Lecture Book
Python for Engineering and Economics

Daniel McGuiness

version 2024

Contents

-- **org-support-shift-select: t; auto-fill-mode: t; --**

Chapter 1.

Welcome

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 Python's Flying Circus.

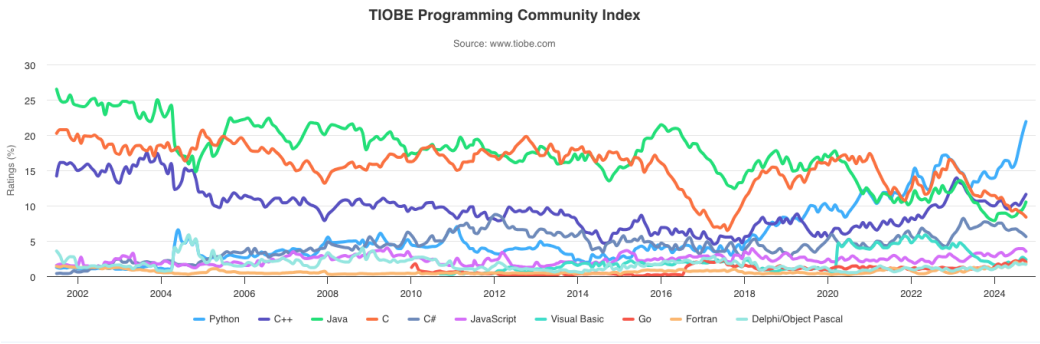
The popularity of Python has grown strongly in recent years, as confirmed by the survey results provided since 2011 by Stack Overflow. Stack Overflow offers its users the opportunity to complete a survey in which they are asked many questions to describe their

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++.

```cpp
#include <iostream>

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

And here is in Java, another major programming language:

```java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
```

```
    }
}
```

And finally, here is in Python:

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

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.

```
username = input("Enter username:")
print("Welcome to the class: " + username + "!")
```

Chapter 2.

Data Types

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:

```
print(type(42))
```

```
<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:

```
42    # Prints out 42
-84   # Prints out -84
0     # Prints out 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:

```
123123123123123123123123123123123123123123123123 + 1
```

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

```
123 ** 10000
```

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:

```
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

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

```
10  # Base 10
print(type(10))

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

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

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

```
<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:

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

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.

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**:

```
(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.

```
42.as_integer_ratio()
  File "<input>", line 1
    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:

```
signal = 0b11010110
set_bits = signal.bit_count()

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

```
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`:

```
print(int())
```

```
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:

```
print(int(42.0))
print(int("42"))
print(int("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:

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

```
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

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:

```
print(type(1.0))
```

```
<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:

```
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:

```
-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**.

```
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:

```
.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:

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

```
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:

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

```
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

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:

```
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:

```
print((42.0).is_integer())
print((42.42).is_integer())
```

```
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:

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

```
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.

- **"ox"** 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`:

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

```
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:

```
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:

```
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 $a + b$**j**, where a is the real part, and b**j** is the imaginary part:

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

```
(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 $a + b\mathbf{j}$.

```
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:

```
print(complex())
print(complex(1))
print(complex(0, 7))
print(complex(2, 3))
print(complex(2.4, 7.5))
print(complex(5, -3))
```

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

When you call `complex()` with no argument, you get $0\mathbf{j}$. If you call the function with a single argument, that argument is the real part, and the imaginary part will be $0\mathbf{j}$. 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:

```
print(complex("5-3j"))
print(complex("5"))
```

```
print(complex("5 - 3j"))
print(complex("5", "3"))
```

```
(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 (\) 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

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 `\"` 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 \t escape sequence:

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

```
Before After
```

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

Of these escape sequences, the newline or linefeed character (\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

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:

Method	Result
`.endswith(suffix[, start[, end]])`	True if the string ends with the specified suffix, False otherwise

Continued on next page

Continued from previous page

Method	Result
`.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

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
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:

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

Continued on next page

Continued from previous page

Method	Description
`.translate(table)`	Returns a translated string

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)

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`

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, `\xc3\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
    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

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 colors. 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:

```
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. In the rest of this tutorial, you'll dive deeper into all the above characteristics of lists and more. Are you ready? To kick things off, you'll start by learning the different ways to create lists.

