Python Programming

Basics

PJ

Contents

1 Python Introduction

1.1 What is Python?

Python is a popular programming language. It was created by *Guido van Rossum*, and released in 1991.
It is used for:

- web development (server-side)
- software development
- mathematics
- system scripting

1.2 What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development

1.3 Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

1.4 Good to know

- The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.

- In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

1.5 Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Python

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

C

```c
#include <stdio.h>
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

Java

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

2 Getting Started

2.1 Installations

Many PCs and Macs will have python already installed.
To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

```
C:\Users\Your Name> python --version
```

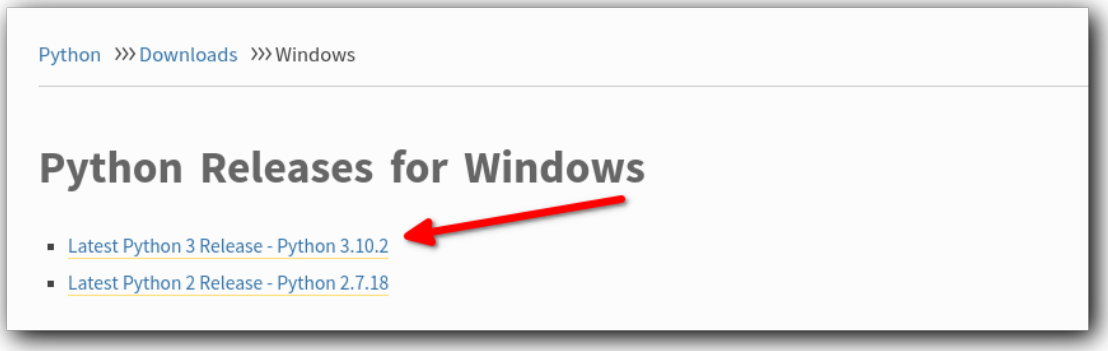
To check if you have python installed on a Linux or Mac, then on linux open the command line or on Mac open the Terminal and type:

```
$ python --version
```

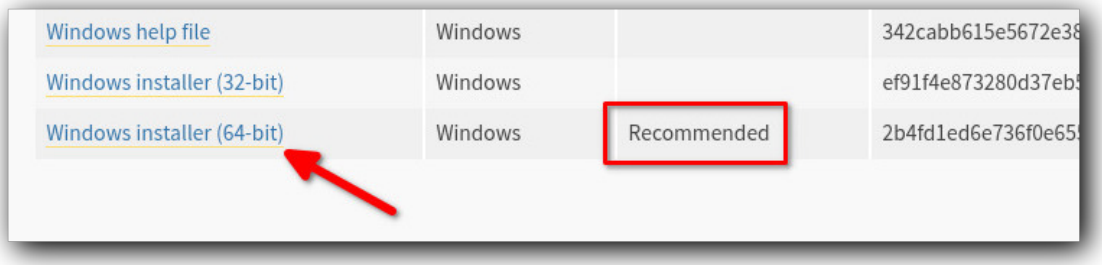
2.1.1 Install Python on Windows

If you find that you do not have Python installed on your computer, then you can download it for free from the following website: python.org.

Download and install Latest Python 3 Release - Python 3.X



I'd prefer to go with Recommended versions



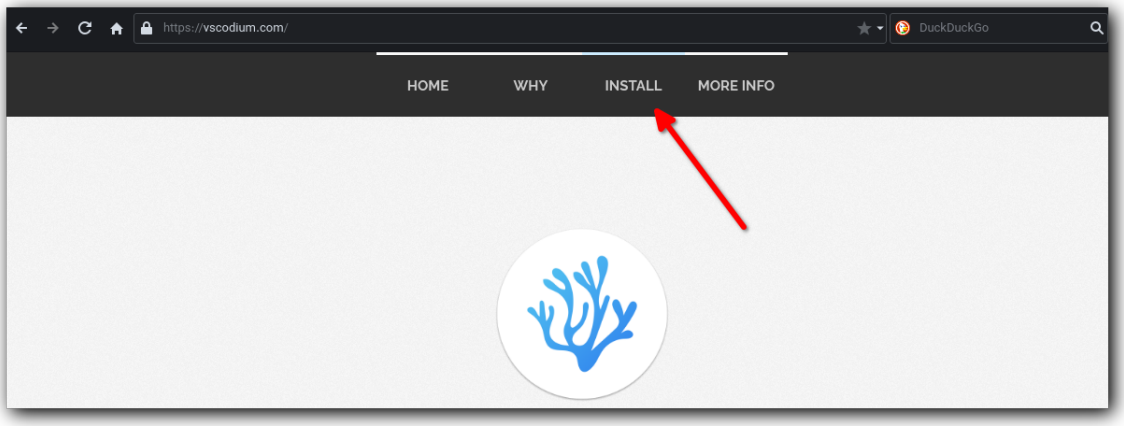
2.1.2 Install Python on Linux

The best way to install Python on Linux, is installing it from our distribution's `Package Manager`.

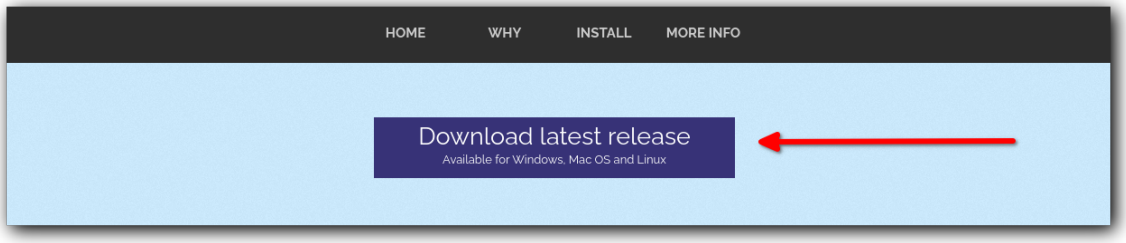
- Ubuntu/Mint/Debian: `$ sudo apt install python3`
- Arch/Manajaro: `$ sudo pacman -Sy python`
- Redhat/Fedora: `$ sudo dnf install python`
- OpenSUSE: `$ sudo zypper install python`

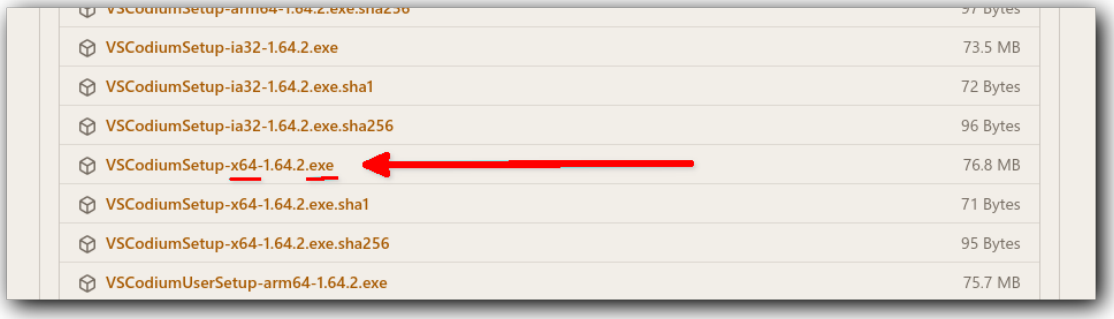
2.1.3 Install Text-editor/IDE

To writing Python programs, we can use any text editor or IDE which we are comfortable with. For example we are going to install VSCodium here and set it up for Python development.
First of all, we need to go to its website and download the software from its webstie: vscodium.com.



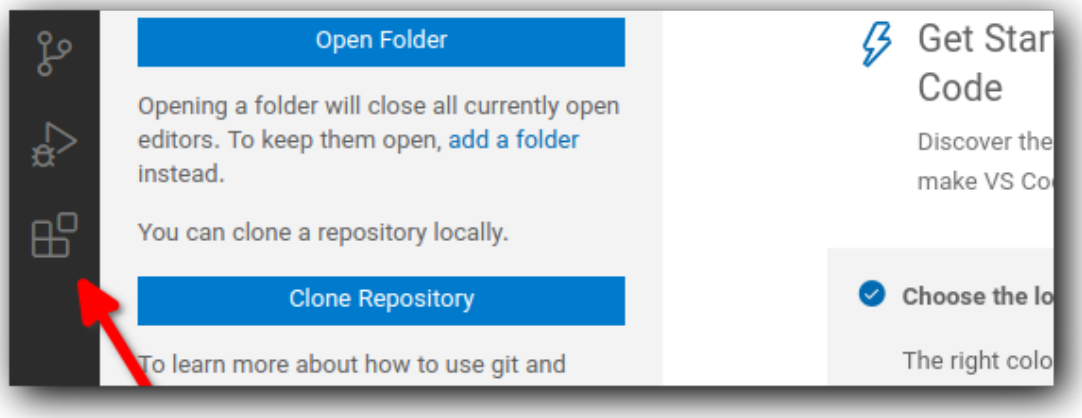
Go download and install the latest version of VSCodium.

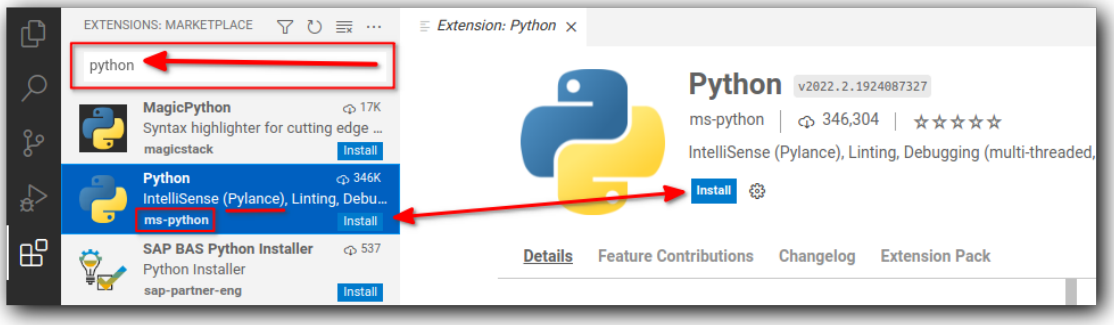


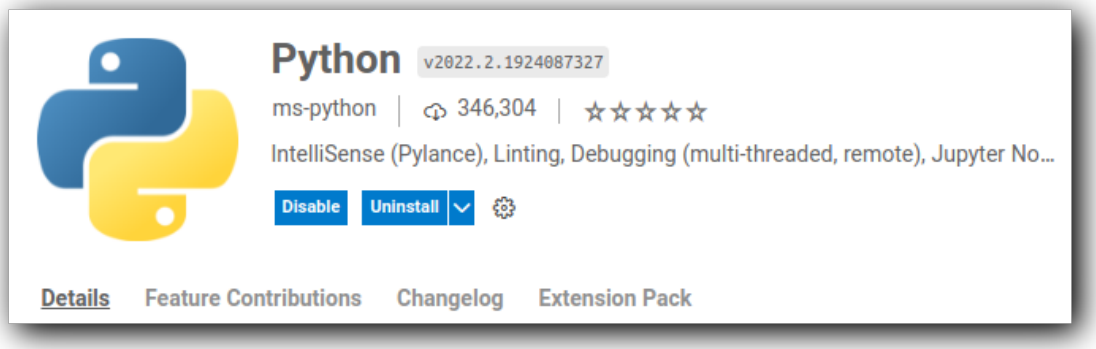


2.1.4 Setup VSCodium

After installing VSCodium, we need to install python extension on it to run and debug our python programs:

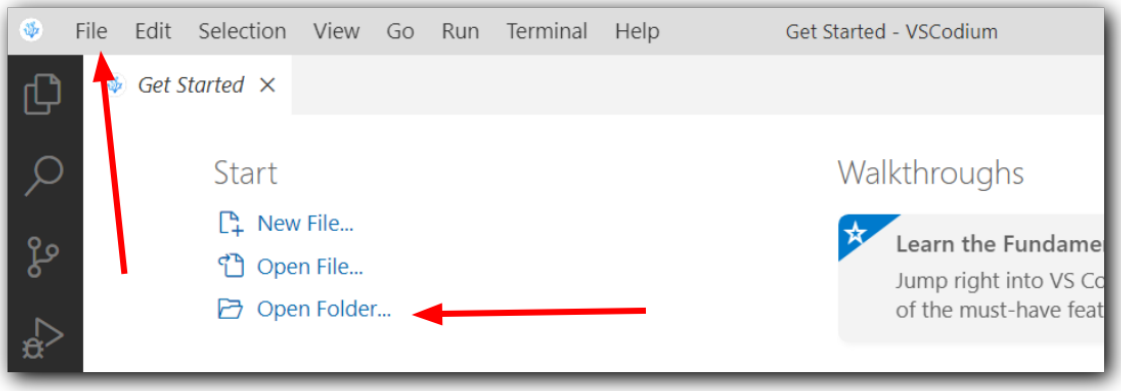


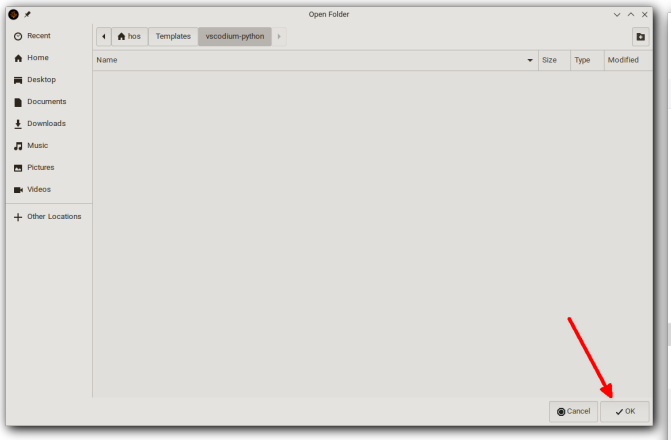




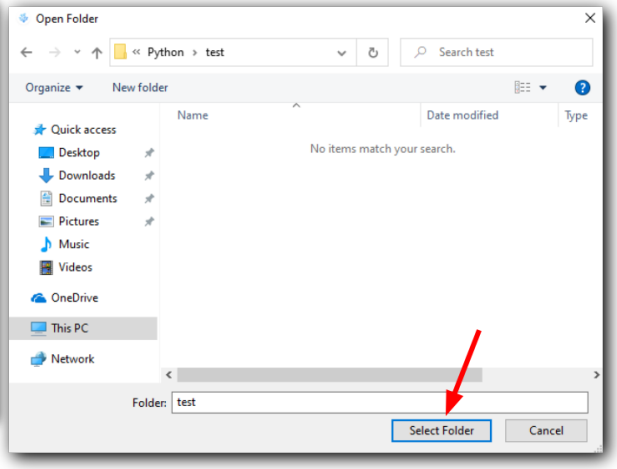
2.1.5 Create and Run a Python program in VSCodium

First of all, we need to choose a folder/directory to start working in.
You open a folder either from File > Open Folder, from the Getting Started tab or with `control-k control-o` key combination:



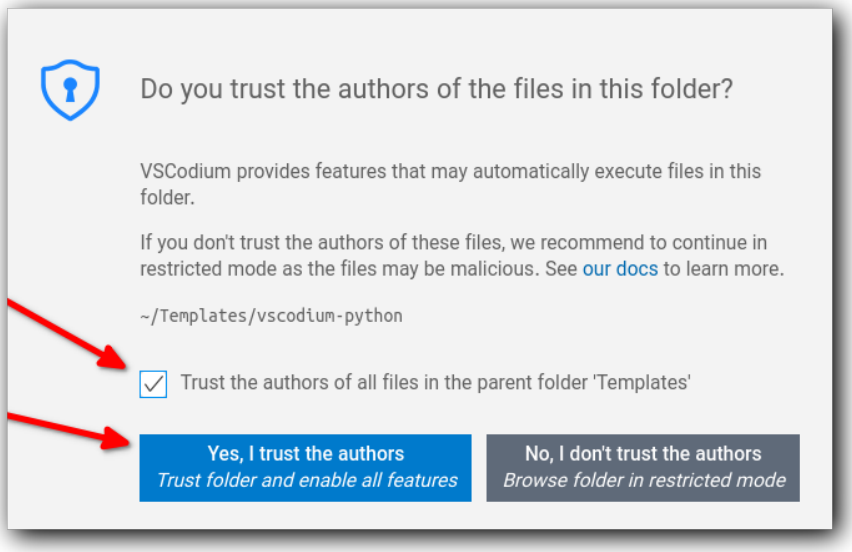


Linux

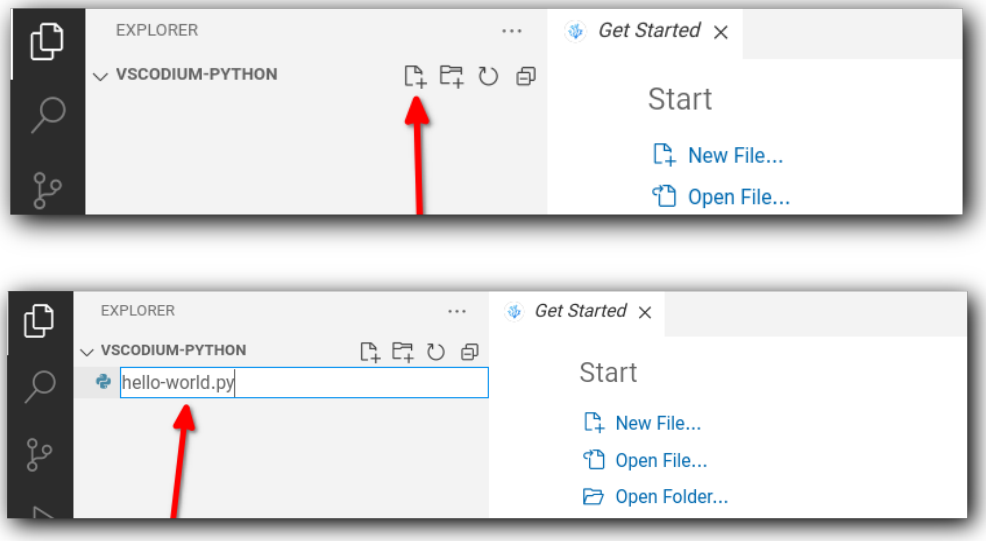


Windows

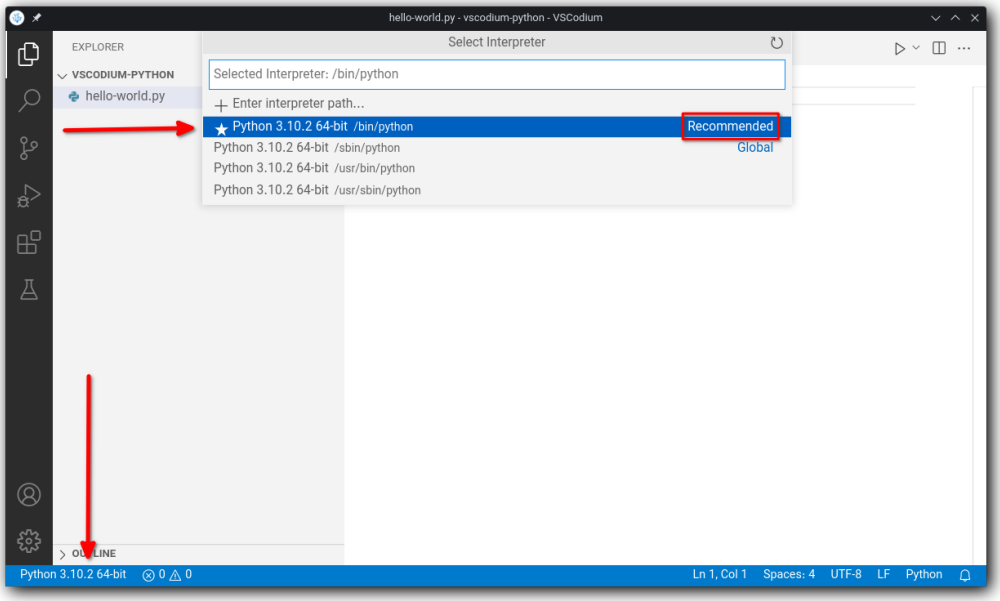
Then you need to confirm that you trust the author and the owner of this folder/directory:



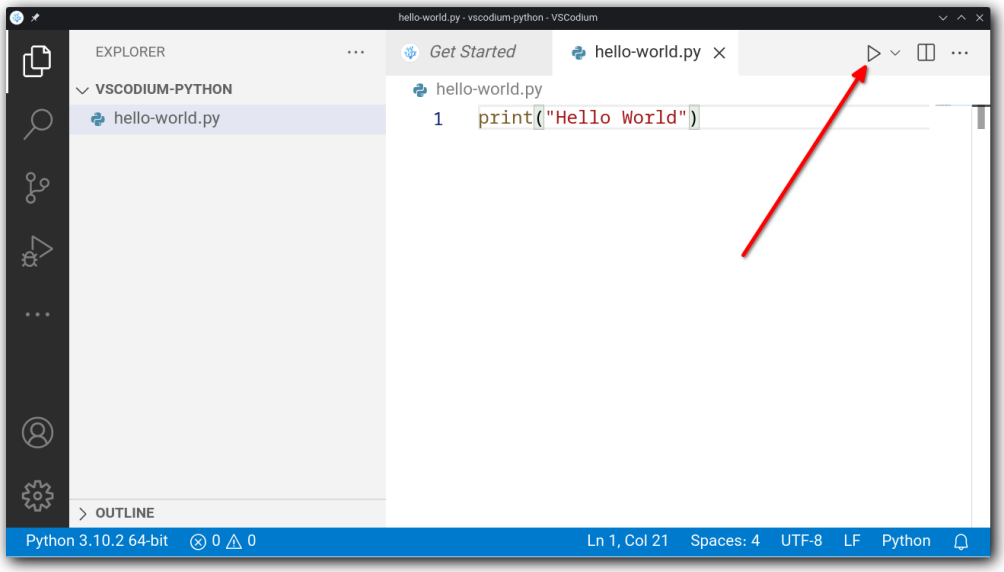
Now that you successfully opened a folder, it's time create a file:

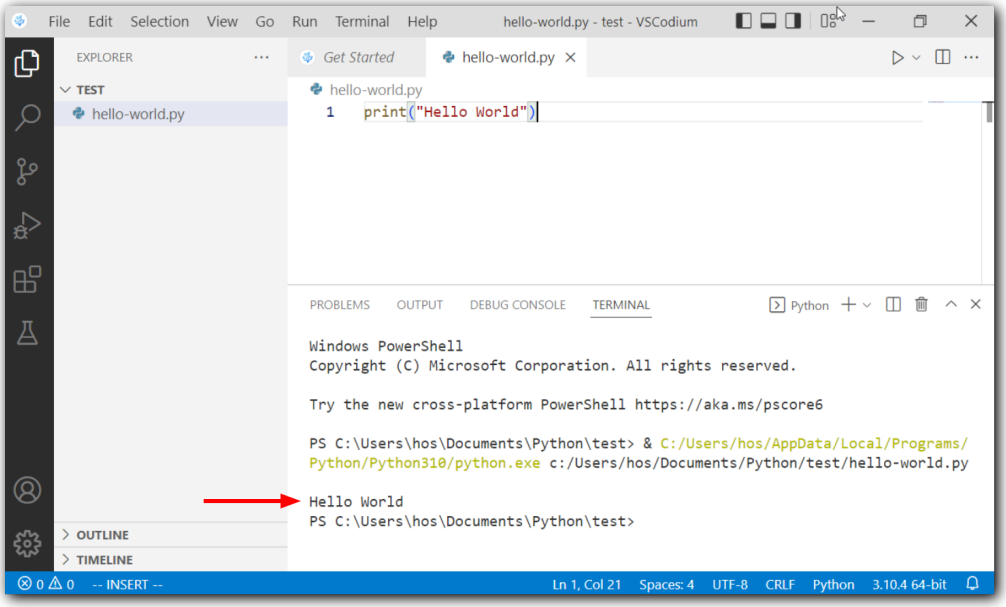


Now you can write and execute your python program. Before we go further, let's check our python's version:



Alright! We are good to go and write our first Python program and execute it:





2.2 Python Quickstart

Python is an interpreted programming language, this means that as a developer you write Python (`.py`) files in a text editor and then put those files into the python interpreter to be executed.
The way to run a python file is like this on the command line:

```
C:\Users\Your Name>python helloworld.py
```

Where `helloworld.py` is the name of your python file.
Let's write our first Python file, called `helloworld.py`, which can be done in any text editor.

Example

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

Simple as that. Save your file. Open your command line, navigate to the directory where you saved your file, and run:

```
$ python helloworld.py_
```

The output should read:

```
Hello, World!
$_
```

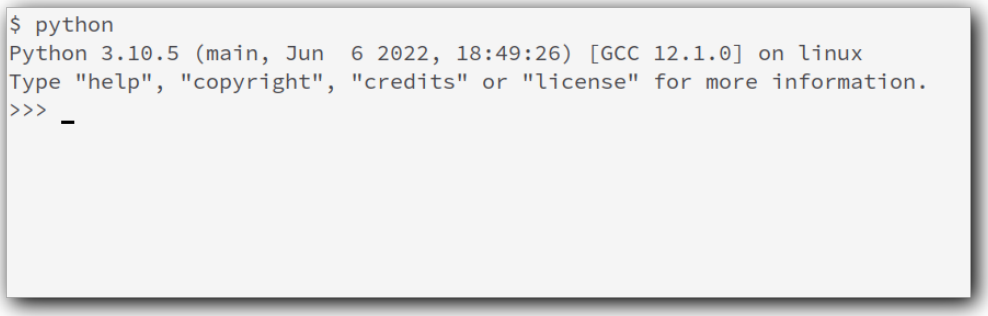
Congratulations, you have written and executed your first Python program.

2.3 The Python Command Line

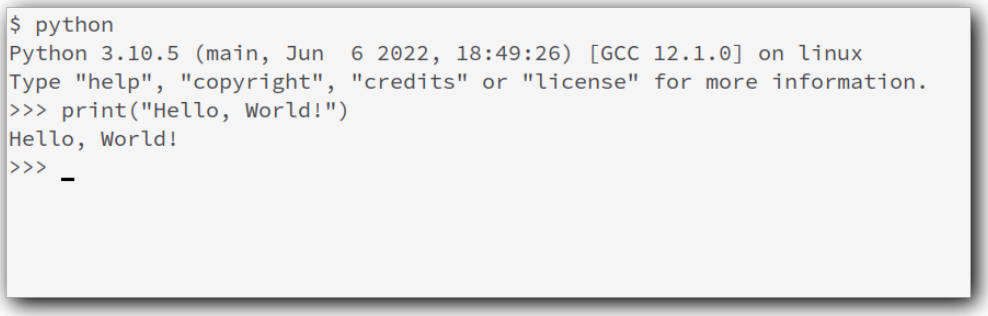
To test a short amount of code in python sometimes it is quickest and easiest not to write the code in a file. This is made possible because Python can be run as a command line itself.
Type the following on the Windows, Mac or Linux command line:

```
$ python
```

From there you can write any python, including our hello world example from earlier in the tutorial:



Which will write “Hello, World!” in the command line:



Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:

```
$ python
Python 3.10.5 (main, Jun  6 2022, 18:49:26) [GCC 12.1.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
>>> exit()
$ _
```

3 Syntax

3.1 Execute Python Syntax

As we learned in the previous page, Python syntax can be executed by writing directly in the Command Line:

Example

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

Or by creating a python file on the server, using the .py file extension, and running it in the Command Line:

Example

```
C:\Users\Your Name> python myfile.py
```

Or on linux:

```
$ python myfile.py
Hello, World!
$ _
```

3.2 Python Indentation

Indentation refers to the spaces at the beginning of a code line.
Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:
    print("Five is greater than two!")
```

Python will give you an error if you skip the indentation:

Error

```
if 5 > 2:
print("Five is greater than two!")
```

The number of spaces is up to you as a programmer, the most common use is four, but it has to be at least one.

Example

```
if 5 > 2:
    print("Five is greater than two!")
if 5 > 2:
        print("Five is greater than two!")
```

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

Error

```
if 5 > 2:
    print("Five is greater than two!")
        print("Five is greater than two!")
```

3.3 Python Variable

In Python, variables are created when you assign a value to it:

Example

```
x = 5
y = "Hello, World!"
```

Python has no command for declaring a variable.
You will learn more about variables in the chapter ??.

3.4 Comments

Python has commenting capability for the purpose of in-code documentation.
Comments start with a `#`, and Python will render the rest of the line as a comment:

Example

```
#This is a comment.
print("Hello, World!")
```

Test

Insert the missing part of the code below to output “Hello World”.

```
    ("Hello World")
```

4 Python Comments

Comments can be used to explain Python code.
Comments can be used to make the code more readable.
Comments can be used to prevent execution when testing code.

4.1 Creating a Comment

Comments starts with a `#`, and Python will ignore them:

Example

```
#This is a comment
print("Hello, World!")
```

Comments can be placed at the end of a line, and Python will ignore the rest of the line:

Example

```
print("Hello, World!") #This is a comment
```

A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code:

Example

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

4.2 Multi Line Comments

Python does not really have a syntax for multi line comments.
To add a multiline comment you could insert a `#` for each line:

Example

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Or, not quite as intended, you can use a multiline string.
Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

Example

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

Test

Comments in Python are written with a special character, which one?

```
This is a comment
```

5 Python Variables

5.1 Variables

Variables are containers for storing data values.

