Lecture Book
Python for Eng. and Eco.

D. T. McGuiness, PhD

Version: 2024

Topics on Computer Science

(2024, D. T. McGuiness, PhD)

Current version is 2024.

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

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

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

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

The current maintainer of this work along with the primary lecturer

is D. T. McGuiness, PhD (dtm@mci4me.at).

Contents

Part I.

Fundamentals

Chapter 1

Welcome to Python

Table of Contents

Python is a multiplatform programming language, written in **C**, under a free license. It is an interpreted language, i.e., it requires an interpreter to execute commands, and has **no compilation phase**. Its first public version dates from 1991.

Figure 1.1.: The name Python comes from the cult classic "Monthy Python's Flying Circus"

Compiled v. Interpreted

In a compiled language, the target machine directly translates the program. In an interpreted language, the source code is not directly translated by the target machine. Instead, a different program, aka the interpreter, reads and executes the code

The main programmer, Guido van Rossum, had started working on this programming language in the late 1980s. The name given to the Python language comes from the interest of its main creator in a British television series broadcast on the BBC called Monty Pyth'on's Flying Circus.

The popularity of Python has grown strongly in recent years, as confirmed by the survey results provided since 2011 by Stack Overflow. Stack Overflow offers its users the opportunity to complete a survey in which they are asked many questions to describe their experience as a developer. The results of the 2019 survey show a new breakthrough in the use of Python by developers.

1.1 Features

1. **Simple and Easy to Learn:** Python has a simple syntax, which makes it easy to learn and read. It's a great language for beginners who are new to programming.

2. **Interpreted:** Python is an interpreted language, which means that the Python code is executed line by line. This makes it easy to test and debug code.

3. **High-Level:** Python is a high-level language, which means that it abstracts away low-level details like memory management and hardware interaction. This makes it easier to write and understand code.

4. **Dynamic Typing:** Python is dynamically typed, which means that you don't need to declare the data type of a variable explicitly. Python will automatically infer the data type based on the value assigned to the variable.

5. **Strong Typing:** Python is strongly typed, which means that the data type of a variable is enforced at runtime. This helps prevent errors and makes the code more robust.

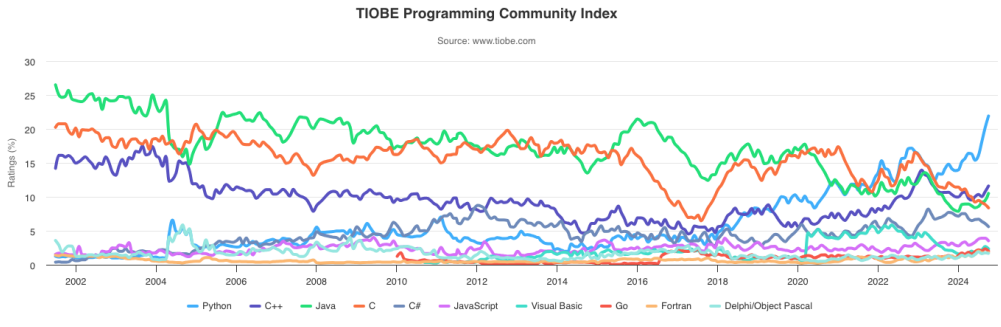
6. **Extensive Standard Library:** Python comes with a large standard library that provides tools and modules for various tasks, such as file I/O, networking, and more. This makes it easy to build complex applications without having to write everything from scratch.

7. **Cross-Platform:** Python is a cross-platform language, which means that Python code can run on different operating systems without modification. This makes it easy to develop and deploy Python applications on different platforms.

8. **Community and Ecosystem:** Python has a large and active community, which contributes to its ecosystem. There are many third-party libraries and frameworks available for various purposes, making Python a versatile language for many applications.

9. **Versatile:** Python is a versatile language that can be used for various purposes, including web development, data science, artificial intelligence, game development, and more.

Since 2003, Python has consistently ranked in the top ten most popular programming languages in the TIOBE Programming Community Index where as of December 2022 it was the most popular language (ahead of C, C++, and Java). It was selected as Programming Language of the Year (for "the highest rise in ratings in a year") in 2007, 2010, 2018, and 2020 (the only language to have done so four times as of 2020).



Large organizations that use Python include Wikipedia, Google, Yahoo!, CERN, NASA, Facebook, Amazon, Instagram, Spotify, and some smaller entities like Industrial Light & Magic and ITA. The social news networking site Reddit was written mostly in Python. Organizations that partially use Python include Discord and Baidu.

1.2 Hello, World !

A "Hello, World!" program is generally a simple computer program that emits (or displays) to the screen (often the console) a message similar to "Hello, World!". A small piece of code in most general-purpose programming languages, this program is used to illustrate a language's basic syntax. A "Hello, World!" program is often the first written by a student of a new programming language, but such a program can also be used as a sanity check to ensure that the computer software intended to compile or run source code is correctly installed, and that its operator understands how to use it.

Python is known to be exceptionally user-friendly compared to other languages. To show, here is a Hello, World! written in C++.

C.R. 1 cpp

```cpp
#+begin_src cpp :session :results output
#include <iostream>

int main() {
    std::cout << "Hello, World!";
    return 0;
```

And here is in Java, another major programming language:

C.R. 2 java

```java
#+begin_src java :session :results output
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
```

And finally, here is in Python:

C.R. 3 python

```python
#+begin_src python :session :results output
```

From here we will start to work on the concepts of programming. While some parts are unique to Python and explained in its notation, the things you learn here will be applicable to the majority of the programming languages you will encounter in academic or industrial environments.
Here is a tiny code to get us started.

C.R. 4 python

```python
#+begin_src python :session :results output
username = input("Enter username:")
```

Chapter 2

Data Types

Table of Contents

Python has several basic data types that are built into the language. With these types, you can represent numeric values, text and binary data, and Boolean values in your code. So, these data types are the basic building blocks of most Python programs and projects.

2.1 Basic Data Types

Python has several built-in data types that you can use out of the box as they're built into the language. From all the built-in types available, you'll find that a few of them represent basic objects, such as **numbers**, **strings** and **characters**, **bytes**, and **Boolean** values.

> the term basic refers to objects that can represent data you typically find in real life, such as numbers and text. It doesn't include composite data types, such as lists, tuples, dictionaries, and others.

In Python, the built-in data types that you can consider basic are the following:

Class	Basic Type
int	Integer numbers
float	Floating-point numbers
complex	Complex numbers
str	Strings and characters
bytes, bytearray	Bytes
bool	Boolean values

In the following sections, you'll learn the basics of how to create, use, and work with all of these built-in data types in Python.

2.2 Integer Numbers

Integer numbers are whole numbers with **NO** decimal places. They can be positive or negative numbers. For example, 0, 1, 2, 3, -1, -2, and -3 are all integers. Usually, you'll use positive integer numbers to count things. In Python, the integer data type is represented by the `int` class:

C.R. 1 python
```python
print(type(42))
```

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

In the following sections, you'll learn the basics of how to create and work with integer numbers in Python.

2.2.1 Integer Literals

When you need to use integer numbers in your code, you'll often use integer literals directly. Literals are constant values of built-in types spelled out literally, such as integers. Python provides a few different ways to create integer literals. The most common way is to use base-ten literals that look the same as integers look in math:

C.R. 2
```python
print(42)    # Prints out 42
print(-84)   # Prints out -84
print(0)     # Prints out 0
```

```text
42
-84
0
```

Here, you have three (3) integer numbers:

1. a positive one,
2. a negative one,
3. and zero.

> To create negative integers, you need to prepend the minus sign (-) to the number.

Python **HAS NO LIMIT** to how long an integer value can be. The only constraint is the amount of memory your system has. Beyond that, an integer can be as long as you need:

C.R. 3
```python
print(123123123123123123123123123123123123123123123123 + 1)
```

```text
123123123123123123123123123123123123123123123124
```

> For a really, really long integer, you can get a `ValueError` when converting it to a string.

C.R. 4
```python
print(123 ** 10000)
```

```text
ValueError: Exceeds the limit (4300 digits) for integer string conversion;
use sys.set_int_max_str_digits() to increase the limit
```

If you need to print an integer number beyond the 4300-digit limit, then you can use the `sys.set_int_max_str_digits()` function to increase the limit and make your code work.

When you're working with long integers, you can use the underscore (_) character to make the literals more readable:

C.R. 5
```python
1_000_000
```

With the underscore as a thousands separator, you can make your integer literals more readable for fellow programmers reading your code. You can also use other bases to represent integers. You can prepend the following characters to an integer value to indicate a base other than 10:

Prefix	Representation	Base
0b or 0B (Zero + b or B)	Binary	2
0o or 0O (Zero + o or O)	Octal	8
0x or 0X (Zero + x or X)	Hexadecimal	16

Table 2.1.

Using the above characters, you can create integer literals using binary, octal, and hexadecimal representations. For example:

C.R. 6
python
```python
10  # Base 10
print(type(10))

0b10  # Base 2
print(type(0b10))

0o10  # Base 8
print(type(0o10))

0x10  # Base 16
print(type(0x10))
```

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

> The underlying type of a Python integer is always `int`.

So, in all cases, the built-in `type()` function returns int, irrespective of the base you use to build the literal.

2.2.2 Integer Methods

The built-in int type has a few methods that you can use in some situations. Here's a quick summary of these methods:
When you call the `.as_integer_ratio()` method on an integer value, you get the integer as the numerator and 1 as the denominator. As you'll see in a moment, this method is more useful in floating-point numbers.

> the `int` type also has a method called `.is_integer()`, which always returns `True`. This method exists for duck typing compatibility with floating-point numbers, which have the method as part of their public interface.

Method	Description
`.as_integer_ratio()`	Returns a pair of integers whose ratio is equal to the original integer and has a positive denominator
`.bit_count()`	Returns the number of ones in the binary representation of the absolute value of the integer
`.bit_length()`	Returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros
`.from_bytes()`	Returns the integer represented by the given array of bytes
`.to_bytes()`	Returns an array of bytes representing an integer
`.is_integer()`	Returns True

Table 2.2.

Duck Typing

An application of the duck test—"If it walks like a duck and it quacks like a duck, then it must be a duck"—to determine whether an object can be used for a particular purpose.

To access an integer method on a literal, you need to wrap the literal in **parentheses**:

C.R. 7
```python
(42).as_integer_ratio()
```

The parentheses are required because the dot character (`.`) also defines floating-point numbers, as you'll learn in a moment. If you don't use the parentheses, then you get a SyntaxError.

C.R. 8
```python
42.as_integer_ratio()
```

```text
    42.as_integer_ratio()
      ^
SyntaxError: invalid decimal literal
```

The `.bit_count()` and `.bit_length()` methods can help you when working on digital signal processing. For example, you may want every transmitted signal to have an even number of set bits:

C.R. 9
```python
signal = 0b11010110
set_bits = signal.bit_count()

if set_bits % 2 == 0:
    print("Even parity")
else:
    print("Odd parity")
```

```text
Odd parity
```

In this toy example, you use `.bit_count()` to ensure that the received signal has the correct parity. This way, you implement a basic **error detection mechanism**.

Finally, the `.from_bytes()` and `.to_bytes()` methods can be useful in network programming. Often, you need to send and receive data over the network in binary format. To do this, you can use `.to_bytes()` to convert the message for network transmission. Similarly, you can use `.from_bytes()` to convert the message back.

2.2.3 The Built-in int() Function

The built-in `int()` function provides another way to create integer values using different representations. With **NO** arguments, the function returns `0`:

C.R. 10
```python
print(int())
```

```text
0
```

This feature makes `int()` especially useful when you need a factory function for classes like `defaultdict` from the `collections` module.

> In Python, the built-in functions associated with data types, such as `int()`, `float()`, `str()`, and `bytes()`, are classes with a **function-style name**. The Python documentation calls them functions, so we'll follow that practice. However, keep in mind that something like `int()` is really a class constructor rather than a regular function.

The `int()` function is commonly used to convert other data types into integers, provided that they're valid numeric values:

C.R. 11
```python
print(int(42.0))
print(int("42"))
print(int("one"))
```

```text
42
42
 print(int("one"))
           ^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'one'
```

In these examples, you first use `int()` to convert a floating-point number into an integer. Then, you convert a string into an integer.

> that when it comes to strings, you must ensure that the input string is a **valid numeric value**. Otherwise, you'll get a ValueError exception.

> When you use the `int()` function to convert floating-point numbers, you must be aware that the function just removes the decimal or fractional part.

This function can take an additional argument called **base**, which defaults to 10 for decimal integers. This argument allows you to convert strings that represent integer values, which are

expressed using a different base:

C.R. 12

```python
print(int("0b10", base=2))
print(int("10", base=8))
print(int("10", base=16))
```

```text
2
8
16
```

In this case, the first argument must be a string representing an integer value with or without a prefix. Then, you must provide the appropriate base in the second argument to run the conversion. Once you call the function, you get the resulting integer value.

2.2.4 Operations Table

Operator	Type	Operation	Sample Expression	Result
+	Unary	Positive	+a	a without any transformation since this is simply a complement to negation
+	Binary	Addition	a + b	The arithmetic sum of a and b
-	Unary	Negation	-a	The value of a but with the opposite sign
-	Binary	Subtraction	a - b	b subtracted from a
*	Binary	Multiplication	a * b	The product of a and b
%	Binary	Modulo	a % b	The remainder of a divided by b
//	Binary	Floor division or integer division	a // b	The quotient of a divided by b, rounded to the next smallest whole number
**	Binary	Exponentiation	a**b	a raised to the power of b

Table 2.3.

2.2.5 Exercises

1. Write a program which multiplies two numbers and prints out the result.
2. Write a program which converts a number in base 10 to base 7
3. Write a program which sums the first five positive integers
4. What is the average age of Sara (age 23), Mark (age 19), and Fatima (age 31)

2.3 Floating-Point Numbers

Floating-point numbers, or just float, are numbers with a decimal place. For example, 1.0 and 3.14 are floating-point numbers. You can also have negative float numbers, such as -2.75. In Python, the name of the `float` class represents floating-point numbers:

C.R. 13
```python
print(type(1.0))
```

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

In the following sections, you'll learn the basics of how to create and work with floating-point numbers in Python.

2.3.1 Floating-Point Literals

The float type in Python designates floating-point numbers. To create these types of numbers, you can also use literals, similar to what you use in math. However, in Python, the dot character (`.`) is what you must use to create floating-point literals:

C.R. 14
```python
4.2
4.
.2
```

In these quick examples, you create floating-point numbers in three different ways. First, you have a literal build using an integer part, the dot, and the decimal part. You can also create a literal using the dot without specifying the decimal part, which defaults to 0. Finally, you make a literal without specifying the integer part, which also defaults to 0.

You can also have negative float numbers:

C.R. 15
```python
-42.0
```

To create a negative floating-point number using a literal, you need to prepend the minus sign (-) to the number.

Similar to integer numbers, if you're working with long floating-point numbers, you can use the underscore character as a **thousands separator**.

C.R. 16
```python
1_000_000.0
```

By using an underscore, you can make your floating-point literals more readable for humans, which is great. Optionally, you can use the characters `e` or `E` followed by a positive or negative integer to express the number using scientific notation:

C.R. 17
```python
.4e7
4.2E-4
```

Scientific Notation

Calculators and computer programs typically present very large or small numbers using scientific notation, and some can be configured to uniformly present all numbers that way. Because superscript exponents like 10^7 can be inconvenient to display or type,

the letter "E" or "e" (for "exponent") is often used to represent "times ten raised to the power of", so that the notation mEn for a decimal significand m and integer exponent n means the same as $m \times 10^n$. For example 6.022×10^{23} is written as $6.022E23$ or $6.022e23$, and 1.6×10^{-35} is written as $1.6E-35$ or $1.6e-35$.

By using the e or E character, you can represent any floating-point number using scientific notation, as you did in the above examples.

2.3.2 Floating-Point Numbers Representation

Now, you can take a more in-depth look at how Python internally represents floating-point numbers. You can readily use floating-point numbers in Python without understanding them to this level, so don't worry if this seems overly complicated. The information in this section is only meant to satisfy your curiosity and is not a strict requirement of learning the fundamentals.

For additional information on the floating-point representation in Python and the potential pitfalls, see Floating Point Arithmetic: Issues and Limitations in the Python documentation.

Almost all platforms represent Python float values as 64-bit (double-precision) values, according to the IEEE 754 standard. In that case, a floating-point number's maximum value is approximately 1.8×10308. Python will indicate this number, and any numbers greater than that, by the "inf" string:

C.R. 18

```python
print(1.79e308)
print(1.8e308)
```

```text
1.79e+308
inf
```

The closest a nonzero number can be to zero is approximately 5.0×10^{-324}. Anything closer to zero than that is effectively considered to be zero:

C.R. 19

```python
print(5e-324)
print(1e-324)
```

```text
5e-324
0.0
```

Python internally represents floating-point numbers as binary (base-2) fractions. Most decimal fractions can't be represented exactly as binary fractions. So, in most cases, the internal representation of a floating-point number is an approximation of its actual value.

In practice, the difference between the actual and represented values is small and should be manageable.

2.3.3 Floating-Point Methods

The built-in `float` type has a few methods and attributes which can be useful in some situations. Here's a quick summary of them:

Method	Description
`.as_integer_ratio()`	Returns a pair of integers whose ratio is exactly equal to the original float
`.is_integer()`	Returns `True` if the float instance is finite with integral value, and `False` otherwise
`.hex()`	Returns a representation of a floating-point number as a hexadecimal string
`.fromhex(string)`	Builds the float from a hexadecimal string

Table 2.4.

The `.as_integer_ratio()` method on a float value returns a pair of integers whose ratio equals the original number. You can use this method in scientific computations that require high precision. In these situations, you may need to avoid precision loss due to floating-point rounding errors. For example, say that you need to perform computations with the gravitational constant:

C.R. 20 python
```python
G = 6.67430e-11
print(G.as_integer_ratio())
```

With this exact ratio, you can perform calculations and prevent floating-point errors that may alter the results of your research. The `.is_integer()` method allows you to check whether a given float value is an integer:

C.R. 21 python
```python
print((42.0).is_integer())
print((42.42).is_integer())
```

text
```
True
False
```

When the number after the decimal point is 0, the `.is_integer()` method returns True. Otherwise, it returns False. Finally, the `.hex()` and `.fromhex()` methods allow you to work with floating-point values using a hexadecimal representation:

C.R. 22 python
```python
print((42.0).hex())
print(float.fromhex("0x1.5000000000000p+5"))
```

text
```
0x1.5000000000000p+5
42.0
```

The `.hex()` method returns a string that represents the target float value as a hexadecimal value. Note that `.hex()` is an instance method. The `.fromhex()` method takes a string that

represents a floating-point number as an argument and builds an actual float number from it. In both methods, the hexadecimal string has the following format:

```
[sign] ["0x"] integer ["." fraction] ["p" exponent]
```

In this template, apart from the integer identifier, the components are optional. Here's what they mean:

`sign:` defines whether the number is positive or negative. It may be either + or -. Only the - sign is required because + is the default.

`"0x"` is the hexadecimal prefix.

`integer` is a string of hexadecimal digits representing the whole part of the float number.

`"."` is a dot that separates the whole and fractional parts. fraction is a string of hexadecimal digits representing the fractional part of the float number.

`"p"` allows for adding an exponent value. exponent is a decimal integer with an optional leading sign.

With these components, you'll be able to create valid hexadecimal strings to process your floating-point numbers with the `.hex()` and `.fromhex()` methods.

2.3.4 The Built-in float() Function

The built-in `float()` function provides another way to create floating-point values. When you call `float()` with no argument, then you get `0.0`:

C.R. 23

```python
print(float())
```

```text
0.0
```

Again, this feature of `float()` allows you to use it as a factory function.

The `float()` function also helps you convert other data types into float, provided that they're valid numeric values:

C.R. 24

```python
print(float(42))
print(float("42"))
print(float("one"))
```

In these examples, you first use `float()` to convert an integer number into a float. Then, you convert a string into a float. Again, with strings, you need to make sure that the input string is a valid numeric value. Otherwise, you get a ValueError exception.

2.3.5 Exercises

1. Write a program which **removes** the decimal value of an integer of user input and print is out
2. Write a simple program which adds two values together and gives it to the user.

2.4 Complex Numbers

Python has a built-in type for **complex numbers**. Complex numbers are composed of real and imaginary parts. They have the form $a + bi$, where a and b are real numbers, and i is the imaginary unit. In Python, you'll use a j instead of an i. For example:

C.R. 25
```python
type(2 + 3j)
```

In this example, the argument to `type()` may look like an expression. However, it's a literal of a complex number in Python. If you pass the literal to the `type()` function, then you'll get the complex type back.

2.4.1 Complex Number Literals

In Python, you can define complex numbers using literals that look like $a + b\mathbf{j}$, where a is the real part, and $b\mathbf{j}$ is the imaginary part:

C.R. 26
```python
print(2 + 3j)
print(7j)
print(2.4 + 7.5j)
print(3j + 5)
print(5 - 3j)
print(1 + j)
```

```text
(2+3j)
7j
(2.4+7.5j)
(5+3j)
(5-3j)
```

As you can conclude from these examples, there are many ways to create complex numbers using literals. The key is that you need to use the **j** letter in one of the components.

> The **j** can't be used alone.

If you try to do so, you get a NameError exception because Python thinks that you're creating an expression. Instead, you need to write 1**j**.

2.4.2 Complex Number Methods

In Python, the complex type has a single method called `.conjugate()`. When you call this method on a complex number, you get the conjugate:

Complex Conjugate

the complex conjugate of a complex number is the number with an equal real part and an imaginary part equal in magnitude but opposite in sign. That is, if a and b are real numbers then the complex conjugate of $a + b\mathbf{j}$ is $a + b\mathbf{j}$.

C.R. 27
```python
number = 2 + 3j
number.conjugate()
```

The `conjugate()` method flips the sign of the imaginary part, returning the complex conjugate.

2.4.3 The Built-in complex() Function

You can also use the built-in `complex()` function to create complex numbers by providing the real and imaginary parts as arguments:

C.R. 28
```python
print(complex())
print(complex(1))
print(complex(0, 7))
print(complex(2, 3))
print(complex(2.4, 7.5))
print(complex(5, -3))
```

```text
0j
(1+0j)
7j
(2+3j)
(2.4+7.5j)
(5-3j)
```

When you call `complex()` with no argument, you get `0j`. If you call the function with a single argument, that argument is the real part, and the imaginary part will be `0j`. If you want only the imaginary part, you can pass `0` as the first argument. Note that you can also use negative numbers. In general, you can use integers and floating-point numbers as arguments to `complex()`.

You can also use `complex()` to convert strings to complex numbers:

C.R. 29
```python
print(complex("5-3j"))
print(complex("5"))
print(complex("5 - 3j"))
print(complex("5", "3"))
```

```text
(5-3j)
(5+0j)
```

To convert strings into complex numbers, you must provide a string that follows the format of complex numbers. For example, you can't have spaces between the components. If you add spaces, then you get a ValueError exception.

Finally, note that you can't use strings to provide the imaginary part of complex numbers. If you do that, then you get a TypeError exception.

2.5 Strings and Characters

In Python, strings are sequences of character data that you can use to represent and store textual data.

The string type in Python is called `str`:

```
print(type("Hello, World!"))
```

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

In this example, the argument to `type()` is a string literal that you commonly create using double quotes to enclose some text.
In the following sections, you'll learn the basics of how to create, use, format, and manipulate strings in Python.

2.5.1 Regular String Literals

You can also use literals to create strings. To build a single-line string literal, you can use double (”) or single quotes (') and, optionally, a sequence of characters in between them. All the characters between the opening and closing quotes are part of the string:

```
print("I am a string")
print('I am a string too')
```

```
I am a string
I am a string too
```

Python's strings can contain as many characters as you need. The only limit is your computer's memory.

You can define empty strings by using the quotes without placing characters between them:

```
""
''
print(len(""))
```

```
0
```

An empty string doesn't contain any characters, so when you use the built-in `len()` function with an empty string as an argument, you get 0 as a result.

There is yet another way to delimit strings in Python. You can create triple-quoted string literals, which can be delimited using either three single quotes or three double quotes. Triple-quoted strings are commonly used to build multiline string literals. However, you can also use them to create single-line literals:

```
"""A triple-quoted string in a single line"""


'''Another triple-quoted string in a single line'''


"""A triple-quoted string
that spans across multiple
lines"""
```

Even though you can use triple-quoted strings to create single-line string literals, the main use case of them would be to create multiline strings. In Python code, probably the most common use case for these string literals is when you need to provide **docstrings** for your packages, modules, functions, classes, and methods.

What if you want to include a quote character as part of the string itself? Your first impulse might be to try something like this:

```
'This string contains a single quote (') character'
  File "<input>", line 1
    'This string contains a single quote (') character'
                                           ^
SyntaxError: unmatched ')'
```

As you can see, that doesn't work so well. The string in this example opens with a single quote, so Python assumes the next single quote—the one in parentheses—is the closing delimiter. The final single quote is then a stray, which causes the syntax error shown.

If you want to include either type of quote character within the string, then you can delimit the string with the other type. In other words, if a string is to contain a single quote, delimit it with double quotes and vice versa:

```
"This string contains a single quote (') character"


'This string contains a double quote (") character'
```

In these examples, your first string includes a single quote as part of the text. To do this, you use double quotes to delimit the literal. In the second example, you do the opposite.

2.5.2 Escape Sequences in Strings

Sometimes, you want Python to interpret a character or sequence of characters within a string differently. This may occur in one of two ways. You may want to:

1. Apply special meaning to characters
2. Suppress special character meaning

You can accomplish these goals by using a backslash (\textbackslash{}) character to indicate that the characters following it should be interpreted specially. The combination of a backslash and a specific character is called an **escape sequence**. That's because the backslash causes the subsequent character to escape its usual meaning.

You already know that if you use single quotes to delimit a string, then you can't directly embed a single quote character as part of the string because, for that string, the single quote has a special meaning, **it terminates the string**. You can eliminate this limitation by using double quotes ("") to delimit the string.

Alternatively, you can escape the quote character using a backslash:

```
'This string contains a single quote (\') character'
```

In this example, the backslash escapes the single quote character by suppressing its usual meaning. Now, Python knows that your intention isn't to terminate the string but to embed the single quote.
The following is a table of escape sequences that cause Python to suppress the usual special interpretation of a character in a string:

Character	Usual Interpretation	Escape Sequence	Escaped Interpretation
'	Delimit a string literal	\'	Literal single quote (') character
"	Delimit a string literal	\"	Literal double quote (") character
<newline>	Terminates the input line	\<newline>	Newline is ignored
\	Introduces an escape sequence	\	Literal backslash (\) character

Table 2.5.

You already have an idea of how the first two escape sequences work.

Now, how does the newline escape sequence work?

Usually, a newline character terminates a physical line of input. So, pressing Enter in the middle of a string will cause an error:

```
"Hello
  File "<input>", line 1
    "Hello
    ^
SyntaxError: incomplete input
```

When you press Enter after typing Hello, you get a SyntaxError. If you need to break up a string over more than one line, then you can include a backslash before each new line:

```
"Hello\
, World\
!"
```

By using a backslash before pressing enter, you make Python ignore the new line and interpret the whole construct as a single line.

Finally, sometimes you need to include a literal backslash character in a string. If that backslash doesn't precede a character with a special meaning, then you can insert it right away:

```
"This string contains a backslash (\) character"
```

In this example, the character after the backslash doesn't match any known escape sequence, so Python inserts the actual backslash for you. Note how the resulting string automatically doubles the backslash. Even though this example works, the best practice is to always double the backslash when you need this character in a string.

However, you may have the need to include a backslash right before a character that makes up an escape sequence:

```
>>> "In this string, the backslash should be at the end \"
  File "<input>", line 1
    "In this string, the backslash should be at the end \"
    ^
SyntaxError: incomplete input
```

Because the sequence `\textbackslash{}"` matches a known escape sequence, your string fails with a SyntaxError. To avoid this issue, you can double the backslash:

```
"In this string, the backslash should be at the end \\"
```

In this update, you double the backslash to escape the character and prevent Python from raising an error.

> When you use the built-in `print()` function to print a string that includes an escaped backslash, then you won't see the double backslash in the output:

```
print("In this string, the backslash should be at the end \\")
```

```
In this string, the backslash should be at the end \
```

In this example, the output only displays one backslash, producing the desired effect.
Up to this point, you've learned how to suppress the meaning of a given character by escaping it. Suppose you need to create a string containing a tab character. Some text editors may allow you to insert a tab character directly into your code. However, this is considered a poor practice for several reasons:

- Computers can distinguish between tabs and a sequence of spaces, but human beings can't because these characters are visually indistinguishable.
- Some text editors automatically eliminate tabs by expanding them to an appropriate number of spaces.

- Some Python REPL environments will not insert tabs into code.

In Python, you can specify a tab character by the \textbackslash{}t escape sequence:

```
print("Before\tAfter")
```

```
Before After
```

The \textbackslash{}t escape sequence changes the usual meaning of the letter t. Instead, Python interprets the combination as a tab character.
Here is a list of escape sequences that cause Python to apply special meaning to some characters instead of interpreting them literally:

Escape Sequence	Escaped Interpretation
\textbackslash{}a	ASCII Bell (BEL) character
\textbackslash{}b	ASCII Backspace (BS) character
\textbackslash{}f	ASCII Formfeed (FF) character
\textbackslash{}n	ASCII Linefeed (LF) character
\textbackslash{}r	Character from Unicode database with given <name>
\textbackslash{}N\{<name>\}	ASCII Carriage return (CR) character
\textbackslash{}uxxxx	ASCII Horizontal tab (TAB) character
\textbackslash{}t	Unicode character with 16-bit hex value xxxx
\textbackslash{}Uxxxxxxxx	Unicode character with 32-bit hex value xxxxxxxx
\textbackslash{}v	ASCII Vertical tab (VT) character
\textbackslash{}ooo	Character with octal value ooo
\textbackslash{}xhh	Character with hex value hh

Table 2.6.

Of these escape sequences, the newline or linefeed character (\textbackslash{}n) is probably the most popular. This sequence is commonly used to create nicely formatted text outputs.

Here are a few examples of the escape sequences in action:

```
# Tab
print("a\tb")

# Linefeed
print("a\nb")

# Octal
print("\141")

# Hex
print("\x61")

# Unicode by name
print("\N{rightwards arrow}")
```

```
a b
a
b
a
a
→
```

These escape sequences are typically useful when you need to insert characters that aren't readily generated from the keyboard or aren't easily readable or printable.

2.5.3 Raw String Literals

A raw string is a string that doesn't translate the escape sequences. Any backslash characters are left in the string. To create a raw string, you can precede the literal with an `r` or `R`:

```
print("Before\tAfter")  # Regular string
print(r"Before\tAfter")  # Raw string
```

```
Before After
Before\tAfter
```

The raw string suppresses the meaning of the escape sequence and presents the characters as they are. This behavior comes in handy when you're creating **regular expressions** because it allows you to use several different characters that may have special meanings without restrictions.

Regular Expression

A regular expression (shortened as regex or regexp), sometimes referred to as rational expression, is a sequence of characters that specifies a match pattern in text.

1. F-String Literals

Python has another type of string literal called formatted strings or f-strings for short. F-strings allow you to interpolate values into your strings and format them as you need.

To build f-string literals, you must prepend an `f` or `F` letter to the string literal. Because the idea behind f-strings is to interpolate values and format them into the final string, you need to use something called a replacement field in your string literal. You create these fields using curly brackets.

Here's a quick example of an `f-string` literal:

```
name = "Jane"
print(f"Hello, {name}!")
```

```
Hello, Jane!
```

In this example, you interpolate the variable name into your string using an f-string literal and a replacement field.

You can also use f-strings to format the interpolated values. To do that, you can use format specifiers that use the syntax defined in Python's string format mini-language. For example, here's how you can present numeric values using a currency format:

```
income = 1234.1234

print(f"Income: ${income:.2f}")
```

```
Income: $1234.12
```

Inside the replacement field, you have the variable you want to interpolate and the format specifier, which is the string that starts with a colon (:). In this example, the format specifier defines a floating-point number with two decimal places.

2.5.4 String Methods

Python's str data type is probably the built-in type with the **MOST** available methods. In fact, you'll find methods for most string processing operations. Here's a summary of the methods that perform some string processing and return a transformed string object:

Method	Description
`.capitalize()`	Converts the first character to uppercase and the rest to lowercase
`.casefold()`	Converts the string into lowercase
`.center(width[, fillchar])`	Centers the string between width using fillchar
`.encode(encoding, errors)`	Encodes the string using the specified encoding
`.expandtabs(tabsize)`	Replaces tab characters with spaces according to tabsize
`.format(*args, **kwargs)`	Interpolates and formats the specified values
`.format_map(mapping)`	Interpolates and formats the specified values using a dictionary
`.join(iterable)`	Joins the items in an iterable with the string as a separator
`.ljust(width[, fillchar])`	Returns a left-justified version of the string
`.rjust(width[, fillchar])`	Returns a right-justified version of the string
`.lower()`	Converts the string into lowercase
`.strip([chars])`	Trims the string by removing chars from the beginning and end
`.lstrip([chars])`	Trims the string by removing chars from the beginning
`.rstrip([chars])`	Trims the string by removing chars from the end
`.removeprefix(prefix, /)`	Removes prefix from the beginning of the string
`.removesuffix(suffix, /)`	Removes suffix from the end of the string
`.replace(old, new [, count])`	Returns a string where the old substring is replaced with new
`.swapcase()`	Converts lowercase letters to uppercase letters and vice versa
`.title()`	Converts the first character of each word to uppercase and the rest to lowercase
`.upper()`	Converts a string into uppercase
`.zfill(width)`	Fills the string with a specified number of zeroes at the beginning

Table 2.7.

All the above methods allow you to perform a specific transformation on an existing string. In all cases, you get a new string as a result:

```
print("beautiful is better than ugly".capitalize())

name = "Jane"

print("Hello, {0}!".format(name))

print(" ".join(["Now", "is", "better", "than", "never"]))

print("====Header====".strip("="))

print("---Tail---".removeprefix("---"))

print("---Head---".removesuffix("---"))

print("Explicit is BETTER than implicit".title())

print("Simple is better than complex".upper())
```

```
Beautiful is better than ugly
Hello, Jane!
Now is better than never
Header
Tail---
---Head
Explicit Is Better Than Implicit
SIMPLE IS BETTER THAN COMPLEX
```

As you can see, the methods in these examples perform a specific transformation on the original string and return a new string object.
You'll also find that the `str` class has several Boolean-valued methods or predicate methods: All these methods allow you to check for various conditions in your strings.

Here are a few demonstrative examples:

```
filename = "main.py"
if filename.endswith(".py"):
    print("It's a Python file")

print("123abc".isalnum())

print("123abc".isalpha())

print("123456".isdigit())

print("abcdf".islower())
```

```
It's a Python file
```

Method	Result
.endswith(suffix[, start[, end]])	True if the string ends with the specified suffix, False otherwise
.startswith(prefix[, start[, end]])	True if the string starts with the specified prefix, False otherwise
.isalnum()	True if all characters in the string are alphanumeric, False otherwise
.isalpha()	True if all characters in the string are letters, False otherwise
.isascii()	True if the string is empty or all characters in the string are ASCII, False otherwise
.isdecimal()	True if all characters in the string are decimals, False otherwise
.isdigit()	True if all characters in the string are digits, False otherwise
.isidentifier()	True if the string is a valid Python name, False otherwise
.islower()	True if all characters in the string are lowercase, False otherwise
.isnumeric()	True if all characters in the string are numeric, False otherwise
.isprintable()	True if all characters in the string are printable, False otherwise
.isspace()	True if all characters in the string are whitespaces, False otherwise
.istitle()	True if the string follows title case, False otherwise
.isupper()	True if all characters in the string are uppercase, False otherwise

Table 2.8.

```
True
False
True
True
```

In these examples, the methods check for specific conditions in the target string and return a **Boolean** value as a result.

Finally, you'll find a few other methods that allow you to run several other operations on your strings:
The first method counts the number of repetitions of a substring in an existing string. Then, you have four (4) methods that help you find substrings in a string.

The `.split()` method is especially useful when you need to split a string into a list of individual strings using a given character as a separator, which defaults to whitespaces. You can also use `.partition()` or `.rpartition()` if you need to divide the string in exactly two parts:

```
sentence = "Flat is better than nested"
words = sentence.split()

print(words)
```

Method	Description
`.count(sub[, start[, end]])`	Returns the number of occurrences of a substring
`.find(sub[, start[, end]])`	Searches the string for a specified value and returns the position of where it was found
`.rfind(sub[, start[, end]])`	Searches the string for a specified value and returns the last position of where it was found
`.index(sub[, start[, end]])`	Searches the string for a specified value and returns the position of where it was found
`.rindex(sub[, start[, end]])`	Searches the string for a specified value and returns the last position of where it was found
`.split(sep=None, maxsplit=-1)`	Splits the string at the specified separator and returns a list
`.splitlines([keepends])`	Splits the string at line breaks and returns a list
`.partition(sep)`	Splits the string at the first occurance of sep
`.rpartition(sep)`	Splits the string at the last occurance of sep
`.split(sep=None, maxsplit=-1)`	Splits the string at the specified separator and returns a list
`.maketrans(x[, y[, z]])`	Returns a translation table to be used in translations
`.translate(table)`	Returns a translated string

Table 2.9.

```
numbers = "1-2-3-4-5"
head, sep, tail = numbers.partition("-")

numbers.rpartition("-")
```

```
['Flat', 'is', 'better', 'than', 'nested']
```

In these toy examples, you've used the `.split()` method to build a list of words from a sentence. Note that by default, the method uses whitespace characters as separators. You also used `.partition()` and `.rpartition()` to separate out the first and last number from a string with numbers.

The `.maketrans()` and `.translate()` are nice tools for playing with strings. For example, say that you want to implement the Cesar cipher algorithm. This algorithm allows for basic text encryption by shifting the alphabet by a number of letters. For example, if you shift the letter a by three, then you get the letter d, and so on.

The following code implements `cipher()`, a function that takes a character and rotates it by three:

```
def cipher(text):
    alphabet = "abcdefghijklmnopqrstuvwxyz"
    shifted = "defghijklmnopqrstuvwxyzabc"
    table = str.maketrans(alphabet, shifted)
    return text.translate(table)


cipher("python")
```

In this example, you use `.maketrans()` to create a translation table that matches the lowercase alphabet to a shifted alphabet. Then, you apply the translation table to a string using the `.translate()` method.