3.2 Constructing Lists in Python

First things first. 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, you'll learn how to use the three (3_) tools listed above to create new lists in your code. You'll 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. Note that 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. You'll find more examples like these in your walkthrough of using lists.

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. Note that 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:

```
def fibonacci_generator(stop):
    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


fibonacci_generator(10)


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

Chapter 4.

Dictionaries

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:

```
type(d)


d[-1]
```

```
d[0:2]
```

```
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'}
foo



foo[42]

foo[2.78]

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

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 shoe-horned 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.

Chapter 6.

Functions

You 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:

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

Here, f is a function that operates on the inputs x and y. The output of the function is z. However, programming functions are much more generalized and versatile than this mathematical definition. In fact, appropriate function definition and use is so critical to proper software development that virtually all modern programming languages support both built-in and user-defined functions.
In programming, a function is a self-contained block of code that encapsulates a specific task or related group of tasks. In previous tutorials in this series, you've been introduced to some of the built-in functions provided by Python. `id()`, for example, takes one argument and returns that object's unique integer identifier:

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

len() returns the length of the argument passed to it:

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

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

```
any([False, False, False])

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

```
any(['bar' == 'baz', len('foo') == 4, 'qux' in {'foo', 'bar', 'baz'}])

any(['bar' == 'baz', len('foo') == 3, 'qux' in {'foo', 'bar', 'baz'}])
```

Each of these built-in functions performs a specific task. The code that accomplishes the task is defined somewhere, but you don't need to know where or even how the code works. All you need to know about is the function's interface:
What arguments (if any) it takes What values (if any) it returns
Then you 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 your code where it left off. The function may or may not return data for your code to use, as the examples above do.
When you define your own Python function, it works just the same. From somewhere in your code, you'll call your 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 you 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.

6.1 Importance of Python Functions

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

- Subroutines
- Procedures
- Methods
- Subprograms

So, why bother defining functions? There are several very good reasons. Let's go over a few now.

6.1.1 Abstraction and Reusability

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

locations within your application. What should you do? Well, you could just replicate the code over and over again, using your editor's copy-and-paste capability.
Later on, you'll probably decide that the code in question needs to be modified. You'll either find something wrong with it that needs to be fixed, or you'll want to enhance it in some way. If copies of the code are scattered all over your application, then you'll need to make the necessary changes in every location.

> At first blush, that may seem like a reasonable solution, but in the long term, it's likely to be a maintenance nightmare! While your code editor may help by providing a search-and-replace function, this method is error-prone, and you could easily introduce bugs into your code that will be difficult to find.

A better solution is to define a Python function that performs the task. Anywhere in your application that you need to accomplish the task, you simply call the function. Down the line, if you decide to change how it works, then you 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.

6.1.2 Modularity

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

```
# Main program

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

# Code to process file
<statement>
<statement>
<statement>
<statement>
```

```
# Code to write file out
<statement>
<statement>
<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 you'd have an increasingly difficult time wrapping your head around it.
Alternatively, you could structure the code more like the following:

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

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

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


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

This example is modularized. 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, you do this sort of thing all the time, even if you don't explicitly think of it that way. If you wanted to move some shelves full of stuff from one side of your garage to the other, then you hopefully wouldn't just stand there and aimlessly think, "Oh, geez. I need to move all that stuff over there! How do I do that???" You'd divide the job into manageable steps:
Take all the stuff off the shelves. Take the shelves apart. Carry the shelf parts across the garage to the new location. Re-assemble the shelves. Carry the stuff across the garage. Put the stuff back on the shelves.
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.

6.1.3 Namespace Separation

A namespace is a region of a program in which identifiers have meaning. As you'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 you write code within a function, you 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.