5.1.1 Creating Variables

Python has no command for declaring a variable.
A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "John"
print(x)
print(y)
```

```
5
John
```

Variables do not need to be declared with any particular type, and can even change type after they have been set.
Variables do not need to be declared with any particular type, and can even change type after they have been set.

Example

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

```
Sally
```

5.1.2 Casting

If you want to specify the data type of a variable, this can be done with casting.

Example

```
x = str(3)    # x will be '3' -> string
y = int(3)    # y will be 3 -> integer
z = float(3)  # z will be 3.0 -> float
```

5.1.3 Get the Type

You can get the data type of a variable with the `type()` function.

Example

```
x = 5
y = "John"
print(type(x))
print(type(y))
```

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

Note

You will learn more about *data types* and *casting* later in this tutorial.

5.1.4 Single or Double Quotes?

String variables can be declared either by using single or double quotes:

Example

```
x = "John"
# is the same as
x = 'John'
```

5.1.5 Case-Sensitive

Variable names are case-sensitive.

Example

```
a = 4
A = "Sally"
#A will not overwrite a
```

5.2 Variable Names

A variable can have a short name (like x and y)
or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character

- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Example

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

Error

```
2myvar = "John"
my-var = "John"
my var = "John"
```

Note

Remember that variable names are *case-sensitive*

5.3 Multi Words Variable Names

Variable names with more than one word can be difficult to read.
There are several techniques you can use to make them more readable:

5.3.1 Camel Case

Each word, except the first, starts with a capital letter:

Example

```
myVariableName = "John"
```

5.3.2 Pascal Case

Each word starts with a capital letter:

Example

```
MyVariableName = "John"
```

5.3.3 Snake Case

Each word is separated by an underscore character:

Example

```
my_variable_name = "John"
```

5.4 Assign Multiple Values

5.4.1 Many Values to Multiple Variables

Python allows you to assign values to multiple variables in one line:

Example

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

```
Orange
Banana
Cherry
```

Note

Make sure the number of variables matches the number of values, or else you will get an error.

5.4.2 One Value to Multiple Variables

And you can assign the same value to multiple variables in one line:

Example

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

```
Orange
Orange
Orange
```

5.4.3 Unpack a Collection

If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called unpacking.

Example

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

```
apple
banana
cherry
```

You will learn more about this in chapter ??.

5.5 Output Variables

5.5.1 Output Variables

The Python `print()` function is often used to output variables.

Example

```
x = "Python is awesome"
print(x)
```

```
Python is awesome
```

In the `print()` function, you output multiple variables, separated by a comma:

Example

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

```
Python is awesome
```

You can also use the `+` operator to output multiple variables:

Example

```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)
```

```
Python is awesome
```

Note

Notice the space character after "Python " and "is ", without them the result would be "Pythonisawesome".

For numbers, the `+` character works as a mathematical operator:

Example

```
x = 5
y = 10
print(x + y)
```

```
15
```

In the `print()` function, when you try to combine a string and a number with the `+` operator, Python will give you an error:

Error

```
x = 5
y = "John"
print(x + y)
```

```
Traceback (most recent call last):
  File "/home/hos/Documents/teach/python/en/test.py",
  line 3, in <module>
    print(x + y)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The best way to output multiple variables in the `print()` function is to separate them with commas, which even support different data types:

Example

```
x = 5
y = "John"
print(x, y)
```

```
5 John
```

5.6 Global Variables

5.6.1 Global Variables

Variables that are created outside of a function (as in all of the examples above) are known as global variables.
Global variables can be used by everyone, both inside of functions and outside.

Example

```
x = "awesome"

def myfunc():
   print("Python is " + x)

myfunc()
```

```
Python is awesome
```

If you create a variable with the same name inside a function, this variable will be local, and can only

be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example

```
x = "awesome"

def myfunc():
   x = "fantastic"
   print("Python is " + x)

myfunc()

print("Python is " + x)
```

```
Python is fantastic
Python is awesome
```

5.6.2 The *global* Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.
To create a global variable inside a function, you can use the `global` keyword.

Example

If you use the global keyword, the variable belongs to the global scope:

```
def myfunc():
   global x
   x = "fantastic"

myfunc()

print("Python is " + x)
```

```
Python is fantastic
```

Also, use the `global` keyword if you want to change a global variable inside a function.

Example

To change the value of a global variable inside a function, refer to the variable by using the `global` keyword:

```
x = "awesome"

def myfunc():
   global x
   x = "fantastic"

myfunc()

print("Python is " + x)
```

```
Python is fantastic
```

5.7 Test Yourself With Exercises

Now you have learned a lot about variables, and how to use them in Python.
Are you ready for a test?
Try to insert the missing part to make the code work as expected:

Test

Create a variable named `carname` and assign the value `Volvo` to it.

```
          = "      "
```

6 Data Types

6.1 Built-in Data Types

In programming, data type is an important concept.
Variables can store data of different types, and different types can do different things.
Python has the following data types built-in by default, in these categories:

Test type:	`str`
Numeric type:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

6.2 Getting the Data Type

You can get the data type of any object by using the `type()` function:

Example

Print the data type of the variable x:

```
x = 5
print(type(x))
```

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

6.3 Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
`x = "Hello World"`	`str`
`x = 20`	`int`
`x = 20.5`	`float`
`x = 1j`	`complex`
`x = ["apple", "banana", "cherry"]`	`list`
`x = ("apple", "banana", "cherry")`	`tuple`
`x = range(6)`	`range`
`x = "name": "John", "age": 36`	`dict`
`x = "apple", "banana", "cherry"`	`set`
`x = frozenset({"apple", "banana", "cherry"})`	`frozenset`
`x = True`	`bool`
`x = b"Hello"`	`bytes`
`x = bytearray(5)`	`bytearray`
`x = memoryview(bytes(5))`	`memoryview`
`x = None`	`NoneType`

6.4 Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

6.5 Exercise

Test

The following code example would print the data type of x, what data type would that be?

```
x = 5
print(type(x))
```

Example	Data Type
`x = str("Hello World")`	`str`
`x = int(20)`	`int`
`x = float(20.5)`	`float`
`x = complex(1j)`	`complex`
`x = list(("apple", "banana", "cherry"))`	`list`
`x = tuple(("apple", "banana", "cherry"))`	`tuple`
`x = range(6)`	`range`
`x = dict(name="John", age=36)`	`dict`
`x = set(("apple", "banana", "cherry"))`	`set`
`x = frozenset(("apple", "banana", "cherry"))`	`frozenset`
`x = bool(5)`	`bool`
`x = bytes(5)`	`bytes`
`x = bytearray(5)`	`bytearray`
`x = memoryview(bytes(5))`	`memoryview`

7 Numbers

7.1 Python Numbers

There are three numeric types in Python:

- `int`
- `float`
- `complex`

Variables of numeric types are created when you assign a value to them:

Example

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

To verify the type of any object in Python, use the `type()` function:

Example

```
print(type(x))
print(type(y))
print(type(z))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

7.2 Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Example

```
x = 1
y = 35656222554887711
z = -3255522

print(type(x))
print(type(y))
print(type(z))
```

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

7.3 Float

Float, or “floating point number” is a number, positive or negative, containing one or more decimals.

Example

```
x = 1.10
y = 1.0
z = -35.59

print(type(x))
print(type(y))
print(type(z))
```

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

Float can also be scientific numbers with an ”e” to indicate the power of 10.

Example

```
x = 35e3
y = 12E4
z = -87.7e100

print(type(x))
print(type(y))
print(type(z))
```

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

7.4 Complex

Complex numbers are written with a ”j” as the imaginary part:

Example

```
x = 3+5j
y = 5j
z = -5j

print(type(x))
print(type(y))
print(type(z))
```

```
<class 'complex'>
<class 'complex'>
<class 'complex'>
```

7.4.1 Type Conversion

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods:

Example

Convert from one type to another:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

```
1.0
2
(1+0j)
<class 'float'>
<class 'int'>
<class 'complex'>
```

Note

You cannot convert complex numbers into another number type.

7.5 Random Number

Python does not have a `random()` function to make a random number, but Python has a built-in module called `random` that can be used to make random numbers:

Example

Import the random module, and display a random number between 1 and 9:

```
import random

print(random.randrange(1, 10))
```

```
4
```

7.6 Exercise

Test

Insert the correct syntax to convert x into a floating point number.

```
x = 5
x = ____(x)
```

8 Python Casting

8.1 Specify a Variable Type

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.
Casting in python is therefore done using constructor functions:

- `int()` - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
- `float()` - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- `str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Example

Integers:

```
x = int(1)      # x will be 1
y = int(2.8)    # y will be 2
z = int("3")    # z will be 3
```

Floats:

```
x = float(1)       # x will be 1.0
y = float(2.8)     # y will be 2.8
z = float("3")     # z will be 3.0
w = float("4.2")   # w will be 4.2
```

Strings:

```
x = str("s1")  # x will be 's1'
y = str(2)     # y will be '2'
z = str(3.0)   # z will be '3.0'
```

9 Python Strings

9.1 Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.
`'Hello'` is the same as `"Hello"`.
You can display a string literal with the `print()` function:

Example

```
print("Hello")
print('Hello')
```

```
Hello
Hello
```

9.1.1 Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example

```
a = "Hello"
print(a)
```

```
Hello
```

9.1.2 Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Or three single quotes:

```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(a)
```

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.
```

Note

in the result, the line breaks are inserted at the same position as in the code.

9.1.3 Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.
However, Python does not have a character data type, a single character is simply a string with a length of 1.
Square brackets can be used to access elements of the string.

Example

Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1])
```

```
e
```

9.1.4 Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a `for` loop.

Example

Loop through the letters in the word "banana":

```
for x in "banana":
   print(x)
```

```
b
a
n
a
n
a
```

- Learn more about `for` Loops in chapter ??.

9.1.5 String Length

To get the length of a string, use the `len()` function.

Example

The `len()` function returns the length of a string:

```
a = "Hello, World!"
print(len(a))
```

```
13
```

9.1.6 Check String

To check if a certain phrase or character is present in a string, we can use the keyword `in`.

Example

Check if "free" is present in the following text:

```
txt = "The best things in life are free!"
print("free" in txt)
```

```
True
```

Use it in an `if` statement:

Example

```
txt = "The best things in life are free!"
if "free" in txt:
   print("Yes, 'free' is present.")
```

```
Yes, 'free' is present.
```

- Learn more about `if` statements in chapter ??

9.1.7 Check if NOT

To check if a certain phrase or character is NOT present in a string, we can use the keyword `not in`.

Example

Check if "expensive" is NOT present in the following text:

```
txt = "The best things in life are free!"
print("expensive" not in txt)
```

```
True
```

Use it in an `if` statement:

Example

print only if "expensive" is NOT present:

```
txt = "The best things in life are free!"
if "expensive" not in txt:
   print("No, 'expensive' is NOT present.")
```

```
No, 'expensive' is NOT present.
```

9.2 Slicing Strings

You can return a range of characters by using the slice syntax.
Specify the start index and the end index, separated by a colon, to return a part of the string.

Example

Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"
print(b[2:5])
```

```
llo
```

Note

The first character has index 0.

9.2.1 Slice From the Start

By leaving out the start index, the range will start at the first character:

Example

Get the characters from the start to position 5 (not included):

```
b = "Hello, World!"
print(b[:5])
```

```
Hello
```

9.2.2 Slice To the End

By leaving out the end index, the range will go to the end:

Example

Get the characters from position 2, and all the way to the end:

```
b = "Hello, World!"
print(b[2:])
```

```
llo, World!
```

9.2.3 Negative Indexing

Use negative indexes to start the slice from the end of the string:

Example

Get the characters:

From: "o" in "World!" (position -5)

To, but not included: "d" in "World!" (position -2):

```
b = "Hello, World!"
print(b[-5:-2])
```

```
orl
```

9.3 Modify Strings

Python has a set of built-in methods that you can use on strings.

Example

The `upper()` method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

```
HELLO, WORLD!
```

9.3.1 Lower Case

Example

The `lower()` method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

```
hello, world!
```

9.3.2 Remove Whitespace

Whitespace is the space before and/or after the actual text, and very often you want to remove this space.

Example

The `strip()` method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip())   # returns "Hello, World!"
```

```
Hello, World!
```

9.3.3 Replace String

Example

The `replace()` method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

```
Jello, World!
```

9.3.4 Split String

The `split()` method returns a list where the text between the specified separator becomes the list items.

Example

The `split()` method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

```
['Hello', ' World!']
```

- Learn more about Lists in chapter ??.

9.4 String Concatenation

To concatenate, or combine, two strings you can use the `+` operator.

Example

Merge variable `a` with variable `b` into variable `c`:

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

```
HelloWorld
```

Example

To add a space between them, add a `" "`:

```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
```

```
Hello World
```

9.5 Format Strings

As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

Error

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

But we can combine strings and numbers by using the `format()` method!
The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are:

Example

Use the `format()` method to insert numbers into strings:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

```
My name is John, and I am 36
```

The `format()` method takes unlimited number of arguments, and are placed into the respective placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

```
I want 3 pieces of item 567 for 49.95 dollars.
```

You can use index numbers `{0}` to be sure the arguments are placed in the correct placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

```
I want to pay 49.95 dollars for 3 pieces of item 567.
```

- Learn more about String Formatting in chapter ??

9.6 Escape Characters

To insert characters that are illegal in a string, use an escape character.
An escape character is a backslash `\` followed by the character you want to insert.
An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

Error

You will get an error if you use double quotes inside a string that is surrounded by double quotes:

```
txt = "We are the so-called "Vikings" from the north."
```

```
File "/home/hos/Documents/teach/python/en/test.py", line 1
    txt = "We are the so-called "Vikings" from the north."
                                ^^^^^^^
SyntaxError: invalid syntax
```

To fix this problem, use the escape character `\"` :

Example

The escape character allows you to use double quotes when you normally would not be allowed:

```
txt = "We are the so-called \"Vikings\" from the north."
```

Escape Characters

Code	Result
`\'`	Single Quote
`\"`	Double Quote
`\\`	Backslash
`\n`	New Line
`\r`	Carriage Return
`\t`	Tab
`\b`	Backspace
`\f`	Form Feed
`\ooo`	Octal value
`\xhh`	Hex value

9.7 Exercises

Now you have learned a lot about Strings, and how to use them in Python.
Are you ready for a test?
Try to insert the missing part to make the code work as expected:

Test

Use the `len` method to print the length of the string.

```
x = "Hello World"
print(       )
```

10 Python Booleans

Booleans represent one of two values: `True` or `False`.

10.1 Boolean Values

In programming you often need to know if an expression is `True` or `False`.
You can evaluate any expression in Python, and get one of two answers, `True` or `False`.
When you compare two values, the expression is evaluated and Python returns the Boolean answer:

Example

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

```
True
False
False
```

When you run a condition in an if statement, Python returns `True` or `False`:

Example

Print a message based on whether the condition is `True` or `False`:

```
a = 200
b = 33

if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

```
b is not greater than a
```

10.2 Evaluate Values and Variables

The `bool()` function allows you to evaluate any value, and give you `True` or `False` in return,

Example

Evaluate a string and a number:

```
print(bool("Hello"))
print(bool(15))
```

```
True
True
```

Example

Evaluate two variables:

```
x = "Hello"
y = 15

print(bool(x))
print(bool(y))
```

```
True
True
```

10.3 Most Values are True

Almost any value is evaluated to `True` if it has some sort of content.
Any string is `True`, except empty strings.
Any number is `True`, except `0`.
Any list, tuple, set, and dictionary are `True`, except empty ones.

Example

The following will return True:

```
bool("abc")
bool(123)
bool(["apple", "cherry", "banana"])
```

10.4 Some Values are False

In fact, there are not many values that evaluate to `False`, except empty values, such as `()`, `[]`, `{}`, `""`, the number `0`, and the value `None`. And of course the value `False` evaluates to `False`.