2.5.5 Common Sequence Operations on Strings

Python's strings are sequences of characters. As other built-in sequences like lists and tuples, strings support a set of operations that are known as common sequence operations. The table below is a summary of all the operations that are common to most sequence types in Python:

Operation	Example	Result
Length	`len(s)`	The length of `s`
Indexing	`s[index]`	The item at index `i`
Slicing	`s[i:j]`	A slice of s from index `i` to `j`
Slicing	`s[i:j:k]`	A slice of s from index `i` to `j` with step `k`
Minimum	`min(s)`	The smallest item of `s`
Maximum	`max(s)`	The largest item of `s`
Membership	`x in s`	`True` if an item of `s` is equal to `x`, else `False`
Membership	`x not in s`	`False` if an item of `s` is equal to `x`, else `True`
Concatenation	`s + t`	The concatenation of `s` and `t`
Repetition	`s * n or n * s`	The repetition of `s` a number of times specified by `n`
Index	`s.index(x[, i[, j]])`	The index of the first occurrence of `x` in `s`
Count	`s.count(x)`	The total number of occurrences of `x` in `s`

Table 2.10.

Sometimes, you need to determine the number of characters in a string. In this situation, you can use the built-in `len()` function:

```
print(len("Factorio"))
```

```
8
```

When you call `len()` with a string as an argument, you get the number of characters in the string at hand.
Another common operation you'd run on strings is retrieving a single character or a substring from an existing string.

In these situations, you can use indexing and slicing, respectively:

```
print("Factorio"[0])
print("Factorio"[5])
print("Factorio"[4])
print("Factorio"[:3])
```

```
F
r
o
Fac
```

To retrieve a character from an existing string, you use the indexing operator `[index]` with the index of the target character.

> Indices are zero-based, so the first character lives at index 0.

To retrieve a slice or substring from an existing string, you use the slicing operator with the appropriate indices. In the example above, you don't provide the start index `i`, so Python assumes that you want to start from the beginning of the string. Then, you give the end index `j` to tell Python where to stop the slicing.

2.5.6 The Built-in str() and repr() Functions

When it comes to creating and working with strings, you have two functions that can help you out and make your life easier:

1. `str()`
2. `repr()`

The built-in `str()` function allows you to create new strings and also convert other data types into strings:

```
print(str())
print(str(42))
print(str(3.14))
print(str([1, 2, 3]))
print(str({"one": 1, "two": 2, "three": 3}))
print(str({"A", "B", "C"}))
```

```
42
3.14
[1, 2, 3]
{'one': 1, 'two': 2, 'three': 3}
{'A', 'B', 'C'}
```

In these examples, you use the `str()` function to convert objects from different built-in types into strings. In the first example, you use the function to create an empty string. In the other examples, you get strings consisting of the object's literals between quotes, which provide user-friendly representations of the objects.

At first glance, these results may not seem useful. However, there are use cases where you need to use `str()`.

For example, say that you have a list of numeric values and want to join them using the `str.join()` method. This method only accepts iterables of strings, so you need to convert the numbers:

```
print("-".join([1, 2, 3, 4, 5]))
print("-".join(str(value) for value in [1, 2, 3, 4, 5]))
```

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
  File "/var/folders/c2/_hry6n9d5v1527pv4f_gfjxm0000gn/T/babel-0kSMF2/python-ls2Ftj",
  ↪  line 1, in <module>
    print("-".join([1, 2, 3, 4, 5]))
          ^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: sequence item 0: expected str instance, int found
```

If you try to pass a list of numeric values to `.join()`, then you get a TypeError exception because the function only joins strings. To work around this issue, you use a **generator expression** to convert each number to its string representation.

Behind the `str()` function, you'll have the `.__str__()` special method. In other words, when you call `str()`, Python automatically calls the `.__str__()` special method on the underlying object. You can use this special method to support `str()` in your own classes.

Consider the following Person class:

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

    def __str__(self):
        return f"I'm {self.name}, and I'm {self.age} years old."
```

In this class, you have two instance attributes, `.name and .age. Then, you have .__str__()` special methods to provide user-friendly string representations for your class.
Here's how this class works:

```
from person import Person

john = Person("John Doe", 35)
str(john)
```

In this code snippet, you create an instance of Person. Then, you call `str()` using the object as an argument. As a result, you get a descriptive message back, which is the user-friendly string representation of your class.

Similarly, when you pass an object to the built-in `repr()` function, you get a developer-friendly string representation of the object itself:

```
repr(42)
repr(3.14)
repr([1, 2, 3])
repr({"one": 1, "two": 2, "three": 3})
repr({"A", "B", "C"})
```

In the case of built-in types, the string representation you get with `repr()` is the same as the one you get with the `str()` function. This is because the representations are the literals of each object, and you can directly use them to re-create the object at hand.

Ideally, you should be able to re-create the current object using this representation. To illustrate, go ahead and update the Person class:

```
class Person:
    # ...

    def __repr__(self):
        return f"{type(self).__name__}(name='{self.name}', age={self.age})"
```

The `.__repr__()` special method allows you to provide a developer-friendly string representation for your class:

```
from person import Person

jane = Person("Jane Doe", 28)
repr(jane)
```

You should be able to copy and paste the resulting representation to re-create the object. That's why this string representation is said to be developer-friendly.

2.5.7 Exercises

1. Write a Python program to calculate the length of a string.
2. Write a Python function that takes a list of words and return the longest word and the length of the longest one.
3. Write a program which adds two strings together (i.e., "abc" "def" becomes "abc + def")
4. Write a program to reverse a string
5. Write a password generator in Python. Be creative with how you generate passwords - strong passwords have a mix of lowercase letters, uppercase letters, numbers, and symbols. The passwords should be random, generating a new password every time the user asks for a new password. Include your run-time code in a main method.
6. Ask the user for a string and print out whether this string is a palindrome or not. (A palindrome is a string that reads the same forwards and backwards.)

2.6 Bytes and Bytearrays

Bytes are **immutable sequences** of single bytes. In Python, the bytes class allows you to build sequences of bytes. This data type is commonly used for manipulating binary data, encoding and decoding text, processing file input and output, and communicating through networks.

Immutable Sequences

Immutable sequence can't be modified once created but it can be altered by making a copy with the updated data.

Python also has a `bytearray` class as a mutable counterpart to bytes objects:

```
print(type(b"This is a bytes literal"))
print(type(bytearray(b"Form bytes")))
```

```
<class 'bytes'>
<class 'bytearray'>
```

In the following sections, you'll learn the basics of how to create and work with bytes and bytearray objects in Python.

2.6.1 Bytes Literals

To create a bytes literal, you'll use a syntax that's largely the same as that for string literals. The difference is that you need to prepend a b to the string literal. As with string literals, you can use different types of quotes to define bytes literals:

```
print(b'This is a bytes literal in single quotes')
print(b"This is a bytes literal in double quotes")
```

```
b'This is a bytes literal in single quotes'
b'This is a bytes literal in double quotes'
```

There is yet another difference between string literals and bytes literals. To define bytes literals, you can only use ASCII characters. If you need to insert binary values over the 127 characters, then you have to use the appropriate escape sequence:

```
print(b"Espa\xc3\xb1a")
print(b"Espa\xc3\xb1a".decode("utf-8"))
```

```
b'Espa\xc3\xb1a'
España
```

In this example, `\textbackslash{}xc3\textbackslash{}xb1` is the escape sequence for the letter n in the Spanish word "Espana". Note that if you try to use the ñ directly, you get a SyntaxError.

2.6.2 The Built-in bytes() Function

The built-in `bytes()` function provides another way to create bytes objects. With no arguments, the function returns an empty bytes object:

```
print(bytes())
```

```
b''
```

You can use the `bytes()` function to convert string literals to bytes objects:

```
print(bytes("Hello, World!", encoding='utf-8'))

print(bytes("Hello, World!"))
```

```
b'Hello, World!'

Traceback (most recent call last):
  File "<string>", line 3, in <module>
  File "/var/folders/c2/_hry6n9d5v1527pv4f_gfjxm0000gn/
T/babel-0kSMF2/python-BTth0s", line 3, in <module>
    print(bytes("Hello, World!"))
          ^^^^^^^^^^^^^^^^^^^^^^
TypeError: string argument without an encoding
```

In these examples, you first use `bytes()` to convert a string into a bytes object. Note that for this to work, you need to provide the appropriate character encoding. In this example, you use the UTF-8 encoding. If you try to convert a string literal without providing the encoding, then you get a TypeError exception.

You can also use `bytes()` with an iterable of integers where each number is the Unicode code point of the individual characters:

```
print(bytes([65, 66, 67, 97, 98, 99]))
```

```
b'ABCabc'
```

In this example, each number in the list you use as an argument to `bytes()` is the code point for a specific letter. For example, 65 is the code point for A, 66 for B, and so on. You can get the Unicode code point of any character using the built-in `ord()` function.

2.6.3 The Built-in bytearray() Function

Python doesn't have dedicated literal syntax for bytearray objects. To create them, you'll always use the class constructor `bytearray()`, which is also known as a built-in function in Python. Here are a few examples of how to create bytearray objects using this function:

```
print(bytearray())

print(bytearray(5))

print(bytearray([65, 66, 67, 97, 98, 99]))

print(bytearray(b"Using a bytes literal"))
```

```
bytearray(b'')
bytearray(b'\x00\x00\x00\x00\x00')
bytearray(b'ABCabc')
bytearray(b'Using a bytes literal')
```

In the first example, you call `bytearray()` without an argument to create an empty bytearray object. In the second example, you call the function with an integer as an argument. In this case, you create a bytearray with five zero-filled items.

Next, you use a list of code points to create a bytearray. This call works the same as with bytes objects. Finally, you use a bytes literal to build up the bytearray object.

2.6.4 Bytes and Bytearray Methods

In Python, bytes and bytearray objects are quite similar to strings. Instead of being sequences of characters, bytes and bytearray objects are sequences of integer numbers, with values from 0 to 255.

Because of their similarities with strings, the bytes and bytearray types support mostly the same methods as strings, so you won't repeat them in this section. If you need detailed explanations of specific methods, then check out the Bytes and Bytearray Operations section in Python's documentation.

Finally, both bytes and bytearray objects support the common sequence operations that you learned in the Common Sequence Operations on Strings section.

2.7 Booleans

Boolean logic relies on the truth value of expressions and objects. The truth value of an expression or object can take one of two possible values: true or false. In Python, these two values are represented by True and False, respectively:

```
print(type(True))

print(type(False))
```

```
<class 'bool'>
<class 'bool'>
```

Both `True` and `False` are instances of the bool data type, which is built into Python. In the following sections, you'll learn the basics about Python's bool data type.

2.7.1 Boolean Literals

Python provides a built-in Boolean data type. Objects of this type may have one of two possible values: True or False. These values are defined as built-in constants with values of 1 and 0, respectively. In practice, the bool type is a subclass of int. Therefore, True and False are also instances of int:

```
print(issubclass(bool, int))

print(isinstance(True, int))

print(isinstance(False, int))

print(True + True)
```

```
True
True
True
2
```

> In Python, the bool type is a subclass of the int type. It has only two possible values, 0 and 1, which map to the constants False and True.

These constant values are also the literals of the bool type:

```
print(True)
print(False)
```

Boolean objects that are equal to True are truthy, and those equal to False are falsy. In Python, non-Boolean objects also have a truth value. In other words, Python objects are either truthy or falsy.

2.7.2 The Built-in bool() Function

You can use the built-in `bool()` function to convert any Python object to a Boolean value. Internally, Python uses the following rules to identify falsy objects:

- Constants that are defined to be false: None and False
- The zero of any numeric type: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
- Empty sequences and collections: '', (), [], {}, `set()`, range(0)

The rest of the objects are considered truthy in Python. You can use the built-in `bool()` function to explicitly learn the truth value of any Python object:

```
print(bool(0))
print(bool(42))
print(bool(0.0))
print(bool(3.14))
print(bool(""))
print(bool("Hello"))
print(bool([]))
print(bool([1, 2, 3]))
```

```
False
True
False
True
False
True
False
True
```

In these examples, you use `bool()` with arguments of different types. In each case, the function returns a Boolean value corresponding to the object's truth value.

> You rarely need to call `bool()` yourself. Instead, you can rely on Python calling `bool()` under the hood when necessary. For example, you can say if numbers: instead of if bool(numbers): to check whether numbers is truthy.

You can also use the `bool()` function with custom classes:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y


point = Point(2, 4)
print(bool(point))
```

```
True
```

By default, all instances of custom classes are true. If you want to modify this behavior, you can use the `.__bool__()` special method.

Consider the following update of your Point class:

```
class Point:
    def __init__(self, x, y):
```

```
        self.x = x
        self.y = y

    def __bool__(self):
        if self.x == self.y == 0:
            return False
        return True
```

The `.__bool__()` method returns False when both coordinates are equal to 0 and True otherwise.

Chapter 3

Lists

Table of Contents

The list class is a fundamental built-in data type in Python. It has an impressive and useful set of features, allowing you to efficiently organize and manipulate heterogeneous data. Knowing how to use lists is a must-have skill for you as a Python developer. Lists have many use cases, so you'll frequently reach for them in real-world coding.

3.1 Getting Started

Python's list is a flexible, versatile, powerful, and popular built-in data type. It allows you to create variable-length and mutable sequences of objects. In a list, you can store objects of any type. You can also mix objects of different types within the same list, although list elements often share the same type.

Some of the more relevant characteristics of list objects include being:

- **Ordered:** They contain elements or items that are sequentially arranged according to their specific insertion order.
- **Zero-based:** They allow you to access their elements by indices that start from zero.
- **Mutable:** They support in-place mutations or changes to their contained elements.
- **Heterogeneous:** They can store objects of different types.

- **Growable and dynamic:** They can grow or shrink dynamically, which means that they support the addition, insertion, and removal of elements.
- **Nestable:** They can contain other lists, so you can have lists of lists.
- **Iterable:** They support iteration, so you can traverse them using a loop or comprehension while you perform operations on each of their elements.
- **Sliceable:** They support slicing operations, meaning that you can extract a series of elements from them.
- **Combinable:** They support concatenation operations, so you can combine two or more lists using the concatenation operators.
- **Copyable:** They allow you to make copies of their content using various techniques.

Lists are sequences of objects. They're commonly called containers or collections because a single list can contain or collect an arbitrary number of other objects.

> In Python, lists support a rich set of operations that are common to all sequence types, including tuples, strings, and ranges. These operations are known as common sequence operations.

In Python, lists are **ordered**, which means that they keep their elements in the order of insertion:

```
colors = [
    "red",
    "orange",
    "yellow",
    "green",
    "blue",
    "indigo",
    "violet"
]

print(colors)
```

```
['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

The items in this list are strings representing colours. If you access the `list` object, then you'll see that the colors keep the same order in which you inserted them into the list.

> This order remains unchanged during the list's lifetime unless you perform some mutations on it.

You can access an individual object in a list by its position or index in the sequence. Indices start from zero ($_0$):

```
print(colors[0])
print(colors[1])
print(colors[2])
print(colors[3])
```

```
red
orange
yellow
green
```

Positions are numbered from zero to the length of the list minus one. The element at index 0 is the first element in the list, the element at index 1 is the second, and so on.

Lists can contain objects of different types. That's why lists are heterogeneous collections:

```
[42, "apple", True, {"name": "John Doe"}, (1, 2, 3), [3.14, 2.78]]
```

This list contains objects of different data types, including:

- an integer number,
- string,
- Boolean value,
- dictionary,
- tuple, and
- another list.

Even though this feature of lists may seem cool, in practice you'll find that lists typically store homogeneous data.

> One of the most relevant characteristics of lists is that they're **mutable** data types. This feature deeply impacts their behavior and use cases.

Okay! That's enough for a first glance at Python lists. To kick things off, you'll start by learning the different ways to create lists.

3.2 Constructing Lists in Python

If you want to use a list to store or collect some data in your code, then you need to create a list object. You'll find several ways to create lists in Python.

That's one of the features that make lists so versatile and popular.
For example, you can create lists using one of the following tools:

- List Literals
- The `list()`

- A list comprehension

In the following sections, we will discuss on how to use the three ($_3$_) tools listed above to create new lists in your code.

Let's start off with **list literals**.

3.2.1 Creating List Using Literals

List literals are probably the most popular way to create a list object in Python. These literals are fairly straightforward. They consist of a pair of square brackets (`[]`) enclosing a comma-separated series (`,`) of objects.

Here's the general syntax of a list literal:

```
[item_0, item_1, ..., item_n]
```

This syntax creates a list of `n` items by listing the items in an enclosing pair of square brackets.

> You don't have to declare the items' type or the list's size beforehand. Remember that lists have a variable size and can store heterogeneous objects.

Here are a few examples of how to use the literal syntax to create new lists:

```
digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
fruits = ["apple", "banana", "orange", "kiwi", "grape"]
cities = [
    "New York",
    "Los Angeles",
    "Chicago",
    "Houston",
    "Philadelphia"
]

matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

inventory = [
    {"product": "phone", "price": 1000, "quantity": 10},
    {"product": "laptop", "price": 1500, "quantity": 5},
    {"product": "tablet", "price": 500, "quantity": 20}
]

functions = [print, len, range, type, enumerate]
```

```
empty = []
```

In these examples, you use the list literal syntax to create lists containing numbers, strings, other lists, dictionaries, and even function objects. As you already know, lists can store any type of object. They can also be **empty**, like the final list in the above code snippet.

Empty lists are useful in many situations. For example, maybe you want to create a list of objects resulting from computations that run in a loop. The loop will allow you to populate the empty list one element at a time.

Using a list literal is arguably the most common way to create lists. You'll find these literals in many Python examples and codebases. They come in handy when you have a series of elements with closely related meanings, and you want to pack them into a single data structure.

Note that **naming lists as plural nouns** is a common practice that improves readability. However, there are situations where you can use collective nouns as well.

For example, you can have a list called people. In this case, every item will be a person. Another example would be a list that represents a table in a database. You can call the list table, and each item will be a row.

3.2.2 Using the list() Constructor

Another tool that allows you to create list objects is the class constructor, `list()`. You can call this constructor with any iterable object, including other lists, tuples, sets, dictionaries and their components, strings, and many others. You can also call it without any arguments, in which case you'll get an empty list back.

Here's the general syntax:

```
list([iterable])
```

To create a list, you need to call `list()` as you'd call any class constructor or function.

> The square brackets around iterable mean that the argument is optional, so the brackets aren't part of the syntax

Here are a few examples of how to use the constructor:

```
print(list((0, 1, 2, 3, 4, 5, 6, 7, 8, 9)))

print(list({"circle", "square", "triangle", "rectangle", "pentagon"}))

print(list({"name": "John", "age": 30, "city": "New York"}.items()))

print(list("Innsbruck"))
```

```
print(list())
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
['triangle', 'rectangle', 'square', 'pentagon', 'circle']
[('name', 'John'), ('age', 30), ('city', 'New York')]
['I', 'n', 'n', 's', 'b', 'r', 'u', 'c', 'k']
[]
```

In these examples, you create different lists using the `list()` constructor, which accepts any type of iterable object, including tuples, dictionaries, strings, and many more. It even accepts sets, in which case you need to remember that sets are unordered data structures, so you won't be able to predict the final order of items in the resulting list.

Calling `list()` without an argument creates and returns a new empty list. This way of creating empty lists is less common than using an empty pair of square brackets. However, in some situations, it can make your code more explicit by clearly communicating your intent: creating an empty list.

The `list()` constructor is especially useful when you need to create a list out of an iterator object. For example, say that you have a generator function that yields numbers from the Fibonacci sequence on demand, and you need to store the first ten numbers in a list.

In this case, you can use `list()` as in the code below:

```
# We define a function to calculate the fibonacci sequence
# Don't worry if you don't understand as we will look at it later.
def fibonacci_generator(stop):
    """
    Calculates the fibonacci sequence
    """
    current_fib, next_fib = 0, 1
    for _ in range(0, stop):
        fib_number = current_fib
        current_fib, next_fib = next_fib, current_fib + next_fib
        yield fib_number

print(list(fibonacci_generator(10)))
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Calling `fibonacci_generator()` directly returns a generator iterator object that allows you to iterate over the numbers in the Fibonacci sequence up to the index of your choice. However, you don't need an iterator in your code. **You need a list**. A quick way to get that list is to wrap the iterator in a call to `list()`, as you did in the final example.

This technique comes in handy when you're working with functions that return iterators, and you want to construct a list object out of the items that the iterator yields. The `list()` constructor will consume the iterator, build your list, and return it back to you.

> As a side note, you'll often find that built-in and third-party functions return iterators. Functions like reversed(), enumerate(), map(), and filter() are good examples of this practice. It's less common to find functions that directly return list objects, but the built-in sorted() function is one example. It takes an iterable as an argument and returns a list of sorted items.

3.2.3 Building Lists With List Comprehensions

List comprehensions are one of the most distinctive features of Python. They're quite popular in the Python community, so you'll likely find them all around. List comprehensions allow you to quickly create and transform lists using a syntax that mimics a `for` loop but in a single line of code.

The core syntax of list comprehensions looks something like this:

```
[expression(item) for item in iterable]
```

Every list comprehension needs at least three components:

1. `expression()` is a Python expression that returns a concrete value, and most of the time, that value depends on item. Note that it doesn't have to be a function.
2. `item` is the current object from iterable.
3. `iterable` can be any Python iterable object, such as a list, tuple, set, string, or generator.

The for construct iterates over the items in iterable, while expression(item) provides the corresponding list item that results from running the comprehension.

To illustrate how list comprehensions allow you to create new lists out of existing iterables, say that you want to construct a list with the square values of the first ten integer numbers. In this case, you can write the following comprehension:

```
[number ** 2 for number in range(1, 11)]
```

In this example, you use `range()` to get the first ten integer numbers. The comprehension iterates over them while computing the square and building the new list. This example is just a quick sample of what you can do with a list comprehension.
In general, you'll use a list comprehension when you need to create a list of transformed values out of an existing iterable. Comprehensions are a great tool that you need to master as a Python developer. They're optimized for performance and are quick to write.

3.3 Accesing Items in a List: Indexing

You can access individual items from a list using the item's associated index. What's an item's index? Each item in a list has an index that specifies its position in the list. Indices are integer numbers that start at 0 and go up to the number of items in the list minus 1.
To access a list item through its index, you use the following syntax:

```
list_object[index]
```

This construct is known as an indexing operation, and the [index] part is known as the indexing operator. It consists of a pair of square brackets enclosing the desired or target index. You can read this construct as from $\text{list}_{\text{object}}$ give me the item at index.
Here's how this syntax works in practice:

```
languages = ["Python", "Java", "JavaScript", "C++", "Go", "Rust"]

print(languages[0])
print(languages[1])
print(languages[2])
print(languages[3])
print(languages[4])
print(languages[5])
```

```
Python
Java
JavaScript
C++
Go
Rust
```

Indexing your list with different indices gives you direct access to the underlying items.
The number of items in a list is known as the list's length. You can check the length of a list by using the built-in `len()` function:

```
len(languages)
```

So, the index of the last item in languages is 6 - 1 = 5. That's the index of the "Rust" item in your sample list. If you use an index greater than this number in an indexing operation, then you get an IndexError exception:

```
print(languages[6])
```

In this example, you try to retrieve the item at index 6. Because this index is greater than 5, you get an IndexError as a result. Using out-of-range indices can be an incredibly common issue when you work with lists, so keep an eye on your target indices.
Indexing operations are quite flexible in Python. For example, you can also use negative indices while indexing lists. This kind of index gives you access to the list items in backward order:

```
print(languages[-1])
print(languages[-2])
print(languages[-3])
print(languages[-4])
print(languages[-5])
print(languages[-6])
```

```
Rust
Go
C++
JavaScript
Java
Python
```

A negative index specifies an element's position relative to the right end of the list, moving back to the beginning of the list. Here's a representation of the process:

You can access the last item in a list using index -1. Similarly, index -2 specifies the next-to-last item, and so forth. It's important to note that negative indices don't start from 0 because 0 already points to the first item. This may be confusing when you're first learning about negative and positive indices, but you'll get used to it. It just takes a bit of practice.

Note that if you use negative indices, then -len(languages) will be the first item in the list. If you use an index lower than that value, then you get an IndexError:

```
print(languages[-7])
```

When you use an index lower than -len(languages), you get an error telling you that the target index is out of range.

Using negative indices can be very convenient in many situations. For example, accessing the last item in a list is a fairly common operation. In Python, you can do this by using negative indices, like in `languages[-1]`, which is more readable and concise than doing something like `languages[len(languages) - 1]`.

> Negative indices also come in handy when you need to reverse a list.

As you already know, lists can contain items of any type, including other lists and sequences. When you have a list containing other sequences, you can access the items in any nested sequence by chaining indexing operations. Consider the following list of employee records:

```
employees = [
    ("John", 30, "Software Engineer"),
    ("Alice", 25, "Web Developer"),
    ("Bob", 45, "Data Analyst"),
    ("Mark", 22, "Intern"),
```

```
    ("Samantha", 36, "Project Manager")
]
```

How can you access the individual pieces of data from any given employee? You can use the following indexing syntax:

```
list_of_sequences[index_0][index_1]...[index_n]
```

The number at the end of each index represents the level of nesting for the list. For example, your employee list has one level of nesting. Therefore, to access Alice's data, you can do something like this:

```
print(employees[1][0])
print(employees[1][1])
print(employees[1][2])
```

In this example, when you do `employees[1][0]`, index 1 refers to the second item in the employees list. That's a nested list containing three items. The 0 refers to the first item in that nested list, which is "Alice". As you can see, you can access items in the nested lists by applying multiple indexing operations in a row. This technique is extensible to lists with more than one level of nesting.

If the nested items are dictionaries, then you can access their data using keys:

```
employees = [
    {"name": "John", "age": 30, "job": "Software Engineer"},
    {"name": "Alice", "age": 25, "job": "Web Developer"},
    {"name": "Bob", "age": 45, "job": "Data Analyst"},
    {"name": "Mark", "age": 22, "job": "Intern"},
    {"name": "Samantha", "age": 36, "job": "Project Manager"}
]

print(employees[3]["name"])
print(employees[3]["age"])
print(employees[3]["job"])
```

In this example, you have a list of dictionaries. To access the data from one of the dictionaries, you need to use its index in the list, followed by the target key in square brackets.

3.3.1 Retrieving Multiple Items From a List: Slicing

Another common requirement when working with lists is to extract a portion, or slice, of a given list. You can do this with a slicing operation, which has the following syntax:

```
list_object[start:stop:step]
```

The [start:stop:step] part of this construct is known as the slicing operator. Its syntax consists of a pair of square brackets and three optional indices, start, stop, and step. The second colon

is optional. You typically use it only in those cases where you need a step value different from 1.

> Slicing is an operation that's common to all Python sequence data types, including lists, tuples, strings, ranges, and others.

Here's what the indices in the slicing operator mean:

- start specifies the index at which you want to start the slicing. The resulting slice includes the item at this index.
- stop specifies the index at which you want the slicing to stop extracting items. The resulting slice doesn't include the item at this index.
- step provides an integer value representing how many items the slicing will skip on each step. The resulting slice won't include the skipped items.

The minimal working variation of the indexing operator is [:]. In this variation, you rely on the default values of all the indices and take advantage of the fact that the second colon is optional. The [::] variation of the slicing operator produces the same result as [:]. This time, you rely on the default value of the three indices.

> Both of the above variations of the slicing operator ([:] and [::]) allow you to create a shallow copy of your target list.

Now it's time for you to explore some examples of how slicing works:

```
letters = ["A", "a", "B", "b", "C", "c", "D", "d"]

upper_letters = letters[0::2] # Or [::2]
print(upper_letters)


lower_letters = letters[1::2]
print(lower_letters)
```

```
['A', 'B', 'C', 'D']
['a', 'b', 'c', 'd']
```

In this example, you have a list of letters in uppercase and lowercase. You want to extract the uppercase letters into one list and the lowercase letters into another list. The [0::2] operator helps you with the first task, and [1::2] helps you with the second.

In both examples, you've set step to 2 because you want to retrieve every other letter from the original list. In the first slicing, you use a start of 0 because you want to start from the very beginning of the list. In the second slicing, you use a start of 1 because you need to jump over the first item and start extracting items from the second one.

You can use any variation of the slicing operator that fits your needs. In many situations, relying on the default indices is pretty helpful. In the examples above, you rely on the default value of stop, which is len($list_{object}$). This practice allows you to run the slicing all the way up to the last item of the target list.

Here are a few more examples of slicing:

```
digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

first_three = digits[:3]
print(first_three)

middle_four = digits[3:7]
print(middle_four)

last_three = digits[-3:]
print(last_three)

every_other = digits[::2]
print(every_other)

every_three = digits[::3]
print(every_three)
```

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

In these examples, the variable names reflect the portion of the list that you're extracting in every slicing operation. As you can conclude, the slicing operator is pretty flexible and versatile. It even allows you to use negative indices.

Every slicing operation uses a slice object internally. The built-in slice() function provides an alternative way to create slice objects that you can use to extract multiple items from a list. The signature of this built-in function is the following:

```
slice(start, stop, step)
```

It takes three arguments with the same meaning as the indices in the slicing operator and returns a slice object equivalent to [start:stop:step]. To illustrate how slice() works, get back to the letters example and rewrite it using this function instead of the slicing operator. You'll end up with something like the following:

```
letters = ["A", "a", "B", "b", "C", "c", "D", "d"]

upper_letters = letters[slice(0, None, 2)]
print(upper_letters)

lower_letters = letters[slice(1, None, 2)]
print(lower_letters)
```

```
['A', 'B', 'C', 'D']
['a', 'b', 'c', 'd']
```

Passing None to any arguments of slice() tells the function that you want to rely on its internal default value, which is the same as the equivalent index's default in the slicing operator. In these examples, you pass None to stop, which tells slice() that you want to use len(letters) as the value for stop.

As an exercise, you can write the digits examples using slice() instead of the slicing operator. Go ahead and give it a try! This practice will help you better understand the intricacies of slicing operations in Python.

Finally, it's important to note that out-of-range values for start and stop don't cause slicing expressions to raise a TypeError exception. In general, you'll observe the following behaviors:

- If start is before the beginning of the list, which can happen when you use negative indices, then Python will use 0 instead.
- If start is greater than stop, then the slicing will return an empty list.
- If stop is beyond the length of the list, then Python will use the length of the list instead.

```
colors = [
    "red",
    "orange",
    "yellow",
    "green",
    "blue",
    "indigo",
    "violet"
]

print(len(colors))
print(colors[-8:])
print(colors[8:])
print(colors[:8])
```

```
7
['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

```
[]
['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

In these examples, your color list has seven items, so len(colors) returns 7. In the first example, you use a negative value for start. The first item of colors is at index -7. Because -8 < -7, Python replaces your start value with 0, which results in a slice that contains the items from 0 to the end of the list.

> In the examples above, you use only one colon in each slicing. In day-to-day coding, this is common practice. You'll only use the second colon if you need to provide a step different from 1.

In the second example, you use a start value that's greater than the length of colors. Because there's nothing to retrieve beyond the end of the list, Python returns an empty list. In the final example, you use a stop value that's greater than the length of colors. In this case, Python retrieves all the items up to the end of the list.

3.4 Updating Items: Index Assignments

Python lists are mutable data types. This means that you can change their elements without changing the identity of the underlying list. These kinds of changes are commonly known as in-place mutations. They allow you to update the value of one or more items in an existing list.

To change the value of a given element in a list, you can use the following syntax:

```
list_object[index] = new_value
```

The indexing operator gives you access to the target item through its index, while the assignment operator allows you to change its current value.

Here's how this assignment works:

```
numbers = [1, 2, 3, 4]

numbers[0] = "one"
print(numbers)

numbers[1] = "two"
print(numbers)

numbers[-1] = "four"
print(numbers)

numbers[-2] = "three"
print(numbers)
```

```
['one', 2, 3, 4]
```

```
['one', 'two', 3, 4]
['one', 'two', 3, 'four']
['one', 'two', 'three', 'four']
```

In this example, you've replaced all the numeric values in numbers with strings. To do that, you've used their indices and the assignment operator in what you can call index assignments. Note that negative indices also work.

What if you know an item's value but don't know its index in the list? How can you update the item's value? In this case, you can use the .index() method as in the code below:

```
fruits = ["apple", "banana", "orange", "kiwi", "grape"]

fruits[fruits.index("kiwi")] = "mango"
print(fruits)
```

```
['apple', 'banana', 'orange', 'mango', 'grape']
```

The .index() method takes a specific item as an argument and returns the index of the first occurrence of that item in the underlying list. You can take advantage of this behavior when you know the item that you want to update but not its index. However, note that if the target item isn't present in the list, then you'll get a ValueError.

You can also update the value of multiple list items in one go. To do that, you can access the items with the slicing operator and then use the assignment operator and an iterable of new values. This combination of operators can be called slice assignment for short.

Here's the general syntax:

```
list_object[start:stop:step] = iterable
```

In this syntax, the values from iterable replace the portion of $\text{list}_{\text{object}}$ defined by the slicing operator. If iterable has the same number of elements as the target slice, then Python updates the elements one by one without altering the length of $\text{list}_{\text{object}}$.

To understand these behaviors, consider the following examples:

```
numbers = [1, 2, 3, 4, 5, 6, 7]

numbers[1:4] = [22, 33, 44]
print(numbers)
```

```
[1, 22, 33, 44, 5, 6, 7]
```

In this example, you update the items located from 1 to 4, without including the last item. In this slice, you have three items, so you use a list of three new values to update them one by one.

If iterable has more or fewer elements than the target slice, then $\text{list}_{\text{object}}$ will automatically grow or shrink accordingly:

```
numbers = [1, 5, 6, 7]

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

Now the initial list of numbers only has four values. The values 1, 2, and 3 are missing. So, you use a slice to insert them starting at index 1. In this case, the slice has a single index, while the list of values has three new values, so Python grows your list automatically to hold the new values.

You can also use a slice to shrink an existing list:

```
numbers = [1, 2, 0, 0, 0, 0, 4, 5, 6, 7]

numbers[2:6] = [3]
print(numbers)
```

Here, the initial list has a bunch of zeros where it should have a 3. Your slicing operator takes the portion filled with zeros and replaces it with a single 3.

Using the slicing operator to update the value of several items in an existing list is a pretty useful technique that may be hard to grasp at first.

3.5 Exercises

NOTE: Some of the questions may require you to know more information on the topic (i.e., loops and statements however, the questions are still present here for you to work with.
If in doubt on how to tackle the StackOverflow is an invauable site to get more information and solve the following problems.

1. Write a Python program to sum all the items in a list.
2. Write a Python program to multiply all the items in a list.
3. Write a Python program to get the largest number from a list.
4. Write a Python program to get the smallest number from a list.
5. Write a Python program to remove duplicates from a list.
6. Write a Python program to check if a list is empty or not.
7. Write a Python program to find the list of words that are longer than n from a given list of words.

Chapter 4

Dictionaries

Table of Contents

Python provides another composite data type called a dictionary, which is similar to a list in that it is a collection of objects.
Dictionaries and lists share the following characteristics:

- Both are mutable.
- Both are dynamic. They can grow and shrink as needed.
- Both can be nested. A list can contain another list. A dictionary can contain another dictionary. A dictionary can also contain a list, and vice versa.

Dictionaries differ from lists primarily in how elements are accessed:

- List elements are accessed by their position in the list, via indexing.
- Dictionary elements are accessed via keys.

4.1 Defining A Dictionary

Dictionaries are Python's implementation of a data structure that is more generally known as an associative array. A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.
You can define a dictionary by enclosing a comma-separated list of key-value pairs in curly braces (`{}`). A colon (`:`) separates each key from its associated value:

```
d = {
    <key>: <value>,
    <key>: <value>,
```

```
      .
      .
      .
    <key>: <value>
}
```

The following defines a dictionary that maps a location to the name of its corresponding Major League Baseball team:

```
MLB_team = {
    'Colorado' : 'Rockies',
    'Boston'   : 'Red Sox',
    'Minnesota': 'Twins',
    'Milwaukee': 'Brewers',
    'Seattle'  : 'Mariners'
}
```

You can also construct a dictionary with the built-in dict() function. The argument to dict() should be a sequence of key-value pairs. A list of tuples works well for this:

```
d = dict([
    (<key>, <value>),
    (<key>, <value),
      .
      .
      .
    (<key>, <value>)
])
```

`MLB_team` can then also be defined this way:

```
MLB_team = dict([
    ('Colorado', 'Rockies'),
    ('Boston', 'Red Sox'),
    ('Minnesota', 'Twins'),
    ('Milwaukee', 'Brewers'),
    ('Seattle', 'Mariners')
])
```

If the key values are simple strings, they can be specified as keyword arguments. So here is yet another way to define `MLB_team`:

```
MLB_team = dict(
    Colorado='Rockies',
    Boston='Red Sox',
    Minnesota='Twins',
    Milwaukee='Brewers',
    Seattle='Mariners'
)
```

Once you've defined a dictionary, you can display its contents, the same as you can do for a list. All three of the definitions shown above appear as follows when displayed:

```
print(type(MLB_team))
```

```
<class 'dict'>
```

The entries in the dictionary display in the order they were defined. But that is irrelevant when it comes to retrieving them. Dictionary elements are not accessed by numerical index. The following code would produce and **error**

```
MLB_team[1]
```

Perhaps you'd still like to sort your dictionary. If that's the case, then check out Sorting a Python Dictionary: Values, Keys, and More.

4.2 Accessing Dictionary Values

Of course, dictionary elements must be accessible somehow. If you don't get them by index, then how do you get them?
A value is retrieved from a dictionary by specifying its corresponding key in square brackets ([]):

```
MLB_team['Minnesota']
MLB_team['Colorado']
```

If you refer to a key that is not in the dictionary, Python raises an exception:

```
MLB_team['Toronto']
```

Adding an entry to an existing dictionary is simply a matter of assigning a new key and value:

```
MLB_team['Kansas City'] = 'Royals'
```

If you want to update an entry, you can just assign a new value to an existing key:

```
MLB_team['Seattle'] = 'Seahawks'
```

To delete an entry, use the `del` statement, specifying the key to delete:

```
del MLB_team['Seattle']
```

4.3 Dictionary Keys vs. List Indices

You may have noticed that the interpreter raises the same exception, KeyError, when a dictionary is accessed with either an undefined key or by a numeric index:

```
MLB_team['Toronto']
```

In fact, it's the same error. In the latter case, [1] looks like a numerical index, but it isn't. You will see later in this tutorial that an object of any immutable type can be used as a dictionary key. Accordingly, there is no reason you can't use integers:

```
d = {0: 'a', 1: 'b', 2: 'c', 3: 'd'}
d


d[0]

d[2]
```

In the expressions MLB_{team}[1], d[0], and d[2], the numbers in square brackets appear as though they might be indices. But they have nothing to do with the order of the items in the dictionary. Python is interpreting them as dictionary keys. If you define this same dictionary in reverse order, you still get the same values using the same keys:

```
d = {3: 'd', 2: 'c', 1: 'b', 0: 'a'}
d


d[0]

d[2]
```

The syntax may look similar, but you can't treat a dictionary like a list:

```
print(type(d))
print(d[-1])
print(d[0:2])
print(d.append('e'))
```

> Although access to items in a dictionary does not depend on order, Python does guarantee that the order of items in a dictionary is preserved. When displayed, items will appear in the order they were defined, and iteration through the keys will occur in that order as well. Items added to a dictionary are added at the end. If items are deleted, the order of the remaining items is retained.