6.2 Function Calls and Definition

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

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

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

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

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, like an if or while statement.
The syntax for calling a Python function is as follows:

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

¡arguments¿ are the values passed into the function. They correspond to the <parameters> in the Python function definition. You 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, you'll start with a small example and add complexity from there. Keeping the time-honored mathematical tradition in mind, you'll call your first Python function f(). Here's a script file, foo.py, that defines and calls f():

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

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

Here's how this code works:
Line 1 uses the def keyword to indicate that a function is being defined. Execution of the def statement merely creates the definition of f(). All the following lines that are indented (lines 2 to 3) become part of the body of f() and are stored as its definition, but they aren't executed yet.
Line 4 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. To learn more about whitespace around top-level Python function definitions, check out Writing Beautiful Pythonic Code With PEP 8.

Line 5 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.
Line 6 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.
Line 7 is the next line to execute once the body of `f()` has finished. Execution returns to this `print()` statement.
Occasionally, you 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:

```
def f():
    pass

f()
```

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

6.3 Argument Passing

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

6.3.1 Positional Arguments

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

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

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

```
f(6, 'bananas', 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 (6, 'bananas', and 1.74) are bound to the parameters in order, as though by variable assignment:

Parameter	Argument
qty	6
item	bananas
price	1.74

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. For starters, the order of the arguments in the call must match the order of the parameters in the definition. There's nothing to stop you from specifying positional arguments out of order, of course:

```
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. You can't leave any out when calling the function:

```
# Too few arguments
f(6, 'bananas')
```

Nor can you specify extra ones:

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

Positional arguments are conceptually straightforward to use, but they're not very forgiving. You must specify the same number of arguments in the function call as there are

parameters in the definition, and in exactly the same order. In the sections that follow, you'll see some argument-passing techniques that relax these restrictions.

6.3.2 Keyword Arguments

When you're calling a function, you 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:

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

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

```
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 you can specify them in any order and Python will still know which argument goes with which parameter:

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

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

```
# 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.
You can call a function using both positional and keyword arguments:

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


f(6, 'bananas', price=1.74)
```

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

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

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

6.3.3 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:

```
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:

```
f(4, 'apples', 2.24)

f(4, 'apples')


f(4)

f()


f(item='kumquats', qty=9)

f(price=2.29)
```

In summary:
Positional arguments must agree in order and number with the parameters declared in the function definition. Keyword arguments must agree with declared parameters in number, but they may be specified in arbitrary order. Default parameters allow some arguments to be omitted when the function is called.

6.3.4 Mutable Default Parameter Values

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

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

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

```
f(['foo', 'bar', 'baz'])


f([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 '###':

```
f()
```

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

```
f()

f()
```

Oops! You might have 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 you call `f()` without a parameter, you're performing `.append()` on the same list.
You can demonstrate this with `id()`:

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

f()


f()
```

```
f()
```

The object identifier displayed confirms that, when my_{list} is allowed to default, the value is the same object with each call. Since lists are mutable, each subsequent `.append()` call causes the list to get longer. This is a common and pretty well-documented pitfall when you'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:

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


f()

f()

f()


f(['foo', 'bar', 'baz'])


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

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

Chapter 7.

Object Oriented Programming

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

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 preprocessed materials at each step on an assembly line. In addition, the object contains behavior, like the action that each assembly line component performs.

7.1 Defining Object Oriented Programming

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

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

Put another way, object-oriented programming is an approach for modeling 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.

> The key takeaway is that objects are at the center 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.

7.2 Defining a Class

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

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

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

Primitive data structures—like 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 favorite colors, respectively. What if you want to represent something more complex?

For example, you might want to track employees in an organization. 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:

```
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.

First, it can make larger code files more difficult to manage. If you reference `kirk[0]` several lines away from where you declared the kirk list, will you remember that the element with index 0 is the employee's name?

Second, 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.

7.3 Classes v. Instances

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

In this chapter, you'll create a Dog class that stores some information about the characteristics and behaviors 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.

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

7.4 Class Definition

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

Here's an example of a Dog class:

```
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 you'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, and breed. To keep the example small in scope, you'll just use name and age.

You define the properties that all Dog objects must have in a method called `.__init__()`. Every time you create a new Dog object, `.__init__()` sets the initial state of the object by assigning the values of the object's properties. That is, `.__init__()` initializes each new instance of the class.

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.