Example

The following will return False:

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

One more value, or object in this case, evaluates to `False`, and that is if you have an object that is made from a class with a `__len__` function that returns `0` or `False`:

Example

```
class myclass():
   def __len__(self):
      return 0

myobj = myclass()
print(bool(myobj))
```

```
False
```

10.5 Functions can Return a Boolean

You can create functions that returns a Boolean Value:

Example

Print the answer of a function:

```
def myFunction():
   return True

print(myFunction())
```

```
True
```

You can execute code based on the Boolean answer of a function:

Example

Print "YES!" if the function returns True, otherwise print "NO!":

```
def myFunction() :
   return True

if myFunction():
   print("YES!")
else:
   print("NO!")
```

```
YES!
```

Python also has many built-in functions that return a boolean value, like the `isinstance()` function, which can be used to determine if an object is of a certain data type:

Example

Check if an object is an integer or not:

```
x = 200
print(isinstance(x, int))
```

```
True
```

10.6 Test Yourself With Exercises

Test

The statement below would print a Boolean value, which one?

```
print(10 > 9)
```

11 Python Operators

Operators are used to perform operations on variables and values.
In the example below, we use the `+` operator to add together two values:

Example

```
print(10 + 5)
```

```
15
```

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

11.1 Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
`+`	Addition	`x + y`
`-`	Subtraction	`x - y`
`*`	Multiplication	`x * y`
`/`	Division	`x / y`
`%`	Modulus	`x % y`
`**`	Exponentiation	`x ** y`
`//`	Floor division	`x // y`

11.2 Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

11.3 Python Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

11.4 Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

11.5 Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
`is`	Returns True if both variables are the same object	`x is y`
`is not`	Returns True if both variables are not the same object	`x is not y`

11.6 Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
`in`	Returns True if a sequence with the specified value is present in the object	`x in y`
`in not`	Returns True if a sequence with the specified value is not present in the object	`x in not y`

11.7 Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

11.8 Test Yourself With Exercises

Test

Multiply `10` with `5`, and print the result.

```
print(10    5)
```

12 Python Lists

```
mylist = ["apple", "banana", "cherry"]
```

12.1 Lists

Lists are used to store multiple items in a single variable.
Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are

1. Tuple
2. Set
3. Dictionary

all with different qualities and usage.
Lists are created using square brackets:

Example

Create a List:

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

```
['apple', 'banana', 'cherry']
```

12.1.1 List Items

List items are *Ordered*, *Changeable*, and *Allow duplicate* values.
List items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

12.1.2 Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.
If you add new items to a list, the new items will be placed at the end of the list.

Note

There are some list methods that will change the order, but in general: the order of the items will not change.

12.1.3 Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

12.1.4 Allow Duplicates

Since lists are indexed, lists can have items with the same value:

Example

Lists allow duplicate values:

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
```

```
['apple', 'banana', 'cherry', 'apple', 'cherry']
```

12.1.5 List Length

To determine how many items a list has, use the `len()` function:

Example

Print the number of items in the list:

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

```
3
```

12.1.6 List Items - Data Types

List items can be of any data type:

Example

String, int and boolean data types:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

A list can contain different data types:

Example

A list with strings, integers and boolean values:

```
list1 = ["abc", 34, True, 40, "male"]
```

12.1.7 `type()`

From Python's perspective, lists are defined as objects with the data type 'list':

```
<class 'list'>
```

Example

What is the data type of a list?

```
mylist = ["apple", "banana", "cherry"]
print(type(mylist))
```

```
<class 'list'>
```

12.1.8 The list() Constructor

It is also possible to use the `list()` constructor when creating a new list.

Example

Using the list() constructor to make a List:

```
# note the double round-brackets
thislist = list(("apple", "banana", "cherry"))
print(thislist)
```

```
['apple', 'banana', 'cherry']
```

12.1.9 Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable[1], and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered[2] and changeable. No duplicate members.

Note

1. Set items are unchangeable, but you can remove and/or add items whenever you like.
2. As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

12.2 Access List Items

12.2.1 Access Items

List items are indexed and you can access them by referring to the index number:

Example

Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

```
banana
```

Note

The first item has index 0.

12.2.2 Negative Indexing

Negative indexing means start from the end
`-1` refers to the last item, `-2` refers to the second last item etc.

Example

Print the last item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

```
cherry
```

12.2.3 Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.
When specifying a range, the return value will be a new list with the specified items.

Example

Return the third, fourth, and fifth item:

```
thislist = ["apple", "banana", "cherry", "orange",
            "kiwi", "melon", "mango"]
print(thislist[2:5])
```

```
['cherry', 'orange', 'kiwi']
```

Note

- The search will start at index 2 (included) and end at index 5 (not included).
- Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item:

Example

This example returns the items from the beginning to, but NOT including, "kiwi":

```
thislist = ["apple", "banana", "cherry", "orange",
            "kiwi", "melon", "mango"]
print(thislist[:4])
```

```
['apple', 'banana', 'cherry', 'orange']
```

By leaving out the end value, the range will go on to the end of the list:

Example

This example returns the items from "cherry" to the end:

```
thislist = ["apple", "banana", "cherry", "orange",
            "kiwi", "melon", "mango"]
print(thislist[2:])
```

```
['cherry', 'orange', 'kiwi', 'melon', 'mango']
```

12.2.4 Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the list:

Example

This example returns the items from “orange” (-4) to, but NOT including “mango” (-1):

```
thislist = ["apple", "banana", "cherry", "orange",
            "kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

```
['orange', 'kiwi', 'melon']
```

12.2.5 Check if Item Exists

To determine if a specified item is present in a list use the `in` keyword:

Example

Check if “apple” is present in the list:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
   print("Yes, 'apple' is in the fruits list")
```

```
Yes, 'apple' is in the fruits list
```

12.3 Change List Items

12.3.1 Change Item Value

To change the value of a specific item, refer to the index number:

Example

Change the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

```
['apple', 'blackcurrant', 'cherry']
```

12.3.2 Change a Range of Item Values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

Example

Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon":

```
thislist = ["apple", "banana", "cherry", "orange",
            "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

Example

Change the second value by replacing it with two new values:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
['apple', 'blackcurrant', 'watermelon', 'cherry']
```

Note

The length of the list will change when the number of items inserted does not match the number of items replaced.

If you insert less items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

Example

Change the second and third value by replacing it with *one* value:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)
```

```
['apple', 'watermelon']
```

12.3.3 Insert Items

To insert a new list item, without replacing any of the existing values, we can use the `insert()` method.

The `insert()` method inserts an item at the specified index:

Example

Insert "watermelon" as the third item:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

```
['apple', 'banana', 'watermelon', 'cherry']
```

Note

As a result of the example above, the list will now contain 4 items.

12.4 Add List Items

12.4.1 Append Items

To add an item to the end of the list, use the `append()` method:

Example

Using the `append()` method to append an item:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

```
['apple', 'banana', 'cherry', 'orange']
```

12.4.2 Insert Items

To insert a list item at a specified index, use the `insert()` method.
The `insert()` method inserts an item at the specified index:

Example

Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

```
['apple', 'orange', 'banana', 'cherry']
```

Note

As a result of the examples above, the lists will now contain 4 items.

12.4.3 Extend List

To append elements from another list to the current list, use the `extend()` method.

Example

Add the elements of `tropical` to `thislist`:

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

```
['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']
```

The elements will be added to the end of the list.

12.4.4 Add Any Iterable

The `extend()` method does not have to append lists, you can add any iterable object (tuples, sets, dictionaries etc.).

Example

Add elements of a tuple to a list:

```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist)
```

```
['apple', 'banana', 'cherry', 'kiwi', 'orange']
```

12.5 Remove List Items

12.5.1 Remove Specified Item

The `remove()` method removes the specified item.

Example

Remove “banana”:

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

```
['apple', 'cherry']
```

12.5.2 Remove Specified Index

The `pop()` method removes the specified index.

Example

Remove the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

```
['apple', 'cherry']
```

If you do not specify the index, the `pop()` method removes the last item.

Example

Remove the last item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

```
['apple', 'banana']
```

The `del` keyword also removes the specified index:

Example

Remove the first item:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

```
['banana', 'cherry']
```

The `del` keyword can also delete the list completely.

Error

Delete the entire list:

```
thislist = ["apple", "banana", "cherry"]
del thislist
print(thislis)
```

```
Traceback (most recent call last):
        File "/home/hos/Documents/teach/python/en/test.py",
        line 4, in <module>
                print(thislis)
NameError: name 'thislis' is not defined
```

12.5.3 Clear the List

The `clear()` method empties the list.
The list still remains, but it has no content.

Example

Clear the list content:

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

```
[]
```

12.6 Loop Lists

12.6.1 Loop Through a List

You can loop through the list items by using a `for` loop:

Example

Print all items in the list, one by one:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
   print(x)
```

```
apple
banana
cherry
```

- Learn more about `for` loops in chapter ??.

12.6.2 Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.
Use the `range()` and `len()` functions to create a suitable iterable.

Example

Print all items by referring to their index number:

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
   print(thislist[i])
```

```
apple
banana
cherry
```

The iterable created in the example above is `[0, 1, 2]`.

12.6.3 Using a While Loop

You can loop through the list items by using a `while` loop.
Use the `len()` function to determine the length of the list, then start at 0 and loop your way through the list items by refering to their indexes.
Remember to increase the index by 1 after each iteration.

Example

Print all items, using a `while` loop to go through all the index numbers

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
   print(thislist[i])
   i = i + 1
```

```
apple
banana
cherry
```

- Learn more about while loops in chapter ??.

12.6.4 Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

Example

A short hand `for` loop that will print all items in a list:

```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

```
apple
banana
cherry
```

12.7 List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example

Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.
Without list comprehension you will have to write a `for` statement with a conditional test inside:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
   if "a" in x:
      newlist.append(x)

print(newlist)
```

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

With list comprehension you can do all that with only one line of code:

Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

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

12.7.1 The Syntax

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

12.7.2 Condition

The condition is like a filter that only accepts the items that valuate to `True`.

Example

Only accept items that are not "apple":

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if x != "apple"]
print(newlist)
```

```
['banana', 'cherry', 'kiwi', 'mango']
```

The condition `if x != "apple"` will return `True` for all elements other than "apple", making the new list contain all fruits except "apple".
The condition is optional and can be omitted:

Example

With no `if` statement:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits]
print(newlist)
```

```
['banana', 'cherry', 'kiwi', 'mango']
```

12.7.3 Iterable

The iterable can be any iterable object, like a list, tuple, set etc.

Example

You can use the `range()` function to create an iterable:

```
newlist = [x for x in range(10)]
```

Same example, but with a condition:

Example

Accept only numbers lower than 5:

```
newlist = [x for x in range(10) if x < 5]
```

12.7.4 Expression

The *expression* is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

Example

Set the values in the new list to upper case:

```
newlist = [x.upper() for x in fruits]
```

You can set the outcome to whatever you like:

Example

Set all values in the new list to 'hello':

```
newlist = ['hello' for x in fruits]
```

The *expression* can also contain conditions, not like a filter, but as a way to manipulate the outcome:

Example

Return "orange" instead of "banana":

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

The expression in the example above says:
"*Return the item if it is not banana, if it is banana return orange*".

12.8 Sort Lists

12.8.1 Sort List Alphanumerically

List objects have a `sort()` method that will sort the list alphanumerically, ascending, by default:

Example

Sort the list alphabetically:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

```
['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

Example

Sort the list numerically:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

```
[23, 50, 65, 82, 100]
```

12.8.2 Sort Descending

To sort descending, use the keyword argument `reverse = True`:

Example

Sort the list descending:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

```
['pineapple', 'orange', 'mango', 'kiwi', 'banana']
```

Example

Sort the list descending:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)
```

```
[100, 82, 65, 50, 23]
```

12.8.3 Customize *sort* Function

You can also customize your own function by using the keyword argument `key = function`.
The function will return a number that will be used to sort the list (the lowest number first):

Example

Sort the list based on how close the number is to 50:

```
def myfunc(n):
   return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)
```

```
[50, 65, 23, 82, 100]
```

12.8.4 Case Insensitive Sort

By default the `sort()` method is case sensitive, resulting in all capital letters being sorted before lower case letters:

Example

Case sensitive sorting can give an unexpected result:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)
```

```
['Kiwi', 'Orange', 'banana', 'cherry']
```

Luckily we can use built-in functions as key functions when sorting a list.
So if you want a case-insensitive sort function, use `str.lower` as a key function:

Example

Perform a case-insensitive sort of the list:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

```
['banana', 'cherry', 'Kiwi', 'Orange']
```

12.8.5 Reverse Order

What if you want to reverse the order of a list, regardless of the alphabet?
The `reverse()` method reverses the current sorting order of the elements.

Example

Reverse the order of the list items:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

```
['cherry', 'Kiwi', 'Orange', 'banana']
```

12.9 Copy Lists

You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`.
There are ways to make a copy, one way is to use the built-in List method `copy()`.

Example

Make a copy of a list with the `copy()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

```
['apple', 'banana', 'cherry']
```

Another way to make a copy is to use the built-in method `list()`.

Example

Make a copy of a list with the `list()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

```
['apple', 'banana', 'cherry']
```

12.10 Join Lists

12.10.1 Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python.
One of the easiest ways are by using the `+` operator.

Example

Join two list:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

```
['a', 'b', 'c', 1, 2, 3]
```

Another way to join two lists is by appending all the items from list2 into list1, one by one:

Example

Append list2 into list1:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

for x in list2:
   list1.append(x)

print(list1)
```

```
['a', 'b', 'c', 1, 2, 3]
```

Or you can use the `extend()` method, which purpose is to add elements from one list to another list:

Example

Use the `extend()` method to add list2 at the end of list1:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

```
['a', 'b', 'c', 1, 2, 3]
```

You can see a set of built-in String methods in section ??

12.11 List Exercises

Now you have learned a lot about lists, and how to use them in Python.

Are you ready for a test?

Try to insert the missing part to make the code work as expected:

Test

Print the second item in the fruits list.