4.4 Building a Dictionary Incrementally

Defining a dictionary using curly braces and a list of key-value pairs, as shown above, is fine if you know all the keys and values in advance. But what if you want to build a dictionary on the fly?

You can start by creating an empty dictionary, which is specified by empty curly braces. Then you can add new keys and values one at a time:

```
person = {}
type(person)


person['fname'] = 'Joe'
person['lname'] = 'Fonebone'
person['age'] = 51
person['spouse'] = 'Edna'
person['children'] = ['Ralph', 'Betty', 'Joey']
person['pets'] = {'dog': 'Fido', 'cat': 'Sox'}
```

Once the dictionary is created in this way, its values are accessed the same way as any other dictionary:

```
person
person['fname']
person['age']
person['children']
```

Retrieving the values in the sublist or subdictionary requires an additional index or key:

```
person['children'][-1]
person['pets']['cat']
```

This example exhibits another feature of dictionaries: the values contained in the dictionary don't need to be the same type. In person, some of the values are strings, one is an integer, one is a list, and one is another dictionary.

Just as the values in a dictionary don't need to be of the same type, the keys don't either:

```
foo = {42: 'aaa', 2.78: 'bbb', True: 'ccc'}
print(foo)
print(foo[42])
print(foo[2.78])
print(foo[True])
```

Here, one of the keys is an integer, one is a float, and one is a Boolean. It's not obvious how this would be useful, but you never know.

Notice how versatile Python dictionaries are. In MLB_{team}, the same piece of information (the baseball team name) is kept for each of several different geographical locations. person, on the other hand, stores varying types of data for a single person.

You can use dictionaries for a wide range of purposes because there are so few limitations on the keys and values that are allowed. But there are some. Read on!

4.5 Restrictions on Dictionary Keys

Almost any type of value can be used as a dictionary key in Python. You just saw this example, where integer, float, and Boolean objects are used as keys:

```
foo = {42: 'aaa', 2.78: 'bbb', True: 'ccc'}
foo
```

You can even use built-in objects like types and functions:

```
d = {int: 1, float: 2, bool: 3}
d

d[float]


d = {bin: 1, hex: 2, oct: 3}
d[oct]
```

However, there are a couple restrictions that dictionary keys must abide by.
First, a given key can appear in a dictionary only once. Duplicate keys are not allowed. A dictionary maps each key to a corresponding value, so it doesn't make sense to map a particular key more than once.

You saw above that when you assign a value to an already existing dictionary key, it does not add the key a second time, but replaces the existing value:

Chapter 5

Conditional Statements

Table of Contents

Everything we have seen so far has consisted of **sequential execution**, in which statements are always performed one after the next, in exactly the order specified.

But the world is often more complicated than that. Frequently, a program needs to skip over some statements, execute a series of statements repetitively, or choose between alternate sets of statements to execute.

That is where control structures come in. A control structure directs the order of execution of the statements in a program (referred to as the program's control flow).

In the real world, we commonly must evaluate information around us and then choose one course of action or another based on what we observe:

If the weather is nice, then I'll mow the lawn. (It's implied that if the weather isn't nice, then I won't mow the lawn.)

In a Python program, the if statement is how you perform this sort of decision-making. It allows for conditional execution of a statement or group of statements based on the value of an expression.

5.1 A Gentle Introduction

We'll start by looking at the most basic type of if statement. In its simplest form, it looks like this:

```
if <expr>:
    <statement>
```

In the form shown above:

- <expr> is an expression evaluated in a Boolean context.
- <statement> is a valid Python statement, which **MUST** be indented. (You will see why very soon.)

If <expr> is true (evaluates to a value that is "truthy"), then <statement> is executed. If <expr> is false, then <statement> is skipped over and not executed.

Note that the colon (:) following <expr> is required. Some programming languages require <expr> to be enclosed in parentheses, but Python does not.

Here are several examples of this type of if statement:

```
x = 0
y = 5

if x < y:                           # Truthy
    print('yes')

if y < x:                           # Falsy
    print('yes')

if x:                               # Falsy
    print('yes')

if y:                               # Truthy
    print('yes')

if x or y:                          # Truthy
    print('yes')

if x and y:                         # Falsy
    print('yes')

if 'aul' in 'grault':               # Truthy
    print('yes')

if 'quux' in ['foo', 'bar', 'baz']: # Falsy
    print('yes')
```

If you are trying these examples interactively in a REPL session, you'll find that, when you hit Enter after typing in the print('yes') statement, nothing happens.
Because this is a multiline statement, you need to hit Enter a second time to tell the interpreter that you're finished with it. This extra newline is not necessary in code executed from a script file.

5.2 Grouping Statements: Indentation and Blocks

let's say you want to evaluate a condition and then do more than one thing if it is true:

If the weather is nice, then I will:

- Mow the lawn
- Weed the garden
- Take the dog for a walk

If the weather isn't nice, then I won't do any of these things.

In all the examples shown above, each if <expr>: has been followed by only a single <statement>. There needs to be some way to say "If <expr> is true, do all of the following things."

The usual approach taken by most programming languages is to define a syntactic device that groups multiple statements into one compound statement or block. A block is regarded syntactically as a single entity. When it is the target of an if statement, and <expr> is true, then all the statements in the block are executed. If <expr> is false, then none of them are.

Virtually all programming languages provide the capability to define blocks, but they don't all provide it in the same way. Let's see how Python does it.

5.2.1 It's All About the Indentation

Python follows a convention known as the off-side rule, a term coined by British computer scientist Peter J. Landin. (The term is taken from football.)

Languages that adhere to the off-side rule define blocks by indentation. Python is one of a relatively small set of off-side rule languages.

Recall from the previous tutorial on Python program structure that indentation has special significance in a Python program. Now you know why: indentation is used to define compound statements or blocks. In a Python program, contiguous statements that are indented to the same level are considered to be part of the same block.

Thus, a compound if statement in Python looks like this:

```
if <expr>:
    <statement>
    <statement>
    ...
    <statement>
<following_statement>
```

Here, all the statements at the matching indentation level (lines 2 to 5) are considered part of the same block. The entire block is executed if <expr> is true, or skipped over if <expr> is false. Either way, execution proceeds with <following_statement> (line 6) afterward.
Notice that there is no token that denotes the end of the block. Rather, the end of the block is indicated by a line that is indented less than the lines of the block itself.

> In the Python documentation, a group of statements defined by indentation is often referred to as a **suite**.

Consider this script file:

```
if 'foo' in ['bar', 'baz', 'qux']:
    print('Expression was true')
    print('Executing statement in suite')
    print('...')
    print('Done.')

print('After conditional')
```

Running it produces this output:

```
After conditional
```

The four print() statements on lines 2 to 5 are indented to the same level as one another. They constitute the block that would be executed if the condition were true. But it is false, so all the statements in the block are skipped. After the end of the compound if statement has been reached (whether the statements in the block on lines 2 to 5 are executed or not), execution proceeds to the first statement having a lesser indentation level: the print() statement on line 6.

Blocks can be nested to arbitrary depth. Each indent defines a new block, and each outdent ends the preceding block. The resulting structure is straightforward, consistent, and intuitive. Here is a more complicated script file called blocks.py:

```
# Does line execute?                        Yes    No
#                                           ---    --
if 'foo' in ['foo', 'bar', 'baz']:        #  x
    print('Outer condition is true')      #  x

    if 10 > 20:                           #  x
        print('Inner condition 1')        #        x
```

```
    print('Between inner conditions')     #  x

    if 10 < 20:                           #  x
        print('Inner condition 2')        #  x

    print('End of outer condition')       #  x
print('After outer condition')            #  x
```

The output generated when this script is run is shown below:

```
Outer condition is true
Between inner conditions
Inner condition 2
End of outer condition
After outer condition
```

> In case you have been wondering, the off-side rule is the reason for the necessity of the extra newline when entering multiline statements in a REPL session. The interpreter otherwise has no way to know that the last statement of the block has been entered.

5.2.2 Advantages and Disadvantages

On the plus side:

- Python's use of indentation is clean, concise, and consistent.
- In programming languages that do not use the off-side rule, indentation of code is completely independent of block definition and code function. It's possible to write code that is indented in a manner that does not actually match how the code executes, thus creating a mistaken impression when a person just glances at it. This sort of mistake is virtually impossible to make in Python.
- Use of indentation to define blocks forces you to maintain code formatting standards you probably should be using anyway.

On the negative side:

- Many programmers don't like to be forced to do things a certain way. They tend to have strong opinions about what looks good and what doesn't, and they don't like to be shoehorned into a specific choice
- Some editors insert a mix of space and tab characters to the left of indented lines, which makes it difficult for the Python interpreter to determine indentation levels. On the other hand, it is frequently possible to configure editors not to do this. It generally isn't considered desirable to have a mix of tabs and spaces in source code anyhow, no matter the language.

5.3 The else and elif Clauses

Now you know how to use an if statement to conditionally execute a single statement or a block of several statements. It's time to find out what else you can do.
Sometimes, you want to evaluate a condition and take one path if it is true but specify an alternative path if it is not. This is accomplished with an else clause:

```
if <expr>:
    <statement(s)>
else:
    <statement(s)>
```

If <expr> is true, the first suite is executed, and the second is skipped. If <expr> is false, the first suite is skipped and the second is executed. Either way, execution then resumes after the second suite. Both suites are defined by indentation, as described above.
In this example, x is less than 50, so the first suite (lines 4 to 5) are executed, and the second suite (lines 7 to 8) are skipped:

```
x = 20

if x < 50:
    print('(first suite)')
    print('x is small')
else:
    print('(second suite)')
    print('x is large')
```

Here, on the other hand, x is greater than 50, so the first suite is passed over, and the second suite executed:

```
x = 120

if x < 50:
    print('(first suite)')
    print('x is small')
else:
    print('(second suite)')
    print('x is large')
```

There is also syntax for branching execution based on several alternatives. For this, use one or more elif (short for else if) clauses. Python evaluates each <expr> in turn and executes the suite corresponding to the first that is true. If none of the expressions are true, and an else clause is specified, then its suite is executed:

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
```

```
elif <expr>:
    <statement(s)>
    ...
else:
    <statement(s)>
```

An arbitrary number of elif clauses can be specified. The else clause is optional. If it is present, there can be only one, and it must be specified last:

```
name = 'Joe'
if name == 'Fred':
    print('Hello Fred')
elif name == 'Xander':
    print('Hello Xander')
elif name == 'Joe':
    print('Hello Joe')
elif name == 'Arnold':
    print('Hello Arnold')
else:
    print("I don't know who you are!")
```

At most, one of the code blocks specified will be executed. If an else clause isn't included, and all the conditions are false, then none of the blocks will be executed.

A Cleaner Way

Using a lengthy if/elif/else series can be a little inelegant, especially when the actions are simple statements like print(). In many cases, there may be a more Pythonic way to accomplish the same thing.
Here's one possible alternative to the example above using the dict.get() method:

```
names = {
    'Fred': 'Hello Fred',
    'Xander': 'Hello Xander',
    'Joe': 'Hello Joe',
    'Arnold': 'Hello Arnold'
}

print(names.get('Joe', "I don't know who you are!"))

print(names.get('Rick', "I don't know who you are!"))
```

An if statement with elif clauses uses short-circuit evaluation, analogous to what you saw with the and and or operators. Once one of the expressions is found to be true and its block is executed, none of the remaining expressions are tested. This is demonstrated below:

```
var  # Not defined
```

```
if 'a' in 'bar':
    print('foo')
elif 1/0:
    print("This won't happen")
elif var:
    print("This won't either")
```

The second expression contains a division by zero, and the third references an undefined variable var. Either would raise an error, but neither is evaluated because the first condition specified is true.

5.4 One Line if Statements

It is customary to write if <expr> on one line and <statement> indented on the following line like this:

```
if <expr>:
    <statement>
```

But it is permissible to write an entire if statement on one line. The following is functionally equivalent to the example above:

```
if <expr>: <statement>
```

There can even be more than one <statement> on the same line, separated by semicolons:

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

But what does this mean? There are two possible interpretations:
If <expr> is true, execute <statement$_1$>.
Then, execute <statement_2> ... <statement_n> unconditionally, irrespective of whether <expr> is true or not.
If <expr> is true, execute all of <statement_1> ... <statement$_n$>. Otherwise, don't execute any of them.
Python takes the latter interpretation. The semicolon separating the <statements> has higher precedence than the colon following <expr>—in computer lingo, the semicolon is said to bind more tightly than the colon. Thus, the <statements> are treated as a suite, and either all of them are executed, or none of them are:

```
if 'f' in 'foo': print('1'); print('2'); print('3')



if 'z' in 'foo': print('1'); print('2'); print('3')
```

Multiple statements may be specified on the same line as an elif or else clause as well:

```
x = 2
if x == 1: print('foo'); print('bar'); print('baz')
elif x == 2: print('qux'); print('quux')
else: print('corge'); print('grault')




x = 3
if x == 1: print('foo'); print('bar'); print('baz')
elif x == 2: print('qux'); print('quux')
else: print('corge'); print('grault')
```

While all of this works, and the interpreter allows it, it is generally discouraged on the grounds that it leads to poor readability, particularly for complex if statements. PEP 8 specifically recommends against it.

As usual, it is somewhat a matter of taste. Most people would find the following more visually appealing and easier to understand at first glance than the example above:

```
x = 3
if x == 1:
    print('foo')
    print('bar')
    print('baz')
elif x == 2:
    print('qux')
    print('quux')
else:
    print('corge')
    print('grault')
```

If an if statement is simple enough, though, putting it all on one line may be reasonable. Something like this probably wouldn't raise anyone's hackles too much:

```
debugging = True  # Set to True to turn debugging on.

    .
    .
    .

if debugging: print('About to call function foo()')
foo()
```

5.5 Conditional Expressions

Python supports one additional decision-making entity called a conditional expression. (It is also referred to as a conditional operator or ternary operator in various places in the Python

documentation.) Conditional expressions were proposed for addition to the language in PEP 308 and green-lighted by Guido in 2005.
In its simplest form, the syntax of the conditional expression is as follows:

```
<expr1> if ~<conditional_expr>~ else <expr2>
```

This is different from the if statement forms listed above because it is not a control structure that directs the flow of program execution. It acts more like an operator that defines an expression. In the above example, <conditional$_{expr}$> is evaluated first. If it is true, the expression evaluates to <expr1>. If it is false, the expression evaluates to <expr2>.

Notice the non-obvious order: the middle expression is evaluated first, and based on that result, one of the expressions on the ends is returned. Here are some examples that will hopefully help clarify:

```
raining = False
print("Let's go to the", 'beach' if not raining else 'library')

raining = True
print("Let's go to the", 'beach' if not raining else 'library')


age = 12
s = 'minor' if age < 21 else 'adult'
s


'yes' if ('qux' in ['foo', 'bar', 'baz']) else 'no'
```

> Python's conditional expression is similar to the `<conditional_expr> ? <expr1> : <expr2>` syntax used by many other languages—C, Perl and Java to name a few. In fact, the ?: operator is commonly called the ternary operator in those languages, which is probably the reason Python's conditional expression is sometimes referred to as the Python ternary operator.
> You can see in PEP 308 that the `<conditional_expr> ? <expr1> : <expr2>` syntax was considered for Python but ultimately rejected in favor of the syntax shown above.

A common use of the conditional expression is to select variable assignment. For example, suppose you want to find the larger of two numbers. Of course, there is a built-in function, max(), that does just this (and more) that you could use. But suppose you want to write your own code from scratch.
You could use a standard if statement with an else clause:

```
if a > b:
    m = a
else:
    m = b
```

But a conditional expression is shorter and arguably more readable as well:

```
m = a if a > b else b
```

Remember that the conditional expression behaves like an expression syntactically. It can be used as part of a longer expression. The conditional expression has lower precedence than virtually all the other operators, so parentheses are needed to group it by itself.

In the following example, the + operator binds more tightly than the conditional expression, so 1 + x and y + 2 are evaluated first, followed by the conditional expression. The parentheses in the second case are unnecessary and do not change the result:

```
x = y = 40

z = 1 + x if x > y else y + 2
z



z = (1 + x) if x > y else (y + 2)
z
```

If you want the conditional expression to be evaluated first, you need to surround it with grouping parentheses. In the next example, (x if x > y else y) is evaluated first. The result is y, which is 40, so z is assigned 1 + 40 + 2 = 43:

```
x = y = 40

z = 1 + (x if x > y else y) + 2
z
```

f you are using a conditional expression as part of a larger expression, it probably is a good idea to use grouping parentheses for clarification even if they are not needed.
Conditional expressions also use short-circuit evaluation like compound logical expressions. Portions of a conditional expression are not evaluated if they don't need to be.
In the expression <expr1> if <conditional_expr> else <expr2>:
If <conditional_expr> is true, <expr1> is returned and <expr2> is not evaluated. If <conditional_expr> is false, <expr2> is returned and <expr1> is not evaluated.
As before, you can verify this by using terms that would raise an error:

```
'foo' if True else 1/0

1/0 if False else 'bar'
```

In both cases, the 1/0 terms are not evaluated, so no exception is raised.
Conditional expressions can also be chained together, as a sort of alternative if/elif/else structure, as shown here:

```
s = ('foo' if (x == 1) else
     'bar' if (x == 2) else
     'baz' if (x == 3) else
     'qux' if (x == 4) else
     'quux'
)
s
```

It's not clear that this has any significant advantage over the corresponding if/elif/else statement, but it is syntactically correct Python.

5.6 The Python pass Statement

Occasionally, you may find that you want to write what is called a code stub: a placeholder for where you will eventually put a block of code that you haven't implemented yet.
In languages where token delimiters are used to define blocks, like the curly braces in Perl and C, empty delimiters can be used to define a code stub. For example, the following is legitimate Perl or C code:

```
# This is not Python
if (x)
{
}
```

Here, the empty curly braces define an empty block. Perl or C will evaluate the expression x, and then even if it is true, quietly do nothing.
Because Python uses indentation instead of delimiters, it is not possible to specify an empty block. If you introduce an if statement with if <expr>:, something has to come after it, either on the same line or indented on the following line.
Consider this script foo.py:

```
if True:

print('foo')
```

If you try to run foo.py, you'll get this:

```
C:\> python foo.py
  File "foo.py", line 3
    print('foo')
        ^
IndentationError: expected an indented block
```

The Python pass statement solves this problem. It doesn't change program behavior at all. It is used as a placeholder to keep the interpreter happy in any situation where a statement is syntactically required, but you don't really want to do anything:

```
if True:
    pass

print('foo')
```

Now foo.py runs without error:

```
C:\> python foo.py
foo
```

5.7 Exercises

1. Write a program which asks user input and the program will tell the user whether it is an odd number or an even number
2. Write a simple calculator app. The code will ask the user which operator to choose which the options are: addition, subtraction, multiplication and division. Once the operation has been decided by the computer, the chosen operation will be done on the two numbers.
3. Write a Python program to find those numbers which are divisible by 7 and multiples of 5, between 1500 and 2700 (both included).
4. Write a Python program to find those numbers which are divisible by 7 and multiples of 5, between 1500 and 2700 (both included).

Chapter 6

For Loops

Table of Contents

Definite iteration loops are frequently referred to as for loops because for is the keyword that is used to introduce them in nearly all programming languages, including Python.

Historically, programming languages have offered a few assorted flavors of for loop.

6.1 Numeric Range Loop

The most basic for loop is a simple numeric range statement with start and end values. The exact format varies depending on the language but typically looks something like this:

```
for i = 1 to 10
    <loop body>
```

Here, the body of the loop is executed ten times. The variable i assumes the value 1 on the first iteration, 2 on the second, and so on. This sort of for loop is used in the languages BASIC, Algol, and Pascal.

6.2 Three-Expression Loop

Another form of for loop popularized by the C programming language contains three parts:
An initialization An expression specifying an ending condition An action to be performed at the end of each iteration.
This type of loop has the following form:

```
for (i = 1; i <= 10; i++)
    <loop body>
```

This loop is interpreted as follows:
Initialize i to 1. Continue looping as long as i <= 10. Increment i by 1 after each loop iteration.
Three-expression for loops are popular because the expressions specified for the three parts can be nearly anything, so this has quite a bit more flexibility than the simpler numeric range form shown above. These for loops are also featured in the C++, Java, PHP, and Perl languages.

6.3 Collection-Based or Iterator-Based Loop

This type of loop iterates over a collection of objects, rather than specifying numeric values or conditions:
Each time through the loop, the variable i takes on the value of the next object in <collection>.
This type of for loop is arguably the most generalized and abstract. Perl and PHP also support this type of loop, but it is introduced by the keyword foreach instead of for.

6.4 The Python for Loop

Of the loop types listed above, Python only implements the last: collection-based iteration. At first blush, that may seem like a raw deal, but rest assured that Python's implementation of definite iteration is so versatile that you won't end up feeling cheated!

Shortly, you'll dig into the guts of Python's for loop in detail. But for now, let's start with a quick prototype and example, just to get acquainted.

Python's for loop looks like this:

```
for <var> in <iterable>:
    <statement(s)>
```

<iterable> is a collection of objects—for example, a list or tuple. The <statement(s)> in the loop body are denoted by indentation, as with all Python control structures, and are executed once for each item in <iterable>. The loop variable <var> takes on the value of the next element in <iterable> each time through the loop.

Here is a representative example:

```
a = ['foo', 'bar', 'baz']
for i in a:
    print(i)
```

In this example, <iterable> is the list a, and <var> is the variable i. Each time through the loop, i takes on a successive item in a, so print() displays the values 'foo', 'bar', and 'baz', respectively. A for loop like this is the Pythonic way to process the items in an iterable.
But what exactly is an iterable? Before examining for loops further, it will be beneficial to delve more deeply into what iterables are in Python.

6.4.1 Iterables

In Python, iterable means an object can be used in iteration. The term is used as:
An adjective: An object may be described as iterable. A noun: An object may be characterized as an iterable.
If an object is iterable, it can be passed to the built-in Python function iter(), which returns something called an iterator. Yes, the terminology gets a bit repetitive. Hang in there. It all works out in the end.
Each of the objects in the following example is an iterable and returns some type of iterator when passed to iter():

```
iter('foobar')                          # String

iter(['foo', 'bar', 'baz'])             # List

iter(('foo', 'bar', 'baz'))             # Tuple

iter({'foo', 'bar', 'baz'})             # Set

iter({'foo': 1, 'bar': 2, 'baz': 3})    # Dict
```

These object types, on the other hand, aren't iterable:

```
iter(42)                                # Integer

iter(3.1)                               # Float

iter(len)                               # Built-in function
```

All the data types you have encountered so far that are collection or container types are iterable. These include the string, list, tuple, dict, set, and frozenset types.

But these are by no means the only types that you can iterate over. Many objects that are built into Python or defined in modules are designed to be iterable. For example, open files

in Python are iterable. As you will see soon in the tutorial on file I/O, iterating over an open file object reads data from the file.

In fact, almost any object in Python can be made iterable. Even user-defined objects can be designed in such a way that they can be iterated over. (You will find out how that is done in the upcoming article on object-oriented programming.)

6.4.2 Iterators

Okay, now you know what it means for an object to be iterable, and you know how to use iter() to obtain an iterator from it. Once you've got an iterator, what can you do with it?
An iterator is essentially a value producer that yields successive values from its associated iterable object. The built-in function next() is used to obtain the next value from in iterator. Here is an example using the same list as above:

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

itr = iter(a)
itr


next(itr)

next(itr)

next(itr)
```

In this example, a is an iterable list and itr is the associated iterator, obtained with iter(). Each next(itr) call obtains the next value from itr.

Notice how an iterator retains its state internally. It knows which values have been obtained already, so when you call next(), it knows what value to return next.

What happens when the iterator runs out of values? Let's make one more next() call on the iterator above:

```
print(next(itr))
```

If all the values from an iterator have been returned already, a subsequent next() call raises a StopIteration exception. Any further attempts to obtain values from the iterator will fail.
You can only obtain values from an iterator in one direction. You can't go backward. There is no prev() function. But you can define two independent iterators on the same iterable object:

```
a


itr1 = iter(a)
```

```
itr2 = iter(a)

next(itr1)

next(itr1)

next(itr1)


next(itr2)
```

Even when iterator itr1 is already at the end of the list, itr2 is still at the beginning. Each iterator maintains its own internal state, independent of the other.

If you want to grab all the values from an iterator at once, you can use the built-in list() function. Among other possible uses, list() takes an iterator as its argument, and returns a list consisting of all the values that the iterator yielded:

```
a = ['foo', 'bar', 'baz']
itr = iter(a)
list(itr)
```

Similarly, the built-in tuple() and set() functions return a tuple and a set, respectively, from all the values an iterator yields:

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

itr = iter(a)
tuple(itr)


itr = iter(a)
set(itr)
```

It isn't necessarily advised to make a habit of this. Part of the elegance of iterators is that they are "lazy." That means that when you create an iterator, it doesn't generate all the items it can yield just then. It waits until you ask for them with next(). Items are not created until they are requested.

When you use list(), tuple(), or the like, you are forcing the iterator to generate all its values at once, so they can all be returned. If the total number of objects the iterator returns is very large, that may take a long time.

In fact, it is possible to create an iterator in Python that returns an endless series of objects using generator functions and itertools. If you try to grab all the values at once from an endless iterator, the program will hang.

6.4.3 The Structure of the Python For Loop

You now have been introduced to all the concepts you need to fully understand how Python's for loop works. Before proceeding, let's review the relevant terms:

Term	Meaning
Iteration	The process of looping through the objects or items in a collection
Iterable	An object (or the adjective used to describe an object) that can be iterated over
Iterator	The object that produces successive items or values from its associated iterable
iter()	The built-in function used to obtain an iterator from an iterable

Table 6.1.

Now, consider again the simple for loop presented at the start of this tutorial:

```
a = ['foo', 'bar', 'baz']
for i in a:
    print(i)
```


```
a = ['foo', 'bar', 'baz']
for i in a:
    print(i)
```


The loop body is executed once for each item next() returns, with loop variable i set to the given item for each iteration.
Python treats looping over all iterables in exactly this way, and in Python, iterables and iterators abound:
Many built-in and library objects are iterable.
There is a Standard Library module called itertools containing many functions that return iterables.
User-defined objects created with Python's object-oriented capability can be made to be iterable.
Python features a construct called a generator that allows you to create your own iterator in a simple, straightforward way.
You will discover more about all the above throughout this series. They can all be the target of a for loop, and the syntax is the same across the board. It's elegant in its simplicity and eminently versatile.

6.5 Iterating Through a Dictionary

You saw earlier that an iterator can be obtained from a dictionary with iter(), so you know dictionaries must be iterable. What happens when you loop through a dictionary? Let's see:

```
d = {'foo': 1, 'bar': 2, 'baz': 3}
for k in d:
    print(k)
```

As you can see, when a for loop iterates through a dictionary, the loop variable is assigned to the dictionary's keys.

To access the dictionary values within the loop, you can make a dictionary reference using the key as usual:

```
for k in d:
    print(d[k])
```

You can also iterate through a dictionary's values directly by using .values():

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

In fact, you can iterate through both the keys and values of a dictionary simultaneously. That is because the loop variable of a for loop isn't limited to just a single variable. It can also be a tuple, in which case the assignments are made from the items in the iterable using packing and unpacking, just as with an assignment statement:

```
i, j = (1, 2)
print(i, j)


for i, j in [(1, 2), (3, 4), (5, 6)]:
    print(i, j)
```

As noted in the tutorial on Python dictionaries, the dictionary method .items() effectively returns a list of key/value pairs as tuples:

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

d.items()
```

Thus, the Pythonic way to iterate through a dictionary accessing both the keys and values looks like this:

```
d = {'foo': 1, 'bar': 2, 'baz': 3}
for k, v in d.items():
    print('k =', k, ', v =', v)
```

6.6 The range() Function

In the first section of this tutorial, you saw a type of for loop called a numeric range loop, in which starting and ending numeric values are specified. Although this form of for loop isn't directly built into Python, it is easily arrived at.

For example, if you wanted to iterate through the values from 0 to 4, you could simply do this:

```
for n in (0, 1, 2, 3, 4):
    print(n)
```

This solution isn't too bad when there are just a few numbers. But if the number range were much larger, it would become tedious pretty quickly.

Happily, Python provides a better option—the built-in range() function, which returns an iterable that yields a sequence of integers.

range(<end>) returns an iterable that yields integers starting with 0, up to but not including <end>:

```
x = range(5)
x

type(x)
```

Note that range() returns an object of class range, not a list or tuple of the values. Because a range object is an iterable, you can obtain the values by iterating over them with a for loop:

```
for n in x:
    print(n)
```

6.7 Altering for loop Behaviour

while loop can be interrupted with break and continue statements and modified with an else clause. These capabilities are available with the for loop as well.

6.7.1 Break and Continue Statements

break and continue work the same way with for loops as with while loops. break terminates the loop completely and proceeds to the first statement following the loop:

```
for i in ['foo', 'bar', 'baz', 'qux']:
    if 'b' in i:
        break
    print(i)
```

continue terminates the current iteration and proceeds to the next iteration:

```
for i in ['foo', 'bar', 'baz', 'qux']:
    if 'b' in i:
        continue
    print(i)
```

6.7.2 Else Clause

A for loop can have an else clause as well. The interpretation is analogous to that of a while loop. The else clause will be executed if the loop terminates through exhaustion of the iterable:

```
for i in ['foo', 'bar', 'baz', 'qux']:
    print(i)
else:
    print('Done.')  # Will execute
```

The else clause won't be executed if the list is broken out of with a break statement:

```
for i in ['foo', 'bar', 'baz', 'qux']:
    if i == 'bar':
        break
    print(i)
else:
    print('Done.')  # Will not execute
```

Chapter 7

Functions

Table of Contents

List of Examples

7.1 Introduction

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

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

where, f is a function operating on inputs x and y. The output of the function is z. However, programming functions are much more generalised and versatile compared to its mathematical counterpart. In fact, appropriate function definition and use is so critical to proper soft-

ware development that virtually all modern programming languages support both built-in and user-defined functions.

In programming, a function is defined as:

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

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

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

C.R. 1
```python
s = 'foobar'
id(s)
```

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

C.R. 2
```python
a = ['foo', 'bar', 'baz', 'qux']
print(len(a))
```

```text
4
```

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

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

```text
False
True
False
True
```

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

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

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

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

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

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

7.2 Importance of Python Functions

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

- Subroutines
- Procedures
- Methods
- Subprograms

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

7.2.1 Abstraction and Re-usability

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

What should we do?

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

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

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

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

anywhere the function is called.

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

This is arguably the strongest motivation for using functions.

7.2.2 Modularity

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

C.R. 4 python

```
# Main program

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

# Code to process file
<statement>
<statement>

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

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

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

C.R. 5 python

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

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

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

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

C.R. 6
```python
write_file()
```

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

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

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

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

7.2.3 Namespace Separation

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

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

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

7.3 Function Calls and Definition

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

C.R. 7
```python
def <function_name>([<parameters>]):
    <statement(s)>
```

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

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

The syntax for calling a Python function is as follows:

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

Table 7.1.: The components of what makes a function.

C.R. 8
```python
<function_name>([<arguments>])
```

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

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

C.R. 9
```python
def f():
    s = '-- Inside f()'
    print(s)

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

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

Here's how this code works:

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

proceeds to `f()` and the statements in the body of `f()` are executed.

6. Here execute once the body of `f()` has finished. Execution returns to this `print()` statement.

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

C.R. 10 python
```python
def f():
    pass

f()
```

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

Sum of Two Numbers 1 **Example**

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

Solution

C.R. 11 python
```python
def sum_values(a, b):
    result = a + b
    return result

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

Maximum of Three Numbers 2 **Example**

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

Solution

C.R. 12 python
```python
# Define a function that returns the maximum of two numbers
def max_of_two(x, y):
    # Check if x is greater than y
    if x > y:
        # If x is greater, return x
        return x
    # If y is greater or equal to x, return y
    return y

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

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

Example **Sum of a List** 3

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

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

Solution

C.R. 13 python

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

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

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

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

Example **Multiply Values in a List** 4

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

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

Solution

C.R. 14 python

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

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

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

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

Example **Printing a Sentence Multiple Times** 5

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

Solution

C.R. 15 python

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


print_sentence_multiple_times()
```

Functions to make a list 6 Example

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

Solution

C.R. 16 python

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

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

Reversing a List 7 Example

Write a Python program to reverse a string.

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

Solution

C.R. 17 python

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

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

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

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

C.R. 18 python

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

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

Example Counting Cases 8

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

- Sample String : 'The quick Brow Fox'
- Expected Output :
- No. of Upper case characters : 3
- No. of Lower case Characters : 12

Solution

C.R. 19 python

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

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

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

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

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

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

7.4 Argument Passing

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

7.4.1 Positional Arguments

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

C.R. 20 python
```python
def f(qty, item, price):
    print(f'{qty} {item} cost ${price:.2f}')
```

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

C.R. 21 python
```python
f(6, 'bananas', 1.74)
```

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

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

Parameter	Argument
qty	6
item	bananas
price	1.74

Table 7.2.: A closer look at the parameters.

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

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

C.R. 22 python
```python
f('bananas', 1.74, 6)
```

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

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

C.R. 23
```python
# Too few arguments and would give an error
f(6, 'bananas')
```

We can't also specify extra ones:

C.R. 24
```python
# Too many arguments
f(6, 'bananas', 1.74, 'kumquats')
```

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

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

7.4.2 Keyword Arguments

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

C.R. 25
```python
f(qty=6, item='bananas', price=1.74)
```

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

C.R. 26
```python
f(qty=6, item='bananas', cost=1.74)
```

> Using keyword arguments lifts the restriction on argument order.

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

C.R. 27
```python
f(item='bananas', price=1.74, qty=6)
```

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

C.R. 28
```python
# Still too few arguments
f(qty=6, item='bananas')
```

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

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

C.R. 29
```python
print(f(6, price=1.74, item='bananas'))
print(f(6, 'bananas', price=1.74))
```

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

C.R. 30
```python
f(6, item='bananas', 1.74)
```

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

7.4.3 Default Parameters

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

C.R. 31
```python
def f(qty=6, item='bananas', price=1.74):
    print(f'{qty} {item} cost ${price:.2f}')
```

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

C.R. 32
```python
print(f(4, 'apples', 2.24))
print(f(4, 'apples'))
print(f(4))
print(f())
print(f(item='kumquats', qty=9))
print(f(price=2.29))
```

7.4.4 Mutable Default Parameter Values

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

mutable is the ability of objects to change their values

C.R. 33
```python
def f(my_list=[]):
    my_list.append('###')
    return my_list
```

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

C.R. 34
```python
print(f(['foo', 'bar', 'baz']))
print(f([1, 2, 3, 4, 5]))
```

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

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

C.R. 35
```python
print(f())
```

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

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

C.R. 36
```python
print(f())
print(f())
```

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

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

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

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

C.R. 37
```python
def f(my_list=[]):
    print(id(my_list))
    my_list.append('###')
    return my_list

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

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

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

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

C.R. 38

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

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

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

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

Celcius to Fahrenheit 9 **Example**

Write a program which converts celcius to fahrenheit.

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

Solution

C.R. 39

```python
# creating a function that converts the given celsius degree temperature
# to Fahrenheit degree temperature
def convertCelsiustoFahrenheit(c):
   # converting celsius degree temperature to Fahrenheit degree temperature
```

C.R. 40 python

```
    f = (9/5)*c + 32
    # returning Fahrenheit degree temperature of given celsius temperature
    return (f)
# input celsius degree temperature
celsius_temp = 80
print("The input Temperature in Celsius is ",celsius_temp)
# calling convertCelsiustoFahrenheit() function by passing
# the input celsius as an argument
fahrenheit_temp = convertCelsiustoFahrenheit(celsius_temp)
# printing the Fahrenheit equivalent of the given celsius degree temperature
print("The Fahrenheit equivalent of input celsius degree = ", fahrenheit_temp)
```

7.5 The return Statement

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

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

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

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

7.5.1 Exiting a Function

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

C.R. 41 python

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

print(f())
```

text

```
foo
bar
None
```

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

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

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

C.R. 42
python
```
def f(x):
    if x < 0:
        return
    if x > 100:
        return
    print(x)

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

text
```
None
None
64
None
```

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

C.R. 43
python
```
def f():
    if error_cond1:
        return
    if error_cond2:
        return
    if error_cond3:
        return

    <normal processing>
```

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

7.5.2 Returning Data to the Caller

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

C.R. 44
python
```
def f():
    return 'foo'

s = f()
s
```

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

Here, the value of expression `f()` is `foo`, which is subsequently assigned to variable `s`.
A function can return any type of object. In Python, that means pretty much anything whatsoever. In the calling environment, the function call can be used syntactically in any way that makes sense for the type of object the function returns.
For example, in the code below, `f()` returns a dictionary. In the calling environment then, the expression `f()` represents a dictionary, and `f()['baz']` is a valid key reference into that dictionary:

C.R. 45 python
```python
def f():
    return dict(foo=1, bar=2, baz=3)

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

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

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

C.R. 46 python
```python
def f():
    return 'foobar'

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

text
```
ob
```

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

C.R. 47 python
```python
def f():
    return ['foo', 'bar', 'baz', 'qux']


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

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

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

C.R. 48 python
```python
def f():
    return 'foo', 'bar', 'baz', 'qux'

print(type(f()))

```

C.R. 49
python
```
t = f()
print(t)

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

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

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

C.R. 50
python
```
def f():
    return

print(f())
```

text
```
None
```

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

C.R. 51
python
```
def g():
    pass

print(g())
```

text
```
None
```

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

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

C.R. 52
python
```
def f():
    return

def g():
    pass

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

text
```
no
```

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

7.6 Variable-Length Argument Lists

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

C.R. 53 python
```python
def avg(a, b, c):
    return (a + b + c) / 3
```

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

C.R. 54 python
```python
print(avg(1, 2, 3))
```

```text
2.0
```

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

C.R. 55 python
```python
avg(1, 2, 3, 4)
```

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

C.R. 56 python
```python
def avg(a, b=0, c=0, d=0, e=0):
    .
    .
    .
```

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

C.R. 57 python
```python
avg(1)
avg(1, 2)
avg(1, 2, 3)
avg(1, 2, 3, 4)
avg(1, 2, 3, 4, 5)
```

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

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

C.R. 58
```python
def avg(a, b=0, c=0, d=0, e=0):
    return (a + b + c + d + e) / # Divided by what???
```

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

C.R. 59
```python
def avg(a):
    total = 0
    for v in a:
            total += v
    return total / len(a)


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

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

C.R. 60
```python
t = (1, 2, 3, 4, 5)
print(avg(t))
```

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

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

7.6.1 Argument Tuple Packing

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

Here's an example:

C.R. 61
```python
def f(*args):
    print(args)
    print(type(args), len(args))
    for x in args:
            print(x)

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

```text
(1, 2, 3)
<class 'tuple'> 3
```

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

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

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

C.R. 62 python
```python
def avg(*args):
    total = 0
    for i in args:
        total += i
    return total / len(args)

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

C.R. 63 python
```python
def avg(*args):
    total = 0
    for i in args:
        total += i
    return total / len(args)

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

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

C.R. 64 python
```python
def avg(*args):
    return sum(args) / len(args)

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

C.R. 65 python
```python
def avg(*args):
    return sum(args) / len(args)

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

C.R. 66
```python
print(avg(1, 2, 3, 4, 5))
```

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

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

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

C.R. 67
```python
avg(1, 'foo', 3)
```

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

7.6.2 Argument Tuple Unpacking

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

C.R. 68
```python
def f(x, y, z):
    print(f'x = {x}')
    print(f'y = {y}')
    print(f'z = {z}')

print(f(1, 2, 3))

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

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

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

C.R. 69
```python
a = ['foo', 'bar', 'baz']

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

C.R. 70
python
```
s = {1, 2, 3}
print(type(s))
print(f(*s))
```

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

7.6.3 Argument Dictionary Unpacking

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

C.R. 71
python
```
def f(a, b, c):
    print(F'a = {a}')
    print(F'b = {b}')
    print(F'c = {c}')

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

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

C.R. 72
python
```
f(a='foo', b=25, c='qux')
```

In fact, we can see it in the code below

C.R. 73
python
```
f(**dict(a='foo', b=25, c='qux'))
```

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

7.6.4 Putting it all together

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

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

C.R. 74

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

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

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

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

Chapter 8

Object Oriented Programming

Table of Contents

8.1 Introduction

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

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

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

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

8.2 Defining Object Oriented Programming

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

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

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

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

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

8.3 Defining a Class

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

C.R. 1

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

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

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

What if we want to represent something more complex?