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

```
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 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":

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

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

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

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

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

7.5 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:

```
class Dog:
    pass

Dog()
```

You first create a new Dog class with no attributes or methods, and then you instantiate the Dog class to create a Dog object.

In the output above, you can see that you now have a new Dog object at 0x106702d30. 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. Note that the address on your screen will be different.

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

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

```
<__main__.Dog object at 0x132c27dd0>
```

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 you created. To see this another way, type the following:

```
a = Dog()
b = Dog()
a == b
```

In this code, you create two 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.

7.6 Class and Instance Attributes

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

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

> If you don't, then Python raises a TypeError:

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

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

This creates two new Dog instances—one for a four-year-old dog named Miles and one for a nine-year-old dog named Buddy.
The Dog class's `.__init__()` method has three 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 you create the Dog instances, you can access their instance attributes using dot notation:

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

```
Miles
4
Buddy
9
```

You can access class attributes the same way:

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

```
Canis familiaris
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. All Dog instances have .species, .name, and .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:

```
buddy.age = 10
buddy.age


miles.species = "Felis silvestris"
miles.species
```

In this example, you change 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. That makes Miles a pretty strange dog, but it's valid Python!
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.

7.6.1 Instance Methods

Instance methods are functions that you define inside a class and can only call on an instance of that class. Just like .__init__(), an instance method always takes self as its first parameter.
Open a new editor window and type in the following Dog class:

```
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 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.

Save the modified Dog class to a file called dog.py and press F5 to run the program. Then open the interactive window and type the following to see your instance methods in action:

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

miles.description()


miles.speak("Woof Woof")


miles.speak("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:

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

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

```
print(miles)
```

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__().
In the editor window, change the name of the Dog class's `.description()` method to .__str__():

```
class Dog:
    # ...

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

Save the file, Now, when you print miles, you get a much friendlier output:

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

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

7.7 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.
You inherit from a parent class by creating a new class and putting the name of the parent class into parentheses:

```
# inheritance.py

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.$\text{hair}_{\text{color}}$ 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:

```
# inheritance.py

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:

```
# inheritance.py

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, you'll take a walk to a dog park to better understand why you might want to use inheritance in your own code.

7.8 Parent Classes v. Child Classes

In this section, you'll create a child class for each of the three breeds mentioned above: Jack Russell terrier, dachshund, and bulldog.
For reference, here's the full definition of the Dog class that you're currently working with:

```
# dog.py
```

```
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}"
```

After doing the dog park example in the previous section, you've removed .breed again. You'll now write code to keep track of a dog's breed using child classes instead.
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 to the dog.py file to create three new child classes of the Dog class:

```
# dog.py

# ...

class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

Press F5 to save and run the file. With the child classes defined, you can now create some dogs of specific breeds in the interactive window:

```
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:

```
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()`:

```
isinstance(miles, Dog)
```

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

```
isinstance(miles, Bulldog)


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, you can give each breed its own sound.

7.8.1 Extending Parent Class Functionality

Since different breeds of dogs have slightly different barks, you want to provide a default value for the sound argument of their respective `.speak()` methods. To do this, you need to override `.speak()` in the class definition for each breed.
To override a method defined on the parent class, you define a method with the same name on the child class. Here's what that looks like for the JackRussellTerrier class:

```
# dog.py

# ...

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".
Update dog.py with the new JackRussellTerrier class and press F5 to save and run the file. You can now call `.speak()` on a JackRussellTerrier instance without passing an argument to sound:

```
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.

Chapter 8.

Computer Algebra System - SymPy

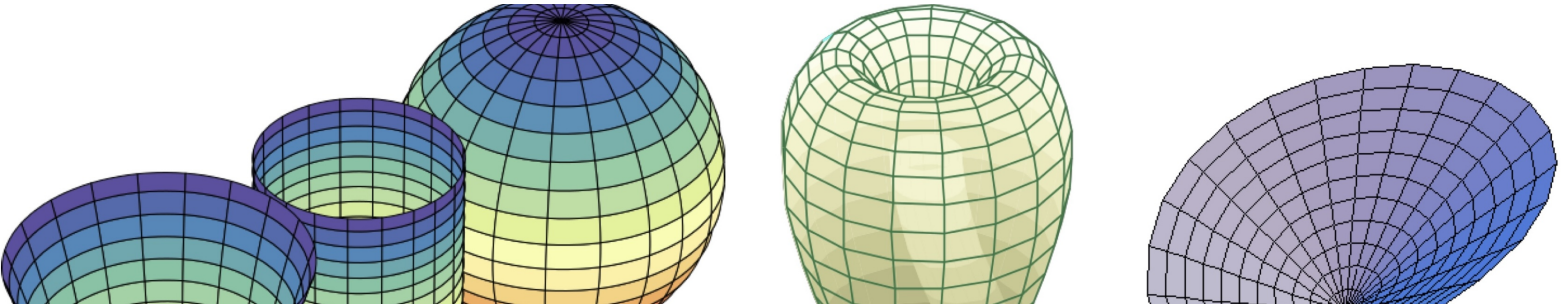


Figure 8.1.

There are two (2_) notable Computer Algebra Systems (CAS) for Python:

1. **SymPy** A python module that can be used in any Python program, or in an IPython session, that provides powerful CAS features.
2. **Sage** A full-featured and very powerful CAS environment that aims to provide an open source system that competes with Mathematica and Maple.

> Sage is not a regular Python module, but rather a CAS environment that uses Python as its programming language.

> Sage is in some aspects more powerful than SymPy, but both offer very comprehensive CAS functionality. The advantage of SymPy is that it is a regular Python module and integrates well with the IPython notebook.

In this lecture we will therefore look at how to use SymPy with IPython notebooks. If you are interested in an open source CAS environment I also recommend to read more about Sage as it is a great tool to study linear algebra and differential equations.

To get started using SymPy in a Python program or notebook, import the module `sympy`:

```
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)
```

8.1 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))
```

$$(x + \pi)^2$$

```
# 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
```

8.1.1 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))
```

$$(ix + 1)^2$$

8.1.2 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))
```

$$\frac{16}{25}$$

8.2 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)))
```

$$\left(1.5 + \pi\right)^{2}$$

```
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)))
```

$$\left(a + 2\pi\right)^{2}$$

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
```

8.3 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.

8.3.1 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)))
```

$$\sin\left(a + b\right)$$

```
print(sp.latex(sp.expand(sp.sin(a+b), trig=True)))
```

$$\sin\left(a\right)\cos\left(b\right) + \sin\left(b\right)\cos\left(a\right)$$

See `help(expand)` for a detailed explanation of the various types of expansions the expand functions can perform.
The opposite of product expansion is of course factoring. To factor an expression in SymPy use the `factor` function:

```
print(sp.latex(sp.factor(x**3 + 6 * x**2 + 11*x + 6)))
```

$$\left(x + 1\right)\left(x + 2\right)\left(x + 3\right)$$

8.3.2 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))))
```

$$\left(x + 1\right)\left(x + 2\right)\left(x + 3\right)$$

```
# simplify uses trigonometric identities
print(sp.latex(sp.simplify(sp.sin(a)**2 +sp. cos(a)**2)))
```

$$1$$

```
print(sp.latex(sp.simplify(sp.cos(x)/sp.sin(x))))
```

$$\frac{1}{\tan\left(x\right)}$$

8.3.3 Apart and Together

```
f1 = 1/((a+1)*(a+2))
```

```
print(sp.latex(f1))
```

$$\frac{1}{(a+1)(a+2)}$$

```
print(sp.latex(sp.apart(f1)))
```

$$-\frac{1}{a+2}+\frac{1}{a+1}$$

```
f2 = 1/(a+2) + 1/(a+3)
```

```
print(sp.latex(f2))
```

$$\frac{1}{a+3}+\frac{1}{a+2}$$

```
print(sp.latex(sp.together(f2)))
```

$$\frac{2a+5}{(a+2)(a+3)}$$

Simplify usually combines fractions but **does not factor**:

```
print(sp.latex(sp.simplify(f2)))
```

$$\frac{2a+5}{(a+2)(a+3)}$$

8.4 Calculus

In addition to algebraic manipulations, the other main use of CAS is to do calculus, like derivatives and integrals of algebraic expressions.