```
fruits = ["apple", "banana", "cherry"]
print(          )
```

13 Python Tuples

```
mytuple = ("apple", "banana", "cherry")
```

13.1 Tuple

Tuples are used to store multiple items in a single variable.
Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are:

1. List
2. Set
3. Dictionary

all with different qualities and usage.
A tuple is a collection which is ordered and **unchangeable**.
Tuples are written with round brackets.

Example

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

```
('apple', 'banana', 'cherry')
```

13.1.1 Tuple Items

Tuple items are ordered, unchangeable, and allow duplicate values.
Tuple items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

13.1.2 Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

13.1.3 Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

13.1.4 Allow Duplicates

Since tuples are indexed, they can have items with the same value:

Example

Tuples allow duplicate values:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```

```
('apple', 'banana', 'cherry', 'apple', 'cherry')
```

13.1.5 Tuple Length

To determine how many items a tuple has, use the `len()` function:

Example

Print the number of items in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

```
3
```

13.1.6 Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Example

One item tuple, remember the comma:

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

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

13.1.7 Tuple Items - Data Types

Tuple items can be of any data type:

Example

String, int and boolean data types:

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

A tuple can contain different data types:

Example

A tuple with strings, integers and boolean values:

```
tuple1 = ("abc", 34, True, 40, "male")
```

13.1.8 `type()`

From Python's perspective, tuples are defined as objects with the data type 'tuple':

```
<class 'tuple'>
```

Example

What is the data type of a tuple?

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

```
<class 'tuple'>
```

13.1.9 The `tuple()` Constructor

It is also possible to use the `tuple()` constructor to make a tuple.

Example

Using the tuple() method to make a tuple:

```
# note the double round-brackets
thistuple = tuple(("apple", "banana", "cherry"))
print(thistuple)
```

```
('apple', 'banana', 'cherry')
```

13.2 Access Tuple Items

13.2.1 Access Items

Tuple items are indexed and you can access them by referring to the index number:

Example

Print the second item of the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

```
banana
```

Note

The first item has index 0.

13.2.2 Negative Indexing

Negative indexing means start from the end
`-1` refers to the last item, `-2` refers to the second last item etc.

Example

Print the last item of the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

```
cherry
```

13.2.3 Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.
When specifying a range, the return value will be a new tuple with the specified items.

Example

Return the third, fourth, and fifth item:

```
thistuple = ("apple", "banana", "cherry", "orange",
             "kiwi", "melon", "mango")
print(thistuple[2:5])
```

```
('cherry', 'orange', 'kiwi')
```

Note

- The search will start at index 2 (included) and end at index 5 (not included).
- Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item:

Example

This example returns the items from the beginning to, but NOT including, "kiwi":

```
thistuple = ("apple", "banana", "cherry", "orange",
             "kiwi", "melon", "mango")
print(thistuple[:4])
```

```
('apple', 'banana', 'cherry', 'orange')
```

By leaving out the end value, the range will go on to the end of the tuple:

Example

This example returns the items from "cherry" to the end:

```
thistuple = ("apple", "banana", "cherry", "orange",
             "kiwi", "melon", "mango")
print(thistuple[2:])
```

```
('cherry', 'orange', 'kiwi', 'melon', 'mango')
```

13.2.4 Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

Example

This example returns the items from “orange” (-4) to, but NOT including “mango” (-1):

```
thistuple = ("apple", "banana", "cherry", "orange",
             "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

```
('orange', 'kiwi', 'melon')
```

13.2.5 Check if Item Exists

To determine if a specified item is present in a tuple use the `in` keyword:

Example

Check if “apple” is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
   print("Yes, 'apple' is in the fruits tuple")
```

```
Yes, 'apple' is in the fruits tuple
```

Note

Accessing tuple items is just like accessing list items

13.3 Update Tuples

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.
But there are some workarounds.

13.3.1 Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.
But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example

Convert the tuple into a list to be able to change it:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

```
('apple', 'kiwi', 'cherry')
```

13.3.2 Add Items

Since tuples are immutable, they do not have a build-in `append()` method, but there are other ways to add items to a tuple.

1. **Convert into a list:** Just like the workaround for changing a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Example

Convert the tuple into a list, add "orange", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
print(thistuple)
```

```
('apple', 'banana', 'cherry', 'orange')
```

2. **Add tuple to a tuple:** You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

Example

Create a new tuple with the value "orange", and add that tuple:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)
```

```
('apple', 'banana', 'cherry', 'orange')
```

Note

When creating a tuple with only one item, remember to include a comma after the item, otherwise it will not be identified as a tuple.

13.3.3 Remove Items

Note

You cannot remove items in a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

Example

Convert the tuple into a list, remove "apple", and convert it back into a tuple:

```
 thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

Or you can delete the tuple completely:

Error

The `del` keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
#this will raise an error because the tuple no longer exists
print(thistuple)
```

```
Traceback (most recent call last):
  File "/home/hos/Documents/teach/python/en/test.py", line 3,
  in <module>
    print(thistuple) #this will raise an error
    because the tuple no longer exists
NameError: name 'thistuple' is not defined
```

13.4 Unpack Tuples

13.4.1 Unpacking a Tuple

When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

Example

Packing a tuple:

```
fruits = ("apple", "banana", "cherry")
```

But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking":

Example

Unpacking a tuple:

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

```
apple
banana
cherry
```

Note

The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

13.4.2 Using Asterisk*

If the number of variables is less than the number of values, you can add an `*` to the variable name and the values will be assigned to the variable as a list:

Example

Assign the rest of the values as a list called “red”:

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits

print(green)
print(yellow)
print(red)
```

```
apple
banana
['cherry', 'strawberry', 'raspberry']
```

If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

Example

Add a list of values the “tropic” variable:

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")

(green, *tropic, red) = fruits

print(green)
print(tropic)
print(red)
```

```
apple
['mango', 'papaya', 'pineapple']
cherry
```

13.5 Loop Tuples

13.5.1 Loop Through a Tuple

You can loop through the tuple items by using a `for` loop.

Example

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
   print(x)
```

```
apple
banana
cherry
```

Learn more about `for` loops in chapter ??.

13.5.2 Loop Through the Index Numbers

You can also loop through the tuple items by referring to their index number.
Use the `range()` and `len()` functions to create a suitable iterable.

Example

Print all items by referring to their index number:

```
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
   print(thistuple[i])
```

```
apple
banana
cherry
```

13.5.3 Using a While Loop

You can loop through the list items by using a `while` loop.
Use the `len()` function to determine the length of the tuple, then start at 0 and loop your way through the tuple items by refering to their indexes.
Remember to increase the index by 1 after each iteration.

Example

Print all items, using a `while` loop to go through all the index numbers:

```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
   print(thistuple[i])
   i = i + 1
```

```
apple
banana
cherry
```

Learn more about `while` loops in chapter ??

13.6 Join Tuples

13.6.1 Join Two Tuples

To join two or more tuples you can use the `+` operator:

Example

Join two tuples:

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

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

13.6.2 Multiply Tuples

If you want to multiply the content of a tuple a given number of times, you can use the `*` operator:

Example

Multiply the fruits tuple by 2:

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)
```

```
('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

You can see a set of built-in Tuple methods in section ??

13.7 Tuple Exercises

Test

Print the first item in the `fruits` tuple.

```
fruits = ("apple", "banana", "cherry")
print(          )
```

14 Python Sets

```
myset = {"apple", "banana", "cherry"}
```

14.1 Sets

Sets are used to store multiple items in a single variable.
Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are:

1. List
2. Tuple
3. Dictionary

all with different qualities and usage.
A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.

Note

* Set *items* are unchangeable, but you can remove items and add new items.

Sets are written with curly brackets.

Example

Create a Set:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

```
{'cherry', 'banana', 'apple'}
```

Note

Sets are unordered, so you cannot be sure in which order the items will appear.

14.1.1 Set Items

Set items are unordered, unchangeable, and do not allow duplicate values.

14.1.2 Unordered

Unordered means that the items in a set do not have a defined order.
Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

14.1.3 Unchangeable

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

Note

Once a set is created, you cannot change its items, but you can remove items and add new items.

14.1.4 Duplicates Not Allowed

Sets cannot have two items with the same value.

Example

Duplicate values will be ignored:

```
thisset = {"apple", "banana", "cherry", "apple"}

print(thisset)
```

```
{'banana', 'apple', 'cherry'}
```

14.1.5 Get the Length of a Set

To determine how many items a set has, use the `len()` function.

Example

Get the number of items in a set:

```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))
```

```
3
```

14.1.6 Set Items - Data Types

Set items can be of any data type:

Example

String, int and boolean data types:

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

A set can contain different data types:

Example

A set with strings, integers and boolean values:

```
set1 = {"abc", 34, True, 40, "male"}
```

14.1.7 `type()`

From Python's perspective, sets are defined as objects with the data type 'set':

```
<class 'set'>
```

Example

What is the data type of a set?

```
myset = {"apple", "banana", "cherry"}
print(type(myset))
```

```
<class 'set'>
```

14.1.8 The set() Constructor

It is also possible to use the `set()` constructor to make a set.

Example

Using the set() constructor to make a set:

```
# note the double round-brackets
thisset = set(("apple", "banana", "cherry"))
print(thisset)
```

```
{'banana', 'cherry', 'apple'}
```

14.2 Access Set Items

14.2.1 Access Items

You cannot access items in a set by referring to an index or a key.
But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

Example

Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}

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

```
apple
banana
cherry
```

Example

Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

14.2.2 Change Items

Note

Once a set is created, you cannot change its items, but you can add new items.

14.3 Add Set Items

14.3.1 Add Items

To add one item to a set use the `add()` method.

Example

Add an item to a set, using the `add()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

```
{'banana', 'apple', 'orange', 'cherry'}
```

14.3.2 Add Sets

To add items from another set into the current set, use the `update()` method.

Example

Add elements from `tropical` into `thisset`:

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)
```

```
{'apple', 'pineapple', 'cherry', 'papaya', 'mango', 'banana'}
```

14.3.3 Add Any Iterable

The object in the `update()` method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

Example

Add elements of a list to at set:

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]

thisset.update(mylist)

print(thisset)
```

```
{'orange', 'cherry', 'banana', 'kiwi', 'apple'}
```

14.4 Remove Set Items

To remove an item in a set, use the `remove()`, or the `discard()` method.

Example

Remove "banana" by using the `remove()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

```
{'apple', 'cherry'}
```

Note

If the item to remove does not exist, `remove()` will raise an error.

Example

Remove "banana" by using the `discard()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

```
{'apple', 'cherry'}
```

Note

If the item to remove does not exist, `discard()` will **NOT** raise an error.

You can also use the `pop()` method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.
The return value of the `pop()` method is the removed item.

Example

Remove the last item by using the `pop()` method:

```
thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

print(thisset)
```

```
apple
{'banana', 'cherry'}
```

Note

Sets are *unordered*, so when using the `pop()` method, you do not know which item that gets removed.

Example

The `clear()` method empties the set:

```
thisset = {"apple", "banana", "cherry"}

thisset.clear()

print(thisset)
```

```
set()
```

Error

The `del` keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}

del thisset

print(thisset)
```

```
Traceback (most recent call last):
  File "/home/hos/Documents/teach/python/en/test.py", line 6
  in <module>
    print(thisset)
NameError: name 'thisset' is not defined
```

14.5 Loop Sets

You can loop through the set items by using a `for` loop:

Example

Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}

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

```
cherry
apple
banana
```

You can see a set of built-in Set methods in section ??

14.6 Join Sets

14.6.1 Join Two Sets

There are several ways to join two or more sets in Python.
You can use the `union()` method that returns a new set containing all items from both sets, or the `update()` method that inserts all the items from one set into another:

Example

The `union()` method returns a new set with all items from both sets:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set3 = set1.union(set2)
print(set3)
```

```
{1, 2, 3, 'b', 'c', 'a'}
```

Example

The `update()` method inserts the items in set2 into set1:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set1.update(set2)
print(set1)
```

```
{'b', 1, 'c', 2, 3, 'a'}
```

Note

Both `union()` and `update()` will exclude any duplicate items.

14.6.2 Keep ONLY the Duplicates

The `intersection_update()` method will keep only the items that are present in both sets.

Example

Keep the items that exist in both set x, and set y:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.intersection_update(y)

print(x)
```

```
{'apple'}
```

The `intersection()` method will return a *new* set, that only contains the items that are present in both sets.

Example

Return a set that contains the items that exist in both set x, and set y:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.intersection(y)

print(z)
```

```
{'apple'}
```

14.6.3 Keep All, But NOT the Duplicates

The `symmetric_difference_update()` method will keep only the elements that are NOT present in both sets.

Example

Keep the items that are not present in both sets:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.symmetric_difference_update(y)

print(x)
```

```
{'microsoft', 'banana', 'cherry', 'google'}
```

The `symmetric_difference()` method will return a new set, that contains only the elements that are NOT present in both sets.

Example

Return a set that contains all items from both sets, except items that are present in both:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.symmetric_difference(y)

print(z)
```

```
{'microsoft', 'banana', 'google', 'cherry'}
```

14.6.4 Set Exercises

Test

Check if "apple" is present in the fruits set.

```
fruits = {"apple", "banana", "cherry"}
if "apple"     fruits:
   print("Yes, apple is a fruit!")
```

15 Python Dictionaries

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

15.1 Dictionary

Dictionaries are used to store data values in key:value pairs.
A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

Note

As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Dictionaries are written with curly brackets, and have keys and values:

Example

Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

15.1.1 Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates.
Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Example

Print the "brand" value of the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

```
Ford
```

15.1.2 Ordered or Unordered?

Note

As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.
Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

15.1.3 Changeable

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

15.1.4 Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

Example

Duplicate values will overwrite existing values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

15.1.5 Dictionary Length

To determine how many items a dictionary has, use the `len()` function:

Example

Print the number of items in the dictionary:

```
print(len(thisdict))
```

```
3
```

15.1.6 Dictionary Items - Data Types

The values in dictionary items can be of any data type:

Example

String, int, boolean, and list data types:

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

15.1.7 `type()`

From Python's perspective, dictionaries are defined as objects with the data type 'dict':

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

Example

Print the data type of a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(type(thisdict))
```

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

15.2 Access Dictionary Items

15.2.1 Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example

Get the value of the “model” key:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
print(x)
```

```
Mustang
```

There is also a method called `get()` that will give you the same result:

Example

Get the value of the “model” key:

```
x = thisdict.get("model")
```

```
Mustang
```

15.2.2 Get Keys

The `keys()` method will return a list of all the keys in the dictionary.

Example

Get a list of the keys:

```
x = thisdict.keys()
```

The list of the keys is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

Example

Add a new item to the original dictionary, and see that the keys list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change
```

```
dict_keys(['brand', 'model', 'year'])
dict_keys(['brand', 'model', 'year', 'color'])
```

15.2.3 Get Values

The `values()` method will return a list of all the values in the dictionary.

Example

Get a list of the values:

```
x = thisdict.values()
```

The list of the values is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

Example

Make a change in the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

```
dict_values(['Ford', 'Mustang', 1964])
dict_values(['Ford', 'Mustang', 2020])
```

Example

Add a new item to the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

```
dict_values(['Ford', 'Mustang', 1964])
dict_values(['Ford', 'Mustang', 1964, 'red'])
```

15.2.4 Get Items

The `items()` method will return each item in a dictionary, as tuples in a list.

Example

Get a list of the key:value pairs

```
x = thisdict.items()
```

The returned list is a *view* of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.

Example

Make a change in the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.items()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2020)])
```

Example

Add a new item to the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.items()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964),
        ('color', 'red')])
```

15.2.5 Check if Key Exists

To determine if a specified key is present in a dictionary use the `in` keyword:

Example

Check if "model" is present in the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
   print("Yes, 'model' is one of the keys in the thisdict
      dictionary")
```

```
Yes, 'model' is one of the keys in the thisdict dictionary
```

15.3 Change Dictionary Items

15.3.1 Change Values

You can change the value of a specific item by referring to its key name:

Example

Change the "year" to 2018:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

15.3.2 Update Dictionary

The `update()` method will update the dictionary with the items from the given argument. The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Update the "year" of the car by using the `update()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

15.4 Add Dictionary Items

15.4.1 Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

Example

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

15.4.2 Update Dictionary

The `update()` method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.
The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Add a color item to the dictionary by using the `update()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"color": "red"})
```

15.5 Remove Dictionary Items

There are several methods to remove items from a dictionary:

Example

The `pop()` method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

```
{'brand': 'Ford', 'year': 1964}
```

Example

The `popitem()` method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang'}
```

Example

The `del` keyword removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

```
{'brand': 'Ford', 'year': 1964}
```

Error

The `del` keyword can also delete the dictionary completely:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
#this will cause an error because "thisdict" no longer exists.
print(thisdict)
```

```
Traceback (most recent call last):
  File "/home/hos/Documents/teach/python/en/test.py", line 7,
  in <module>
    print(thisdict) #this will cause an error because "thisdict"
    no longer exists.
NameError: name 'thisdict' is not defined
```

Example

The clear() method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

```
{}
```

15.6 Loop Through a Dictionary

You can loop through a dictionary by using a `for` loop.
When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

Example

Print all key names in the dictionary, one by one:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(x)
```

```
brand
model
year
```

Example

Print all values in the dictionary, one by one:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(thisdict[x])
```

```
Ford
Mustang
1964
```

Example

You can also use the `values()` method to return values of a dictionary:

```
for x in thisdict.values():
  print(x)
```

```
Ford
Mustang
1964
```

Example

You can use the `keys()` method to return the keys of a dictionary:

```
for x in thisdict.keys():
  print(x)
```

```
brand
model
year
```

Example

Loop through both keys and values, by using the `items()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x, y in thisdict.items():
  print(x, y)
```

```
brand Ford
model Mustang
year 1964
```

15.7 Copy a Dictionary

You cannot copy a dictionary simply by typing `dict2 = dict1`, because: `dict2` will only be a *reference* to `dict1`, and changes made in `dict1` will automatically also be made in `dict2`.
There are ways to make a copy, one way is to use the built-in Dictionary method `copy()`.

Example

Make a copy of a dictionary with the `copy()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Another way to make a copy is to use the built-in function `dict()`.

Example

Make a copy of a dictionary with the `dict()` function:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

15.8 Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

Example

Create a dictionary that contain three dictionaries:

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

Or, if you want to add three dictionaries into a new dictionary:

Example

Create three dictionaries, then create one dictionary that will contain the other three dictionaries:

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

You can see a set of built-in Dictionary methods in section ??

15.9 Dictionary Exercises

Test

Use the `get` method to print the value of the "model" key of the `car` dictionary.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(                    )
```

16 Python *if ... else*

16.1 Python Conditions

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Greater than: `a > b`
- Less than or equal to: `a <= b`
- Greater than or equal to: `a >= b`

These conditions can be used in several ways, most commonly in "if statements" and loops.