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

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

C.R. 2
python
```
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

There are a number of issues with this approach.

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

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

8.4 Classes v. Instances

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

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

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

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

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

8.5 Class Definition

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

Here's an example of a Dog class:

C.R. 3
```python
class Dog:
    pass
```

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

the `pass` keyword.

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

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

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

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

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

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

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

The Role of `.__init__()`

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

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

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

the instance to the self parameter in `.__init__()` so that Python can define the new attributes on the object.

The role of `self`

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

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

C.R. 4

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

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

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

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

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

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

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

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

C.R. 5

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

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

C.R. 6
```python
        self.age = age
```

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

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

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

Creating a Class with no Attributes 1 **Example**

Create a Vehicle class without any variables and methods.

Solution

Creating a Class with no Attributes

C.R. 7
```python
class Vehicle:
    pass
```

8.6 To Instance a Class

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

C.R. 8
```python
class Dog:
    pass

print(Dog())
```

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

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

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

> The address on your screen will be different.

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

C.R. 9
```python
print(Dog())
```

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

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

To see this another way, type the following:

C.R. 10
```python
a = Dog()
b = Dog()
print(a == b)
```

```
False
```
text

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

Example **Creating a Class** 2

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

Solution

Creating a Class

C.R. 11
```python
class Vehicle:
    def __init__(self, max_speed, mileage):
        self.max_speed = max_speed
        self.mileage = mileage

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

8.7 Class and Instance Attributes

C.R. 12
```python
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

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

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

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

C.R. 13
```python
miles = Dog("Miles", 4)
buddy = Dog("Buddy", 9)
```

This creates two (2) new Dog instances:

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

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

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

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

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

C.R. 14

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

```text
Miles
4
Buddy
9
```

You can access class attributes the same way:

C.R. 15

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

```text
Canis familiaris
```

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

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

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

C.R. 16

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

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

```text
10
Felis silvestris
```

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

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

8.7.1 Instance Methods

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

C.R. 17 python

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

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

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

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

This Dog class has two (2) instance methods:

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

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

C.R. 18 python

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

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

text

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

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

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

C.R. 19
```python
names = ["Miles", "Buddy", "Jack"]
print(names)
```

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

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

C.R. 20
```python
print(miles)
```

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

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

C.R. 21
```python
class Dog:
    species = "Canis familiaris"

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

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

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

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

C.R. 22
```python
miles = Dog("Miles", 4)
print(miles)
```

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

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

8.8 Inheritance

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

from are called parent classes.

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

C.R. 23
```python
class Parent:
    hair_color = "brown"

class Child(Parent):
    pass
```

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

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

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

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

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

C.R. 24
```python
class Parent:
    hair_color = "brown"

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

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

C.R. 25
```python
class Parent:
    speaks = ["English"]

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

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

The `super()` Function

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

Class Inheritance 3 **Example**

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

C.R. 26

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

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

Solution

Class Inheritance

C.R. 27

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

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

8.9 Parent Classes v. Child Classes

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

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

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

C.R. 28
```python
class Dog:
    species = "Canis familiaris"

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

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

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

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

C.R. 29
```python
class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

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

C.R. 30
```python
miles = JackRussellTerrier("Miles", 4)
buddy = Dachshund("Buddy", 9)
jack = Bulldog("Jack", 3)
jim = Bulldog("Jim", 5)
```

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

C.R. 31
```python
miles.species
buddy.name
print(jack)
jim.speak("Woof")
```

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

```
type(miles)
```

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

C.R. 32
```python
print(isinstance(miles, Dog))
```

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

```text
True
```

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

C.R. 33
```python
print(isinstance(miles, Bulldog))
print(isinstance(jack, Dachshund))
```

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

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

Creating a Child Class 4 **Example**

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

C.R. 34
```python
class Vehicle:

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

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

Solution

Creating a Child Class

C.R. 35
```python
class Vehicle:

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

class Bus(Vehicle):
    pass

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

8.9.1 Extending Parent Class Functionality

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

`.speak()` in the class definition for each breed.

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

C.R. 36 python
```
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return f"{self.name} says {sound}"
```

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

C.R. 37 python
```
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

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

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

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

```
# dog.py

class Dog:
    # ...

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

# ...
```

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

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

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

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

Sometimes it makes sense to completely override a method from a parent class. But in this case, you don't want the JackRussellTerrier class to lose any changes that you might make to the formatting of the Dog.speak() output string.

To do this, you still need to define a `.speak()` method on the child JackRussellTerrier class. But instead of explicitly defining the output string, you need to call the Dog class's .speak() from inside the child class's `.speak()` using the same arguments that you passed to JackRussellTerrier.speak().

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

```
# dog.py

# ...

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

# ...
```

When you call super().speak(sound) inside JackRussellTerrier, Python searches the parent class, Dog, for a `.speak()` method and calls it with the variable sound.

Update dog.py with the new JackRussellTerrier class. Save the file and press F5 so you can test it in the interactive window:

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

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

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

8.10 The classmethod() function

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

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

8.10.1 Class v. Static Method

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

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

Example **An Example of @classmethod** 5

Create a simple classmethod

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

C.R. 38 python

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

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

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

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

#+end_src

** Practicals

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

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

C.R. 39

```python
    @classmethod
```

Solution

An Example of @classmethod

Part II.

Python For Applications

Chapter 9

Numpy

Table of Contents

Figure 9.1.: The logo of numpy.

9.1 Introduction

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

the n-dimensional array.

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

9.2 Why use Numpy

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

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

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

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

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

9.3 An Introductionary Example - Calculating Grades

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

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

These concepts are the core of using NumPy effectively.
Imagine a scenario: You're a teacher who has just graded your students on a recent test. Unfortunately, you may have made the test too challenging, and most of the students did worse than expected. To help everybody out, you're going to curve everyone's grades[2].

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

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

C.R. 1
```python
import numpy as np
CURVE_CENTER = 80
grades = np.array([72, 35, 64, 88, 51, 90, 74, 12])
def curve(grades):
    average = grades.mean()
```

C.R. 2

```python
    change = CURVE_CENTER - average
    new_grades = grades + change
    return np.clip(new_grades, grades, 100)

curve(grades)
```

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

Here are the important highlights:

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

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

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

1. **Vectorization:** process of performing the same operation in the same way for each element in an array. This removes for loops from your code but achieves the same result.

2. **Broadcasting:** process of extending two arrays of different shapes and figuring out how to perform a vectorized calculation between them

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

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

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

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

9.4 Getting Into Shape: Array Shapes and Axes

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

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

9.4.1 Understanding Shapes

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

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

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

C.R. 3 python

```python
import numpy as np

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

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

text

```
(2, 2, 3)

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

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

```
   [38.6 22.2]]]
```

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

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

9.4.2 Understanding Axes

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

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

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

C.R. 4 python

```python
import numpy as np

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

print(table.max())

print(table.max(axis=0))

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

text

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

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

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

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

9.4.3 Broadcasting

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

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

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

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

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

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

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

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

C.R. 5

```python
import numpy as np

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

print(A)

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

print(B)
```

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

```
   [40 41 42 43 44 45 46 47]]]
```

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

Add the two (2) arrays together:

C.R. 6

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

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

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

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

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

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

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

9.5 Data Science Operations: Filter, Order, Aggregate

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

filtering, sorting, and aggregating data.

9.5.1 Indexing

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

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

Dürer's magic square

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

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

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

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

Let's prove this statement:

C.R. 7 python

```python
import numpy as np

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

for i in range(4):
```

C.R. 8
python
```
    assert square[:, i].sum() == 34
    assert square[i, :].sum() == 34


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

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

The keyword assert

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

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

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

9.6 Masking and Filtering

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

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

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

C.R. 9
python
```
import numpy as np

numbers = np.linspace(5, 50, 24, dtype=int).reshape(4, -1)

print(numbers)

```

C.R. 10

```python
mask = numbers % 4 == 0

print(mask)

numbers[mask]

by_four = numbers[numbers % 4 == 0]

print(by_four)
```

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

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

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

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

- Line 7 creates the mask by performing a vectorized Boolean computation, taking each element and checking to see if it divides evenly by four. This returns a mask array of the same shape with the element-wise results of the computation.
- Line 13 uses this mask to index into the original numbers array. This causes the array to lose its original shape, reducing it to one dimension, but you still get the data you're looking for.
- Line 13 provides a more traditional, idiomatic masked selection that you might see in the wild, with an anonymous filtering array created inline, inside the selection brackets.

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

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

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

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

Because of the particular calculation in this example, it makes life easier to have integers in the numbers array. But because the space between 5 and 50 doesn't divide evenly by 24, the

resulting numbers would be floating-point numbers. You specify a dtype of int to force the function to round down and give you whole integers. You'll see a more detailed discussion of data types later on.

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

A Simple Filter Exercise 1 **Example**

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

Solution

C.R. 11

python

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

# Creating a NumPy array 'nums' containing values in a 3x3 matrix
nums = np.array([[5.54, 3.38, 7.99],
                 [3.54, 4.38, 6.99],
                 [1.54, 2.39, 9.29]])

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

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

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

text

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

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

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

C.R. 12
python
```python
import numpy as np

from numpy.random import default_rng

rng = default_rng()

values = rng.standard_normal(10000)

print(values[:5])

std = values.std()

print(std)

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

print(filtered.size)

print(values.size)

filtered.size / values.size
```

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

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

Example **Creating a Numpy Array** 2

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

Solution

C.R. 13
python
```python
# Importing the NumPy library with an alias 'np'
import numpy as np

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

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

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

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

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

An Identity Matrix 3 Example

Write a 3-by-3 identity matrix.

Solution

C.R. 14

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

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

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

# Printing the 3x3 identity matrix
print(array_2D)

```

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

A Random Value 4 Example

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

Solution

C.R. 15

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

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

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

# Printing the generated random number
print(rand_num)

```

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

text

Example **A Random Array** 5

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

Solution

C.R. 16

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

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

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

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

```

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

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

text

9.7 Transposing, Sorting, and Concatenating

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

Here's transposing an array:

C.R. 17

```python
import numpy as np

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

print(a.T)

print(a.transpose())
```

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

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

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

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

C.R. 18

```python
import numpy as np

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

print(np.sort(data))

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

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

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

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

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

C.R. 19

```python
import numpy as np

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

C.R. 20 python

```python
])

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

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

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

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

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

text

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

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

Example **Sum of an Array** 6

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

Solution

C.R. 21 python

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

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

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

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

C.R. 22

```python
print(np.sum(x))

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

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

```

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

9.8 Aggregating

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

9.9 Practical Exercise Implementing Maclauring Series

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

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

Using NumPy allows you to keep closer to a one-to-one representation from equation to code. In this next example, you'll encode the Maclaurin series[6] for ex. Maclaurin series are a way of approximating more complicated functions with an infinite series of summed terms centered about zero.

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

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

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

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

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

C.R. 23
python
```
from math import e, factorial

import numpy as np

fac = np.vectorize(factorial)

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

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

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

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

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

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

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

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

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

C.R. 24
python
```
n = np.arange(terms)
return np.sum((x ** n) / fac(n))
```

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

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

9.10 Optimizing Storage: Data Types

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

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

9.11 Numerical Types

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

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

Table 9.1.: The data types supported by Python and Numpy.

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

Find the Missing Values 7 **Example**

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

Solution

C.R. 25 python

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

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

# Printing a message indicating the original array 'nums'
```

C.R. 26 python

```python
print("Original array:")
print(nums)

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

text

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

Example **Array Value Parity** 8

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

Solution

C.R. 27 python

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

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

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

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

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

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

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

```

C.R. 28 python

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

text

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

Chapter 10

Matplotlib

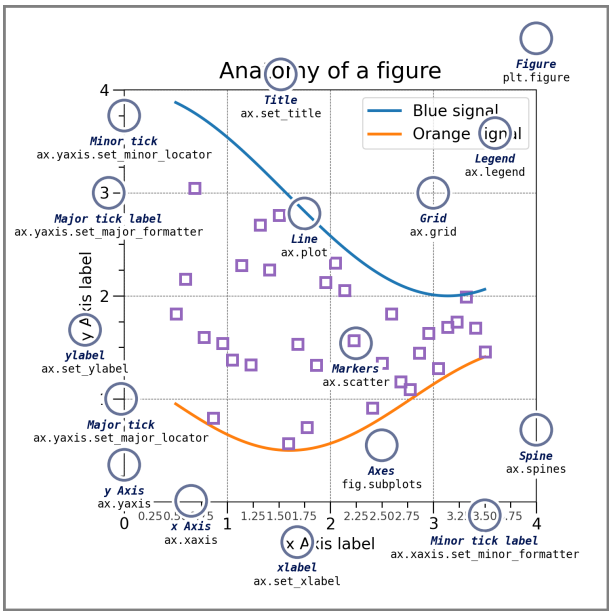
Table of Contents

10.1 Introduction

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

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

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



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

C.R. 1
```python
import matplotlib
import matplotlib.pyplot as plt
```

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

C.R. 2
```python
import numpy as np
```

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

C.R. 3
```python
x = np.linspace(0, 5, 10)
y = x ** 2
```

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

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

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

C.R. 4

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

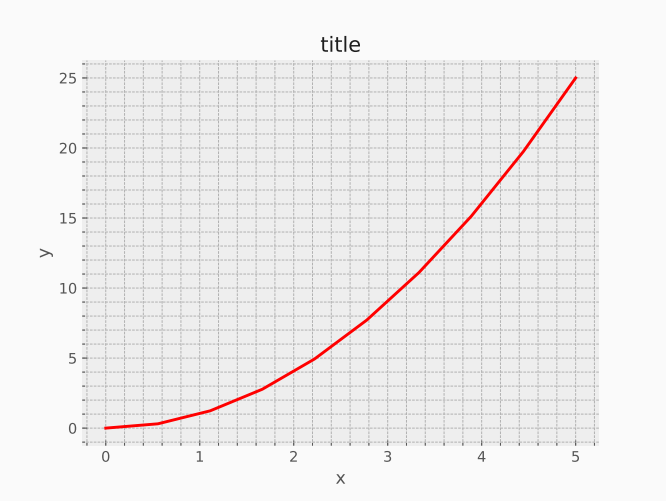


Figure 10.1.

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

C.R. 5

```python
fig = plt.figure()

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

# main figure
axes1.plot(x, y, 'r')
axes1.set_xlabel('x')
axes1.set_ylabel('y')
axes1.set_title('Primary Plot')

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

# save figure and close
```

C.R. 6
```python
plt.savefig("images/Matplotlib/figure-in-a-figure.pdf")
plt.close()
```

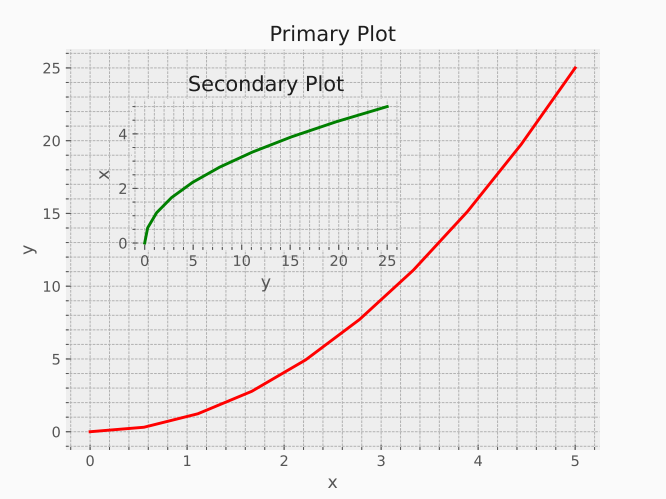


Figure 10.2.

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

C.R. 7
```python
fig, axes = plt.subplots()

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

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

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

C.R. 8
```python
fig, axes = plt.subplots(nrows=1, ncols=2)

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

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

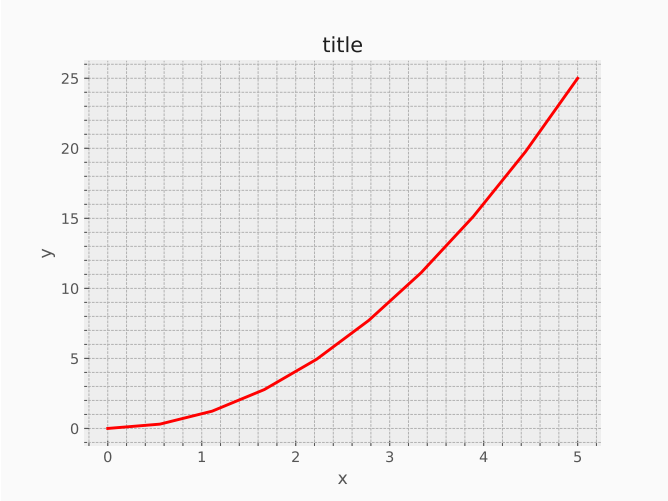


Figure 10.3.

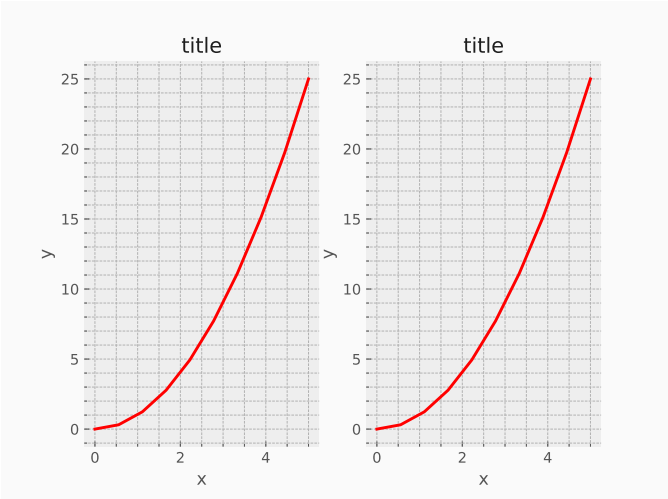


Figure 10.4.

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

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

C.R. 9

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

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

fig.tight_layout()

# save figure and close
```

C.R. 10
```python
plt.savefig("images/Matplotlib/tight-subplots-figure.pdf")
plt.close()
```

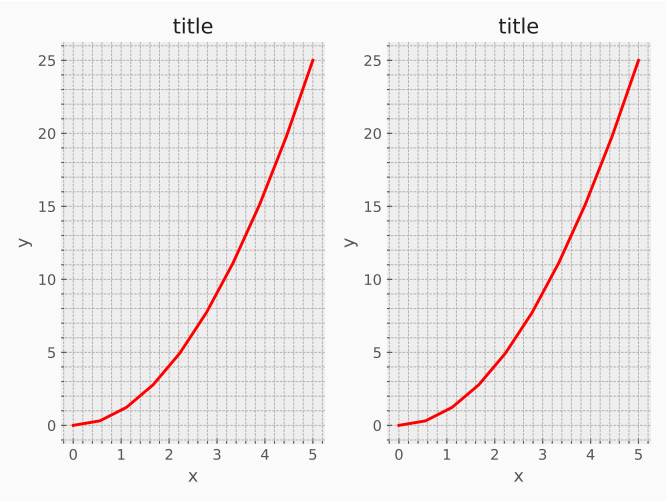


Figure 10.5.

10.1.1 Figure size, aspect ratio and DPI

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

C.R. 11
```python
fig = plt.figure(figsize=(8,4), dpi=100)
```

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

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

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

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

Saving Figures

As you have seen, the method `savefig` method is used quite frequently. This method allows you to save any figure generated by matplotlib. As it stands you can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF.

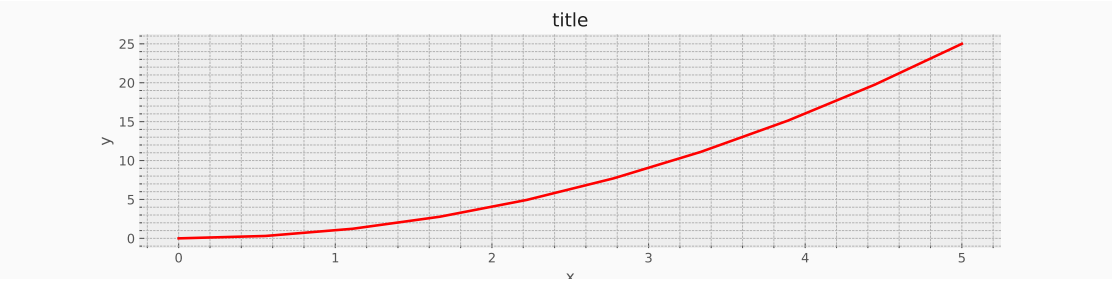


Figure 10.6.

If you are using any non-vector image formats, you can also set the DPI settings, for example

C.R. 13
```python
fig.savefig("filename.png", dpi=200)
```

10.1.2 Legends, labels and titles

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

Figure Titles

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

C.R. 14
```python
ax.set_title("title");
```

Axis Labels

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

C.R. 15
```python
ax.set_xlabel("x")
ax.set_ylabel("y")
```

Legends

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

C.R. 16
```python
ax.legend(["curve1", "curve2", "curve3"])
```

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

[1] A software used by engineers for calculations

labelled curve).

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

C.R. 17
```python
ax.plot(x, x**2, label="curve1")
ax.plot(x, x**3, label="curve2")
ax.legend();
```

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

The `legend` function takes an optional keyword argument `loc` that can be used to specify where in the figure the legend is to be drawn. The allowed values of `loc` are numerical codes for the various places the legend can be drawn. See `http://matplotlib.org/users/legend_guide.html#legend-location` for details. Some of the most common `loc` values are:

C.R. 18
```python
ax.legend(loc=0) # let matplotlib decide the optimal location
ax.legend(loc=1) # upper right corner
ax.legend(loc=2) # upper left corner
ax.legend(loc=3) # lower left corner
ax.legend(loc=4) # lower right corner
# .. many more options are available
```

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

C.R. 19
```python
fig, ax = plt.subplots()

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

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

10.1.3 Formatting text: LaTeX, fontsize, font family

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

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

Matplotlib has great support for LaTeX. All we need to do is to use dollar signs encapsulate LaTeX

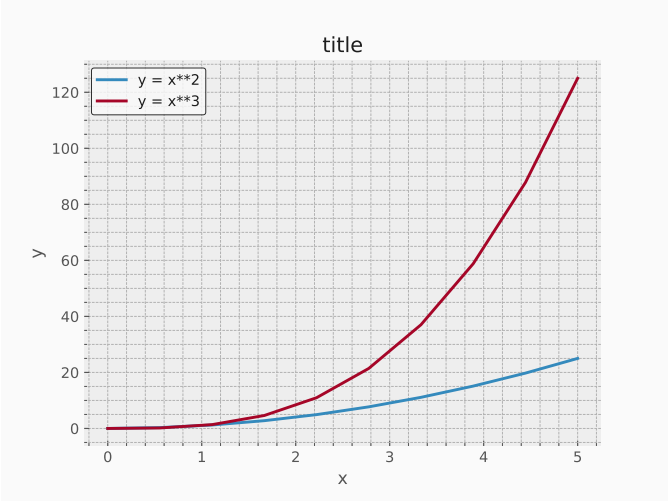


Figure 10.7.

in any text (legend, title, label, etc.).

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

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

C.R. 20
python
```
fig, ax = plt.subplots()

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

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

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

C.R. 21
python
```
matplotlib.rcParams.update({'font.size': 18, 'font.family': 'serif'})
```

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

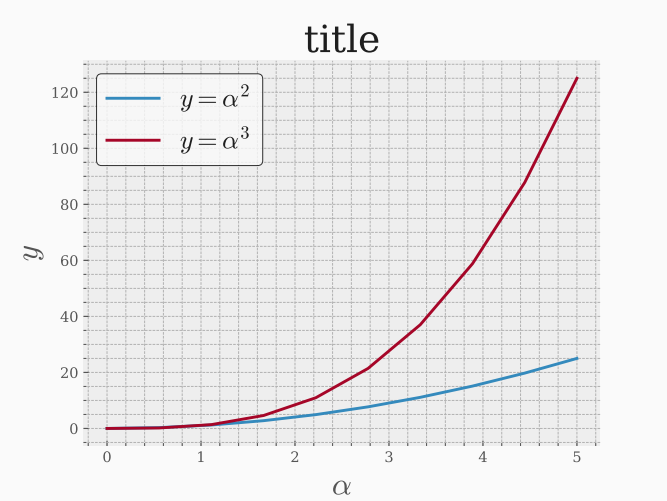


Figure 10.8.

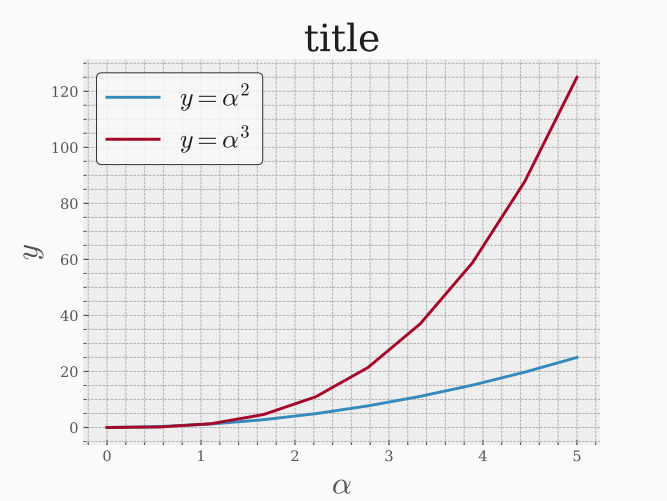


Figure 10.9.

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

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

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

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

10.1.4 Setting colors, linewidths, linetypes

Colours

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

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

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

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

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

Line and marker styles

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

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

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

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

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

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2', '3', '4',
↪  ...
ax.plot(x, x+ 9, color="green", lw=2, ls='--', marker='+')
ax.plot(x, x+10, color="green", lw=2, ls='--', marker='o')
ax.plot(x, x+11, color="green", lw=2, ls='--', marker='s')
ax.plot(x, x+12, color="green", lw=2, ls='--', marker='1')
```

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

10.1.5 Control over axis appearance

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

Plot range

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

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

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

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

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

Logarithmic scale

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

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

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

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

10.1.6 Placement of ticks and custom tick labels

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

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

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

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

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

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

Scientific notation

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

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

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

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

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

10.1.7 Axis number and axis label spacing

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

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

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

ax.set_title("label and axis spacing")

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

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

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

Axis Position Adjustement

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

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

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

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

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

10.1.8 Axis grid

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

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

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

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

10.1.9 Axis Spines

We can also change the properties of axis spines:

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

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

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

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

10.1.10 Twin Axes

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

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

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

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

10.1.11 Axes where x and y is zero

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

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

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

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

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

10.1.12 Other 2D plot styles

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

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

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

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

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

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

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

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

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

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

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

10.1.13 Text annotation

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

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

ax.plot(xx, xx**2, xx, xx**3)

ax.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="blue")
ax.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="green");
```

10.1.14 Figures with multiple subplots and insets

Axes can be added to a matplotlib Figure canvas manually using fig.add$_{\text{axes}}$ or using a sub-figure layout manager such as subplots, subplot2grid, or gridspec:

subplots

```
fig, ax = plt.subplots(2, 3)
fig.tight_layout()
```

subplot2grid

```
fig = plt.figure()
ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
ax2 = plt.subplot2grid((3,3), (1,0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1,2), rowspan=2)
ax4 = plt.subplot2grid((3,3), (2,0))
ax5 = plt.subplot2grid((3,3), (2,1))
fig.tight_layout()
```

gridspec

```
import matplotlib.gridspec as gridspec
```

```
fig = plt.figure()

gs = gridspec.GridSpec(2, 3, height_ratios=[2,1], width_ratios=[1,2,1])
for g in gs:
    ax = fig.add_subplot(g)

fig.tight_layout()
```

add_axes

Manually adding axes with add_{axes} is useful for adding insets to figures:

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

ax.plot(xx, xx**2, xx, xx**3)
fig.tight_layout()

# inset
inset_ax = fig.add_axes([0.2, 0.55, 0.35, 0.35]) # X, Y, width, height

inset_ax.plot(xx, xx**2, xx, xx**3)
inset_ax.set_title('zoom near origin')

# set axis range
inset_ax.set_xlim(-.2, .2)
inset_ax.set_ylim(-.005, .01)

# set axis tick locations
inset_ax.set_yticks([0, 0.005, 0.01])
inset_ax.set_xticks([-0.1,0,.1]);
```

10.2 Colormap and contour figures

Colormaps and contour figures are useful for plotting functions of two (2) variables. In most of these functions we will use a colormap to encode one dimension of the data. There are a number of predefined colormaps. It is relatively straightforward to define custom colormaps.

For a list of pre-defined colormaps, see: `http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps`

C.R. 22
```python
alpha = 0.7
phi_ext = 2 * np.pi * 0.5

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * np.cos(phi_p) * np.cos(phi_m) \
        - alpha * np.cos(phi_ext - 2*phi_p)
```

C.R. 23
```python
phi_m = np.linspace(0, 2*np.pi, 100)
phi_p = np.linspace(0, 2*np.pi, 100)
X,Y = np.meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
```

pcolor

C.R. 24
```python
fig, ax = plt.subplots()

p = ax.pcolor(X/(2*np.pi), Y/(2*np.pi), Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(),
↪  vmax=abs(Z).max())
cb = fig.colorbar(p, ax=ax)

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

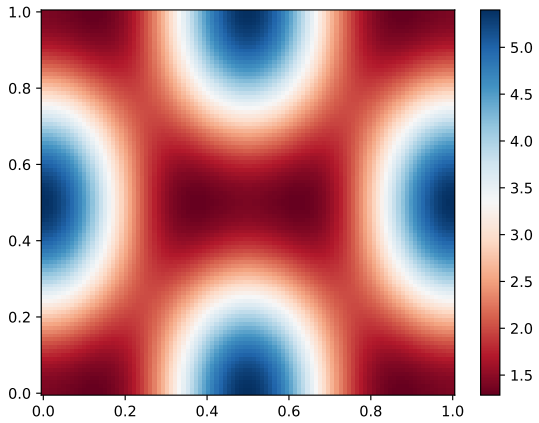


Figure 10.10.

imshow

C.R. 25
```python
fig, ax = plt.subplots()

im = ax.imshow(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(), extent=[0,
↪  1, 0, 1])
im.set_interpolation('bilinear')
```

C.R. 26
python
```

cb = fig.colorbar(im, ax=ax)

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

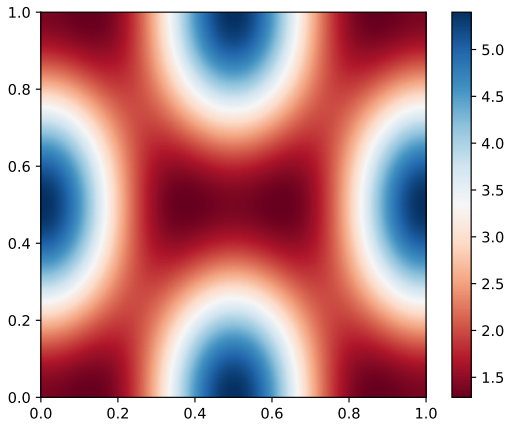


Figure 10.11.

contour

C.R. 27
python
```
fig, ax = plt.subplots()

cnt = ax.contour(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(), extent=[0,
↪  1, 0, 1])

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

10.2.1 3D Figures

To use 3D graphics in matplotlib, we first need to create an instance of the Axes3D class. 3D axes can be added to a matplotlib figure canvas in exactly the same way as 2D axes; or, more conveniently, by passing a `projection='3d'` keyword argument to the `add_axes` or `add_subplot` methods.

C.R. 28
python
```
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

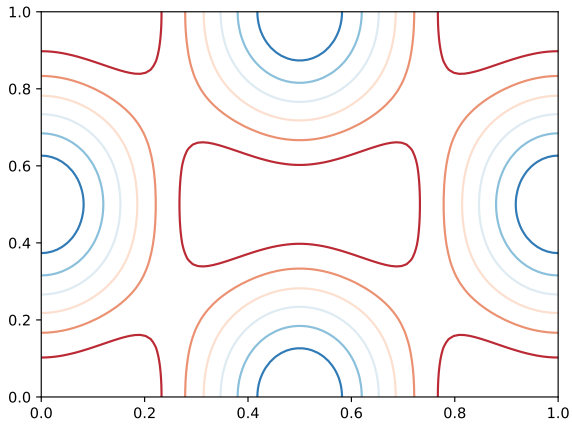


Figure 10.12.

Surface Plots

C.R. 29

```python
fig = plt.figure(figsize=(14,6))

# `ax` is a 3D-aware axis instance because of the projection='3d' keyword argument to add_subplot
ax = fig.add_subplot(1, 2, 1, projection='3d')

p = ax.plot_surface(X, Y, Z, rstride=4, cstride=4, linewidth=0)

# surface_plot with color grading and color bar
ax = fig.add_subplot(1, 2, 2, projection='3d')
p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm, linewidth=0, antialiased=False)
cb = fig.colorbar(p, shrink=0.5)

# save figure and close
plt.savefig("images/Matplotlib/3d-surface-plot.pdf")
plt.close()
```

Wire-Frame Plots

C.R. 30

```python
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1, 1, 1, projection='3d')

p = ax.plot_wireframe(X, Y, Z, rstride=4, cstride=4)

# save figure and close
plt.savefig("images/Matplotlib/3d-wire-plot.pdf")
plt.close()
```

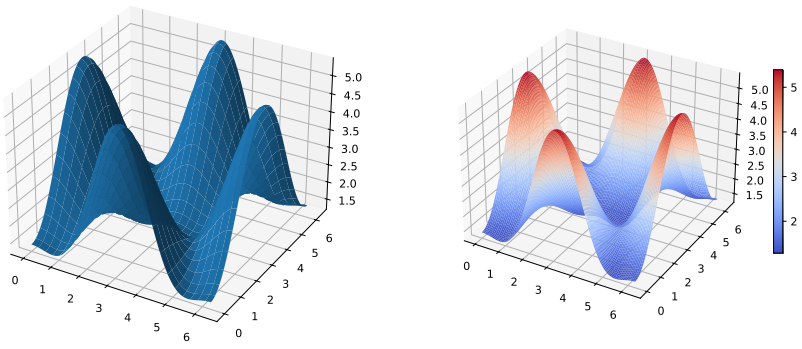


Figure 10.13.

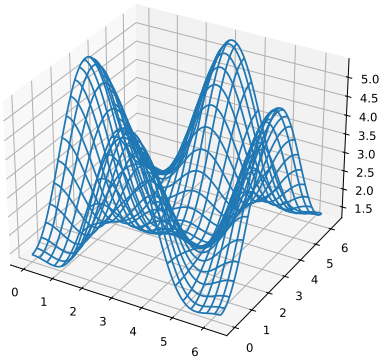


Figure 10.14.

Contour Plots with Projections

C.R. 31

```python
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*np.pi, cmap=matplotlib.cm.coolwarm)

ax.set_xlim3d(-np.pi, 2*np.pi);
ax.set_ylim3d(0, 3*np.pi);
ax.set_zlim3d(-np.pi, 2*np.pi);

# save figure and close
plt.savefig("images/Matplotlib/contour-plot-with-projections.pdf")
```

```python
plt.close()
```
C.R. 32

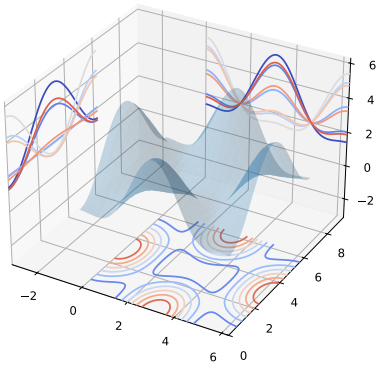


Figure 10.15.

Change the view angle

We can change 3D plot perspective using the method `view_init`, which takes two (2) arguments:

- elevation angle,
- azimuth angle.

