain_distances.csv', delimiter=',')
```

This gives:

```
>>> cities
                    City        Lat       Long
 0              Albacete  38.993361  -1.857049
 1      Alcalá de Henares  40.483345  -3.366698
 2            Alcobendas  40.537026  -3.637362
 3              Alcorcón  40.346813  -3.826688
 4    Alcázar de San Juan  39.390069  -3.210149
...                   ...        ...        ...
95            Valladolid  41.651979  -4.728730
96                  Vigo  42.240557  -8.722127
97       Vitoria-Gasteiz  42.850905  -2.673291
98                 Zafra  38.423940  -6.417711
99              Zaragoza  41.651983  -0.880433

100 rows × 3 columns
```

and a 100×100 distance matrix. For example the distance between Albacete (index 0) and Alcalá de Henares (index 1) is 210.03 miles:

```
>>> distances.shape
(100, 100)

>>> distances[0, 1]
210.02459584632692
```

We can write a class to perform the random descent here:

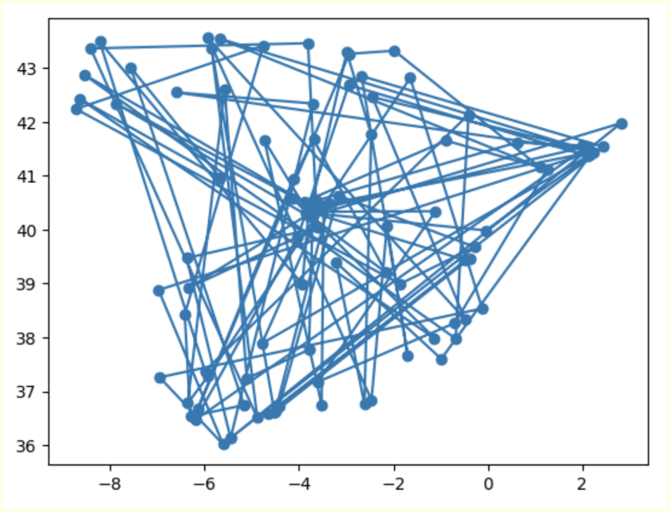
- A solution s is an ordering of the cities, that is a list of numbers from 0 to 99 in a particular order;
- The evaluation function $v(s)$ is the successive distances between those cities in that order, plus the distance from the final city back to the first city;
- The neighbourhood operator that we will use is to swap the order of two cities.

The full class is given by:

```
>>> class TSP_RandomDescent:
...     def __init__(self, cities, distances):
...         self.cities = cities
...         self.n_cities = len(cities)
...         self.distances = distances
...         self.get_initial_solution()
...
...     def get_initial_solution(self):
...         solution = list(self.cities.index)
...         random.shuffle(solution)
...         self.solution = solution
...
...     def plot_solution(self):
...         fig, ax = plt.subplots()
...         lats = [self.cities.loc[s, 'Lat'] for s in self.solution]
...         longs = [self.cities.loc[s, 'Long'] for s in self.solution]
...         plt.scatter(longs, lats)
...         plt.plot(longs + [longs[0]], lats + [lats[0]])
...         return fig
...
...     def evaluate_solution(self, sol):
...         return sum(
...             self.distances[i, j] for i, j in zip(sol, sol[1:] + [sol[0]])
...         )
...
...     def neighbourhood_operator(self):
...         c1, c2 = random.sample(list(range(self.n_cities)), k=2)
...         new_sol = list(self.solution)
...         new_sol[c1], new_sol[c2] = new_sol[c2], new_sol[c1]
...         return new_solution
...
...     def optimise(self, n_iters):
...         self.best_score = self.evaluate_solution(self.solution)
...         for iter in range(n_iters):
...             new_sol = self.neighbourhood_operator()
...             new_score = self.evaluate_solution(new_sol)
...             if new_score < self.best_score:
...                 self.best_score = new_score
...                 self.solution = new_sol
```

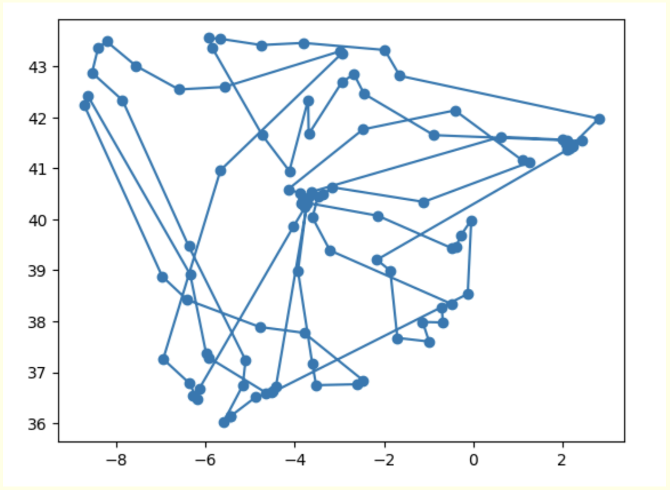
Now we can use this on the Spanish cities. Consider what the initial solution looks like *before* optimisation:

```
>>> random.seed(0)
>>> A = TSP_RandomDescent(cities, distances)
>>> A.plot_solution();
```



Then running the random descent for 20,000 iterations we get a total route length of 11,314.2 miles, and see that the route looks a lot less untangled:

```
>>> A.optimise(n_iters=20000)
>>> A.best_score
11315.194960209916

>>> A.plot_solution();
```

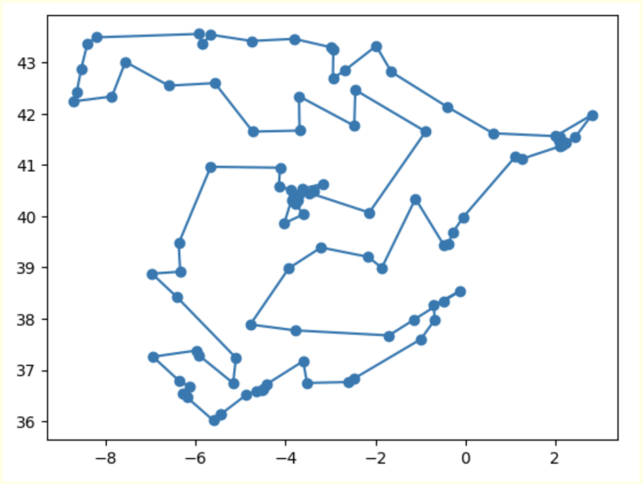


The neighbourhood operator of swapping two cities is a little naive. A better neighbourhood operator for the travelling salesman problem is called the 2-opt operator: rather than swapping two randomly chosen cities, it reverses the path between two randomly chosen cities. It is known to perform better - let's try it out. We only need to replace one method, the `neighbourhood_operator`, and so we can use Python's class inheritance to quickly try it out:

```
>>> class TSP_2Opt(TSP_RandomDescent):
...     def neighbourhood_operator(self):
...         c1, c2 = random.sample(list(range(self.n_cities)), k=2)
...         c1, c2 = sorted([c1, c2])
...         new_sol = self.solution[:c1]
...         new_sol += self.solution[c1:c2][::-1]
...         new_sol += self.solution[c2:]
...         return new_sol
```

Using the same random seed, and so starting from the same initial solution, with the same number of iterations, we see that using the 2-opt operator improves the solution - nearly cutting the total route in half, and producing a much less tangled route:

```
>>> random.seed(0)
>>> B = TSP_2Opt(cities, distances)
>>> B.optimise(n_iters=20000)
>>> B.best_score
6183.848066059243
>>> B.plot_solution();
```

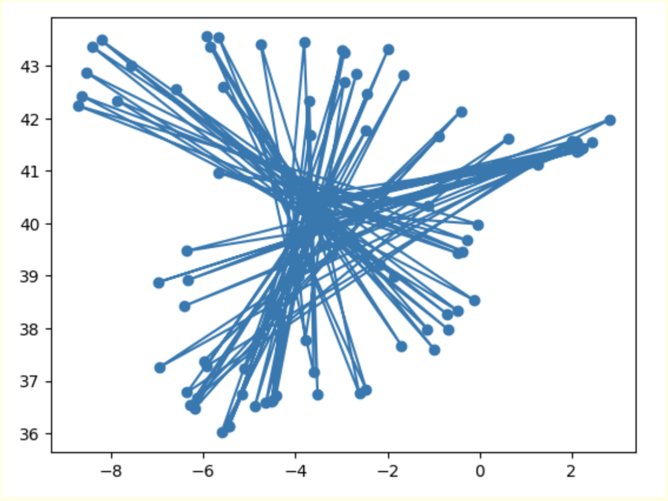


This demonstrates the effectiveness of using modular classes and inheritance, where various elements of the problem or algorithm can be swapped out in a plug-and-play fashion. To further emphasise the point, imagine we wish to maximise the route. Them all we need to do is create a new class that inherits from the old one, re-writing the evaluation function:

```
>>> class TSP_Max(TSP_RandomDescent):
...     def evaluate_solution(self, sol):
...         return 1 / sum(
...             self.distances[i, j] for i, j in zip(sol, sol[1:] + [sol[0]])
...         )
```

Giving:

```
>>> random.seed(0)
>>> C = TSP_Max(cities, distances)
>>> C.optimise(n_iters=20000)
>>> C.best_score
1.6139972180336087e-05
>>> C.plot_solution();
```



Descent methods are not perfect, they can become 'stuck' in a local optimum, that is a solution in which all instances of the neighbourhood operator will lead to a worse solution; however there remains an better global optimum unfound. In order to overcome this, we may introduce rules in which sometimes worse solutions are accepted, examples include:

- **Tabu search**: In which a partial list of previously visited solutions are kept in a list, and the algorithm is forbidden from returning to these solutions, possibly choosing worse candidates in their place;
- **Simulated annealing**: In which there is a probability of accepting worse solutions at each iteration, and that probability decreases with the iterations.

Population-Based Methods

In population-based methods a number of possible solutions are evaluated simultaneously, and survive to the next iteration based on their evaluations, and 'breeding' new solutions. These are akin to an evolutionary process. In general:

Algorithm 3: Population-Based

```
Generate a population of random solutions $P = \{s_1, s_2, s_3, \dots, s_N\}$;
for $g$ generations do
    Rank each solution in the population by the evaluation function;
    Remove the worst performing solutions;
    Repopulate the population by 'breeding' the remaining solutions;
end
```

Here 'breeding' solutions is open to interpretation. We could for example:

- copy a surviving solution and apply a neighbourhood operator on it: this is called **mutation**;
- combine two surviving functions in some manner: this is called **crossover**.

As an example, say we with to find six positive integers u, v, w, x, y, z that maximise:

$$f(u, v, w, x, y, z) = \left|\cos\left(\frac{x}{z+1}\right)\right| + \left|\sin\left(\frac{wz}{xu + vy + 1}\right)\right| + \left|\cos\left(\frac{x + 5 - yw}{uyz + 1}\right)\right| + \left|\sin\left(\frac{w^2 + yu}{z+1}\right)\right|$$

subject to $u + v + w + x + y + z = 100$.

This complex and non-linear optimisation problem can be solved with a population-based method. First, the function for the objective itself:

```
>>> import math
>>> def f(solution):
...     u, v, w, x, y, z = solution
...     term_1 = abs(math.cos(x / (z + 1)))
...     term_2 =  abs(math.sin((w * z) / ((x * u) + (v * y) + 1)))
...     term_3 = abs(math.cos((x + 5 - (y * w)) / ((u * y * z) + 1)))
...     term_4 = abs(math.sin(((w ** 2) + (y * u)) / (z + 1)))
...     return term_1 + term_2 + term_3 + term_4
```

Now we will write the class for the population-based method, where:

- A solution s is a list of size integers that sum to 100;
- There is no crossover, and mutation involves adding and subtracting 1 from two randomly chosen indices of the list:

```
>>> class PopulationBased:
...     def __init__(self, f, psize, ksize):
...         self.f = f
...         self.psize = psize
...         self.ksize = ksize
...
...     def generate_solution(self):
...         solution = [0, 0, 0, 0, 0, 0]
...         for b in range(100):
...             index = random.choice(range(6))
...             solution[index] += 1
...         return solution
...
...     def get_initial_population(self):
...         self.pop = [self.generate_solution() for _ in range(self.psize)]
...
...     def rank_and_kill_population(self):
...         self.pop = sorted(self.pop, key=f, reverse=True)[:self.ksize]
...
...     def mutate(self, sol):
...         i1, i2 = random.sample(list(range(6)), k=2)
...         new_sol = list(sol)
...         if sol[i1] > 0:
...             new_sol[i1] -= 1
...             new_sol[i2] += 1
...         return new_sol
...
...     def repopulate(self):
...         mutated = [
...             self.mutate(sol) for sol in random.choices(
...                 self.pop,
...                 k=self.psize-self.ksize
...             )
...         ]
...         self.pop += mutated
...
...     def optimise(self, n_gens):
...         self.get_initial_population()
...         for gen in range(n_gens):
...             self.rank_and_kill_population()
...             self.repopulate()
...         self.best_solution = self.pop[0]
```

Running this with a population size of 200, keeping the top 50 solutions each generation, over 5000 generations gives:

```
>>> random.seed(0)
>>> D = PopulationBased(f=f, psize=200, ksize=50)
>>> D.optimise(n_gens=5000)
>>> D.best_solution
[16, 28, 18, 1, 10, 27]
>>> f(D.best_solution)
3.996378698113503
```

Which gives quite a good solution, close to f's theoretical maximum of 4.