16.1.1 *if* statements

An "if statement" is written by using the `if` keyword.

Example

If statement:

```
a = 33
b = 200
if b > a:
   print("b is greater than a")
```

```
b is greater than a
```

In this example we use two variables, `a` and `b`, which are used as part of the if statement to test whether `b` is greater than `a`. As `a` is 33, and `b` is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

16.1.2 Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Error

If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

```
  File "/home/hos/Documents/teach/python/en/test.py", line 4
    print("b is greater than a") # you will get an error
    ^
IndentationError: expected an indented block after 'if' statement on
line 3
```

16.1.3 *elif*

The `elif` keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

Example

```
a = 33
b = 33
if b > a:
   print("b is greater than a")
elif a == b:
   print("a and b are equal")
```

```
a and b are equal
```

In this example `a` is equal to `b`, so the first condition is not true, but the `elif` condition is true, so we print to screen that "a and b are equal".

16.1.4 *else*

The `else` keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200
b = 33
if b > a:
   print("b is greater than a")
elif a == b:
   print("a and b are equal")
else:
   print("a is greater than b")
```

```
a is greater than b
```

In this example `a` is greater than `b`, so the first condition is not true, also the `elif` condition is not true, so we go to the `else` condition and print to screen that "a is greater than b".
You can also have an else without the `elif`:

Example

```
a = 200
b = 33
if b > a:
   print("b is greater than a")
else:
   print("b is not greater than a")
```

```
b is not greater than a
```

16.1.5 Nested If

You can have `if` statements inside `if` statements, this is called *nested* `if` statements.

Example

```
x = 41

if x > 10:
   print("Above ten,")
   if x > 20:
      print("and also above 20!")
   else:
      print("but not above 20.")
```

```
Above ten,
and also above 20!
```

16.2 Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example

One line if statement:

```
if a > b: print("a is greater than b")
```

16.2.1 Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

One line if else statement:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

```
B
```

Note

This technique is known as **Ternary Operators**, or **Conditional Expressions**.

You can also have multiple else statements on the same line:

Example

One line if else statement, with 3 conditions:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

```
=
```

16.3 *and*

The `and` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, AND if c is greater than a:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

```
Both conditions are True
```

16.4 *or*

The `or` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, OR if c is greater than a:

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

```
At least one of the conditions is True
```

16.5 The *pass* Statement

`if` statements cannot be empty, but if you for some reason have an `if` statement with no content, put in the `pass` statement to avoid getting an error.

Example

```
a = 33
b = 200

if b > a:
   pass
```

16.6 Test Yourself With Exercises

Test

Print "Hello World" if `a` is greater than `b`.

```
a = 50
b = 10
   a   b
   print("Hello World")
```

17 Python Loops

Python has two primitive loop commands:

- `while` loops
- `for` loops

17.1 *while* Loops

With the `while` loop we can execute a set of statements as long as a condition is true.

Example

Print i as long as i is less than 6:

```
i = 1
while i < 6:
   print(i)
   i += 1
```

```
1
2
3
4
5
```

Note

remember to increment i, or else the loop will continue forever.

17.1.1 The *break* Statement

With the `break` statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
   print(i)
   if i == 3:
      break
   i += 1
```

```
1
2
3
```

17.1.2 The *continue* Statement

With the `continue` statement we can stop the current iteration, and continue with the next:

Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
   i += 1
   if i == 3:
      continue
   print(i)
```

```
1
2
4
5
6
```

17.1.3 The *else* Statement

With the `else` statement we can run a block of code once when the condition no longer is true:

Example

Print a message once the condition is false:

```
i = 1
while i < 6:
   print(i)
   i += 1
else:
   print("i is no longer less than 6")
```

```
1
2
3
4
5
i is no longer less than 6
```

17.1.4 Exercises

Test

Print `i` as long as `i` is less than 6.

```
i = 1
       i < 6
   print(i)
   i += 1
```

17.2 *for* Loops

A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
With the `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example

Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
   print(x)
```

```
apple
banana
cherry
```

The `for` loop does not require an indexing variable to set beforehand.

17.2.1 Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word “banana”:

```
for x in "banana":
   print(x)
```

```
b
a
n
a
n
a
```

17.2.2 The *break* Statement

With the `break` statement we can stop the loop before it has looped through all the items:

Example

Exit the loop when x is "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
   print(x)
   if x == "banana":
      break
```

```
apple
banana
```

Example

Exit the loop when x is "banana", but this time the break comes before the print:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
   if x == "banana":
      break
   print(x)
```

```
apple
```

17.2.3 The *continue* Statement

With the `continue` statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
   if x == "banana":
      continue
   print(x)
```

```
apple
cherry
```

17.2.4 The range() Function

To loop through a set of code a specified number of times, we can use the `range()` function,
The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the `range()` function:

```
for x in range(6):
   print(x)
```

```
0
1
2
3
4
5
```

Note

Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):

Example

Using the start parameter:

```
for x in range(2, 6):
   print(x)
```

```
2
3
4
5
```

The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: `range(2, 30, 3)`:

Example

Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):
   print(x)
```

```
2
5
8
11
14
17
20
23
26
29
```

17.2.5 *else* in *for* Loop

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

Example

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):
   print(x)
else:
   print("Finally finished!")
```

```
0
1
2
3
4
5
Finally finished!
```

Note

The `else` block will NOT be executed if the loop is stopped by a `break` statement.

Example

Break the loop when x is 3, and see what happens with the `else` block:

```
for x in range(6):
   if x == 3: break
   print(x)
else:
   print("Finally finished!")
```

```
0
1
2
```

17.2.6 Nested Loops

A nested loop is a loop inside a loop.
The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
   for y in fruits:
      print(x, y)
```

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

17.2.7 The *pass* Statement

`for` loops cannot be empty, but if you for some reason have a `for` loop with no content, put in the `pass` statement to avoid getting an error.

Example

```
for x in [0, 1, 2]:
    pass
```

17.2.8 Test Yourself With Exercises

Test

Loop through the items in the `fruits` list.

```
fruits = ["apple", "banana", "cherry"]
    x   fruits
    print(x)
```

18 Functions

A function is a block of code which only runs when it is called.
You can pass data, known as parameters, into a function.
A function can return data as a result.

18.1 Creating a Function

In Python a function is defined using the `def` keyword:

Example

```
def my_function():
   print("Hello from a function")
```

18.2 Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
   print("Hello from a function")

my_function()
```

```
Hello from a function
```

18.3 Arguments

Information can be passed into functions as arguments.
Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.
The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):
   print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

Note

Arguments are often shortened to *args* in Python documentations.

18.3.1 Parameters or Arguments?

The terms parameter and argument can be used for the same thing: information that are passed into a function.

Note

From a function's perspective:

- A parameter is the variable listed inside the parentheses in the function definition.
- An argument is the value that is sent to the function when it is called.

18.3.2 Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less

Example

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
   print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

Example

This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):
   print(fname + " " + lname)

my_function("Emil")
```

18.3.3 Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.
This way the function will receive a tuple of arguments, and can access the items accordingly:

Example

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
   print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

```
The youngest child is Linus
```

Note

Arbitrary Arguments are often shortened to **args* in Python documentations.

18.3.4 Keyword Arguments

You can also send arguments with the *key* = *value* syntax.
This way the order of the arguments does not matter.

Example

```
def my_function(child3, child2, child1):
   print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

```
The youngest child is Linus
```

Note

The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

18.3.5 Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: `**` before the parameter name in the function definition.
This way the function will receive a dictionary of arguments, and can access the items accordingly:

Example

If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**kid):
   print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

```
His last name is Refsnes
```

Note

Arbitrary Kword Arguments are often shortened to ***kwargs* in Python documentations.

18.4 Default Parameter Value

The following example shows how to use a default parameter value.
If we call the function without argument, it uses the default value:

Example

```
def my_function(country = "Norway"):
   print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

18.4.1 Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.
E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example

```
def my_function(food):
   for x in food:
      print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

```
apple
banana
cherry
```

18.5 Return Values

To let a function return a value, use the `return` statement:

Example

```
def my_function(x):
    return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

```
15
25
45
```

18.6 The pass Statement

`function` definitions cannot be empty, but if you for some reason have a `function` definition with no content, put in the `pass` statement to avoid getting an error.

Example

```
def myfunction():
    pass
```

18.7 Recursion

Python also accepts function recursion, which means a defined function can call itself.
Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.
The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.
In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements `(-1)` every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).
To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Example

Recursion Example

```
def tri_recursion(k):
    if(k > 0):
        result = k + tri_recursion(k - 1)
        print(result)
    else:
        result = 0
    return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

```
Recursion Example Results
1
3
6
10
15
21
```

18.8 Test Yourself With Exercises

Test

Create a function named `my_function`.

```
                    :
    print("Hello from a function")
```

19 Python *lambda*

A lambda function is a small anonymous function.
A lambda function can take any number of arguments, but can only have one expression.

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned:

Example

Add 10 to argument `a`, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

```
15
```

Lambda functions can take any number of arguments:

Example

Multiply argument `a` with argument `b` and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

```
30
```

Example

Summarize argument `a`, `b`, and `c` and return the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

```
13
```

19.1 Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.
Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
   return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

Example

```
def myfunc(n):
   return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

```
22
```

Or, use the same function definition to make a function that always triples the number you send in:

Example

```
def myfunc(n):
   return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

```
33
```

Or, use the same function definition to make both functions, in the same program:

Example

```
def myfunc(n):
   return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

```
22
33
```

Note

Use lambda functions when an anonymous function is required for a short period of time.

19.2 Exercise

Test

Create a lambda function that takes one parameter (`a`) and returns it.

```
x =
```

20 Python Arrays

Note

Python does not have built-in support for Arrays, but *Python Lists* can be used instead.

Note

This page shows you how to use **Lists** as **Arrays**, however, to work with arrays in Python you will have to import a library, like the *NumPy library*.

Arrays are used to store multiple values in one single variable:

Example

Create an array containing car names:

```
cars = ["Ford", "Volvo", "BMW"]
```

20.1 What is an Array?

An array is a special variable, which can hold more than one value at a time.
If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
car1 = "Ford"
car2 = "Volvo"
car3 = "BMW"
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?
The solution is an array!
An array can hold many values under a single name, and you can access the values by referring to an index number.

20.2 Access the Elements of an Array

You refer to an array element by referring to the index number.

Example

Get the value of the first array item:

```
x = cars[0]
```

Example

Modify the value of the first array item:

```
cars[0] = "Toyota"
```

20.3 The Length of an Array

Use the `len()` method to return the length of an array (the number of elements in an array).

Example

Return the number of elements in the `cars` array:

```
x = len(cars)
```

Note

The length of an array is always one more than the highest array index.

20.4 Looping Array Elements

You can use the `for in` loop to loop through all the elements of an array.

Example

Print each item in the `cars` array:

```
cars = ["Ford", "Volvo", "BMW"]
for x in cars:
   print(x)
```

```
Ford
Volvo
BMW
```

20.5 Adding Array Elements

You can use the `append()` method to add an element to an array.

Example

Add one more element to the `cars` array:

```
cars = ["Ford", "Volvo", "BMW"]
cars.append("Honda")
for x in cars:
   print(x)
```

```
Ford
Volvo
BMW
Honda
```

20.6 Removing Array Elements

You can use the `pop()` method to remove an element from the array.

Example

Delete the second element of the `cars` array:

```
cars.pop(1)
```

You can also use the `remove()` method to remove an element from the array.

Example

Delete the element that has the value "Volvo":

```
cars.remove("Volvo")
```

Note

The list's `remove()` method only removes the first occurrence of the specified value.

You can see a set of built-in Array methods in section ??

21 Python Classes/Objects

Python is an object oriented programming language.
Almost everything in Python is an object, with its properties and methods.
A Class is like an object constructor, or a "blueprint" for creating objects.

21.1 Create a Class

To create a class, use the keyword `class`:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:
    x = 5
```

21.2 Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
class MyClass:
    x = 5

p1 = MyClass()
print(p1.x)
```

```
5
```

21.3 The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.
To understand the meaning of classes we have to understand the built-in `__init__()` function.
All classes have a function called __init__(), which is always executed when the class is being initiated.
Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

Create a class named Person, use the __init__() function to assign values for name and age:

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

```
John
36
```

Note

The `__init__()` function is called automatically every time the class is being used to create a new object.

21.4 The self Parameter

The `self` parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named `self` , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example

Use the words *mysillyobject* and *abc* instead of *self*:

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

   def myfunc(abc):
      print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

```
Hello my name is John
```

21.5 Modify Object Properties

You can modify properties on objects like this:

Example

Set the age of p1 to 40:

```
p1.age = 40
```

21.6 Delete Object Properties

You can delete properties on objects by using the `del` keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

21.7 Delete Object

You can delete objects by using the `del` keyword:

Example

Delete the p1 object:

```
del p1
```

21.8 Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.
Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

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

   def myfunc(self):
      print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

```
Hello my name is John
```

Note

The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

21.9 The pass Statement

`class` definitions cannot be empty, but if you for some reason have a `class` definition with no content, put in the `pass` statement to avoid getting an error.

Example

```
class Person:
   pass
```

You can see a set of built-in Object methods in section ??

21.10 Test Yourself With Exercises

Test

Create a class named MyClass:

```
    MyClass:
  x = 5
```

22 Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.
Parent class is the class being inherited from, also called base class.
Child class is the class that inherits from another class, also called derived class.

22.1 Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Example

Create a class named `Person`, with `firstname` and `lastname` properties, and a `printname` method:

```
class Person:
   def __init__(self, fname, lname):
      self.firstname = fname
      self.lastname = lname

   def printname(self):
      print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the
   printname method:

x = Person("John", "Doe")
x.printname()
```

```
John Doe
```

22.2 Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Example

Create a class named `Student`, which will inherit the properties and methods from the `Person` class:

```
class Student(Person):
   pass
```

Note

Use the `pass` keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Example

Use the `Student` class to create an object, and then execute the `printname` method:

```
class Student:
   def __init__(self, fname, lname):
      self.firstname = fname
      self.lastname = lname

   def printname(self):
      print(self.firstname, self.lastname)

x = Student("Mike", "Olsen")
x.printname()
```

```
Mike Olsen
```

22.3 Add the __init__() Function

So far we have created a child class that inherits the properties and methods from its parent.
We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

Note

The `__init__()` function is called automatically every time the class is being used to create a new object.