> Both these values are in degrees and not radians.

```python
fig = plt.figure(figsize=(12,6))

ax = fig.add_subplot(1,2,1, projection='3d')
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
ax.view_init(30, 45)

ax = fig.add_subplot(1,2,2, projection='3d')
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
ax.view_init(70, 30)

fig.tight_layout()


# save figure and close
plt.savefig("images/Matplotlib/view-angle.pdf")
plt.close()
```
C.R. 33

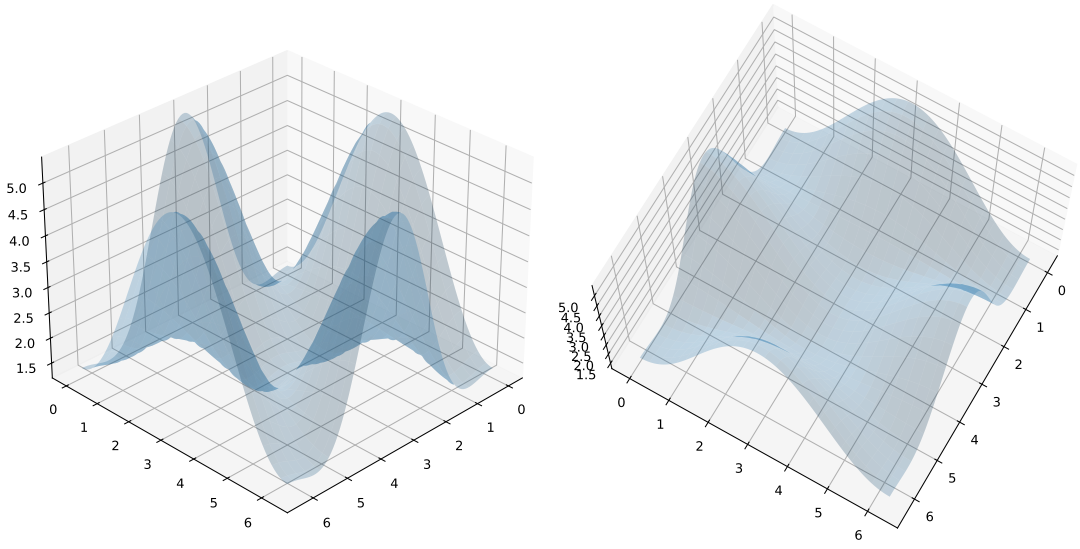


Figure 10.16.

Chapter 11

Scipy

Table of Contents

11.1 Introduction

The SciPy framework builds on top of the low-level NumPy framework for multidimensional arrays, and provides a large number of higher-level scientific algorithms. Some of the topics that SciPy covers are shown in **Table** 11.1.
Each of these submodules provides a number of functions and classes that can be used to solve problems in their respective topics. In this chapter we will look at how to use some of these subpackages. However, the reader is encouraged to look at other aspects of the package.

To access the SciPy package in a Python program, we start by importing everything from the scipy module.

C.R. 1
```python
#+NAME: PRE-1
```

Topic	Module
Special functions	`scipy.special`
Integration	`scipy.integrate`
Optimization	`scipy.optimize`
Interpolation	`scipy.interpolate`
Fourier Transforms	`scipy.fftpack`
Signal Processing	`scipy.signal`
Linear Algebra	`scipy.linalg`
Sparse Eigenvalue Problems	`scipy.sparse`
Statistics	`scipy.stats`
Multi-dimensional image processing	`scipy.ndimage`
File IO	`scipy.io`

Table 11.1.: Scipy offers a wide variety of packages the user can work with. Above are some of them but the reader is recommended to look at other packages described within the documentation.

If we only need to use part of the SciPy framework we can selectively include only those modules we are interested in. For example, to include the linear algebra package under the name `la` [1], we can do:

[1] As usual, it is up to the programmer to pick their alias, but some aliases have standard notation which is commonly accepted such as np for numpy.

C.R. 2
```python
#+NAME: PRE-2
```

11.2 Special Functions

A large number of mathematical special functions are important for many computational physics problems. SciPy provides implementations of a very extensive set of special functions. For more information on the list of functions, the reference documentation is at

> Special functions are particular mathematical functions that have more or less established names and notations due to their importance in mathematical analysis, functional analysis, geometry, physics, or other applications.
>
> The term is defined by consensus, and thus lacks a general formal definition.

To demonstrate the typical usage of special functions we will look in more detail at the Bessel functions:

11.2.1 The Bessel Function

Bessel functions, first defined by the mathematician Daniel Bernoulli and then generalised by *Friedrich Bessel*, are canonical solutions $y(x)$ of Bessel's differential equation:

$$x^2 \frac{d^2y}{dx^2} + x\frac{dy}{dx} + \left(x^2 - \alpha^2\right) y = 0$$

where for an arbitrary complex number α, represents the **order** of the Bessel function. Although α and $-\alpha$ produce the same differential equation, it is conventional to define different Bessel functions for these two values in such a way that the Bessel functions are mostly

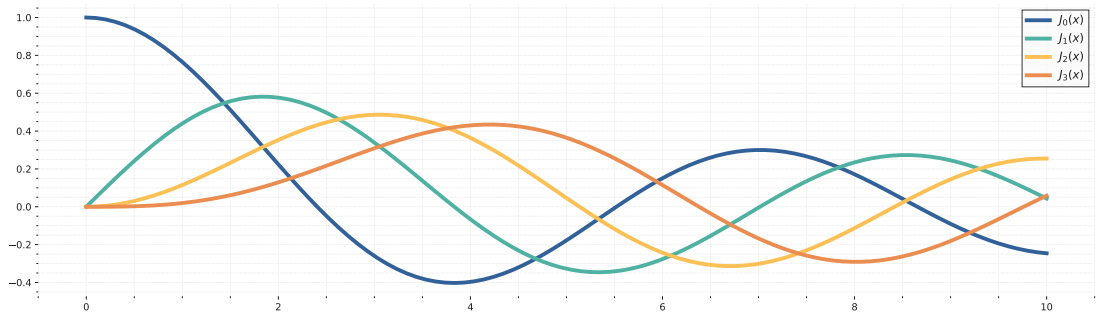


Figure 11.1.: The Bessel functions of the first kind (J_α) with different orders.

smooth functions of α. The Bessel functions of the first kind (J_α) with different orders can be seen in **Fig.** 11.1 .

The most important cases are when α is an integer or half-integer. Bessel functions for integer α are also known as cylinder functions or the cylindrical harmonics because they appear in the solution to Laplace's equation in cylindrical coordinates. Spherical Bessel functions with half-integer α are obtained when solving the Helmholtz equation in spherical coordinates.

C.R. 3 python

```
*** Bessel Function

#+NAME: SPE-1
#+begin_src python :session :results output
#
# The scipy.special module includes a large number of Bessel-functions
# Here we will use the functions jn and yn, which are the Bessel functions
# of the first and second kind and real-valued order. We also include the
# function jn_zeros and yn_zeros that gives the zeroes of the functions jn
```

C.R. 4 python

```
#+end_src

#+NAME: SPE-2
#+begin_src python :session :results output
n = 0    # order
x = 0.0

# Bessel function of first kind
print("J_%d(%f) = %f" % (n, x, jn(n, x)))
```

text

```
#+end_src


```

C.R. 5 python

```

#+NAME: SPE-3
#+begin_src python :session :results output
# zeros of Bessel functions
n = 0 # order
```

11.3 Integration

In mathematics, an integral is the continuous analog of a sum, which is used to calculate areas, volumes, and their generalizations. Integration, the process of computing an integral, is one of the two fundamental operations of calculus, the other being differentiation. Integration was initially used to solve problems in mathematics and physics, such as finding the area under a curve, or determining displacement from velocity. Usage of integration expanded to a wide variety of scientific fields thereafter.

11.3.1 Numerical Integration

Numerical integration comprises a broad family of algorithms for calculating the numerical value of a definite integral. The term numerical quadrature (often abbreviated to quadrature) is more or less a synonym for *numerical integration*, especially as applied to one-dimensional integrals. Some function have no integration (i.e, no anti-derivative) and it is necessary to approximate the results using numerical methods. The application of numerical integration is an indispensable tool for an engineer.

Numerical evaluation of a function of the type

$$\int_a^b f(x)\ \mathrm{d}x$$

is called numerical quadrature, or simply quadature. SciPy provides a series of functions for different kind of quadrature, for example the `quad`, `dblquad` and `tplquad` for single, double and triple integrals, respectively.

C.R. 6
```
1
```
python

The quad function takes a large number of optional arguments, which can be used to fine-tune the behaviour of the function [2].

[2] To get more information you can use `help(quad)`.

The basic usage is as follows. We first define a function as such:

C.R. 7
```
1 #+begin_src python :session :results output
2 # define a simple function for the integrand
3 def f(x):
```
python

Then we define our limit for the integral and invoke the `quad()` function.

C.R. 8
```
1 #+begin_src python :session :results output
2 x_lower = 0 # the lower limit of x
3 x_upper = 1 # the upper limit of x
4
5 val, abserr = quad(f, x_lower, x_upper)
6
```
python

```
1 #+begin_example
```
text

If we need to pass extra arguments to integrand function we can use the `args` keyword argument:

C.R. 9
```python
#+begin_src python :session :results output
def integrand(x, n):
    """
    Bessel function of first kind and order n.
    """
    return jn(n, x)


x_lower = 0  # the lower limit of x
x_upper = 10 # the upper limit of x

val, abserr = quad(integrand, x_lower, x_upper, args=(3,))

```

```text
#+begin_example
```

For simple functions we can use a lambda[3] function (name-less function) instead of explicitly defining a function for the integrand:

[3] A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

C.R. 10
```python
#+begin_src python :session :results output
val, abserr = quad(lambda x: np.exp(-x ** 2), -np.Inf, np.Inf)

print("numerical  =", val, abserr)

analytical = sqrt(pi)

```

```text
#+begin_example
```

As show in the example above, we can also use 'Inf' or '-Inf' as integral limits. Higher-dimensional integration works in the same way:

C.R. 11
```python
#+begin_src python :session :results output
def integrand(x, y):
    return np.exp(-x**2-y**2)

x_lower = 0
x_upper = 10
y_lower = 0
y_upper = 10

val, abserr = dblquad(integrand, x_lower, x_upper, lambda x : y_lower, lambda x: y_upper)

print(val, abserr)
```

```text
#+begin_example
```

> we had to pass lambda functions for the limits for the y integration, since these in general can be functions of x.

11.4 Ordinary Differential Equations

An ordinary differential equation (ODE) is a differential equation dependent on only a single independent variable. As with other DE, its unknown(s) consists of one (or more) function(s) and involves the derivatives of those functions.

SciPy provides two (2) different ways to solve ODEs:

1. An API based on the function `odeint`,
2. object-oriented API based on the class `ode`.

> Usually `odeint` is easier to get started with, but the `ode` class offers some finer level of control.

Here we will use the `odeint` functions. For more information about the class ode, try `help(ode)`. It does pretty much the same thing as odeint, but in an object-oriented fashion. To use `odeint`, first import it from the `scipy.integrate` module.

C.R. 12 python

```
#+begin_src python :session :results output
from scipy.integrate import odeint, ode
```

A system of ODEs are usually formulated on standard form before it is attacked numerically. The standard form is:

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

where

$$y = [y_1(t), y_2(t), \ldots, y_n(t)]$$

and f is some function that gives the derivatives of the function $y_i(t)$. To solve an ODE we need to know the function f and an initial condition $y(0)$.

> Note that higher-order ODEs can always be written in this form by introducing new variables for the intermediate derivatives

Once we have defined the Python function f and array y_0, we can use the `odeint` function as:

```
y_t = odeint(f, y_0, t)
```

where t is and array with time-coordinates for which to solve the ODE problem, `y_t` is an array with one row for each point in time in t, where each column corresponds to a solution `y_i(t)` at that point in time.

We will see how we can implement f and `y_0` in Python code in the examples below.

Let's consider a physical examples.

11.4.1 Double Pendulum

a double pendulum, also known as a chaotic pendulum, is a pendulum with another pendulum attached to its end, forming a simple physical system that exhibits rich dynamic behavior with a strong sensitivity to initial conditions. The motion of a double pendulum is governed by a pair of coupled ordinary differential equations and is chaotic. A model showcasing the setup can be seen in **Fig.** 11.1 .

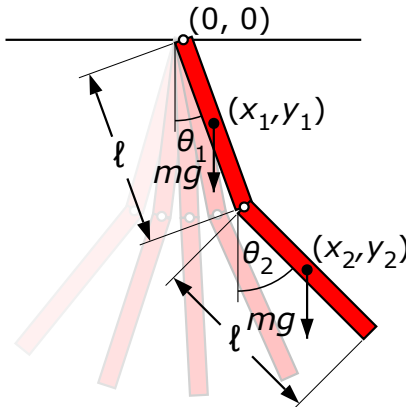


Figure 11.2.: A model of the double compound pendulum with physical description of its parameters.

We are not going to bother deriving the solution and instead use the resources on the wiki page to derive the solution:

$$
\begin{aligned}
\dot{\theta_1} &= \frac{6}{m\ell^2}\frac{2p_{\theta_1} - 3\cos(\theta_1 - \theta_2)p_{\theta_2}}{16 - 9\cos^2(\theta_1 - \theta_2)}, \\
\dot{\theta_2} &= \frac{6}{m\ell^2}\frac{8p_{\theta_2} - 3\cos(\theta_1 - \theta_2)p_{\theta_1}}{16 - 9\cos^2(\theta_1 - \theta_2)}, \\
\dot{p_{\theta_1}} &= -\frac{1}{2}m\ell^2\left[\dot{\theta_1}\dot{\theta_2}\sin(\theta_1 - \theta_2) + 3\frac{g}{\ell}\sin\theta_1\right], \\
\dot{p_{\theta_2}} &= -\frac{1}{2}m\ell^2\left[-\dot{\theta_1}\dot{\theta_2}\sin(\theta_1 - \theta_2) + \frac{g}{\ell}\sin\theta_2\right].
\end{aligned}
$$

To make the Python code simpler to follow, let's introduce new variable names and the vector notation $x = \left[\theta_1,\, \theta_2,\, p_{\theta_1},\, p_{\theta_2}\right]$. We can now write our function describing the problem.

$$
\begin{aligned}
\dot{x_1} &= \frac{6}{m\ell^2}\frac{2x_3 - 3\cos(x_1 - x_2)x_4}{16 - 9\cos^2(x_1 - x_2)}, \\
\dot{x_2} &= \frac{6}{m\ell^2}\frac{8x_4 - 3\cos(x_1 - x_2)x_3}{16 - 9\cos^2(x_1 - x_2)}, \\
\dot{x_3} &= -\frac{1}{2}m\ell^2\left[\dot{x_1}\dot{x_2}\sin(x_1 - x_2) + 3\frac{g}{\ell}\sin x_1\right], \\
\dot{x_4} &= -\frac{1}{2}m\ell^2\left[-\dot{x_1}\dot{x_2}\sin(x_1 - x_2) + \frac{g}{\ell}\sin x_2\right].
\end{aligned}
$$

C.R. 13

python

```
#+begin_src python :session :results output
g = 9.82
L = 0.5
m = 0.1
```

C.R. 14 python

```
def dx(x, t):
    """
    The right-hand side of the pendulum ODE
    """
    x1, x2, x3, x4 = x[0], x[1], x[2], x[3]

    dx1 = 6.0/(m*L**2) * (2 * x3 - 3 * np.cos(x1-x2) * x4)/(16 - 9 * np.cos(x1-x2)**2)
    dx2 = 6.0/(m*L**2) * (8 * x4 - 3 * np.cos(x1-x2) * x3)/(16 - 9 * np.cos(x1-x2)**2)
    dx3 = -0.5 * m * L**2 * ( dx1 * dx2 * np.sin(x1-x2) + 3 * (g/L) * np.sin(x1))
    dx4 = -0.5 * m * L**2 * (-dx1 * dx2 * np.sin(x1-x2) + (g/L) * np.sin(x2))

```

We then need to tell the system where we are going to start with our simulation:

C.R. 15 python

```
x0 = [np.pi/4, np.pi/2, 0, 0]
#+end_src
```

Continuing on, we need to describe the time-scale in which the simulation will take place and solve the differential equation:

C.R. 16 python

```

#+NAME: ODE-5
```

C.R. 17 python

```
#+begin_src python :session :results output
# plot the angles as a function of time
```

We then can finally plot our result and see for ourselves.

C.R. 18 python

```
y1 = - L * np.cos(x[:, 0])

x2 = x1 + L * np.sin(x[:, 1])
y2 = y1 - L * np.cos(x[:, 1])

axes[1].plot(x1, y1, label="pendulum1")
axes[1].plot(x2, y2, label="pendulum2")
axes[1].set_ylim([-1, 0])
axes[1].set_xlim([1, -1]);
#+end_src

#+NAME: ODE-P
#+begin_src python :session :results output
import matplotlib as mpl

axes[0].tick_params(axis='x', labelsize=16)
```

11.4.2 Damped Harmonic Oscillator

ODE problems are important in computational physics, so we will look at one more example: the damped harmonic oscillation. This problem is well described on the wiki page:

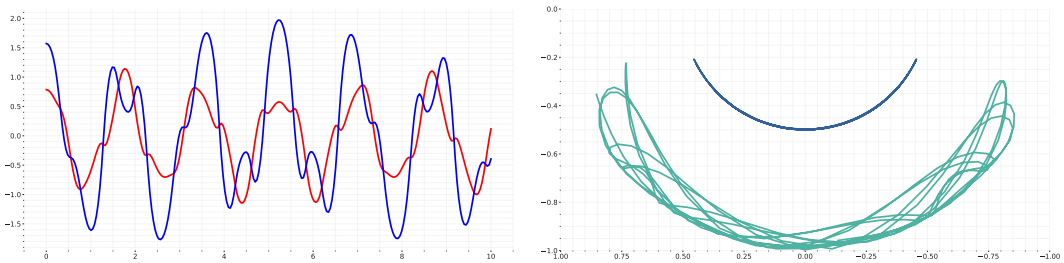


Figure 11.3.: The results of the double pendulum experiment. **(a)** The angles of θ_1 and θ_2 with respect to time **(b)** The spatial position of the pendulum as it swings around its pivot point.

The equation of motion for the damped oscillator is:

$$\frac{d^2x}{dt^2} + 2\zeta\omega_0\frac{dx}{dt} + \omega_0^2 x = 0$$

where x is the position of the oscillator, ω_0 is the frequency, and ζ is the damping ratio. To write this second-order ODE on standard form we introduce $p = dx/dt$.

$$\frac{dp}{dt} = -2\zeta\omega_0 p - \omega_0^2 x$$
$$\frac{dx}{dt} = p$$

In the implementation of this example we will add extra arguments to the right-hand side function for the ODE, rather than using global variables as we did in the previous example. As a consequence of the extra arguments to the right-hand size, we need to pass an keyword argument `args` to the `odeint` function.

We first define a function which takes the ODE defined above and convert to their state space representation:

C.R. 19 python

```
** Interpolation

#+NAME: IP-1
#+begin_src python :session :results output
from scipy.interpolate import *
#+end_src

#+NAME: IP-2
#+begin_src python :session :results output
```

We then write our initial state value for our equation.

C.R. 20 python

```
x = np.linspace(0, 9, 100)

```

We then define the time axis for our solution to be calculated in.

C.R. 21
```python
cubic_interpolation = interp1d(n, y_meas, kind='cubic')
y_interp2 = cubic_interpolation(x)
#+end_src
```

After this we calculate the solutions with different damping ratios:

C.R. 22
```python
plt.plot(x, y_interp1, 'r', label='linear interp')
plt.plot(x, y_interp2, 'g', label='cubic interp')
plt.legend(loc=3);
cp.store_fig("interpolation", close=True)
#+end_src
```

Once the calculations are done we can plot our results (we will discuss the plotting in the matplotlib section)

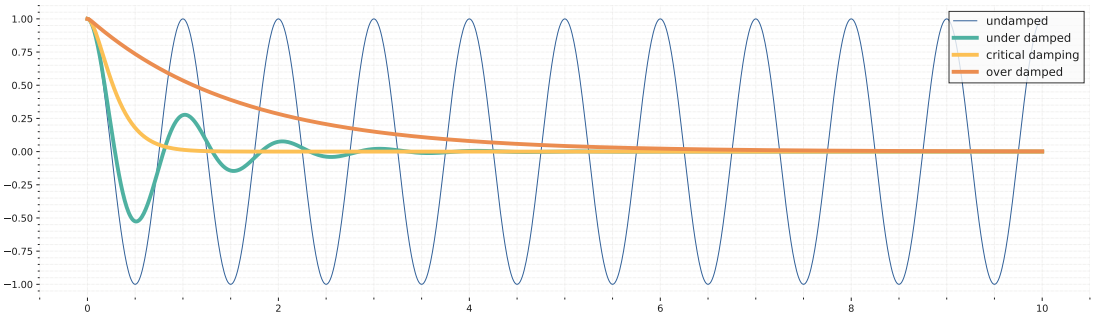


Figure 11.4.: In classical mechanics, a harmonic oscillator is a system that, when displaced from its equilibrium position, experiences a restoring force F proportional to the displacement x with k being a positive constant.

11.5 Fourier Transform

In physics, engineering and mathematics, the Fourier transform is an integral transform that takes a function as input and outputs another function that describes the extent to which various frequencies are present in the original function. The output of the transform is a complex-valued function of frequency.

The term Fourier transform refers to both this complex-valued function and the mathematical operation. When a distinction needs to be made, the output of the operation is sometimes called the frequency domain representation of the original function. The Fourier transform is analogous to decomposing the sound of a musical chord into the intensities of its constituent pitches.

Fourier transforms are one of the **universal tools** in computational physics, which appear over and over again in different contexts. SciPy provides functions for accessing the classic FFTPACK library from NetLib (a repository of software for scientific computing maintained by AT&T, Bell Laboratories, the University of Tennessee and Oak Ridge National Laboratory), which is an efficient and well tested FFT library written in Fortran[4]. The SciPy API has a few additional

[4] A 3rd generation, compiled, imperative programming language that is especially suited to numeric computation and scientific computing.

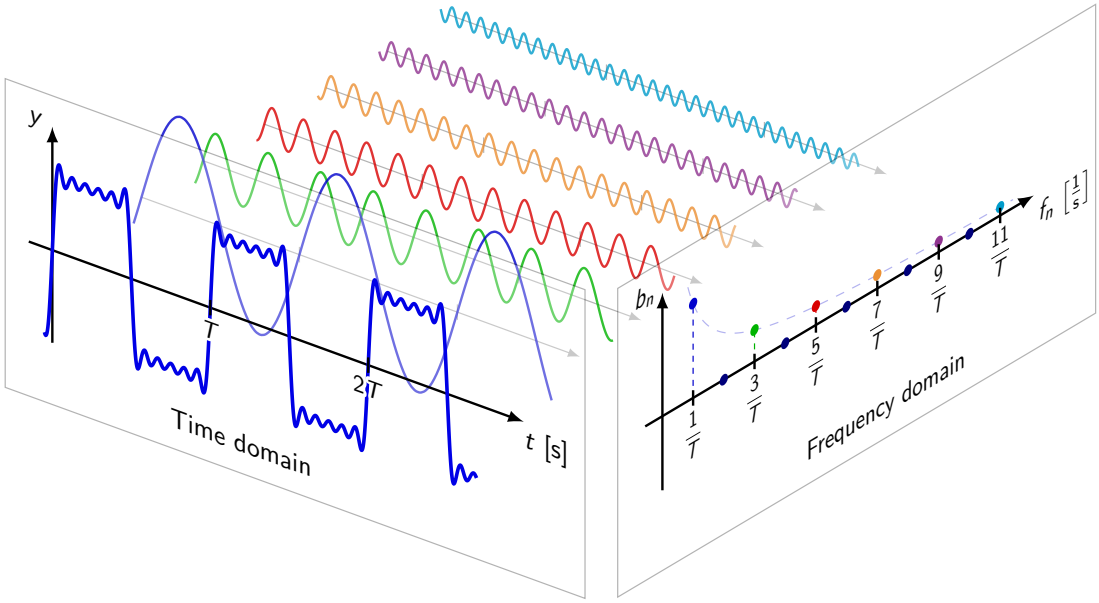


Figure 11.5.: Fourier transform can be taught of the deconstruction of a signal to their sine and cosine components with a bias term which is generally known as DC term. The transform takes a signal in its time domain and transform it to their frequency components.

convenience functions, but overall the API is closely related to the original Fortran library.

To use the `fftpack` module in a python program, include it using:

C.R. 23 python

```
#+end_src

```

To demonstrate how to do a fast Fourier transform with SciPy, let's look at the FFT[5] of the solution to the damped oscillator from the previous section:

[5] allows the decomposition of the signal to its frequency components with high speed. It has been called one of the most important numerical algorithms ever designed

C.R. 24 python

```
#+begin_src python :session :results output
# select only indices for elements that corresponds to positive frequencies
indices = np.where(w > 0)
w_pos = w[indices]
F_pos = F[indices]
#+end_src

#+NAME: FF-5
#+begin_src python :session :results output
```

Since the signal is real, the spectrum is symmetric. We therefore only need to plot the part that corresponds to the positive frequencies. To extract that part of the `w` and `F` we can use some of the indexing tricks for NumPy arrays.

C.R. 25 python

```
** Linear Algebra

```

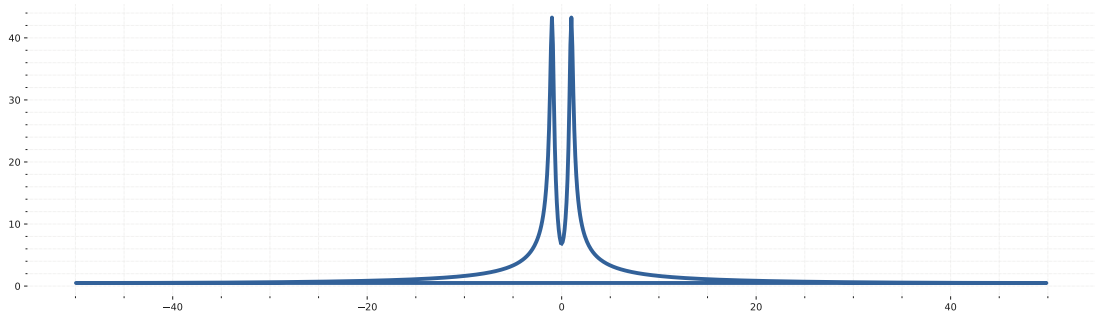


Figure 11.6.: The FFT analysis of the solution to the under-damped oscillator from the previous section.

C.R. 26 python

```
b = array([1,2,3])
#+end_src

#+NAME: LA-3
```

C.R. 27 python

```
#+RESULTS: LA-3
#+begin_example
[-0.23333333  0.46666667  0.1       ]
```

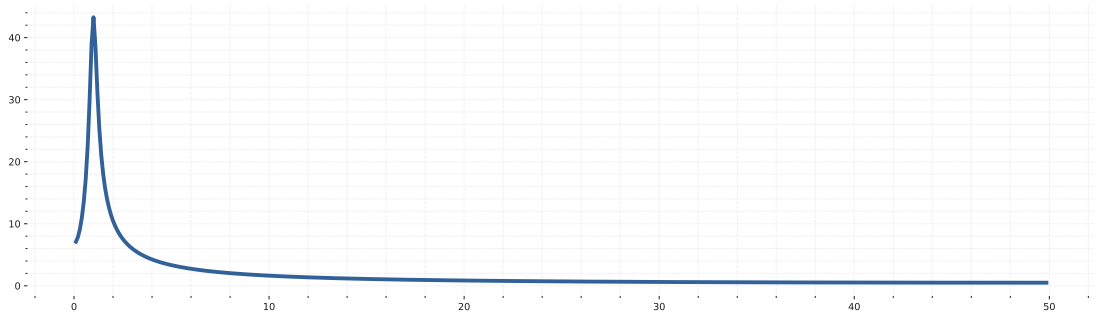


Figure 11.7.: As the signal was symmetric, we only need to see the positive side of the spectrum.

As expected, we now see a peak in the spectrum that is centered around 1, which is the frequency we used in the damped oscillator example.

11.6 Linear Algebra

The linear algebra module contains a lot of matrix related functions, including:

- linear equation solvers,
- eigenvalue solvers,
- matrix functions (for example matrix-exponentiation),
- a number of different decompositions (SVD, LU, cholesky)

Detailed documentation is available at:
Here we will look at how to use some of these functions:

11.6.1 Linear Equation Systems

Linear equations systems on the matrix form

$$\mathbf{Ax} = \mathbf{b}$$

where A is a matrix and x, b are vector can be solved like:

C.R. 28
```
#+begin_example
[0. 0. 0.]
```
python

C.R. 29
```
#+end_src
```
python

C.R. 30
```


#+RESULTS: LA-6
#+begin_example
```
python

```
```
text

We can also do a check to see if our calculations are correct by using numpy:

C.R. 31
```

```
python

```

```
text

We can also do the same with:

$$\mathbf{AX} = \mathbf{B}$$

where **A**, **B**, **C** are matrices:

C.R. 32
```
evals = eigvals(A)
print(evals)
```
python

C.R. 33
```


#+NAME: EV-2
#+begin_src python :session :results output
```
python

```
#+RESULTS: EV-2
#+begin_example
[ 1.37593026+0.j -0.61877061+0.j -0.14902867+0.j]
```
text

C.R. 34
```
print(evecs)
```
python

```
print(evecs)
```
C.R. 35 python

11.6.2 Eigenvalues and Eigenvectors

The eigenvalue problem for a matrix **A**:

$$\mathbf{A}\mathbf{x}_n = \lambda_n \mathbf{x}_n$$

where **x** is the eigenvector and λ is the eigenvalue. To calculate eigenvalues of a matrix, use the `eigvals` and for calculating both eigenvalues and eigenvectors, use the function `eig`:

```
n = 1


```
C.R. 36 python

```
#+begin_example
```
text

```


#+NAME: MO-1
```
C.R. 37 python

```
#+RESULTS: MO-1
```
text

```
#+end_example
```
C.R. 38 python

```
#+end_src

#+RESULTS: MO-2
```
text

The eigenvectors corresponding to the n^{th} eigenvalue (stored in `evals[n]`) is the n^{th} column in `evecs`, i.e., `evecs[:,n]`. To verify this, let's try multiplying eigenvectors with the matrix and compare to the product of the eigenvector and the eigenvalue:

```
#+NAME: MO-3
#+begin_src python :session :results output
print(norm(A, ord=2), norm(A, ord=Inf))
```
C.R. 39 python

```
1.4746360038079478 1.7781999584117365
```
text

> There are also more specialized eigensolvers, like the `eigh` for Hermitian matrices.

11.7 Interpolation

In the mathematical field of numerical analysis, interpolation is a type of estimation, a method of constructing (finding) new data points based on the range of a discrete set of known data

points.

In engineering and science, one often has a number of data points, obtained by sampling or experimentation, which represent the values of a function for a limited number of values of the independent variable. It is often required to interpolate; that is, estimate the value of that function for an intermediate value of the independent variable.

A closely related problem is the approximation of a complicated function by a simple function. Suppose the formula for some given function is known, but too complicated to evaluate efficiently. A few data points from the original function can be interpolated to produce a simpler function which is still fairly close to the original. The resulting gain in simplicity may outweigh the loss from interpolation error and give better performance in calculation process.

Interpolation is simple and convenient in scipy: The `interp1d` function, when given arrays describing `X` and `Y` data, returns and object that behaves like a function that can be called for an arbitrary value of x (in the range covered by X), and it returns the corresponding interpolated y value:

C.R. 40 python

```
1
```

C.R. 41 python

```
# initial state:
y0 = [1.0, 0.0]
```

C.R. 42 python

```
# time coordinate to solve the ODE for
t = np.linspace(0, 10, 1000)
w0 = 2*np.pi*1.0
#+end_src

#+NAME: DHO-4
#+begin_src python :session :results output
# solve the ODE problem for three different values of the damping ratio
y1 = odeint(dy, y0, t, args=(0.0, w0)) # undamped
y2 = odeint(dy, y0, t, args=(0.2, w0)) # under damped
y3 = odeint(dy, y0, t, args=(1.0, w0)) # critical damping
```

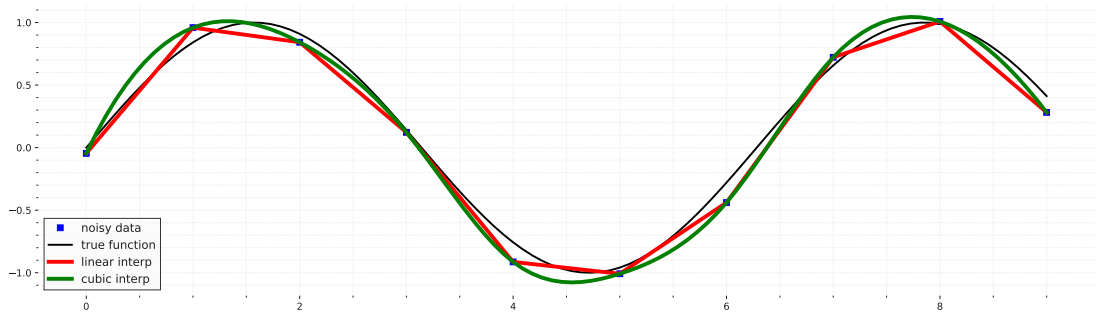


Figure 11.8.: An example of interpolation of a function using linear, and cubic interpolation.

11.8 Statistics

The `scipy.stats` module contains a large number of statistical distributions, statistical functions and tests. There is also a very powerful python package for statistical modelling called statsmodels.

C.R. 43
python
```
#+begin_src python :session :results output
# create a (continuous) random variable with normal distribution
```

C.R. 44
python
```


fig, axes = plt.subplots(3,1, sharex=True, figsize=(18, 10))

```

C.R. 45
python
```
# plot histogram of 1000 random realizations of the stochastic variable Y
axes[2].hist(Y.rvs(size=1000), bins=50);

#+end_src

#+NAME: FF-P-2
#+begin_src python :session :results output
cp.store_fig("stats-plot-norm", close=True)
#+end_src

#+NAME: ST-6
#+begin_src python :session :results output
print(X.mean(), X.std(), X.var()) # Poisson distribution
```

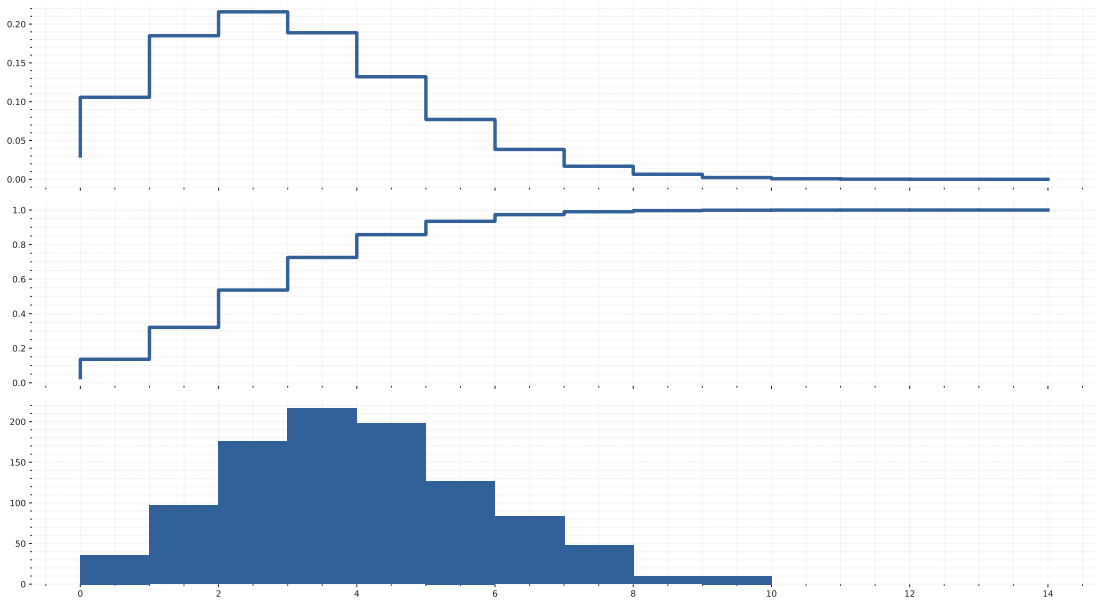


Figure 11.9.: he plots of three statistical properties for a discrete distribution. From top to bottom **(a)** Probability density function **(b)** Cumulative density function **(c)** Histogram.

C.R. 46
```python
#+RESULTS: ST-7
```

C.R. 47
```python
```

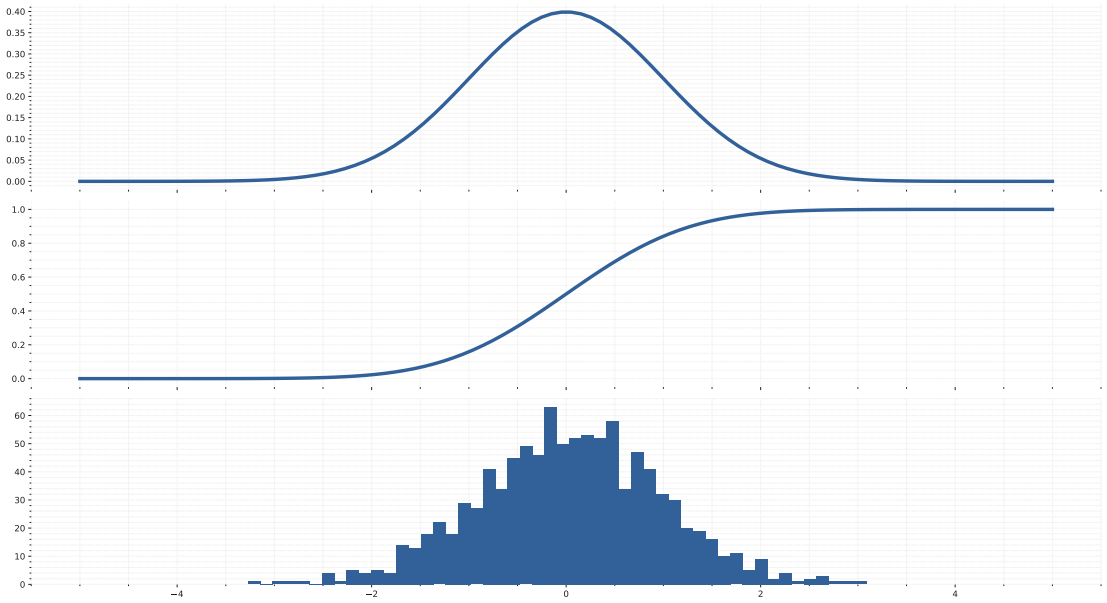


Figure 11.10.: The plots of three statistical properties for a continuous distribution. From top to bottom **(a)** Probability density function **(b)** Cumulative density function **(c)** Histogram.