8.4.1 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)))
```

$$4\left(x + \pi\right)^3$$

For higher order derivatives we can do:

```
print(sp.latex(sp.diff(y**2, x, x)))
```

$$12\left(x + \pi\right)^2$$

```
print(sp.latex(sp.diff(y**2, x, 2))) # same as above
```

$$12\left(x + \pi\right)^2$$

To calculate the derivative of a multivariate expression, we can do:

```
x, y, z = sp.symbols("x,y,z")
```

```
f = sp.sin(x*y) + sp.cos(y*z)
```

```
print(sp.latex(sp.diff(f, x, 1, y, 2)))
```

$$-x\left(xy\cos\left(xy\right) + 2\sin\left(xy\right)\right)$$

8.4.2 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)))
```

$$x\cos(yz) + \begin{cases} -\frac{\cos(xy)}{y} & \text{for } y \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

By providing limits for the integration variable we can evaluate definite integrals:

```
print(sp.latex(sp.integrate(f, (x, -1, 1))))
```

$$2\cos(yz)$$

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.

8.4.3 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!
```

$$\prod_{n=1}^{10} n$$

8.4.4 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)))
```

$$y\cos(xy)$$

```
h = sp.Symbol("h")
```

```
print(sp.latex(sp.limit((f.subs(x, x+h) - f)/h, h, 0)))
```

$$y\cos(xy)$$

We can change the direction from which we approach the limiting point using the `dir` keyword argument:

```
print(sp.latex(sp.limit(1/x, x, 0, dir="+")))
```

$$\infty$$

```
print(sp.latex(sp.limit(1/x, x, 0, dir="-")))
```

$$-\infty$$

8.4.5 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)
```

8.5 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()))
```

$$\begin{bmatrix} \frac{m_{22}}{m_{11}m_{22}-m_{12}m_{21}} & -\frac{m_{12}}{m_{11}m_{22}-m_{12}m_{21}} \\ -\frac{m_{21}}{m_{11}m_{22}-m_{12}m_{21}} & \frac{m_{11}}{m_{11}m_{22}-m_{12}m_{21}} \end{bmatrix}$$

8.6 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])))
```

$$\left\{x : \frac{a}{2}+\frac{c}{2},\ y : \frac{a}{2}-\frac{c}{2}\right\}$$

Part I.

Economic Analysis

Chapter 9.

Macro and Micro Economics

Economics is generally classified under two ($_2$) topics:

1. Micro economic analysis
2. Macro economic analysis

Micro economics was the basis of study in ancient times because they had never thought of approaching the macro aspect with the changing conditions, the economic problems took a new turn resulting in the change in approach of economic analysis. Thus more importance was now given to Microanalysis than Macro analysis.

Now a days both macro and microanalysis are quite popular which will be the topic of discussion for todays topic.

9.1 Micro Economics

Classic economists have established and developed micro economics. The word 'Micro' has been derived from a Greek word 'Mikros'. It means micro or small. As the name suggests Micro Economics studies theories related with individual produces, individual firms and individual industries. Classical economists have been important decision or Micro Economics. Adam smith himself has accepted the fact that an individual works only by inducing the feeling of selfishness. This is an example of his recognition of micro economics .Even the neo classical economist Marshal has made an individual material welfare as the basis of subject matter of economics. In fact classical economists had studied macro economics and not separately. The fundamental tendencies of Micro Economics can be classified by following definitions.

1. Micro Economics is the study of particular economic organisms and their interaction and of particular economic quantities and their determination.

2. Micro Economics is the study of economics actions of individuals and well- defined group of individuals

The definitions of the above economists clearly indicate that in Micro Economics is a study of behavior of some particular economic units .For example _with the rise in price of a commodity , the economics decreases its consumption but the producer increases its production. The study of these types of private or personal events are possible only in Micro Economics we also study an industry with innumerable firms in Micro Economics. This is because a single unit is represented by an industry for the whole economy. Countless economics work in a nations economy. In this condition it is essential to study industries under Micro Economics.

9.1.1 Characteristics