Example

Add the `__init__()` function to the `Student` class:

```
class Student(Person):
   def __init__(self, fname, lname):
      #add properties etc.
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Note

The child's `__init__()` function overrides the inheritance of the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

Example

```
class Student(Person):
   def __init__(self, fname, lname):
      Person.__init__(self, fname, lname)
```

Now we have successfully added the __init__() function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

22.4 Use the super() Function

Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:

Example

```
class Person:
   def __init__(self, fname, lname):
      self.firstname = fname
      self.lastname = lname

   def printname(self):
      print(self.firstname, self.lastname)

class Student(Person):
   def __init__(self, fname, lname):
      super().__init__(fname, lname)
```

By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Example

Add a property called `graduationyear` to the `Student` class:

```
class Person:
   def __init__(self, fname, lname):
      self.firstname = fname
      self.lastname = lname

class Student(Person):
   def __init__(self, fname, lname):
      super().__init__(fname, lname)
      self.graduationyear = 2019

x = Student("Mike", "Olsen")
print(x.graduationyear)
```

```
2019
```

In the example below, the year `2019` should be a variable, and passed into the `Student` class when creating student objects. To do so, add another parameter in the __init__() function:

Example

Add a `year` parameter, and pass the correct year when creating objects:

```
class Student(Person):
   def __init__(self, fname, lname, year):
      super().__init__(fname, lname)
      self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
print(x.graduationyear)
```

```
2019
```

22.5 Add Methods

Example

Add a method called `welcome` to the `Student` class:

```
class Person:
   def __init__(self, fname, lname):
      self.firstname = fname
      self.lastname = lname

   def printname(self):
      print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class
       of", self.graduationyear)

x.welcome()
```

```
Welcome Mike Olsen to the class of 2019
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

22.6 Test Yourself With Exercises

Test

What is the correct syntax to create a class named `Student` that will inherit properties and methods from a class named `Person`?

```
class                :
```

23 Python Iterators

An iterator is an object that contains a countable number of values.
An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

23.1 Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.
All these objects have a `iter()` method which is used to get an iterator:

Example

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

```
apple
banana
cherry
```

Even strings are iterable objects, and can return an iterator:

Example

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

```
b
a
n
a
n
a
```

23.2 Looping Through an Iterator

We can also use a `for` loop to iterate through an iterable object:

Example

Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")

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

```
apple
banana
cherry
```

Example

Iterate the characters of a string:

```
mystr = "banana"

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

```
b
a
n
a
n
a
```

The for `loop` actually creates an iterator object and executes the next() method for each loop.

23.3 Create an Iterator

To create an object/class as an iterator you have to implement the methods `__iter__()` and `__next__()` to your object.
As you have learned in the Python Classes/Objects chapter, all classes have a function called `__init__()`, which allows you to do some initializing when the object is being created.
The `__iter__()` method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.
The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

Example

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
   def __iter__(self):
      self.a = 1
      return self

   def __next__(self):
      x = self.a
      self.a += 1
      return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

```
1
2
3
4
5
```

23.4 StopIteration

The example above would continue forever if you had enough next() statements, or if it was used in a `for` loop.

To prevent the iteration to go on forever, we can use the `StopIteration` statement.

In the `__next__()` method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example

Stop after 20 iterations:

```
class MyNumbers:
   def __iter__(self):
      self.a = 1
      return self

   def __next__(self):
      if self.a <= 20:
         x = self.a
         self.a += 1
         return x
      else:
         raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

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

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

24 Python Scope

A variable is only available from inside the region it is created. This is called **scope**.

24.1 Local Scope

A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

Example

A variable created inside a function is available inside that function:

```
def myfunc():
  x = 300
  print(x)

myfunc()
```

```
300
```

24.2 Function Inside Function

As explained in the example above, the variable `x` is not available outside the function, but it is available for any function inside the function:

Example

The local variable can be accessed from a function within the function:

```
def myfunc():
  x = 300
  def myinnerfunc():
    print(x)
  myinnerfunc()

myfunc()
```

```
300
```

24.3 Global Scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope.
Global variables are available from within any scope, global and local.

Example

A variable created outside of a function is global and can be used by anyone:

```
x = 300

def myfunc():
   print(x)

myfunc()

print(x)
```

```
300
300
```

24.3.1 Naming Variables

If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

Example

The function will print the local `x`, and then the code will print the global `x`:

```
x = 300

def myfunc():
   x = 200
   print(x)

myfunc()

print(x)
```

```
200
300
```

24.4 Global Keyword

If you need to create a global variable, but are stuck in the local scope, you can use the `global` keyword.

The `global` keyword makes the variable global.

Example

If you use the `global` keyword, the variable belongs to the global scope:

```
def myfunc():
   global x
   x = 300

myfunc()

print(x)
```

```
300
```

Also, use the `global` keyword if you want to make a change to a global variable inside a function.

Example

To change the value of a global variable inside a function, refer to the variable by using the `global` keyword:

```
x = 300

def myfunc():
   global x
   x = 200

myfunc()

print(x)
```

```
200
```

25 Python Modules

25.1 What is a Module?

Consider a module to be the same as a code library.
A file containing a set of functions you want to include in your application.

25.1.1 Create a Module

To create a module just save the code you want in a file with the file extension `.py`:

Example

Save this code in a file named `mymodule.py`

```
def greeting(name):
    print("Hello, " + name)
```

25.1.2 Use a Module

Now we can use the module we just created, by using the `import` statement:

Example

Import the module named mymodule, and call the greeting function:

```
import mymodule

mymodule.greeting("Jonathan")
```

```
Hello Jonathan
```

Note

When using a function from a module, use the syntax: `module_name.function_name`.

25.1.3 Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example

Save this code in the file `mymodule.py`

```
person1 = {
    "name": "John",
    "age": 36,
    "country": "Norway"
}
```

Example

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

```
36
```

25.1.4 Naming a Module

You can name the module file whatever you like, but it must have the file extension `.py`

25.1.5 Re-naming a Module

You can create an alias when you import a module, by using the `as` keyword:

Example

Create an alias for `mymodule` called `mx`:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

```
36
```

25.1.6 Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

Example

Import and use the `platform` module:

```
import platform

x = platform.system()
print(x)
```

```
Linux
```

25.1.7 Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module. The `dir()` function:

Example

List all the defined names belonging to the platform module:

```
import platform

x = dir(platform)
print(x)
```

```
['DEV_NULL', '_UNIXCONFDIR', 'WIN32_CLIENT_RELEASES',
'WIN32_SERVER_RELEASES', '__builtins__', '__cached__',
'__copyright__', '__doc__', '__file__', '__loader__', '__name__',
'__package __', '__spec__', '__version__',
'_default_architecture', '_dist_try_harder', '_follow_symlinks',
'_ironpython26_sys_version_parser',
'_ironpython_sys_version_parser', '_java_getprop', '_libc_search',
'_linux_distribution', '_lsb_release_version', '_mac_ver_xml',
'_node', '_norm_version', '_perse_release_file', '_platform',
'_platform_cache', '_pypy_sys_version_parser',
'_release_filename', '_release_version', '_supported_dists',
'_sys_version', '_sys_version_cache', '_sys_version_parser',
'_syscmd_file', '_syscmd_uname', '_syscmd_ver', '_uname_cache',
'_ver_output', 'architecture', 'collections', 'dist', 'java_ver',
'libc_ver', 'linux_distribution', 'mac_ver', 'machine', 'node',
'os', 'platform', 'popen', 'processor', 'python_branch',
'python_build', 'python_compiler', 'python_implementation',
'python_revision', 'python_version', 'python_version_tuple', 're',
'release', 'subprocess', 'sys', 'system', 'system_aliases',
'uname', 'uname_result', 'version', 'warnings', 'win32_ver']
```

Note

The `dir()` function can be used on all modules, also the ones you create yourself.

25.1.8 Import From Module

You can choose to import only parts from a module, by using the `from` keyword.

Example

The module named `mymodule` has one function and one dictionary:

```
def greeting(name):
    print("Hello, " + name)

person1 = {
    "name": "John",
    "age": 36,
    "country": "Norway"
}
```

Example

Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

```
36
```

Note

When importing using the `from` keyword, do not use the module name when referring to elements in the module. Example: `person1["age"]`, **not** ~~`mymodule.person1["age"]`~~

25.1.9 Test Yourself With Exercises

Test

What is the correct syntax to import a module named "mymodule"?

```
        mymodule
```

25.2 Datetime Module

25.2.1 Python Dates

A date in Python is not a data type of its own, but we can import a module named `datetime` to work with dates as date objects.

Example

Import the datetime module and display the current date:

```
import datetime

x = datetime.datetime.now()
print(x)
```

```
2022-07-23 13:13:36.521911
```

25.2.2 Date Output

When we execute the code from the example above the result will be:
`2022-07-23 13:11:52.114696`
The date contains year, month, day, hour, minute, second, and microsecond.
The `datetime` module has many methods to return information about the date object.
Here are a few examples, you will learn more about them later in this chapter:

Example

Return the year and name of weekday:

```
import datetime

x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))
```

```
2022
Saturday
```

25.2.3 Creating Date Objects

To create a date, we can use the `datetime()` class (constructor) of the `datetime` module.
The `datetime()` class requires three parameters to create a date: year, month, day.

Example

Create a date object:

```
import datetime

x = datetime.datetime(2020, 5, 17)

print(x)
```

```
2020-05-17 00:00:00
```

The `datetime()` class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of `0`, (`None` for timezone).

25.2.4 The strftime() Method

The `datetime` object has a method for formatting date objects into readable strings.
The method is called `strftime()`, and takes one parameter, format, to specify the `format` of the returned string:

Example

Display the name of the month:

```
import datetime

x = datetime.datetime(2018, 6, 1)

print(x.strftime("%B"))
```

```
June
```

A reference of all the legal format codes:

Directive	Description	Example
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	3
%d	Day of month 01-31	31
%b	Month name, short version	Dec
%B	Month name, full version	December
%m	Month as a number 01-12	12
%y	Year, short version, without century	18
%Y	Year, full version	2018
%H	Hour 00-23	17
%I	Hour 00-12	05
%p	AM/PM	PM
%M	Minute 00-59	41
%S	Second 00-59	08
%f	Microsecond 000000-999999	548513
%z	UTC offset	+0100
%Z	Timezone	CST
%j	Day number of year 001-366	365
%U	Week number of year, Sunday as the first day of week, 00-53	52
%W	Week number of year, Monday as the first day of week, 00-53	52
%c	Local version of date and time	Mon Dec 31 17:41:00 2018
%C	Century	20
%x	Local version of date	12/31/18
%X	Local version of time	17:41:00
%%	A % character	%
%G	ISO 8601 year	2018
%u	ISO 8601 weekday (1-7)	1
%V	ISO 8601 weeknumber (01-53)	01

25.3 Math

Python has a set of built-in math functions, including an extensive math module, that allows you to perform mathematical tasks on numbers.

25.3.1 Built-in Math Functions

The `min()` and `max()` functions can be used to find the lowest or highest value in an iterable:

Example

```
x = min(5, 10, 25)
y = max(5, 10, 25)

print(x)
print(y)
```

```
5
25
```

The `abs()` function returns the absolute (positive) value of the specified number:

Example

```
x = abs(-7.25)

print(x)
```

```
7.25
```

The `pow(x, y)` function returns the value of x to the power of y (x^y).

Example

Return the value of 4 to the power of 3 (same as 4 * 4 * 4):

```
x = pow(4, 3)

print(x)
```

```
64
```

25.3.2 The Math Module

Python has also a built-in module called `math`, which extends the list of mathematical functions. To use it, you must import the `math` module:

```
import math
```

When you have imported the `math` module, you can start using methods and constants of the module.

The `math.sqrt()` method for example, returns the square root of a number:

Example

```
import math

x = math.sqrt(64)

print(x)
```

```
8.0
```

The `math.ceil()` method rounds a number upwards to its nearest integer, and the `math.floor()` method rounds a number downwards to its nearest integer, and returns the result:

Example

```
import math

x = math.ceil(1.4)
y = math.floor(1.4)

print(x) # returns 2
print(y) # returns 1
```

```
2
1
```

The `math.pi` constant, returns the value of PI (3.14...):

Example

```
import math

x = math.pi

print(x)
```

```
3.141592653589793
```

25.3.3 Complete Math Module Reference

In Math Module Reference you will find a complete reference of all methods and constants that belongs to the Math module (Section ??).

25.4 JSON Module

JSON is a syntax for storing and exchanging data.
JSON is text, written with JavaScript object notation.

25.4.1 JSON in Python

Python has a built-in package called `json`, which can be used to work with JSON data.

Example

Import the json module:

```
import json
```

25.4.2 Parse JSON - Convert from JSON to Python

If you have a JSON string, you can parse it by using the `json.loads()` method.

Note

The result will be a Python dictionary.

Example

Convert from JSON to Python:

```
import json

# some JSON:
x = '{ "name":"John", "age":30, "city":"New York" }'

# parse x:
y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])
```

```
30
```

25.4.3 Convert from Python to JSON

If you have a Python object, you can convert it into a JSON string by using the `json.dumps()` method.