C.R. 48
```python
```

```text
```

C.R. 49
```python
```

```text
```

Chapter 12

SymPy

Table of Contents

There are two (2) notable Computer Algebra Systems (CAS) for Python:

1. **SymPy** A python module that can be used in any Python program, or in an IPython session, that provides powerful CAS features.

2. **Sage** A full-featured and very powerful CAS environment that aims to provide an open source system that competes with Mathematica and Maple.

> Sage is not a regular Python module, but rather a CAS environment that uses Python as its programming language.

> Sage is in some aspects more powerful than SymPy, but both offer very comprehensive CAS functionality. The advantage of SymPy is that it is a regular Python module and integrates well with the IPython notebook.

In this chapter we will therefore look at how to use SymPy with IPython notebooks. If you are interested in an open source CAS environment I also recommend to read more about Sage as it is a great tool to study linear algebra and differential equations.

To get started using SymPy in a Python program or notebook, import the module `sympy`:

```
import sympy as sp # for symbolic calculations
import matplotlib.pyplot as plt # for plotting applications
```

To print them in LaTeX format, we can invoke the following command:

```
sp.init_printing(use_latex=True)
```

12.1 Symbolic Variables

In SymPy we need to create symbols for the variables we want to work with. We can create a new symbol using the `Symbol` class:

```
x = sp.Symbol('x')
```

```
print(sp.latex((sp.pi + x)**2))
```

$$\left(x + \pi\right)^{2}$$

```
# alternative way of defining symbols
a, b, c = sp.symbols("a, b, c")
```

And if we want to look at the type of a data in Python, remember to just use `type()` which will give you the data type of the variable in question.

```
print(type(a))
```

```
<class 'sympy.core.symbol.Symbol'>
```

We can add assumptions to symbols when we create them just write it in the paranthesis as so:

```
x = sp.Symbol('x', real=True)
```

And if we assume a false assumption, we should get `False`.

```
print(x.is_imaginary)
```

```
False
```

We can also do other assumptions:

```
x = sp.Symbol('x', positive=True)
```

And testing this with a `True` statement, we find:

```
print(x > 0)
```

```
True
```

12.1.1 Complex Numbers

The imaginary unit is denoted `I` in SymPy.

Imaginary Numbers

An imaginary number is the product of a real number and the imaginary unit i, which is defined by its property $i^2 = -1$. The square of an imaginary number bi is $-b^2$. For example, $5i$ is an imaginary number, and its square is -25. The number zero is considered to be both real and imaginary.

Originally coined in the 17th century by René Descartes as a derogatory term and regarded as fictitious or useless, the concept gained wide acceptance following the work of Leonhard Euler (in the 18th century) and Augustin-Louis Cauchy and Carl Friedrich Gauss (in the early 19th century).

An imaginary number bi can be added to a real number a to form a complex number of the form $a + bi$, where the real numbers a and b are called, respectively, the real part and the imaginary part of the complex number.

```
print(sp.latex(1+1*sp.I))
```

$$1 + i$$

```
print(sp.latex(sp.I**2))
```

$$-1$$

```
print(sp.latex((x * sp.I + 1)**2))
```

$$(ix + 1)^2$$

12.1.2 Rational Numbers

There are three different numerical types in SymPy: `Real`, `Rational`, `Integer`:

Real - Rational - Integer

Real Numbers: A number that can be used to measure a continuous one-dimensional quantity such as a distance, duration or temperature. Here, continuous means that pairs of values can have arbitrarily small differences.

Rational Number: a rational number is a number that can be expressed as the quotient or fraction $\frac{p}{q}$ of two integers, a numerator p and a non-zero denominator q.

Integer: is the number zero (0), a positive natural number (1, 2, 3, . . .), or the negation of a positive natural number (-1, -2, -3, . . .).

```
r1 = sp.Rational(4,5)
r2 = sp.Rational(5,4)
```

```
print(sp.latex(r1))
```

$$\frac{4}{5}$$

```
print(sp.latex(r1 + r2))
```

$$\frac{41}{20}$$

```
print(sp.latex(r1 / r2))
```

$$\frac{16}{25}$$

12.2 Numerical Evaluation

SymPy uses a library for arbitrary precision as numerical backend, and has predefined SymPy expressions for a number of mathematical constants, such as: `pi`, `e`, `oo` for infinity.
To evaluate an expression numerically we can use the evalf function (or `N`). It takes an argument `n` which specifies the number of significant digits.

```
print(sp.latex(sp.pi.evalf(n=50)))
```

$$3.1415926535897932384626433832795028841971693993751$$

```
y = (x + sp.pi)**2
```

```
print(sp.latex(sp.N(y, 5))) # same as evalf
```

$$9.8696 \left(0.31831 x + 1\right)^{2}$$

When we numerically evaluate algebraic expressions we often want to substitute a symbol with a numerical value. In SymPy we do that using the subs function

```
print(sp.latex(y.subs(x, 1.5)))
```

$$(1.5 + \pi)^2$$

```
print(sp.latex(sp.N(y.subs(x, 1.5))))
```

21.5443823618587

The `subs` function can of course also be used to substitute Symbols and expressions:

```
print(sp.latex(y.subs(x, a+sp.pi)))
```

$$(a + 2\pi)^2$$

We can also combine numerical evaluation of expressions with NumPy arrays:

```
import numpy
```

```
x_vec = numpy.arange(0, 10, 0.1)
```

```
y_vec = numpy.array([sp.N(((x + sp.pi)**2).subs(x, xx)) for xx in x_vec])
```

```
fig, ax = plt.subplots()
ax.plot(x_vec, y_vec);
plt.savefig("images/sympy/fplot.pdf")
```

However, this kind of numerical evaluation can be very slow, and there is a much more efficient way to do it: Use the function `lambdify` to "compile" a SymPy expression into a function that is much more efficient to evaluate numerically:

```
f = sp.lambdify([x], (x + sp.pi)**2, 'numpy')
# the first argument is a list of variables that
# f will be a function of: in this case only x -> f(x)
```

```
y_vec = f(x_vec)  # now we can directly pass a numpy array and f(x) is efficiently
↪  evaluated
```

12.3 Algebraic manipulations

One of the main uses of an CAS is to perform algebraic manipulations of expressions. For example, we might want to expand a product, factor an expression, or simplify an expression. The functions for doing these basic operations in SymPy are demonstrated in this section.

12.3.1 Expand and Factor

The first steps in an algebraic manipulation

```
print(sp.latex((x+1)*(x+2)*(x+3)))
```

$$(x+1)(x+2)(x+3)$$

```
print(sp.latex(sp.expand((x+1)*(x+2)*(x+3))))
```

$$x^3 + 6x^2 + 11x + 6$$

The `expand` function takes a number of keywords arguments which we can tell the functions what kind of expansions we want to have performed. For example, to expand trigonometric expressions, use the `trig=True` keyword argument:

```
print(sp.latex(sp.sin(a+b)))
```

$$\sin(a+b)$$

```
print(sp.latex(sp.expand(sp.sin(a+b), trig=True)))
```

$$\sin(a)\cos(b) + \sin(b)\cos(a)$$

See `help(expand)` for a detailed explanation of the various types of expansions the expand functions can perform.
The opposite of product expansion is of course factoring. To factor an expression in SymPy use the `factor` function:

```
print(sp.latex(sp.factor(x**3 + 6 * x**2 + 11*x + 6)))
```

$$(x+1)(x+2)(x+3)$$

12.3.2 Simplify

The `simplify` tries to simplify an expression into a nice looking expression, using various techniques. More specific alternatives to the `simplify` functions also exists: `trigsimp`, `powsimp`, `logcombine`, etc. The basic usages of these functions are as follows:

```
# simplify (sometimes) expands a product
print(sp.latex(sp.simplify((x+1)*(x+2)*(x+3))))
```

$$(x+1)(x+2)(x+3)$$

```
# simplify uses trigonometric identities
print(sp.latex(sp.simplify(sp.sin(a)**2 +sp. cos(a)**2)))
```

```
print(sp.latex(sp.simplify(sp.cos(x)/sp.sin(x))))
```

$$\frac{1}{\tan\left(x\right)}$$

12.3.3 Apart and Together

```
f1 = 1/((a+1)*(a+2))
```

```
print(sp.latex(f1))
```

$$\frac{1}{\left(a+1\right)\left(a+2\right)}$$

```
print(sp.latex(sp.apart(f1)))
```

$$-\frac{1}{a+2}+\frac{1}{a+1}$$

```
f2 = 1/(a+2) + 1/(a+3)
```

```
print(sp.latex(f2))
```

$$\frac{1}{a+3}+\frac{1}{a+2}$$

```
print(sp.latex(sp.together(f2)))
```

$$\frac{2a+5}{\left(a+2\right)\left(a+3\right)}$$

Simplify usually combines fractions but **does not factor**:

```
print(sp.latex(sp.simplify(f2)))
```

$$\frac{2a+5}{\left(a+2\right)\left(a+3\right)}$$

12.4 Calculus

In addition to algebraic manipulations, the other main use of CAS is to do calculus, like derivatives and integrals of algebraic expressions.

12.4.1 Differentiation

Differentiation is usually simple. Use the `diff` function. The first argument is the expression to take the derivative of, and the second argument is the symbol by which to take the derivative:

```
print(sp.latex(y))
```

$$(x + \pi)^2$$

```
print(sp.latex(sp.diff(y**2, x)))
```

$$4 (x + \pi)^3$$

For higher order derivatives we can do:

```
print(sp.latex(sp.diff(y**2, x, x)))
```

$$12 (x + \pi)^2$$

```
print(sp.latex(sp.diff(y**2, x, 2))) # same as above
```

$$12 (x + \pi)^2$$

To calculate the derivative of a multivariate expression, we can do:

```
x, y, z = sp.symbols("x,y,z")
```

```
f = sp.sin(x*y) + sp.cos(y*z)
```

```
print(sp.latex(sp.diff(f, x, 1, y, 2)))
```

$$-x \left(xy \cos\left(xy\right) + 2 \sin\left(xy\right)\right)$$

12.4.2 Integration

Integration is done in a similar fashion:

```
print(sp.latex(f))
```

$$\sin\left(xy\right) + \cos\left(yz\right)$$

```
print(sp.latex(sp.integrate(f, x)))
```

$$x \cos\left(yz\right) + \begin{cases} -\frac{\cos\left(xy\right)}{y} & \text{for } y \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

By providing limits for the integration variable we can evaluate definite integrals:

```
print(sp.latex(sp.integrate(f, (x, -1, 1))))
```

$$2 \cos\left(yz\right)$$

and also improper integrals

```
print(sp.latex(sp.integrate(sp.exp(-x**2), (x, -sp.oo, sp.oo))))
```

$$\sqrt{\pi}$$

> Remember, oo is the SymPy notation for inifinity.

12.4.3 Sums and Products

We can evaluate sums using the function Sum:

```
n = sp.Symbol("n")
```

```
print(sp.latex(sp.Sum(1/n**2, (n, 1, 10))))
```

$$\sum_{n=1}^{10} \frac{1}{n^2}$$

```
print(sp.latex(sp.Sum(1/n**2, (n,1, 10)).evalf()))
```

$$1.54976773116654$$

```
print(sp.latex(sp.Sum(1/n**2, (n, 1, sp.oo)).evalf()))
```

$$1.64493406684823$$

Products work much the same way:

```
print(sp.latex(sp.Product(n, (n, 1, 10)))) # 10!
```

$$\prod_{n=1}^{10} n$$

12.4.4 Limits

Limits can be evaluated using the `limit` function. For example,

```
print(sp.latex(sp.limit(sp.sin(x)/x, x, 0)))
```

$$1$$

We can use 'limit' to check the result of derivation using the `diff` function:

```
print(sp.latex(f))
```

$$\sin(xy) + \cos(yz)$$

```
print(sp.latex(sp.diff(f, x)))
```

$$y\cos\left(xy\right)$$

```
h = sp.Symbol("h")
```

```
print(sp.latex(sp.limit((f.subs(x, x+h) - f)/h, h, 0)))
```

$$y\cos\left(xy\right)$$

We can change the direction from which we approach the limiting point using the `dir` keyword argument:

```
print(sp.latex(sp.limit(1/x, x, 0, dir="+")))
```

$$\infty$$

```
print(sp.latex(sp.limit(1/x, x, 0, dir="-")))
```

$$-\infty$$

12.4.5 Series

Series expansion is also one of the most useful features of a CAS. In SymPy we can perform a series expansion of an expression using the series function:

```
print(sp.latex(sp.series(sp.exp(x), x)))
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + O\left(x^6\right)$$

By default it expands the expression around $x = 0$, but we can expand around any value of x by explicitly include a value in the function call:

```
print(sp.latex(sp.series(sp.exp(x), x, 1)))
```

$$e + e\left(x-1\right) + \frac{e\left(x-1\right)^2}{2} + \frac{e\left(x-1\right)^3}{6} + \frac{e\left(x-1\right)^4}{24} + \frac{e\left(x-1\right)^5}{120} + O\left(\left(x-1\right)^6; x \to 1\right)$$

And we can explicitly define to which order the series expansion should be carried out:

```
print(sp.latex(sp.series(sp.exp(x), x, 1, 5)))
```

$$e + e\left(x-1\right) + \frac{e\left(x-1\right)^2}{2} + \frac{e\left(x-1\right)^3}{6} + \frac{e\left(x-1\right)^4}{24} + O\left(\left(x-1\right)^5; x \to 1\right)$$

The series expansion includes the order of the approximation, which is very useful for keeping track of the order of validity when we do calculations with series expansions of different order:

```
s1 = sp.cos(x).series(x, 0, 5)
print(sp.latex(s1))
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} + O\left(x^5\right)$$

```
s2 = sp.sin(x).series(x, 0, 2)
print(sp.latex(s2))
```

$$x + O\left(x^2\right)$$

```
print(sp.latex(sp.expand(s1 * s2)))
```

$$x + O\left(x^2\right)$$

If we want to get rid of the order information we can use the removeO method:

```
print(sp.latex(sp.expand(s1.removeO() * s2.removeO())))
```

$$\frac{x^5}{24} - \frac{x^3}{2} + x$$

But note that this is not the correct expansion $\cos x \sin x$ of to 5th order:

```
(cos(x)*sin(x)).series(x, 0, 6)
```

12.5 Linear Algebra

Matrices are defined using the Matrix class:

```
m11, m12, m21, m22 = sp.symbols("m11, m12, m21, m22")
b1, b2 = sp.symbols("b1, b2")
```

```
A = sp.Matrix([[m11, m12],[m21, m22]])
print(sp.latex(A))
```

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$$

```
b = sp.Matrix([[b1], [b2]])
print(sp.latex(b))
```

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

With `Matrix` class instances we can do the usual matrix algebra operations:

```
print(sp.latex(A**2))
```

$$\begin{bmatrix} m_{11}^2 + m_{12}m_{21} & m_{11}m_{12} + m_{12}m_{22} \\ m_{11}m_{21} + m_{21}m_{22} & m_{12}m_{21} + m_{22}^2 \end{bmatrix}$$

```
print(sp.latex(A * b))
```

$$\begin{bmatrix} b_1 m_{11} + b_2 m_{12} \\ b_1 m_{21} + b_2 m_{22} \end{bmatrix}$$

And calculate determinants and inverses, and the like:

```
print(sp.latex(A.det()))
```

$$m_{11}m_{22} - m_{12}m_{21}$$

```
print(sp.latex(A.inv()))
```

$$\begin{bmatrix} \frac{m_{22}}{m_{11}m_{22} - m_{12}m_{21}} & -\frac{m_{12}}{m_{11}m_{22} - m_{12}m_{21}} \\ -\frac{m_{21}}{m_{11}m_{22} - m_{12}m_{21}} & \frac{m_{11}}{m_{11}m_{22} - m_{12}m_{21}} \end{bmatrix}$$

12.6 Solving equations

For solving equations and systems of equations we can use the `solve` function:

```
print(sp.latex(sp.solve(x**2 - 1, x)))
```

$$[-1, 1]$$

```
print(sp.latex(sp.solve(x**4 - x**2 - 1, x)))
```

$$\left[-i\sqrt{-\frac{1}{2} + \frac{\sqrt{5}}{2}}, i\sqrt{-\frac{1}{2} + \frac{\sqrt{5}}{2}}, -\sqrt{\frac{1}{2} + \frac{\sqrt{5}}{2}}, \sqrt{\frac{1}{2} + \frac{\sqrt{5}}{2}}\right]$$

System of equations:

```
print(sp.latex(sp.solve([x + y - 1, x - y - 1], [x,y])))
```

$$\{x : 1, y : 0\}$$

In terms of other symbolic expressions:

```
print(sp.latex(sp.solve([x + y - a, x - y - c], [x,y])))
```

$$\left\{x : \frac{a}{2} + \frac{c}{2}, y : \frac{a}{2} - \frac{c}{2}\right\}$$

Chapter 13

Introduction to Artificial Neural Networks

Table of Contents

13.1 Introduction

It is quite apparent that life imitates life and engineers are inspired by nature. It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine. This is the logic that sparked Artificial Neural Networks (ANN)s, Machine Learning

Figure 13.1.: Nature always is a great source of inspiration for good design. For example, the beak of a bird is aerodynamically efficient and was used in designing the Bullet train [3].The field is of emulating models, systems, and elements of nature for the purpose of solving complex human problems is called biomimetics [25].

(ML) models inspired by the networks of biological neurons found in our brains. However, although planes were inspired by birds, they don't have to flap their wings to fly. Similarly, ANNs have gradually become quite different from their biological cousins. Some researchers even argue that we should drop the biological analogy altogether such as calling them units rather than neurons [1], as some consider this naming to decrease the amount of creativity we can give to the topic .

ANNs are at the very core of **deep learning**. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex ML tasks such as classifying billions of images (e.g., Google Images), powering speech recognition services (e.g., Apple's Siri), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning to beat the world champion at the game of Go (DeepMind's AlphaGo [13]).

The will treat this chapter as a formal introduction to ANN, starting with a tour of the very first ANN architectures and leading up to multilayer perceptrons, which are heavily used today.

In the second part, we will look at how to implement neural networks using TensorFlow's Keras API. This is a beautifully designed and simple high-level API for building, training, evaluating, and running neural networks. While it may look simple at first glance, it is expressive and flexible enough to let you build a wide variety of neural network architectures.

> For most of your use cases, using `keras` will be enough.

Figure 13.2.: The prolific advancements of computers and neural networks have allowed us to tackle problems once deemed impossible. A game of GO requires uncountable amount of moves, yet using ML it was possible to create a software capable of beating the world champion.

13.2 From Biology to Silicon: Artificial Neurons

While it may seem they are the cutting edge in ML, ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. In their landmark paper *A Logical Calculus of Ideas Immanent in Nervous Activity*, They presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic. This was the first artificial neural network architecture [18].

Since then many other architectures have been invented. The early successes of ANNs led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear in the 1960s that this promise would go unfulfilled (at least for quite a while), funding flew elsewhere, and ANNs entered a long winter. This is also known as the 1st AI winter [14]. In the early 1980s, new architectures were invented and better training techniques were developed, sparking a revival of interest in connectionism, the study of neural networks. But progress was slow, and by the 1990s other powerful ML techniques had been invented, such as Support Vector Machines (SVM). These techniques seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold and this is known as the 2nd AI winter.

We are now witnessing yet another wave of interest in ANNs. Will this wave die out like the previous ones did? Well, here are a few good reasons to believe that this time is different and that the renewed interest in ANNs will have a much more profound impact on our lives:

There is now a **huge quantity of data available** to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems. One of the major turning points of ANN was the fundamental question of:

Is our understanding of the model at fault or is it merely the lack of data to train?

The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to **Moore's law**, but also thanks to the gaming industry, which has stimulated the production of powerful GPU cards by the millions which have become the norm to train ML instead of CPUs.

Moore's Law

the number of components in integrated circuits has doubled about every 2 years over the last 50 years.

In addition to previous additions, cloud platforms have made this power accessible to everyone. The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have had a huge positive impact.

Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima [8], but it turns out that this is not a big problem in practice, especially for larger neural networks: the local optima often perform almost as well as the global optimum.

ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them, resulting in more and more progress and even more amazing products.

13.2.1 Biological Neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron. It is an unusual-looking cell mostly found in animal brains.

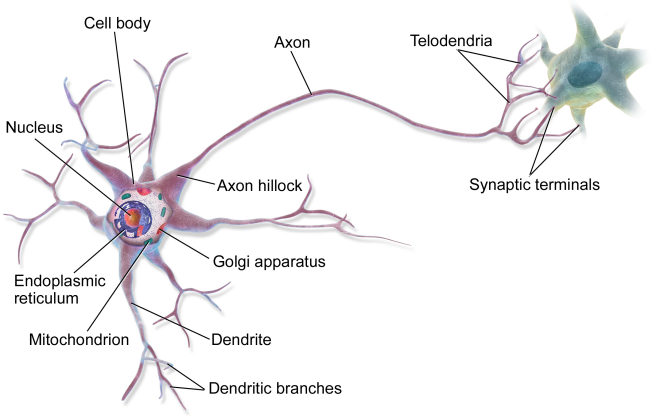


Figure 13.3.: A neuron or nerve cell is an excitable cell that fires electric signals called action potentials across a neural network in the nervous system. Neurons communicate with other cells via synapses, which are specialized connections that commonly use minute amounts of chemical neurotransmitters to pass the electric signal from the presynaptic neuron to the target cell through the synaptic gap [22].

It's composed of a cell body containing the nucleus and most of the cell's complex components, many branching extensions called dendrites, plus one very long extension called the axon. The axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer.
Near its extremity the axon splits off into many branches called *telodendria*, and at the tip of these branches are minuscule structures called synaptic terminals (or simply synapses), which are connected to the dendrites or cell bodies of other neurons. Biological neurons produce short electrical impulses called action potentials (APs, or just signals), which travel along the axons and make the synapses release chemical signals called neurotransmitters. When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses (actually, it depends on the neurotransmitters, as some of them inhibit the neuron from firing).

Therefore, individual biological neurons seem to behave in a simple way, but they're organized in a vast network of billions, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of biological neural networks (BNNs) is the subject of active research, but some parts of the brain have been mapped [2]. These efforts show that neurons are often organized in consecutive layers, especially in the cerebral cortex (the outer layer of the brain).

13.2.2 Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an **artificial neuron**: it has one or more binary (on/off) inputs and one binary output. The artificial neuron activates its output when more than a certain number of its

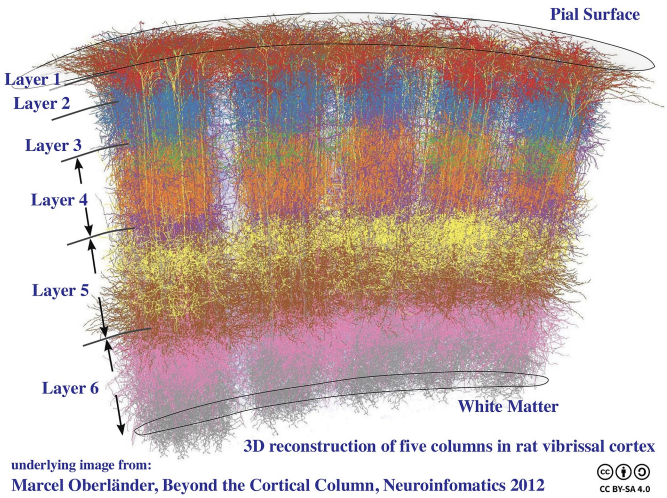


Figure 13.4.: A cortical column is a group of neurons forming a cylindrical structure through the cerebral cortex of the brain perpendicular to the cortical surface. The structure was first identified by Vernon Benjamin Mountcastle in 1957. He later identified minicolumns as the basic units of the neocortex which were arranged into columns. Each contains the same types of neurons, connectivity, and firing properties. Columns are also called hypercolumn, macrocolumn, functional column or sometimes cortical module

inputs are active. In their paper, McCulloch and Pitts showed that even with such a simplified model it is possible to build a network of artificial neurons that can compute any logical proposition you want. To see how such a network works, let's build a few ANNs that perform various logical computationsm, assuming that a neuron is activated when at least two of its input connections are active. Let's see what these networks do:

- The first network on the left is the identity function: if neuron **A** is activated, then neuron **C** gets activated as well (since it receives two input signals from neuron **A**); but if neuron **A** is off, then neuron **C** is off as well.
- The second network performs a logical **AND**: neuron **C** is activated only when both neurons **A** and **B** are activated (a single input signal is not enough to activate neuron **C**).
- The third network performs a logical **OR**: neuron **C** gets activated if either neuron **A** or neuron **B** is activated (or both).
- Finally, if we suppose that an input connection can inhibit the neuron's activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

You can imagine how these networks can be combined to compute complex logical expressions.

13.2.3 The Perceptron

The perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt [5]. It is based on a slightly different artificial neuron called a Threshold Logic Unit (TLU), or sometimes a Linear Threshold Unit (LTU) which can be seen in Fig. 13.5. The inputs and output

are numbers (this is instead of binary on/off values), and each input connection is associated with a weight. The TLU first computes a linear function of its inputs:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \mathbf{x}^T \mathbf{w} + b$$

Then it applies a **step function** to the result:

$$h(x) = \text{step}(z) \qquad \text{where} \qquad z = \mathbf{x}^T \mathbf{w}.$$

It is similar to logistic regression, except it uses a step function instead of the logistic function. Just like in logistic regression, the model parameters are the input weights w and the bias term b.

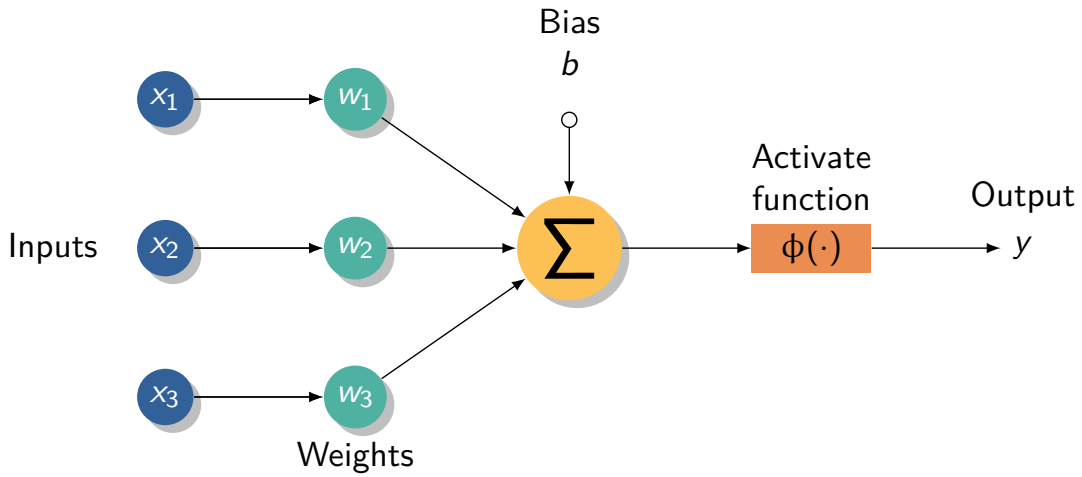


Figure 13.5.: Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a certain activation function.

The most common step function used in perceptrons is the **Heaviside step** and sometimes the **sign function** is used instead [23].

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if} \quad z < 0, \\ 1 & \text{if} \quad z \geq 0. \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if} \quad z < 0, \\ 0 & \text{if} \quad z = 0, \\ +1 & \text{if} \quad z > 0, \end{cases}$$

A single TLU can be used for simple **linear binary classification**:

> *It computes a linear function of its inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise, it outputs the negative class. They exhibit a similar behaviour to logistic regression or linear SVM classification.*

It is possible, for example, use a single TLU to classify iris flowers [6] (a famous dataset used by statisticians and ML researchers) based on petal length and width. Training such a TLU would require finding the right values for w_1, w_2, b.

A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input. Such a layer is called a **fully connected layer**, or a dense layer. The inputs constitute the input layer and since the layer of TLUs produces the final outputs, it is

called the output layer.

This perceptron can classify instances simultaneously into three (3) different binary classes, which makes it a **multilabel classifier**. It may also be used for multiclass classification.

Using linear algebra, the following equation can be used to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

In this equation:

- $\mathbf{X}$ represents the matrix of input features. It has one row per instance and one column per feature.
- $\mathbf{W}$ is the weight matrix containing all the connection weights. It has one row per input and one column per neuron.
- $\mathbf{b}$ is the bias term containing all the bias terms: one per neuron.
- ϕ is the activation function is called the activation function: when the artificial neurons are TLUs, it is a step function.

Now the question is:

> *How is this perceptron train?*

The original perceptron training algorithm proposed by Rosenblatt was largely inspired by Hebb's rule [24]. In his 1949 book *The Organization of Behaviour*, Donald Hebb suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger [9].

Siegrid Löwel later summarized Hebb's idea in the catchy phrase,

> *Cells that fire together, wire together*

This means the connection weight between two neurons tends to increase when they fire simultaneously.

> This rule later became known as Hebb's rule (or Hebbian learning [7])

Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction. The perceptron learning rule **reinforces connections that help reduce the error**.

More specifically, the perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right) x_i x$$

where:

- $w_{i,j}$ is the connection weight between the i^{th} input and the j^{th} neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}^j$ is the output of the j^{th} output neuron for the current training instance.
- y^j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

The decision boundary of each output neuron is **linear**, therefore perceptrons are incapable of learning complex patterns. However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.

> This is called the perceptron convergence theorem.

C.R. 1
python
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values y = (iris.target == 0) # Iris
↪  setosa
per_clf = Perceptron(random_state=42) per_clf.fit(X, y)
X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new) # predicts True and False for these 2 flowers
```

For those of you who have taken a **Data Science II** course, you may have noticed that the perceptron learning algorithm strongly resembles *stochastic gradient descent*. In fact, `sklearn`'s `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters:

- `loss="perceptron"`,
- `learning_rate="constant"`,
- `eta0=1` (the learning rate),
- `penalty=None` (no regularization).

In their 1969 monograph Perceptrons, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of perceptrons: in particular, they are incapable of solving some trivial problems (e.g., the exclusive OR (XOR) classification problem).

XOR Problem

A simple logic gate problem which is proven to be unsolvable using a single-layer perceptron.

This is true of any other linear classification model, but researchers had expected much more from perceptrons, and some were so disappointed, they dropped neural networks altogether in favour of higher-level problems such as logic, problem solving, and search. The lack of practical applications also didn't help.

It turns out that some of the limitations of perceptrons can be eliminated by stacking multiple perceptrons. The resulting ANN is called a MLP and a MLP can solve the XOR problem [19].

> Perceptrons **DO NOT** output a class probability. This is one reason to prefer logistic regression over perceptrons. Moreover, perceptrons do not use any regularization by default, and training stops as soon as there are no more prediction errors on the training set, so the model typically does not generalize as well as logistic regression or a linear SVM classifier. However, perceptrons may train a bit faster.

13.2.4 Multilayer Perceptron and Backpropagation

An MLP is composed of one input layer, one or more layers of TLUs called **hidden layers**, and one final layer of TLUs called the output layer. The layers close to the input layer are usually called the lower layers, and the ones close to the outputs are usually called the upper layers.

> The signal flows only in one direction (inputs to outputs), so this architecture is an example of a Feedforward Neural Networks (FNN) [4].

When an ANN contains a deep stack of hidden layers, it is called a Deep Neural Networks (DNN). The field of deep learning studies DNNs, and more generally it is interested in models containing deep stacks of computations [21].

For many years researchers struggled to find a way to train MLPs, without success. In the

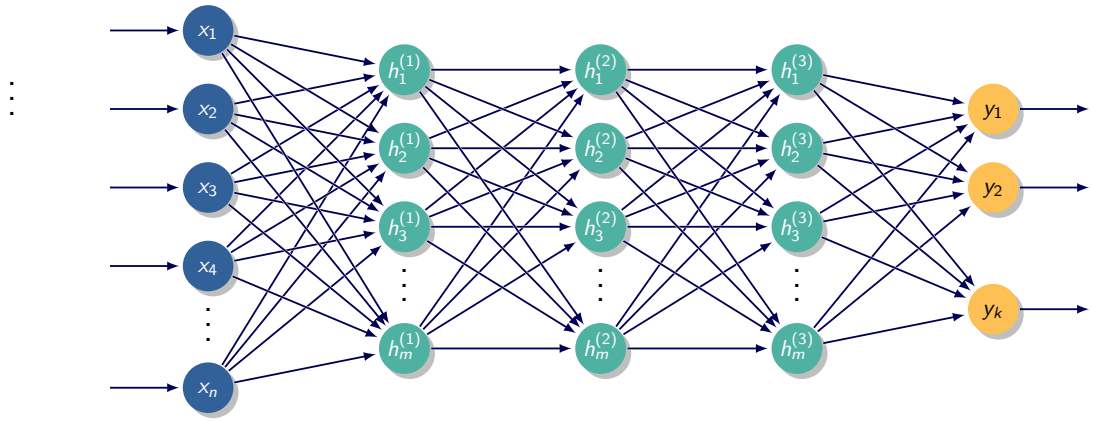


Figure 13.6.: Architecture of a Multilayer Perceptron with five inputs, three hidden layer of four neurons, and three output neurons.

early 1960s several researchers discussed of using gradient descent to train neural networks. This requires computing the gradients of the model's error with regard to the model parameters and at that time, it wasn't clear at the time how to do this efficiently with such a complex model containing so many parameters.

Then, in 1970, a researcher named *Seppo Linnainmaa* introduced in his master's thesis a technique to compute all the gradients automatically and efficiently. This algorithm is now called **reverse-mode automatic differentiation** [17]. In just two passes through the network (one forward, one backward), it is able to compute the gradients of the neural network's error with

regard to every single model parameter.

In other words, it can find out how each connection weight and each bias should be tweaked in order to reduce the neural network's error. These gradients can then be used to perform a gradient descent step. Repeating the process of computing the gradients automatically and taking a gradient descent step, the neural network's error will gradually drop until it eventually reaches a minimum.

This combination of reverse-mode automatic differentiation and gradient descent is now called **backpropagation** [10].

> There are various automatic differentiation techniques (i.e., forward and reverse), with each having its own advantages and disadvantages. Reverse-mode autodiff is well suited when the function to differentiate has many variables (e.g., connection weights and biases) and few outputs (e.g., one loss).

Backpropagation can actually be applied to all sorts of computational graphs, not just neural networks: Linnainmaa's M.Sc thesis was not about neural nets, it was more general. It was several more years before backprop started to be used to train neural networks, but it still wasn't mainstream.

Then, in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper analyzing how backpropagation allowed neural networks to learn useful internal representations [20]. Their results were so impressive that backpropagation was quickly popularized in the field. Today, it is by far the most popular training technique for neural networks.

Let's run through how backpropagation works again in a bit more detail:

1. It handles one mini-batch at a time, and goes through the full training set multiple times. Each pass is called an **epoch**.
2. Each mini-batch enters the network through the input layer. The algorithm then computes the output of all the neurons in the first hidden layer, for every instance in the mini-batch. The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer.
3. Next, the algorithm measures the network's output error. This means, it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error.
4. It then computes how much each output bias and each connection to the output layer contributed to the error. This is done analytically by applying the chain rule, which makes this step fast and precise.
5. The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until it reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all

the connection weights and biases in the network by propagating the error gradient backward through the network.

6. Finally, the algorithm performs a gradient descent step to tweak all the connection weights in the network, using the error gradients it just computed.

> Initialize all the hidden layers' connection weights randomly, or training will fail.

For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and therefore backpropagation will affect them in exactly the same way, so they will remain identical.

In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won't be too smart. If instead you randomly initialize the weights, you **break the symmetry** and allow back-propagation to train a diverse team of neurons.

In short, backpropagation makes predictions for a mini-batch (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each parameter (reverse pass), and finally tweaks the connection weights and biases to reduce the error, which is the gradient descent step.

For back-propagation to work properly, Rumelhart and his colleagues made a key change to the MLP's architecture by replacing the step function with the logistic function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Which is also called the **sigmoid function**. This was an important improvement as step function contains only flat segments, so there is no gradient to work with, while the sigmoid function has a well-defined nonzero derivative everywhere. In fact, the backpropagation algorithm works well with many other activation functions, not just the sigmoid function.

Here are two (2) other popular choices:

The hyperbolic tangent function **Model**

$$\tanh(z) = 2\sigma(2z) - 1$$

Similar sigmoid function, this activation function is also S-shaped, continuous, and differentiable, but its output value ranges from –1 to 1, instead of 0 to 1 like the sigmoid function.

This bigger range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

The rectified linear unit function **Model**

$$ReLU(z) = max(0, z)$$

It is continuous but unfortunately not differentiable at $z = 0$ as the slope changes abruptly, which can make gradient descent bounce around, and its derivative is 0 for z < 0.

In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default.

Importantly, the fact that it does not have a maximum output value helps reduce some issues during gradient descent.

You might wonder what is the point of an activation function, let alone whether it is linear or not? Chaining several linear transformations, gives you only linear transformation. For example:

$$f(x) = 2x + 3 \quad \text{and} \quad g(x) = 5x - 1 \quad \rightarrow \quad f(g(x)) = 2(5x - 1) + 3 = 10x + 1$$

You don't have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can't solve very complex problems with that.

> A large enough DNN with nonlinear activations can theoretically approximate any continuous function.

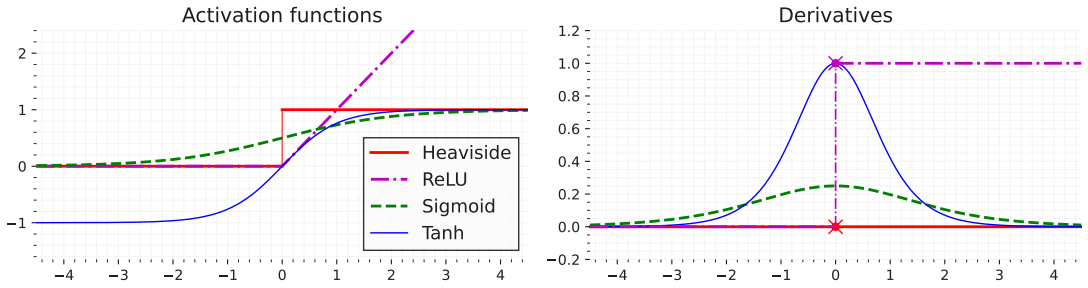


Figure 13.7.: The activation function of a node in an ANN is a function which calculates the output of the node based on its individual inputs and their weights. Nontrivial problems can be solved using only a few nodes if the activation function is nonlinear [15]. Modern activation functions include the smooth version of the ReLU, the GELU, which was used in the 2018 BERT model [11], the logistic (sigmoid) function used in the 2012 speech recognition model developed by Hinton et al [12], the ReLU used in the 2012 AlexNet computer vision model [16] and in the 2015 ResNet model.

13.2.5 Regression MLPs

First, MLPs can be used for regression tasks. If you want to predict a single value (e.g., the price of a house, given many of its features), you just need a single output neuron:

its output is the predicted value

For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. As an example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two (2) output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object.

So, in the end you end up with four (4) output neurons.

`sklearn` includes an `MLPRegressor` class, so let's use it to build an MLP with three hidden layers composed of 50 neurons each, and train it on the California housing dataset.

For simplicity, we will use `sklearn`'s `fetch_california_housing()` function to load the data instead of downloading from a sketchy website.

The following code starts by fetching and splitting the dataset, then it creates a pipeline to standardise the input features before sending them to the `MLPRegressor`. This is very important for neural networks as they are trained using gradient descent, and gradient descent does not converge very well when the features have very different scales.

Finally, the code trains the model and evaluates its validation error. The model uses the ReLU activation function in the hidden layers, and it uses a variant of gradient descent called Adam to minimize the mean squared error, with a little bit of ℓ_2 regularisation:

C.R. 2

python

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False)
```

We get a validation RMSE of about 0.505, which is comparable to what you would get with a random forest classifier.

> This MLP does not use any activation function for the output layer, so it's free to output any value it wants.

This is generally fine, but if you want to guarantee that the output will always be positive, then you should use the ReLU activation function in the output layer, or the softplus activation function, which is a smooth variant of ReLU: softplus(z) = log(1 + exp(z)).

Softplus is close to 0 when z is negative, and close to z when z is positive. Finally, if you want to guarantee that the predictions will always fall within a given range of values, then you should use the sigmoid function or the hyperbolic tangent, and scale the targets to the appropriate range: 0 to 1 for sigmoid and –1 to 1 for tanh.

Sadly, the `MLPRegressor` class does not support activation functions in the output layer.

> Building and training a standard MLP with `sklearn` is very convenient, but features are limited. This is why we will switch to Keras in the second part of this chapter.

The `MLPRegressor` class uses the mean squared error, which is usually what you want for regression, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you may want to use the Huber loss, which is a combination of both. It is quadratic when the error is smaller than a threshold δ (typically 1) but linear when the error is larger than δ. The linear part makes it less sensitive to outliers than the mean squared error, and the quadratic part allows it to converge faster and be more precise than the mean absolute error. However, `MLPRegressor` only supports the MSE.

13.2.6 Classification MLPs

MLPs can also be used for **classification** tasks. For a binary classification problem, you just need a single output neuron using the sigmoid activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class.

> The estimated probability of the negative class is equal to one minus that number.

MLPs can also easily handle multilabel binary classification tasks. For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email.

In this case, you would need two output neurons, both using the sigmoid activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to 1. This lets the model output any combination of labels: you can have nonurgent ham, urgent ham, nonurgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer (see Figure 10-9). The softmax function (introduced in Chapter 4) will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1, since the classes are exclusive. As you saw in Chapter 3, this is called multiclass classification.

Regarding the loss function, since we are predicting probability distributions, the cross-entropy loss (or x-entropy or log loss for short, see Chapter 4) is generally a good choice.

`sklearn` has an `MLPClassifier` class in the `sklearn.neural_network` package. It is almost identical to the `MLPRegressor` class, except that it minimizes the cross entropy rather than the MSE. Give it a try now, for example on the iris dataset. It's almost a linear task, so a single layer with 5 to 10 neurons should suffice (make sure to scale the features).

13.3 Implementing MLPs with Keras

Keras is TensorFlow's high-level deep learning API: it allows you to build, train, evaluate, and execute all sorts of neural networks. The original Keras 12 library was developed by François Chollet as part of a research project and was released as a standalone open source project in March 2015. It quickly gained popularity, owing to its ease of use, flexibility, and beautiful design.

Application

Keras used to support multiple backends, including TensorFlow, PlaidML, Theano, and Microsoft Cognitive Toolkit (CNTK) (the last two are sadly deprecated), but since version 2.4, Keras is TensorFlow-only. Similarly, TensorFlow used to include multiple high-level APIs, but Keras was officially chosen as its preferred high-level API when TensorFlow 2 came out. Installing TensorFlow will automatically install Keras as well, and Keras will not work without TensorFlow installed. In short, Keras and TensorFlow fell in love and got married. Other popular deep learning libraries include PyTorch by Facebook and JAX by Google.13

13.3.1 Building an Image Classifier Using Sequential API

Before we do anything, we need to load a dataset. We will use Fashion MNIST. There are 70,000 grayscale images of 28 × 28 pixels each, with 10 classes where images represent fashion items rather than handwritten digits, so each class is more diverse, and the problem turns out to be significantly challenging.

Using Keras to load the dataset

`keras` provides utility functions to fetch and load common datasets, including MNIST, Fashion MNIST, and a few more.

Let's load Fashion MNIST. It's already shuffled and split into a training set (60,000 images) and a test set (10,000 images), but we'll hold out the last 5,000 images from the training set for validation:

C.R. 3 python

```python
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

TensorFlow is usually imported as `tf`, and the Keras API is available via `tf.keras`.

When loading MNIST or Fashion MNIST using `tf.keras` rather than `sklearn`, an important difference is that every image is represented as a 28-by-28 array rather than a 1D array of size 784 with intensities are represented as integers (from 0 to 255) rather than

floats (from 0.0 to 255.0).

Let's take a look at the shape and data type of the training set:

C.R. 4 python
```
print("The size of the training dataset: ", X_train.shape)
print("The type of the training dataset: ", X_train.dtype)
```

text
```
The size of the training dataset:  (55000, 28, 28)
The type of the training dataset:  uint8
```

To make it simple, we'll scale the pixel intensities down to the 0–1 range by dividing them by 255.0

This operation also converts the integer values to floats.

C.R. 5 python
```
X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test / 255.
```

Using Fashion MNIST, we need the list of class names to know what we are dealing with:

C.R. 6 python
```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

For example, the first image in the training set represents an ankle boot: and below we can

Figure 13.8.: An example of a data within the Fashion MNIST.

see some examples of the Fashion MNIST dataset.

13.3.2 Creating the model using the sequential API

It is time to build the neural network. Here is a classification MLP with two (2) hidden layers:

C.R. 7 python
```
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
```

Figure 13.9.: A random collection of dataset, making the Fashion MNIST.

C.R. 8
python
```
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Let's try to understand the code:

1. Set `tf` random seed to make the results reproducible: the random weights of the hidden layers and the output layer will be the same every time you run your code. You could also choose to use the `tf.keras.utils.set_random_seed()` function, which conveniently sets the random seeds for TensorFlow, Python (`random.seed()`), and NumPy (`np.random.seed()`).

2. Next line creates a *Sequential model*. This is the simplest kind of Keras model for neural networks, composed of a single stack of layers connected sequentially. This is called the sequential API.

3. We build the first layer (an Input layer) and add it to the model. We specify the input shape, which doesn't include the batch size, only the shape of the instances. Keras needs to know the shape of the inputs so it can determine the shape of the connection weight matrix of the first hidden layer.

4. We add a Flatten layer. Its role is to convert each input image into a 1D array: for example, if it receives a batch of shape [32, 28, 28], it will reshape it to [32, 784]. In other words, if it receives input data X, it computes `X.reshape(-1, 784)`. This layer doesn't have any parameters; it's just there to do some simple pre-processing.

5. We add a Dense hidden layer with 300 neurons. It will use the ReLU activation function. Each Dense layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vector of bias terms (one per neuron).

6. We add a second Dense hidden layer with 100 neurons, also using the ReLU activation function.

7. We add a Dense output layer with 10 neurons (one per class), using the softmax activation function because the classes are exclusive.

> Writing the argument `activation="relu"` is equivalent to specifying `activation=tf.keras.activations.relu`. Other activation functions are available in the `tf.keras.activations` package.

Instead of adding the layers one by one as we just did, it's often more convenient to pass a list of layers when creating the Sequential model. You can also drop the Input layer and instead specify the `input_shape` in the first layer:

C.R. 9 python
```
tf.keras.backend.clear_session()
tf.random.set_seed(42)

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

The model's `summary()` method displays all the model's layers, including each layer's name, which is automatically generated, its output shape, and its number of parameters.

The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters:

C.R. 10 python
```
tf.keras.utils.plot_model(model, imagePath+"mnist-_model.pdf", show_shapes=True)
```

Dense layers often have a lot of parameters. For example, the first hidden layer has 784-by-300 connection weights, with 300 bias terms, which adds up to 235,500 parameters.

This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of **over-fitting**, especially when you do not have a lot of training data.

Each layer in a model must have a unique name (e.g., `dense_2`). You can set the layer names explicitly using the constructor's name argument, but generally it's simpler to let Keras name the layers automatically, as we just did. Keras takes the layer's class name and converts it to snake case (i.e., a layer from the `MyCoolLayer` class is named `my_cool_layer` by default). Keras also ensures that the name is **globally unique**, even across models, by appending an index if needed, as in `dense_2`.

This naming scheme makes it possible to merge models easily without getting name conflicts.

> All global state managed by Keras is stored in a Keras session, which you can clear using `tf.keras.backend.clear_session()`.

You can easily get a model's list of layers using the layers attribute, or use the `get_layer()` method to access a layer by name:

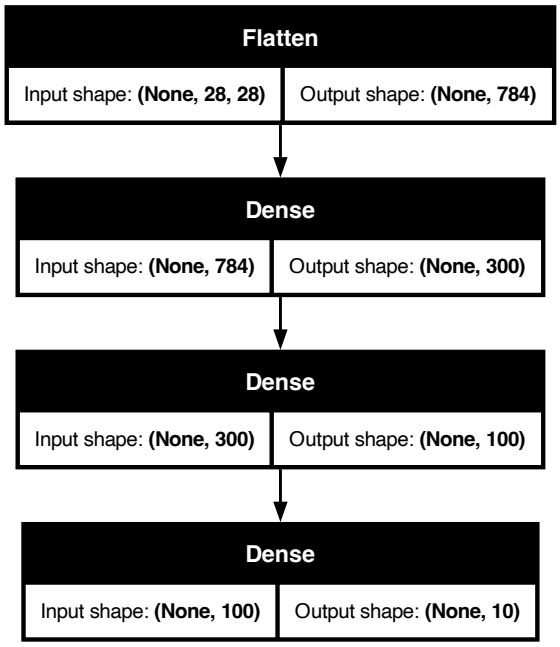


Figure 13.10.: The plot of the neural network, showcasing its layers.

C.R. 11 python
```python
print(model.layers)
```

C.R. 12 text
```
[<Flatten name=flatten, built=True>,
<Dense name=dense, built=True>,
<Dense name=dense_1, built=True>,
<Dense name=dense_2, built=True>]
```

All the parameters of a layer can be accessed using its `get_weights()` and `set_weights()` methods.

For a Dense layer, this includes both the connection weights and the bias terms:

C.R. 13 python
```python
hidden1 = model.layers[1]
weights, biases = hidden1.get_weights()
print(weights)
```

C.R. 14 text
```
[[-0.05415904  0.00010975 -0.00299759 ...  0.05136904  0.0740822
   0.06472497]
 [ 0.05510217 -0.01353022 -0.00363479 ...  0.07100512 -0.04926914
  -0.02905609]
 [-0.07024231  0.02524897 -0.04784295 ... -0.0521326   0.05084455
  -0.06636713]
 ...
```

C.R. 15
text
```
[ 0.0067075  -0.00256791 -0.064556   ...  0.05266081  0.03520959
 -0.02309504]
[ 0.05826265 -0.0361187  -0.04228947 ...  0.05612285 -0.03179397
  0.06843598]
[ 0.06636336 -0.00123435 -0.00247347 ...  0.01809192  0.03434542
  0.00700523]]
```

Notice that the Dense layer initialized the connection weights randomly.

> This is needed to break symmetry.

The biases were initialized to zeros, which is fine.

C.R. 16
python
```python
print(biases)
```

C.R. 17
text
```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

If you want to use a different initialization method, you can set `kernel_initializer` or `bias_initializer` when creating the layer.

Weight Matrix Shape

The shape of the weight matrix depends on the number of inputs, which is why we specified the `input_shape` when creating the model. If you do not specify the input shape, it's OK: Keras will simply wait until it knows the input shape before it actually builds the model parameters. This will happen either when you feed it some data (e.g., during training), or when you call its build() method. Until the model parameters are built, you will not be able to do certain things, such as display the model summary or save the model. So, if you know the input shape when creating the model, it is best to specify it.

Model Compiling

After a model is created, we need to call its `compile()` method to specify the loss function and the optimizer to use, or you can specify a list of extra metrics to compute during training and evaluation:

C.R. 18
```python
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

Before continuing, we need to explain what is going on here.

We use the `sparse_categorical_crossentropy` loss because we have sparse labels (i.e., for each instance, there is just a target class index, from 0 to 9 in this case), and the classes are **exclusive**.

- If we had one target probability per class for each instance (such as one-hot vectors, e.g., [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.] to represent class 3), then we would need to use the `categorical_crossentropy` loss instead.
- If we were doing binary classification or multilabel binary classification, then we would use the sigmoid activation function in the output layer instead of the softmax activation function, and we would use the `binary_crossentropy` loss.

Regarding the optimizer, `sgd` means that we will train the model using stochastic gradient descent. Keras will perform the backpropagation algorithm described earlier (i.e., reverse-mode autodiff plus gradient descent).

Finally, as this is a classifier, it's useful to measure its accuracy during training and evaluation, which is why we set `metrics=["accuracy"]`.

Training and Evaluating Models

Now the model is ready to be trained. For this we simply need to call its `fit()` method:

C.R. 19
```python
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

We pass it the input features (`X_train`) and the target classes (`y_train`), as well as the number of epochs to train (or else it would default to just 1, which would definitely not be enough to converge to a good solution).

We also pass a validation set which is optional. Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs.

> If the performance on the training set is much better than on the validation set, the model is probably overfitting the training set, or there is a bug, such as a data mismatch between the training set and the validation set.

And that's it! The neural network is trained. At each epoch during training, Keras displays the number of mini-batches processed so far on the left side of the progress bar.

The batch size is 32 by default, and since the training set has 55,000 images, the model goes through 1,719 batches per epoch: 1,718 of size 32, and 1 of size 24.
After the progress bar, you can see the mean training time per sample, and the loss and accuracy (or any other extra metrics you asked for) on both the training set and the validation set and notice that the training loss went down, which is a good sign, and the validation accuracy reached 88.94% after 30 epochs.

That's slightly below the training accuracy, so there is a little bit of overfitting going on, but not a huge amount.
If the training set was very skewed, with some classes being overrepresented and others underrepresented, it would be useful to set the `class_weight` argument when calling the `fit()` method, to give a larger weight to underrepresented classes and a lower weight to overrepresented classes.

These weights would be used by Keras when computing the loss. If you need per-instance weights, set the `sample_weight` argument. If both `class_weight` and `sample_weight` are provided, then Keras multiplies them. Per-instance weights could be useful, for example, if some instances were labeled by experts while others were labeled using a crowdsourcing platform: you might want to give more weight to the former.

You can also provide sample weights (but not class weights) for the validation set by adding them as a third item in the `validation_data` tuple. The `fit()` method returns a History object containing the training parameters (history.params), the list of epochs it went through (history.epoch), and most importantly a dictionary (history.history) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any).

C.R. 20 python
```python
print(history.params)
print(history.epoch)
```

C.R. 21 text
```
{'verbose': 'auto', 'epochs': 30, 'steps': 1719}
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
↪  25, 26, 27, 28, 29]
```

If you use this dictionary to create a Pandas DataFrame and call its `plot()` method, you get the learning curves shown in Fig. 13.11.
You can see that both the training accuracy and the validation accuracy steadily increase during training, while the training loss and the validation loss decrease.

This is good.

The validation curves are relatively close to each other at first, but they get further apart over time, which shows that there's a little bit of overfitting. In this particular case, the model looks like it performed better on the validation set than on the training set at the beginning of training, but that's not actually the case.

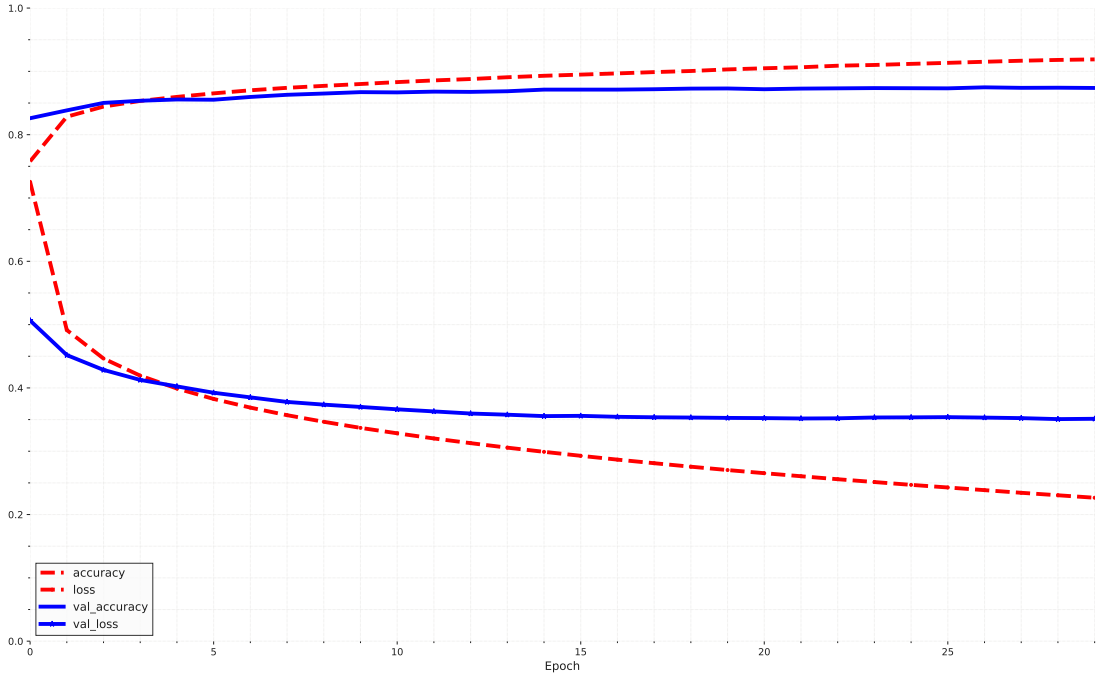


Figure 13.11.: Learning curves: the mean training loss and accuracy measured over each epoch, and the mean validation loss and accuracy measured at the end of each epoch

The validation error is computed at the end of each epoch, while the training error is computed using a running mean during each epoch, so the training curve should be shifted by half an epoch to the left.

If you do that, you will see that the training and validation curves overlap almost perfectly at the beginning of training. The training set performance ends up beating the validation performance, as is generally the case when you train for long enough.

You can tell that the model has not quite converged yet, as the validation loss is still going down, so it would be better to continue training. This is as simple as calling the `fit()` method again, as Keras just continues training where it left off: you should be able to reach about 89.8% validation accuracy, while the training accuracy will continue to rise up to 100%.

> This is not always the case.

If you are not satisfied with the performance of your model, it is a good idea to back and tune the hyperparameters.

1. First check the learning rate (η).
2. If that doesn't help, try another optimizer, and always retune the learning rate after changing any hyperparameter,
3. If the performance is still not great, try tuning model hyperparameters such as the number of layers, the number of neurons per layer, and the types of activation functions to use for each hidden layer.

You can also try tuning other hyperparameters, such as the batch size (it can be set in the `fit()` method using the `batch_size` argument, which defaults to 32).

Once you are satisfied with your model's validation accuracy, you should evaluate it on the test set to estimate the generalization error before you deploy the model to production. You can easily do this using the `evaluate()` method.

> It also supports several other arguments, such as `batch_size` and `sample_weight`.

It is common to get slightly lower performance on the test set than on the validation set, as hyperparameters are **tuned on the validation set**, not the test set. However, in this example, we did not do any hyperparameter tuning, so the lower accuracy is just bad luck.

> Resist the temptation to tweak the hyperparameters on the test set, or else your estimate of the generalization error will be too optimistic.

Using Model to Make Predictions

It is time to use the model's `predict()` method to make predictions on new instances. As we don't have actual new instances, we'll just use the first three (3) instances of the test set:

C.R. 22 python
```python
X_new = X_test[:3]
y_proba = model.predict(X_new)
print(y_proba.round(2))
```

C.R. 23 text
```
[[0.   0.   0.   0.   0.   0.12 0.   0.01 0.   0.87]
 [0.   0.   1.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.   0.   0.   0.   0.  ]]
```

For each instance the model estimates one probability per class, from class 0 to class 9. This is similar to the output of the `predict_proba()` method in `sklearn` classifiers.

For example, for the first image it estimates that the probability of class 9 (ankle boot) is 87%, the probability of class 7 (sneaker) is 1%, the probability of class 5 (sandal) is 12%, and the probabilities of the other classes are negligible.

In other words, it is highly confident that the first image is footwear, most likely ankle boots but possibly sneakers or sandals. If you only care about the class with the highest estimated probability (even if that probability is quite low), then you can use the `argmax()` method to get the highest probability class index for each instance:

C.R. 24 python
```python
y_pred = y_proba.argmax(axis=-1)
print(y_pred)
```

C.R. 25 text
```
[9 2 1]
```

Here, the classifier actually classified all three images correctly, where these images are shown in Fig. 13.12.

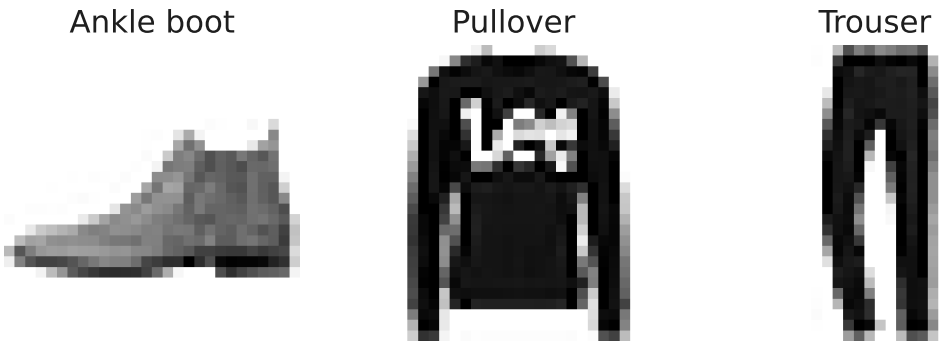


Figure 13.12.: Correctly classified Fashion MNIST images.

13.3.3 Building a Regression MLP Using the Sequential API

Instead of classifying categories, lets try to estimate a value. For this application, we need a different dataset. Let's switch back to the California housing problem and tackle it using the same MLP as earlier, with 3 hidden layers composed of 50 neurons each, but this time building it with `tf.keras`.

Using the sequential API to build, train, evaluate, and use a regression MLP is quite similar to what we did for classification. The main differences in the following code example are the fact that the output layer has a single neuron (since we only want to predict a single value) and it uses no activation function, the loss function is the mean squared error, the metric is the RMSE, and we're using an Adam optimizer like `sklearn`s `MLPRegressor` did.

In addition, in this example we don't need a Flatten layer, and instead we're using a Normalization layer as the first layer: it does the same thing as `sklearn`s `StandardScaler`, but it must be fitted to the training data using its `adapt()` method before you call the model's `fit()` method.

Let's look at the code:

C.R. 26 python

```python
housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)
```

C.R. 27 python

```python
tf.random.set_seed(42)
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([
    norm_layer,
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
```

C.R. 28

```python
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(1)
])
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])
norm_layer.adapt(X_train)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test, rmse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

As you can see, the sequential API is quite clean and straightforward. However, although Sequential models are extremely common, it is sometimes useful to build neural networks with more complex topologies, or with multiple inputs or outputs. For this purpose, Keras offers the functional API.

Chapter 14

pandas

Table of Contents

14.1 Introduction

The pandas DataFrame is a structure that contains two-dimensional data and its corresponding labels. DataFrames are widely used in data science, machine learning, scientific computing, and many other data-intensive fields.

DataFrames are similar to SQL [1] tables or the spreadsheets that you work with in Microsoft Excel or OpenOffice Calc. In many cases, DataFrames are faster, easier to use, and more powerful than tables or spreadsheets as they're an integral part of the Python and NumPy ecosystems. Things that can be done with pandas include:

[1] a domain-specific language used to manage data, especially in a relational database management system (RDBMS)

- Defines fundamental structures for working with data,
- Access methods that facilitate operations,
- Reading in data,
- Adjusting indices,
- Working with dates and time series,
- Sorting, grouping, re-ordering and general data manipulation,
- Sealing with missing values, etc., etc.

More sophisticated statistical functionality is left to other packages, such as statsmodels and scikit-learn, which are built on top of pandas.
Throughout the lecture, it will be assumed that the following imports have taken place and have a reasonable grasp on these packages

C.R. 1
python
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import requests
```

> Requests is a simple yet powerful python library to work with HTTP requests.

There are two (2) important data types defined by pandas:

Series Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index.

DataFrame a two-dimensional, tabular data structure with rows and columns. It is similar to a spreadsheet or a table in a relational database. The DataFrame has three main components: the data, which is stored in rows and columns; the rows, which are labeled by an index; and the columns, which are labeled and contain the actual data.

Let's start working with series.

14.2 Series

Series, as mentioned previously, is a one-dimensional array-like object that can hold data of any type (integer, float, string, etc.). It is labelled, meaning each element has a unique identifier called an index. Think of a Series as a column in a spreadsheet or a single column of a database table.

Series are a fundamental data structure in Pandas and are commonly used for data manipulation and analysis tasks. They can be created from lists, arrays, dictionaries, and existing Series objects. Series are also a building block for the more complex Pandas DataFrame, which is a two-dimensional table-like structure consisting of multiple Series objects.
We begin by creating a series of four random observations

C.R. 2
```python
s = pd.Series(np.random.randn(4), name='daily returns')
print(s)
```

```text
0   -0.775075
1    2.113545
2    0.256437
3    0.093216
Name: daily returns, dtype: float64
```

We can imagine the indices 0, 1, 2, 3 as indexing four (4) listed companies, and the values being daily returns on their shares. As with any respectable spreadsheet software, Pandas has a wide variety of operations thanks to being built on top of numpy. Let's do a simple multiplication of 100 of each element in the series

C.R. 3
```python
print(s * 100)
```

```text
0    -77.507464
1    211.354529
2     25.643659
3      9.321635
Name: daily returns, dtype: float64
```

We can take the absolute value of all values using `np.abs()`:

C.R. 4
```python
print(np.abs(s))
```

```text
0    0.775075
1    2.113545
2    0.256437
3    0.093216
Name: daily returns, dtype: float64
```

As a spreadsheet software, Series provide more than numpy arrays. Not only do they have some additional (statistically oriented) methods. For example here we use `describe()` to view some basic statistical details like percentile, mean, std, etc. of numeric values.

C.R. 5
```python
print(s.describe())
```

```text
count    4.000000
mean     0.422031
std      1.215157
min     -0.775075
25%     -0.123856
50%      0.174826
75%      0.720714
max      2.113545
Name: daily returns, dtype: float64
```

But their indices are more flexible. Here we can insert new values to its indices.

C.R. 6
```python
s.index = ['AMZN', 'AAPL', 'MSFT', 'GOOG']
print(s)
```

```text
AMZN   -0.775075
AAPL    2.113545
MSFT    0.256437
GOOG    0.093216
Name: daily returns, dtype: float64
```

It is not a hard strecth to think of Series as a fast, efficient Python dictionaries [2]. In fact, you can use much of the same syntax as Python dictionaries

[2] with the restriction that the items in the dictionary all have the same type, which for our example, floats.

C.R. 7
```python
print(s['AMZN'])
```

```text
-0.7750746403810563
```

C.R. 8
```python
s['AMZN'] = 0
print(s)
```

```text
AMZN    0.000000
AAPL    2.113545
MSFT    0.256437
GOOG    0.093216
Name: daily returns, dtype: float64
```

C.R. 9
```python
print('AAPL' in s)
```

```text
True
```

14.3 DataFrames

A DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the data, rows, and columns.

> In essence, a DataFrame in pandas is analogous to a (highly optimized) Excel spreadsheet.

Therefore, it is a powerful tool for representing and analyzing data that are naturally organized into rows and columns, often with descriptive indexes for individual rows and individual columns.

Let's look at an example that reads data from the CSV file `pandas/data/test/pwt.csv`, which is taken from the Penn World Tables.
The dataset contains the following indicators
We'll read this in from a URL using the pandas function `.read_csv()`.

C.R. 10
```python
df = pd.read_csv("https://raw.githubusercontent.com/QuantEcon/" + \
                 "lecture-python-programming/master/source/" + \
                 "_static/lecture_specific/pandas/data/test_pwt.csv")
print(type(df))
```

```text
<class 'pandas.core.frame.DataFrame'>
```

Here's the content of `test_pwt.csv`.

C.R. 11
```python
print(df)
```

```text
         country country isocode  year          POP       XRAT         tcgdp         cc
         ↪  cg
0      Argentina             ARG  2000    37335.653   0.999500  2.950722e+05  75.716805
↪  5.578804
1      Australia             AUS  2000    19053.186   1.724830  5.418047e+05  67.759026
↪  6.720098
2          India             IND  2000  1006300.297  44.941600  1.728144e+06  64.575551
↪  14.072206
3         Israel             ISR  2000     6114.570   4.077330  1.292539e+05  64.436451
↪  10.266688
4         Malawi             MWI  2000    11801.505  59.543808  5.026222e+03  74.707624
↪  11.658954
5   South Africa             ZAF  2000    45064.098   6.939830  2.272424e+05  72.718710
↪  5.726546
6  United States             USA  2000   282171.957   1.000000  9.898700e+06  72.347054
↪  6.032454
7        Uruguay             URY  2000     3219.793  12.099592  2.525596e+04  78.978740
↪  5.108068
```

14.3.1 Selecting Data by Position

In practice, one thing that we do all the time is to find, select and work with a subset of the data of our interests. We can select particular rows using standard Python array slicing notation.

C.R. 12
```python
print(df[2:5])
```

```text
  country country isocode  year          POP       XRAT         tcgdp         cc         cg
2   India             IND  2000  1006300.297  44.941600  1.728144e+06  64.575551  14.072206
3  Israel             ISR  2000     6114.570   4.077330  1.292539e+05  64.436451  10.266688
4  Malawi             MWI  2000    11801.505  59.543808  5.026222e+03  74.707624  11.658954
```

To select columns, we can pass a list containing the names of the desired columns represented as strings:

C.R. 13
```python
print(df[['country', 'tcgdp']])
```

```text
         country         tcgdp
0      Argentina  2.950722e+05
1      Australia  5.418047e+05
2          India  1.728144e+06
3         Israel  1.292539e+05
4         Malawi  5.026222e+03
5   South Africa  2.272424e+05
6  United States  9.898700e+06
7        Uruguay  2.525596e+04
```

To select both rows and columns using integers, the `iloc` attribute should be used with the format `.iloc[rows, columns]`.

C.R. 14
```python
print(df.iloc[2:5, 0:4])
```

```text
  country country isocode  year          POP
2   India             IND  2000  1006300.297
3  Israel             ISR  2000     6114.570
4  Malawi             MWI  2000    11801.505
```

To select rows and columns using a mixture of integers and labels, the `loc`attribute can be used in a similar way:

C.R. 15
```python
print(df.loc[df.index[2:5], ['country', 'tcgdp']])
```

```text
  country         tcgdp
2   India  1.728144e+06
3  Israel  1.292539e+05
4  Malawi  5.026222e+03
```

14.3.2 Select Data by Conditions

Instead of indexing rows and columns using integers and names, we can also obtain a sub-dataframe of our interests that satisfies certain (potentially complicated) conditions.
The most straightforward way is with the [] operator.

C.R. 16
```python
print(df[df.POP >= 20000])
```

```text
         country country isocode  year          POP      XRAT         tcgdp         cc
        ↪  cg
0      Argentina             ARG  2000    37335.653   0.99950  2.950722e+05  75.716805
↪  5.578804
2          India             IND  2000  1006300.297  44.94160  1.728144e+06  64.575551
↪  14.072206
```

```
5   South Africa            ZAF  2000    45064.098   6.93983  2.272424e+05  72.718710
↪  5.726546
6  United States            USA  2000   282171.957   1.00000  9.898700e+06  72.347054
↪  6.032454
```

To understand what is going on here, notice that `df.POP >= 20000` returns a series of boolean values.

C.R. 17
```python
print(df.POP >= 20000)
```

```text
0     True
1    False
2     True
3    False
4    False
5     True
6     True
7    False
Name: POP, dtype: bool
```

In this case, `df[]` takes a series of boolean values and only returns rows with the `True` values. Let's have one more example,

C.R. 18
```python
print(df[(df.country.isin(['Argentina', 'India', 'South Africa'])) & (df.POP > 40000)])
```

```text
        country country isocode  year          POP      XRAT         tcgdp         cc
        ↪  cg
2         India             IND  2000  1006300.297  44.94160  1.728144e+06  64.575551
↪  14.072206
5  South Africa             ZAF  2000    45064.098   6.93983  2.272424e+05  72.718710
↪  5.726546
```

However, there is another way of doing the same thing, which can be slightly faster for large dataframes, with more natural syntax.

C.R. 19
```python
print(df.query("POP >= 20000"))
```

```text
         country country isocode  year          POP      XRAT         tcgdp         cc
         ↪  cg
0      Argentina             ARG  2000    37335.653   0.99950  2.950722e+05  75.716805
↪  5.578804
2          India             IND  2000  1006300.297  44.94160  1.728144e+06  64.575551
↪  14.072206
5   South Africa             ZAF  2000    45064.098   6.93983  2.272424e+05  72.718710
↪  5.726546
6  United States             USA  2000   282171.957   1.00000  9.898700e+06  72.347054
↪  6.032454
```

C.R. 20
```python
print(df.query("country in ['Argentina', 'India', 'South Africa'] and POP > 40000"))
```

```text
        country country isocode  year          POP      XRAT         tcgdp         cc
        ↪  cg
2         India             IND  2000  1006300.297  44.94160  1.728144e+06  64.575551
↪  14.072206
5  South Africa             ZAF  2000    45064.098   6.93983  2.272424e+05  72.718710
↪  5.726546
```

We can also allow arithmetic operations between different columns.

C.R. 21
```python
print(df[(df.cc + df.cg >= 80) & (df.POP <= 20000)])
```

```text
   country country isocode  year        POP       XRAT         tcgdp         cc         cg
4   Malawi             MWI  2000  11801.505  59.543808   5026.221784  74.707624  11.658954
7  Uruguay             URY  2000   3219.793  12.099592  25255.961693  78.978740   5.108068
```

C.R. 22
```python
print(df.query("cc + cg >= 80 & POP <= 20000"))
```

```text
   country country isocode  year        POP       XRAT         tcgdp         cc         cg
4   Malawi             MWI  2000  11801.505  59.543808   5026.221784  74.707624  11.658954
7  Uruguay             URY  2000   3219.793  12.099592  25255.961693  78.978740   5.108068
```

For example, we can use the conditioning to select the country with the largest household consumption: gdp share cc.

C.R. 23
```python
print(df.loc[df.cc == max(df.cc)])
```

```text
   country country isocode  year       POP       XRAT         tcgdp        cc        cg
7  Uruguay             URY  2000  3219.793  12.099592  25255.961693  78.97874  5.108068
```

When we only want to look at certain columns of a selected sub-dataframe, we can use the above conditions with the `.loc[]` command. The first argument takes the condition, while the second argument takes a list of columns we want to return.

C.R. 24
```python
print(df.loc[(df.cc + df.cg >= 80) & (df.POP <= 20000), ['country', 'year', 'POP']])
```

```text
   country  year        POP
4   Malawi  2000  11801.505
7  Uruguay  2000   3219.793
```

14.4 Pandas for Panel Data

Econometricians often need to work with more complex data sets, such as panels. Common tasks include:

- Importing data, cleaning it and reshaping it across several axes.
- Selecting a time series or cross-section from a panel.
- Grouping and summarizing data.

pandas contains powerful and easy-to-use tools for solving exactly these kinds of problems. In what follows, we will use a panel data set of real minimum wages from the OECD to create:

- summary statistics over multiple dimensions of our data
- a time series of the average minimum wage of countries in the dataset
- kernel density estimates of wages by continent

We will begin by reading in our long format panel data from a `.csv` file and reshaping the resulting DataFrame with `pivot_table` to build a `MultiIndex`. Additional detail will be added to our DataFrame using pandas' `merge` function, and data will be summarized with the `groupby` function.

C.R. 25
```python
import matplotlib.pyplot as plt
```

14.4.1 Slicing and Reshaping Data

We will read in a dataset from the OECD of real minimum wages in 32 countries and assign it to `realwage`. The dataset can be accessed with the following link:

C.R. 26
```python
url1 = "https://raw.githubusercontent.com/" + \
    "QuantEcon/lecture-python/master/source/" + \
    "_static/lecture_specific/pandas_panel/realwage.csv"
```

C.R. 27
```python
import pandas as pd

# Display 6 columns for viewing purposes
pd.set_option('display.max_columns', 6)

# Reduce decimal points to 2
pd.options.display.float_format = '{:,.2f}'.format

realwage = pd.read_csv(url1)
```

Let's have a look at what we've got to work with which its result can be seen in

C.R. 28
```python
print(realwage.head())  # Show first 5 rows
```

The data is currently in long format, which is difficult to analyze when there are several dimensions to the data. To simplify, we will use `pivot_table` to create a wide format panel,

with a MultiIndex to handle higher dimensional data. `pivot_table` arguments should specify the data (values), the index, and the columns we want in our resulting dataframe.

By passing a list in columns, we can create a MultiIndex in our column axis:

C.R. 29 python

```python
realwage = realwage.pivot_table(values='value',
                                index='Time',
                                columns=['Country', 'Series', 'Pay period'])
realwage.head()
```

text

```
realwage.index = pd.to_datetime(realwage.index)
print(type(realwage.index))
#+end_src

#+RESULTS: PP-6
#+begin_example
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
#+end_example

#+NAME: PP-7
#+begin_src python :session :results output
print(type(realwage.columns))
#+end_src

#+RESULTS: PP-7
#+begin_example
<class 'pandas.core.indexes.multi.MultiIndex'>
#+end_example

#+NAME: PP-8
#+begin_src python :session :results output
print(realwage.columns.names)
#+end_src

#+RESULTS: PP-8
#+begin_example
['Country', 'Series', 'Pay period']
#+end_example

#+NAME: PP-9
#+begin_src python :session :results output
print(realwage['United States'].head())
#+end_src

#+RESULTS: PP-9
#+begin_example
Series     In 2015 constant prices at 2015 USD PPPs         \
```

14.5 Application: Long-Run Growth

In this section we will use pandas and matplotlib to download, organize, and visualise historical data on economic growth. In addition to learning how to deploy these tools more generally, we'll use them to describe facts about economic growth experiences across many countries

over several centuries.
Such growth facts are interesting for a variety of reasons:

- They explain growth facts is a principal purpose of both "development economics" and "economic history".
- They are important inputs into historians' studies of geopolitical forces and dynamics.

Thus, Adam Tooze's account of the geopolitical precedents and antecedents of World War I begins by describing how the Gross Domestic Products (GDP) of European Great Powers had evolved during the 70 years preceding 1914 (see chapter 1 of [Tooze, 2014]). Using the very same data that Tooze used to construct his figure (with a slightly longer timeline), here is our version of his chapter 1 figure.
Chapter 1 of [Tooze, 2014] used his graph to show how US GDP started the 19th century way behind the GDP of the British Empire.
By the end of the nineteenth century, US GDP had caught up with GDP of the British Empire, and how during the first half of the 20th century, US GDP surpassed that of the British Empire.
For Adam Tooze, that fact was a key geopolitical underpinning for the "American century".
Looking at this graph and how it set the geopolitical stage for "the American (20th) century" naturally tempts one to want a counterpart to his graph for 2014 or later.
(An impatient reader seeking a hint at the answer might now want to jump ahead and look at figure Fig. 2.7.)
As we'll see, reasoning by analogy, this graph perhaps set the stage for an "XXX (21st) century", where you are free to fill in your guess for country XXX.
As we gather data to construct those two graphs, we'll also study growth experiences for a number of countries for time horizons extending as far back as possible.
These graphs will portray how the "Industrial Revolution" began in Britain in the late 18th century, then migrated to one country after another.
In a nutshell, this lecture records growth trajectories of various countries over long time periods.
While some countries have experienced long-term rapid growth across that has lasted a hundred years, others have not.
Since populations differ across countries and vary within a country over time, it will be interesting to describe both total GDP and GDP per capita as it evolves within a country.
First let's import the packages needed to explore what the data says about long-run growth

C.R. 30
python
```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
from collections import namedtuple
```

A project initiated by *Angus Maddison* has collected many historical time series related to economic growth, some dating back to the first century. The data can be downloaded from the Maddison Historical Statistics by clicking on the "Latest Maddison Project Release".

We are going to read the data from a QuantEcon GitHub repository. Our objective in this section is to produce a convenient DataFrame instance that contains per capita GDP for different

countries. Here we read the Maddison data into a pandas DataFrame:

C.R. 31

```python
data_url = "https://github.com/QuantEcon/lecture-python-intro/raw/main/lectures/datasets/mpd2020.xlsx"
data = pd.read_excel(data_url,
                     sheet_name='Full data')
print(data.head())
```

```text
  countrycode      country  year    gdppc      pop
0         AFG  Afghanistan  1820      NaN 3,280.00
1         AFG  Afghanistan  1870      NaN 4,207.00
2         AFG  Afghanistan  1913      NaN 5,730.00
3         AFG  Afghanistan  1950 1,156.00 8,150.00
4         AFG  Afghanistan  1951 1,170.00 8,284.00
```

We can see that this dataset contains GDP per capita (gdppc) and population (pop) for many countries and years. Let's look at how many and which countries are available in this dataset:

C.R. 32

```python
countries = data.country.unique()
print(len(countries))
```

```text
169
```

We can now explore some of the 169 countries that are available. Let's loop over each country to understand which years are available for each country

C.R. 33

```python
country_years = []
for country in countries:
    cy_data = data[data.country == country]['year']
    ymin, ymax = cy_data.min(), cy_data.max()
    country_years.append((country, ymin, ymax))
country_years = pd.DataFrame(country_years,
                    columns=['country', 'min_year', 'max_year']).set_index('country')
print(country_years.head())
```

```text
                     min_year  max_year
country
Afghanistan              1820      2018
Angola                   1950      2018
Albania                     1      2018
United Arab Emirates     1950      2018
Argentina                1800      2018
```

Let's now reshape the original data into some convenient variables to enable quicker access to countries' time series data. We can build a useful mapping between country codes and country names in this dataset:

C.R. 34
```python
code_to_name = data[
    ['countrycode',
    'country']].drop_duplicates().reset_index(drop=True).set_index(['countrycode'])
```

Now we can focus on GDP per capita (`gdppc`) and generate a wide data format

C.R. 35
```python
gdp_pc = data.set_index(['countrycode', 'year'])['gdppc']
gdp_pc = gdp_pc.unstack('countrycode')
```

C.R. 36
```python
print(gdp_pc.tail())
```

```text
countrycode         AFG         AGO          ALB          ARE          ARG          ARM         AUS
AUT          AZE  ...          USA          UZB          VEN          VNM          YEM
YUG          ZAF         ZMB         ZWE
year
...
2014         2022.0000  8673.0000   9808.0000  72601.0000  19183.0000   9735.0000  47867.0000
41338.0000  17439.0000  ...  51664.0000   9085.0000  20317.0000  5455.0000  4054.0000
14627.0000  12242.0000  3478.0000  1594.0000
2015         1928.0000  8689.0000  10032.0000  74746.0000  19502.0000  10042.0000  48357.0000
41294.0000  17460.0000  ...  52591.0000   9720.0000  18802.0000  5763.0000  2844.0000
14971.0000  12246.0000  3478.0000  1560.0000
2016         1929.0000  8453.0000  10342.0000  75876.0000  18875.0000  10080.0000  48845.0000
41445.0000  16645.0000  ...  53015.0000  10381.0000  15219.0000  6062.0000  2506.0000
15416.0000  12139.0000  3479.0000  1534.0000
2017         2014.7453  8146.4354  10702.1201  76643.4984  19200.9061  10859.3783  49265.6135
42177.3706  16522.3072  ...  54007.7698  10743.8666  12879.1350  6422.0865  2321.9239
15960.8432  12189.3579  3497.5818  1582.3662
2018         1934.5550  7771.4418  11104.1665  76397.8181  18556.3831  11454.4251  49830.7993
42988.0709  16628.0553  ...  55334.7394  11220.3702  10709.9506  6814.1423  2284.8899
16558.3123  12165.7948  3534.0337  1611.4052
[5 rows x 169 columns]
```

We create a variable `color_mapping` to store a map between country codes and colors for consistency:

C.R. 37
```python
country_names = data['countrycode']

# Generate a colormap with the number of colors matching the number of countries
colors = cm.tab20(np.linspace(0, 0.95, len(country_names)))

# Create a dictionary to map each country to its corresponding color
color_mapping = {country: color for
                 country, color in zip(country_names, colors)}
```

14.5.1 GDP per Capita

In this section we examine GDP per capita over the long run for several different countries.

United Kingdom

First we examine UK GDP growth:

C.R. 38

```python
fig, ax = plt.subplots(figsize = (15,6))
country = 'GBR'
gdp_pc[country].plot(
        ax=ax,
        ylabel='international dollars',
        xlabel='year',
        color=color_mapping[country]
    );
```

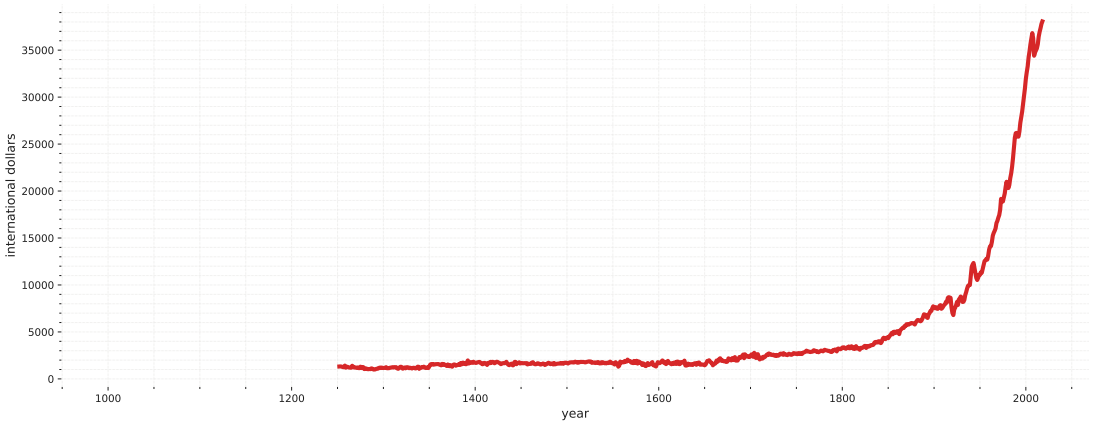


Figure 14.1.: GDP per Capita of United Kingdom.

> International dollars are a hypothetical unit of currency that has the same purchasing power parity that the U.S. Dollar has in the United States at a given point in time. They are also known as Geary–Khamis dollars (GK Dollars).

We can see that the data is non-continuous for longer periods in the early 250 years of this millennium, so we could choose to interpolate to get a continuous line plot. Here we use dashed lines to indicate interpolated trends:

C.R. 39

```python
fig, ax = plt.subplots(figsize = (15,6))
country = 'GBR'
ax.plot(gdp_pc[country].interpolate(),
        linestyle='--',
        lw=2,
        color=color_mapping[country])

ax.plot(gdp_pc[country],
        lw=2,
        color=color_mapping[country])
ax.set_ylabel('international dollars')
ax.set_xlabel('year')
```

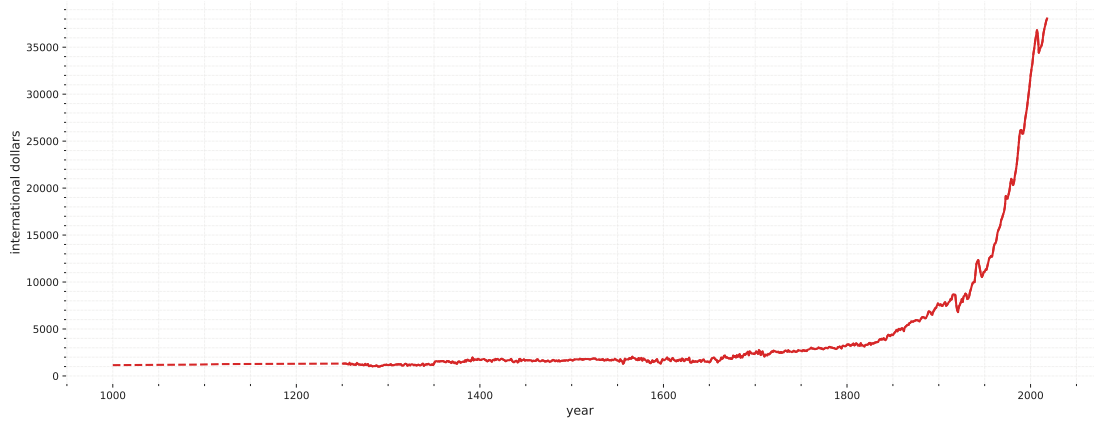


Figure 14.2.: GDP per Capita of United Kingdom with continuation.

Comparing the US, UK, and China

In this section we will compare GDP growth for the US, UK and China. As a first step we create a function to generate plots for a list of countries.

C.R. 40

python

```python
def draw_interp_plots(series,        # pandas series
                      country,       # list of country codes
                      ylabel,        # label for y-axis
                      xlabel,        # label for x-axis
                      color_mapping, # code-color mapping
                      code_to_name,  # code-name mapping
                      lw,            # line width
                      logscale,      # log scale for y-axis
                      ax             # matplolib axis
                     ):

    for c in country:
        # Get the interpolated data
        df_interpolated = series[c].interpolate(limit_area='inside')
        interpolated_data = df_interpolated[series[c].isnull()]

        # Plot the interpolated data with dashed lines
        ax.plot(interpolated_data,
                linestyle='--',
                lw=lw,
                alpha=0.7,
                color=color_mapping[c])

        # Plot the non-interpolated data with solid lines
        ax.plot(series[c],
                lw=lw,
                color=color_mapping[c],
                alpha=0.8,
                label=code_to_name.loc[c]['country'])

        if logscale:
            ax.set_yscale('log')
```

C.R. 41
python
```
    # Draw the legend outside the plot
    ax.legend(loc='upper left', frameon=False)
    ax.set_ylabel(ylabel)
    ax.set_xlabel(xlabel)
```

As you can see from this chart, economic growth started in earnest in the 18th century and continued for the next two hundred years.How does this compare with other countries' growth trajectories?

Let's look at the United States (USA), United Kingdom (GBR), and China (CHN)

C.R. 42
python
```
# Define the namedtuple for the events
Event = namedtuple('Event', ['year_range', 'y_text', 'text', 'color', 'ymax'])

fig, ax = plt.subplots(figsize = (15,6))

country = ['CHN', 'GBR', 'USA']
draw_interp_plots(gdp_pc[country].loc[1500:],
                  country,
                  'international dollars','year',
                  color_mapping, code_to_name, 2, False, ax)

# Define the parameters for the events and the text
ylim = ax.get_ylim()[1]
b_params = {'color':'grey', 'alpha': 0.2}
t_params = {'fontsize': 9,
            'va':'center', 'ha':'center'}

# Create a list of events to annotate
events = [
    Event((1650, 1652), ylim + ylim*0.04,
          'the Navigation Act\n(1651)',
          color_mapping['GBR'], 1),
    Event((1655, 1684), ylim + ylim*0.13,
          'Closed-door Policy\n(1655-1684)',
          color_mapping['CHN'], 1.1),
    Event((1848, 1850), ylim + ylim*0.22,
          'the Repeal of Navigation Act\n(1849)',
          color_mapping['GBR'], 1.18),
    Event((1765, 1791), ylim + ylim*0.04,
          'American Revolution\n(1765-1791)',
          color_mapping['USA'], 1),
    Event((1760, 1840), ylim + ylim*0.13,
          'Industrial Revolution\n(1760-1840)',
          'grey', 1.1),
    Event((1929, 1939), ylim + ylim*0.04,
          'the Great Depression\n(1929-1939)',
          'grey', 1),
    Event((1978, 1979), ylim + ylim*0.13,
          'Reform and Opening-up\n(1978-1979)',
          color_mapping['CHN'], 1.1)
]

```

C.R. 43

python

```
def draw_events(events, ax):
    # Iterate over events and add annotations and vertical lines
    for event in events:
        event_mid = sum(event.year_range)/2
        ax.text(event_mid,
                event.y_text, event.text,
                color=event.color, **t_params)
        ax.axvspan(*event.year_range, color=event.color, alpha=0.2)
        ax.axvline(event_mid, ymin=1, ymax=event.ymax, color=event.color,
                   clip_on=False, alpha=0.15)

# Draw events
draw_events(events, ax)
```

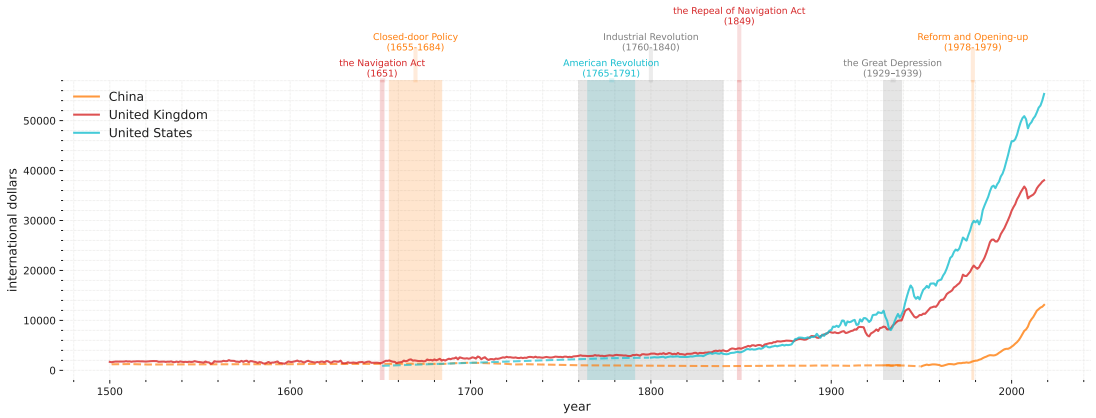


Figure 14.3.: GDP per capita of UK, US, China with important economic event imposed.

The preceding graph of per capita GDP strikingly reveals how the spread of the Industrial Revolution has over time gradually lifted the living standards of substantial groups of people

- most of the growth happened in the past 150 years after the Industrial Revolution.
- per capita GDP in the US and UK rose and diverged from that of China from 1820 to 1940.
- the gap has closed rapidly after 1950 and especially after the late 1970s.
- these outcomes reflect complicated combinations of technological and economic-policy factors that students of economic growth try to understand and quantify.

Focusing on China

It is fascinating to see China's GDP per capita levels from 1500 through to the 1970s. Notice the long period of declining GDP per capital levels from the 1700s until the early 20th century. Thus, the graph indicates

- a long economic downturn and stagnation after the Closed-door Policy by the Qing government.
- China's very different experience than the UK's after the onset of the industrial revolution in the UK.

- how the Self-Strengthening Movement seemed mostly to help China to grow.
- how stunning have been the growth achievements of modern Chinese economic policies by the PRC that culminated with its late 1970s reform and liberalization.

C.R. 44
python
```
fig, ax = plt.subplots(figsize = (15,6))

country = ['CHN']
draw_interp_plots(gdp_pc[country].loc[1600:2000],
                  country,
                  'international dollars','year',
                  color_mapping, code_to_name, 2, True, ax)

ylim = ax.get_ylim()[1]

events = [
Event((1655, 1684), ylim + ylim*0.06,
      'Closed-door Policy\n(1655-1684)',
      'tab:orange', 1),
Event((1760, 1840), ylim + ylim*0.06,
      'Industrial Revolution\n(1760-1840)',
      'grey', 1),
Event((1839, 1842), ylim + ylim*0.2,
      'First Opium War\n(1839-1842)',
      'tab:red', 1.07),
Event((1861, 1895), ylim + ylim*0.4,
      'Self-Strengthening Movement\n(1861-1895)',
      'tab:blue', 1.14),
Event((1939, 1945), ylim + ylim*0.06,
      'WW 2\n(1939-1945)',
      'tab:red', 1),
Event((1948, 1950), ylim + ylim*0.23,
      'Founding of PRC\n(1949)',
      color_mapping['CHN'], 1.08),
Event((1958, 1962), ylim + ylim*0.5,
      'Great Leap Forward\n(1958-1962)',
      'tab:orange', 1.18),
Event((1978, 1979), ylim + ylim*0.7,
      'Reform and Opening-up\n(1978-1979)',
      'tab:blue', 1.24)
]

# Draw events
draw_events(events, ax)
```

Focusing on the US and UK

Now we look at the United States (USA) and United Kingdom (GBR) in more detail. In the following graph, please watch for:

- impact of trade policy (Navigation Act).
- productivity changes brought by the Industrial Revolution.
- how the US gradually approaches and then surpasses the UK, setting the stage for the "Amer-

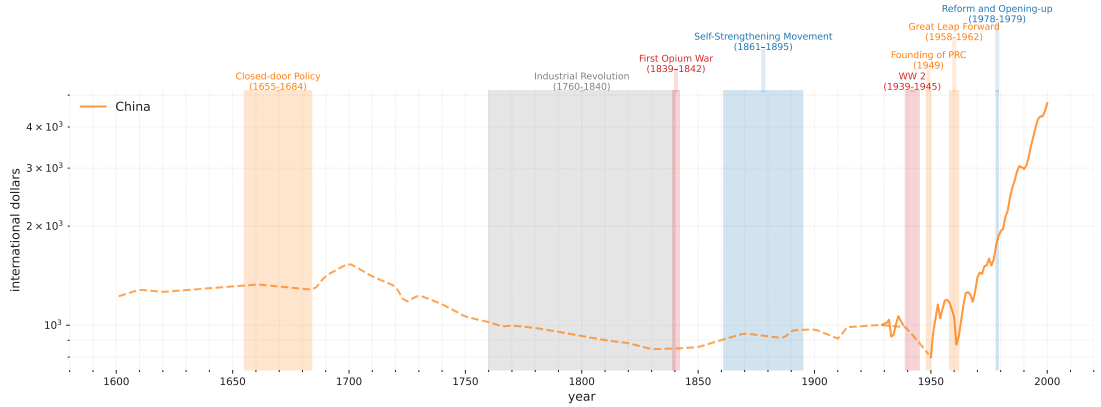


Figure 14.4.: GDP per capita of China with important economic event imposed.

ican Century".

- the often unanticipated consequences of wars.
- interruptions and scars left by business cycle recessions and depressions.

C.R. 45 python

```python
fig, ax = plt.subplots(figsize = (15,6))

country = ['GBR', 'USA']
draw_interp_plots(gdp_pc[country].loc[1500:2000],
                  country,
                  'international dollars','year',
                  color_mapping, code_to_name, 2, True, ax)

ylim = ax.get_ylim()[1]

# Create a list of data points
events = [
    Event((1651, 1651), ylim + ylim*0.15,
          'Navigation Act (UK)\n(1651)',
          'tab:orange', 1),
    Event((1765, 1791), ylim + ylim*0.15,
          'American Revolution\n(1765-1791)',
          color_mapping['USA'], 1),
    Event((1760, 1840), ylim + ylim*0.6,
          'Industrial Revolution\n(1760-1840)',
          'grey', 1.08),
    Event((1848, 1850), ylim + ylim*1.1,
          'Repeal of Navigation Act (UK)\n(1849)',
          'tab:blue', 1.14),
    Event((1861, 1865), ylim + ylim*1.8,
          'American Civil War\n(1861-1865)',
          color_mapping['USA'], 1.21),
    Event((1914, 1918), ylim + ylim*0.15,
          'WW 1\n(1914-1918)',
          'tab:red', 1),
    Event((1929, 1939), ylim + ylim*0.6,
          'the Great Depression\n(1929-1939)',
```

C.R. 46
python
```
            'grey', 1.08),
    Event((1939, 1945), ylim + ylim*1.1,
            'WW 2\n(1939-1945)',
            'tab:red', 1.14)
]

# Draw events
draw_events(events, ax)
```

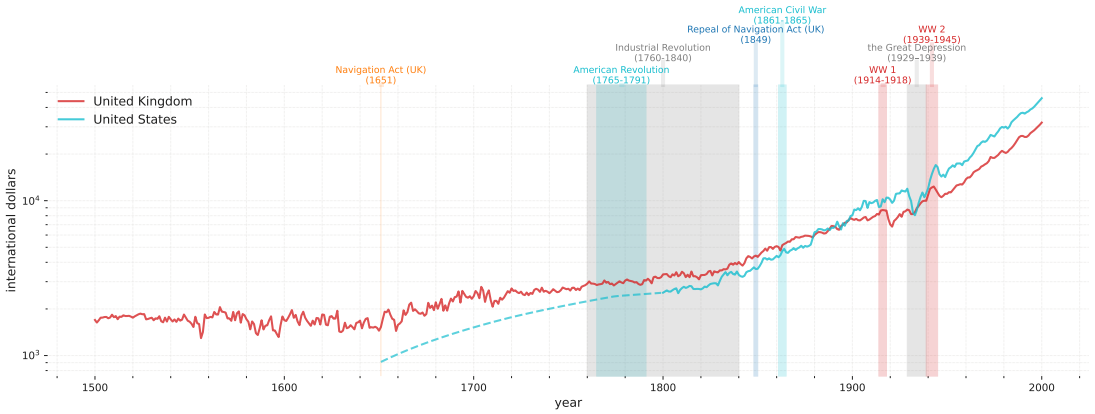


Figure 14.5.: GDP per capita of US and UK with important economic event imposed.

14.5.2 GDP Growth

Now we'll construct some graphs of interest to geopolitical historians like Adam Tooze. We'll focus on total Gross Domestic Product (GDP) (as a proxy for "national geopolitical-military power") rather than focusing on GDP per capita (as a proxy for living standards).

C.R. 47
python
```
data = pd.read_excel(data_url, sheet_name='Full data')
data.set_index(['countrycode', 'year'], inplace=True)
data['gdp'] = data['gdppc'] * data['pop']
gdp = data['gdp'].unstack('countrycode')
```

Early industrialization (1820 to 1940)

We first visualize the trend of China, the Former Soviet Union, Japan, the UK and the US. The most notable trend is the rise of the US, surpassing the UK in the 1860s and China in the 1880s. The growth continued until the large dip in the 1930s when the Great Depression hit. Meanwhile, Russia experienced significant setbacks during World War I and recovered significantly after the February Revolution.

C.R. 48
python
```
fig, ax = plt.subplots(figsize = (15,6))
country = ['CHN', 'SUN', 'JPN', 'GBR', 'USA']
start_year, end_year = (1820, 1945)
draw_interp_plots(gdp[country].loc[start_year:end_year],
```

C.R. 49 python
```
                  country,
                  'international dollars', 'year',
                  color_mapping, code_to_name, 2, False, ax)
```

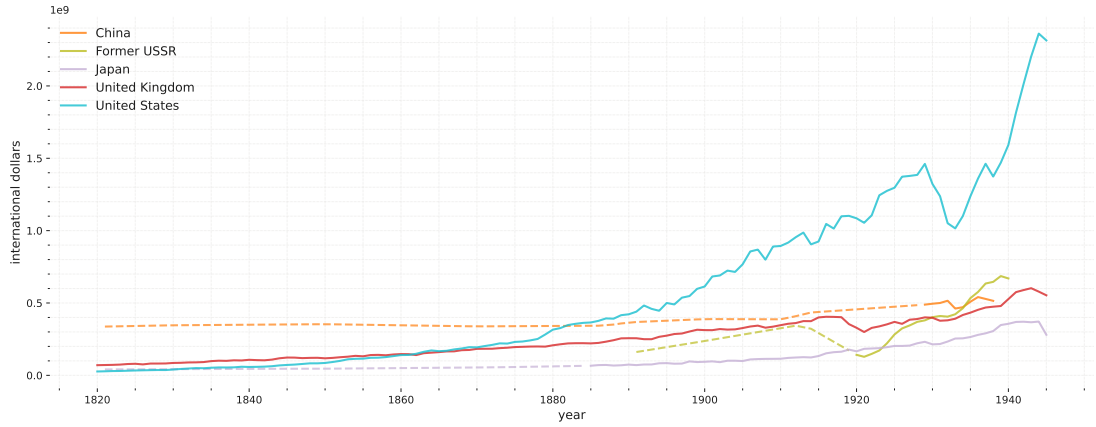


Figure 14.6.: GDP in the early industrialization era.

Constructing a plot similar to Tooze's In this section we describe how we have constructed a version of the striking figure we discussed at the start of this lecture. Let's first define a collection of countries that consist of the British Empire (BEM) so we can replicate that series in Tooze's chart.

C.R. 50 python
```
BEM = ['GBR', 'IND', 'AUS', 'NZL', 'CAN', 'ZAF']
# Interpolate incomplete time-series
gdp['BEM'] = gdp[BEM].loc[start_year-1:end_year].interpolate(method='index').sum(axis=1)
```

Now let's assemble our series and get ready to plot them.

C.R. 51 python
```
# Define colour mapping and name for BEM
color_mapping['BEM'] = color_mapping['GBR']  # Set the color to be the same as Great Britain
# Add British Empire to code_to_name
bem = pd.DataFrame(["British Empire"], index=["BEM"], columns=['country'])
bem.index.name = 'countrycode'
code_to_name = pd.concat([code_to_name, bem])
```

C.R. 52 python
```
fig, ax = plt.subplots(figsize = (15,6))
country = ['DEU', 'USA', 'SUN', 'BEM', 'FRA', 'JPN']
start_year, end_year = (1821, 1945)
draw_interp_plots(gdp[country].loc[start_year:end_year],
                  country,
                  'international dollars', 'year',
                  color_mapping, code_to_name, 2, False, ax)
```

At the start of this lecture, we noted how US GDP came from "nowhere" at the start of the 19th century to rival and then overtake the GDP of the British Empire by the end of the 19th century,

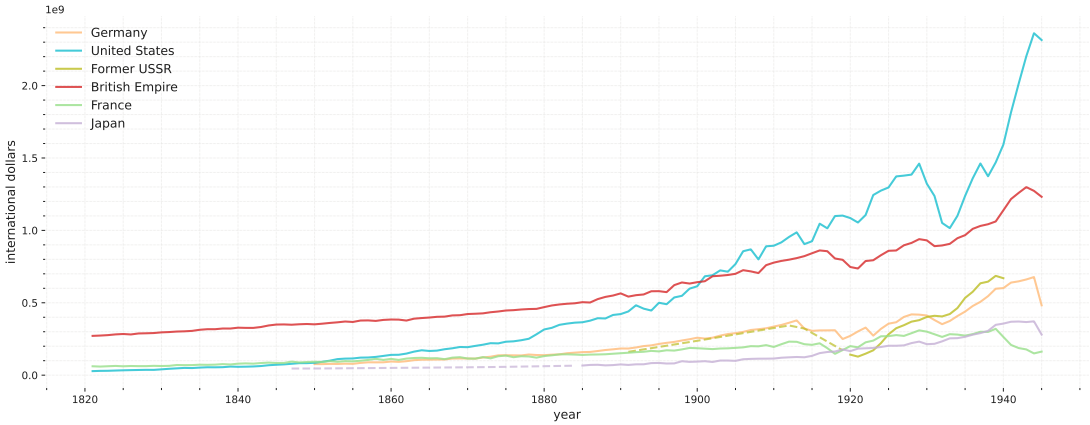


Figure 14.7.

setting the geopolitical stage for the "American (twentieth) century". Let's move forward in time and start roughly where Tooze's graph stopped after World War II. In the spirit of Tooze's chapter 1 analysis, doing this will provide some information about geopolitical realities today.

The modern era (1950 to 2020)

The following graph displays how quickly China has grown, especially since the late 1970s.

C.R. 53 python

```
fig, ax = plt.subplots(figsize = (15,6))
country = ['CHN', 'SUN', 'JPN', 'GBR', 'USA']
start_year, end_year = (1950, 2020)
draw_interp_plots(gdp[country].loc[start_year:end_year],
                  country,
                  'international dollars', 'year',
                  color_mapping, code_to_name, 2, False, ax)
```

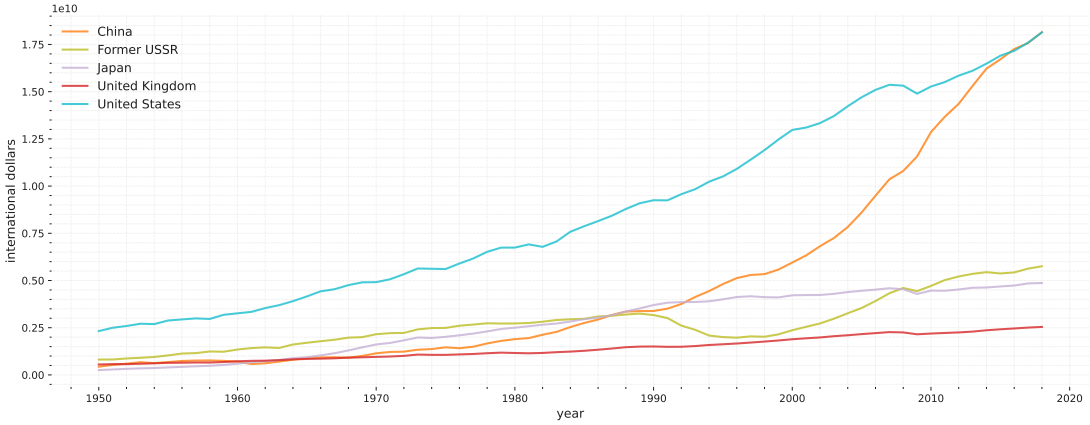


Figure 14.8.: GDP in the modern era.

It is tempting to compare this graph with figure Fig. 2.6 that showed the US overtaking the UK near the start of the "American Century", a version of the graph featured in chapter 1 of [Tooze, 2014].

14.5.3 Regional Analysis

We often want to study the historical experiences of countries outside the club of "World Powers". The Maddison Historical Statistics dataset also includes regional aggregations

C.R. 54
python
```
data = pd.read_excel(data_url,
                     sheet_name='Regional data',
                     header=(0,1,2),
                     index_col=0)
data.columns = data.columns.droplevel(level=2)
```

We can save the raw data in a more convenient format to build a single table of regional GDP per capita.

C.R. 55
python
```
regionalgdp_pc = data['gdppc_2011'].copy()
regionalgdp_pc.index = pd.to_datetime(regionalgdp_pc.index, format='%Y')
```

Let's interpolate based on time to fill in any gaps in the dataset for the purpose of plotting.

C.R. 56
python
```
regionalgdp_pc.interpolate(method='time', inplace=True)
```

Looking more closely, let's compare the time series for Western Offshoots and Sub-Saharan Africa with a number of different regions around the world. Again we see the divergence of the West from the rest of the world after the Industrial Revolution and the convergence of the world after the 1950s.

C.R. 57
python
```
fig, ax = plt.subplots(figsize = (15,6))
regionalgdp_pc.plot(ax=ax, xlabel='year',
                    lw=2,
                    ylabel='international dollars')
ax.set_yscale('log')
plt.legend(loc='lower center',
           ncol=3, bbox_to_anchor=[0.5, -0.5])
```

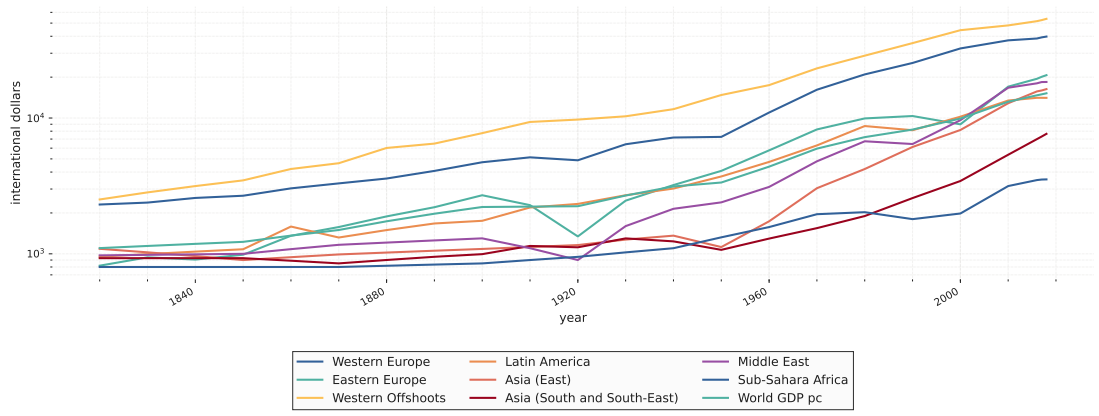


Figure 14.9.: Regional GDP per capita.

Glossary

ANN Artificial Neural Networks. 9–11, 13, 17, 20

DNN Deep Neural Networks. 17, 20

FNN Feedforward Neural Networks. 17

LTU Linear Threshold Unit. 13

ML Machine Learning. 9–11, 14

MLP Multi-layer Perceptrons. 3, 9, 17, 19–25, 27, 29, 31, 33

SVM Support Vector Machines. 11, 14

TLU Threshold Logic Unit. 13–15, 17

Bibliography

[1] S Agatonovic-Kustrin and Rosemary Beresford. “Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research”. In: *Journal of pharmaceutical and biomedical analysis* 22.5 (2000), pp. 717–727.

[2] David GT Barrett, Ari S Morcos, and Jakob H Macke. “Analyzing biological and artificial neural networks: challenges with opportunities for synergy?” In: *Current opinion in neurobiology* 55 (2019), pp. 55–64.

[3] BBC. *How a kingfisher helped reshape Japan’s bullet train*. 2019. URL: https://www.bbc.com/news/av/science-environment-47673287.

[4] George Bebis and Michael Georgiopoulos. “Feed-forward neural networks”. In: *Ieee Potentials* 13.4 (1994), pp. 27–31.

[5] Hans-Dieter Block. “The perceptron: A model for brain functioning. i”. In: *Reviews of Modern Physics* 34.1 (1962), p. 123.

[6] Ronald A Fisher. “The use of multiple measurements in taxonomic problems”. In: *Annals of eugenics* 7.2 (1936), pp. 179–188.

[7] Wulfram Gerstner and Werner M Kistler. “Mathematical formulations of Hebbian learning”. In: *Biological cybernetics* 87.5 (2002), pp. 404–415.

[8] Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. “Qualitatively characterizing neural network optimization problems”. In: *arXiv preprint arXiv:1412.6544* (2014).

[9] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

[10] Robert Hecht-Nielsen. “Theory of the backpropagation neural network”. In: *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[11] Dan Hendrycks and Kevin Gimpel. “Gaussian error linear units (gelus)”. In: *arXiv preprint arXiv:1606.08415* (2016).

[12] Geoffrey Hinton et al. “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups”. In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97.

[13] Sean D Holcomb et al. “Overview on deepmind and its alphago zero ai”. In: *Proceedings of the 2018 international conference on big data and education*. 2018, pp. 67–71.

[14] Jim Howe. “Artificial intelligence at edinburgh university: A perspective”. In: *Archived from the original on* 17 (2007).

[15] Hinkelmann Knut. “Neural Networks p. 7”. In: *University of Applied Sciences Northwestern Switzerland* (2018).

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[17] Seppo Linnainmaa. "Algoritmin kumulatiivinen pyöristysvirhe yksittäisten pyöristysvirheiden Taylor-kehitelmänä". Available in Finnish at `https://people.idsia.ch/~juergen/linnainmaa1970thesis.pdf`. Master's thesis. University of Helsinki, 1970.

[18] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[19] Marius-Constantin Popescu et al. "Multilayer perceptron and neural networks". In: *WSEAS Transactions on Circuits and Systems* 8.7 (2009), pp. 579–588.

[20] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[21] Wojciech Samek et al. "Explaining deep neural networks and beyond: A review of methods and applications". In: *Proceedings of the IEEE* 109.3 (2021), pp. 247–278.

[22] Arnau Sebé-Pedrós. "Stepwise emergence of the neuronal gene expression program in early animal evolution". In: (2023).

[23] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. "Activation functions in neural networks". In: *Towards Data Sci* 6.12 (2017), pp. 310–316.

[24] Haim Sompolinsky. "The theory of neural networks: The Hebb rule and beyond". In: *Heidelberg Colloquium on Glassy Dynamics: Proceedings of a Colloquium on Spin Glasses, Optimization and Neural Networks Held at the University of Heidelberg June 9–13, 1986*. Springer. 2006, pp. 485–527.

[25] Julian FV Vincent et al. "Biomimetics: its practice and theory". In: *Journal of the Royal Society Interface* 3.9 (2006), pp. 471–482.