Example

Convert from Python to JSON:

```
import json

# a Python object (dict):
x = {
  "name": "John",
  "age": 30,
  "city": "New York"
}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
print(y)
```

```
{"name": "John", "age": 30, "city": "New York"}
```

You can convert Python objects of the following types, into JSON strings:

- dict
- list
- tuple
- string
- int
- float
- True
- False
- None

Example

Convert Python objects into JSON strings, and print the values:

```
import json

print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

```
{"name": "John", "age": 30}
["apple", "bananas"]
["apple", "bananas"]
"hello"
42
31.76
true
false
null
```

When you convert from Python to JSON, Python objects are converted into the JSON (JavaScript) equivalent:

Python	JSON
dict	Object
list	Array
tuple	Array
str	String
int	Number
float	Number
True	true
False	false
None	null

Example

Convert a Python object containing all the legal data types:

```
import json

x = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
  "children": ("Ann","Billy"),
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}

print(json.dumps(x))
```

```
{"name": "John", "age": 30, "married": true, "divorced": false,
"children": ["Ann", "Billy"], "pets": null,
"cars": [{"model": "BMW 230", "mpg": 27.5},
{"model": "Ford Edge", "mpg": 24.1}]}
```

25.4.4 Format the Result

The example above prints a JSON string, but it is not very easy to read, with no indentations and line breaks.

The `json.dumps()` method has parameters to make it easier to read the result:

Example

Use the `indent` parameter to define the numbers of indents:

```
json.dumps(x, indent=4)
```

Example

```python
import json

x = {
   "name": "John",
   "age": 30,
   "married": True,
   "divorced": False,
   "children": ("Ann","Billy"),
   "pets": None,
   "cars": [
      {"model": "BMW 230", "mpg": 27.5},
      {"model": "Ford Edge", "mpg": 24.1}
   ]
}

# use four indents to make it easier to read the result:
print(json.dumps(x, indent=4))
```

```
{
    "name": "John",
    "age": 30,
    "married": true,
    "divorced": false,
    "children": [
        "Ann",
        "Billy"
    ],
    "pets": null,
    "cars": [
        {
            "model": "BMW 230",
            "mpg": 27.5
        },
        {
            "model": "Ford Edge",
            "mpg": 24.1
        }
    ]
}
```

You can also define the separators, default value is (`", ", ": "`), which means using a comma and a space to separate each object, and a colon and a space to separate keys from values:

Example

Use the separators parameter to change the default separator:

```
json.dumps(x, indent = 4, separators = (". ", " = "))
```

```
{
    "name" = "John".
    "age" = 30
}
```

25.4.5 Order the Result

The `json.dumps()` method has parameters to order the keys in the result:

Example

Use the `sort_keys` parameter to specify if the result should be sorted or not:

```
import json

x = {
   "name": "John",
   "children": ("Ann","Billy"),
   "cars": [
      {"model": "BMW 230", "mpg": 27.5},
   ]
}

# sort the result alphabetically by keys:
print(json.dumps(x, indent=4, sort_keys=True))
```

```
{
    "cars": [
        {
            "model": "BMW 230",
            "mpg": 27.5
        }
    ],
    "children": [
        "Ann",
        "Billy"
    ],
    "name": "John"
}
```

25.5 RegEx Module

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.
RegEx can be used to check if a string contains the specified search pattern.
Python has a built-in package called `re`, which can be used to work with Regular Expressions.
Import the `re` module:

```
import re
```

25.5.1 RegEx in Python

When you have imported the `re` module, you can start using regular expressions:

Example

Search the string to see if it starts with "The" and ends with "Spain":

```
import re

#Check if the string starts with "The" and ends with "Spain":

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)

if x:
   print("YES! We have a match!")
else:
   print("No match")
```

```
YES! We have a match!
```

25.5.2 RegEx Functions

The `re` module offers a set of functions that allows us to search a string for a match:

Function	Description
`findall`	Returns a list containing all matches
`search`	Returns a Match object if there is a match anywhere in the string
`split`	Returns a list where the string has been split at each match
`sub`	Replaces one or many matches with a string

25.5.3 Metacharacters

Metacharacters are characters with a special meaning:

Character	Description	Example
`[]`	A set of characters	`"[a-m]"`
`\`	Signals a special sequence (can also be used to escape special characters)	`"\d"`
`.`	Any character (except newline character)	`"he..o"`
`^`	Starts with	`"^hello"`
`$`	Ends with	`"planet$"`
`*`	Zero or more occurrences	`"he.*o"`
`+`	One or more occurrences	`"he.+o"`
`?`	Zero or one occurrences	`"he.?o"`
`{}`	Exactly the specified number of occurrences	`"he.{2}o"`
`|`	Either or	`"falls|stays"`
`()`	Capture and group	

25.5.4 Sets

A set is a set of characters inside a pair of square brackets `[]` with a special meaning:

Set	Description
`[arn]`	Returns a match where one of the specified characters (a, r, or n) are present
`[a-n]`	Returns a match for any lower case character, alphabetically between a and n
`[^arn]`	Returns a match for any character EXCEPT a, r, and n
`[0123]`	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
`[0-9]`	Returns a match for any digit between 0 and 9
`[0-5][0-9]`	Returns a match for any two-digit numbers from 00 and 59
`[a-zA-Z]`	Returns a match for any character alphabetically between a and z, lower case OR upper case
`[+]`	In sets, `+`, `*`, `.`, `|`, `()`, `$`,`{}` has no special meaning, so `[+]` means: return a match for any + character in the string

25.5.5 Special Sequences

A special sequence is a `\` followed by one of the characters in the list below, and has a special meaning:

Character	Description	Example
`\A`	Returns a match if the specified characters are at the beginning of the string	`"\AThe"`
`\b`	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	`r"\bain"r"ain\b"`
`\B`	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	`r"\Bain"r"ain\B"`
`\d`	Returns a match where the string contains digits (numbers from 0-9)	`"\d"`
`\D`	Returns a match where the string DOES NOT contain digits	`"\D"`
`\s`	Returns a match where the string contains a white space character	`"\s"`
`\S`	Returns a match where the string DOES NOT contain a white space character	`"\S"`
`\w`	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	`"\w"`
`\W`	Returns a match where the string DOES NOT contain any word characters	`"\W"`
`\Z`	Returns a match if the specified characters are at the end of the string	`"Spain\Z"`

25.5.6 The findall() Function

The `findall()` function returns a list containing all matches.

Example

Print a list of all matches:

```
import re

# Return a list containing every occurrence of "ai":

txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

```
['ai', 'ai']
```

The list contains the matches in the order they are found.
If no matches are found, an empty list is returned:

Example

Return an empty list if no match was found:

```
import re

txt = "The rain in Spain"

# Check if "Portugal" is in the string:

x = re.findall("Portugal", txt)
print(x)

if (x):
   print("Yes, there is at least one match!")
else:
   print("No match")
```

```
[]
No match
```

25.5.7 The search() Function

The `search()` function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

Example

Search for the first white-space character in the string:

```
import re

txt = "The rain in Spain"
x = re.search("\s", txt)

print("The first white-space character is located in position:",
   x.start())
```

```
The first white-space character is located in position: 3
```

If no matches are found, the value `None` is returned:

Example

Make a search that returns no match:

```
import re

txt = "The rain in Spain"
x = re.search("Portugal", txt)
print(x)
```

```
None
```

25.5.8 The split() Function

The `split()` function returns a list where the string has been split at each match:

Example

Split at each white-space character:

```
import re

# Split the string at every white-space character:

txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

```
['The', 'rain', 'in', 'Spain']
```

You can control the number of occurrences by specifying the `maxsplit` parameter:

Example

Split the string only at the first occurrence:

```
import re

# Split the string at the first white-space character:

txt = "The rain in Spain"
x = re.split("\s", txt, 1)
print(x)
```

```
['The', 'rain in Spain']
```

25.5.9 The sub() Function

The `sub()` function replaces the matches with the text of your choice:

Example

Replace every white-space character with the number 9:

```
import re

txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)
```

```
The9rain9in9Spain
```

You can control the number of replacements by specifying the `count` parameter:

Example

Replace the first 2 occurrences:

```
import re

# Replace the first two occurrences of a white-space character with the
    digit 9:

txt = "The rain in Spain"
x = re.sub("\s", "9", txt, 2)
print(x)
```

```
The9rain9in Spain
```

25.5.10 Match Object

A Match Object is an object containing information about the search and the result.

Note

If there is no match, the value `None` will be returned, instead of the Match Object

Example

Do a search that will return a Match Object:

```
import re

txt = "The rain in Spain"
x = re.search("ai", txt)
print(x) # this will print an object
```

```
<re.Match object; span=(5, 7), match='ai'>
```

The Match object has properties and methods used to retrieve information about the search, and the result:

- `.span()` returns a tuple containing the start-, and end positions of the match.
- `.string` returns the string passed into the function
- `.group()` returns the part of the string where there was a match

Example

Print the position (start- and end-position) of the first match occurrence.

The regular expression looks for any words that starts with an upper case "S":

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
```

```
(12, 17)
```

Example

Print the string passed into the function:

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
```

```
The rain in Spain
```

Example

Print the part of the string where there was a match.

The regular expression looks for any words that starts with an upper case "S":

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

```
Spain
```

Note

If there is no match, the value `None` will be returned, instead of the Match Object.

26 Python PIP

26.1 What is PIP?

PIP is a package manager for Python packages, or modules if you like.

Note

If you have Python version 3.4 or later, PIP is included by default.

26.2 What is a Package?

A package contains all the files you need for a module.
Modules are Python code libraries you can include in your project.

26.3 Check if PIP is Installed

Navigate your command line to the location of Python's script directory, and type the following:

```
$ pip --version
pip 22.1.2 from /usr/lib/python3.10/site-packages/pip (python 3.10)
$ _
```

26.4 Install PIP

If you do not have PIP installed, you can download and install it from this page: pypi.org/project/pip

26.5 Download a Package

Downloading a package is very easy.
Open the command line interface and tell PIP to download the package you want.
Navigate your command line to the location of Python's script directory, and type the following:

```
$ pip install camelcase_
```

Now you have downloaded and installed your first package!

26.6 Using a Package

Once the package is installed, it is ready to use.
Import the "camelcase" package into your project.

Example

Import and use "camelcase":

```
import camelcase

c = camelcase.CamelCase()

txt = "lorem ipsum dolor sit amet"

print(c.hump(txt))

#This method capitalizes the first letter of each word.
```

```
Lorem Ipsum Dolor Sit Amet
```

26.7 Find Packages

Find more packages at pypi.org.

26.8 Remove a Package

Use the `uninstall` command to remove a package:

```
$ pip uninstall camelcase_
```

The PIP Package Manager will ask you to confirm that you want to remove the camelcase package:

```
Uninstalling camelcase-02.1:
        Would remove:
                ...
Proceed (y/n)?
```

Press y and the package will be removed.

26.9 List Packages

Use the `list` command to list all the packages installed on your system:

```
$ pip list
Package         Version
------------------------
camelcase       0.2
mysql-connector 2.1.6
pip             18.1
pymongo         3.6.1
setuptools      39.0.1
$_
```

27 try ... execpt

The `try` block lets you test a block of code for errors.
The `except` block lets you handle the error.
The `else` block lets you execute code when there is no error.
The `finally` block lets you execute code, regardless of the result of the try and except blocks.

27.1 Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
These exceptions can be handled using the `try` statement:

Example

The `try` block will generate an exception, because x is not defined:

```
try:
   print(x)
except:
   print("An exception occurred")
```

```
An exception occurred
```

Example

This statement will raise an error, because `x` is not defined:

```
#This will raise an exception, because x is not defined:

print(x)
```

```
Traceback (most recent call last):
  File "demo_try_except_error.py", line 3, in <module>
    print(x)
NameError: name 'x' is not defined
```

27.2 Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a `NameError` and another for other errors:

```
#The try block will generate a NameError, because x is not defined:

try:
   print(x)
except NameError:
   print("Variable x is not defined")
except:
   print("Something else went wrong")
```

```
Variable x is not defined
```

27.3 *else*

You can use the `else` keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the `try` block does not generate any error:

```
try:
   print("Hello")
except:
   print("Something went wrong")
else:
   print("Nothing went wrong")
```

```
Hello
Nothing went wrong
```

27.4 *finally*

The `finally` block, if specified, will be executed regardless if the try block raises an error or not.

Example

```
try:
   print(x)
except:
   print("Something went wrong")
finally:
   print("The 'try except' is finished")
```

```
Something went wrong
The 'try except' is finished
```

This can be useful to close objects and clean up resources:

Example

Try to open and write to a file that is not writable:

```
# The try block will raise an error when trying to write to a read-only
   file:

try:
   f = open("demofile.txt")
   try:
      f.write("Lorum Ipsum")
   except:
      print("Something went wrong when writing to the file")
   finally:
      f.close()
except:
   print("Something went wrong when opening the file")
```

```
Something went wrong when writing to the file
```

The program can continue, without leaving the file object open.

27.5 *raise* an *exception*

As a Python developer you can choose to throw an exception if a condition occurs.
To throw (or raise) an exception, use the `raise` keyword.

Example

Raise an error and stop the program if x is lower than 0:

```
x = -1

if x < 0:
   raise Exception("Sorry, no numbers below zero")
```

```
Traceback (most recent call last):
  File "demo_ref_keyword_raise.py", line 4, in <module>
    raise Exception("Sorry, no numbers below zero")
Exception: Sorry, no numbers below zero
```

The `raise` keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

Example

Raise a TypeError if x is not an integer:

```
x = "hello"

if not type(x) is int:
   raise TypeError("Only integers are allowed")
```

```
Traceback (most recent call last):
  File "demo_ref_keyword_raise2.py", line 4, in <module>
    raise TypeError("Only integers are allowed")
TypeError: Only integers are allowed
```

28 User *input*

Python allows for user input.
That means we are able to ask the user for input.
Python 3.X uses the `input()` method.
The following example asks for the username, and when you entered the username, it gets printed on the screen:

Example

```
username = input("Enter username:")
print("Username is: " + username)
```

```
Enter username: my user name
Username is: my user name
```

Note

Python stops executing when it comes to the `input()` function, and continues when the user has given some input.

29 String Formatting

To make sure a string will display as expected, we can format the result with the `format()` method.

29.1 String format()

The `format()` method allows you to format selected parts of a string.
Sometimes there are parts of a text that you do not control, maybe they come from a database, or user input?
To control such values, add placeholders (curly brackets `{}`) in the text, and run the values through the `format()` method:

Example

Add a placeholder where you want to display the price:

```
price = 49
txt = "The price is {} dollars"
print(txt.format(price))
```

```
The price is 49 dollars
```

You can add parameters inside the curly brackets to specify how to convert the value:

Example

```
price = 49
txt = "The price is {:.2f} dollars"
print(txt.format(price))
```

```
The price is 49.00 dollars
```

Check out all formatting types in String format() Reference (Section ??)

29.2 Multiple Values

If you want to use more values, just add more values to the format() method:

```
print(txt.format(price, itemno, count))
```

And add more placeholders:

Example

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {} pieces of item number {} for {:.2f} dollars.
    "
print(myorder.format(quantity, itemno, price))
```

```
I want 3 pieces of item number 567 for 49.00 dollars.
```

29.3 Index Numbers

You can use index numbers (a number inside the curly brackets `{0}`) to be sure the values are placed in the correct placeholders:

Example

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {0} pieces of item number {1} for {2:.2f}
   dollars."
print(myorder.format(quantity, itemno, price))
```

```
I want 3 pieces of item number 567 for 49.00 dollars.
```

Also, if you want to refer to the same value more than once, use the index number:

Example

```
age = 36
name = "John"
txt = "His name is {1}. {1} is {0} years old."
print(txt.format(age, name))
```

```
His name is John. John is 36 years old.
```

29.4 Named Indexes

You can also use named indexes by entering a name inside the curly brackets `{carname}`, but then you must use names when you pass the parameter values `txt.format(carname = "Ford")`:

Example

```
myorder = "I have a {carname}, it is a {model}."
print(myorder.format(carname = "Ford", model = "Mustang"))
```

```
I have a Ford, it is a Mustang.
```

30 File Handling

File handling is an important part of any web application.
Python has several functions for creating, reading, updating, and deleting files.

30.1 Python File Open

The key function for working with files in Python is the `open()` function.
The `open()` function takes two parameters; filename, and mode.
There are four different methods (modes) for opening a file:

- `"r"` - Read - Default value. Opens a file for reading, error if the file does not exist
- `"a"` - Append - Opens a file for appending, creates the file if it does not exist
- `"w"` - Write - Opens a file for writing, creates the file if it does not exist
- `"x"` - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

- `"t"` - Text - Default value. Text mode
- `"b"` - Binary - Binary mode (e.g. images)

30.1.1 Syntax

To open a file for reading it is enough to specify the name of the file:

Example

```
f = open("demofile.txt")
```

The code above is the same as:

Example

```
f = open("demofile.txt", "rt")
```

Because `"r"` for read, and `"t"` for text are the default values, you do not need to specify them.

Note

Make sure the file exists, or else you will get an error.

30.2 Python File Read

Assume we have the following file, located in the same folder as Python:

demofile.txt:

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in `open()` function.
The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

Example

```
1 f = open("demofile.txt", "r")
2 print(f.read())
```

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

If the file is located in a different location, you will have to specify the file path, like this:

Example

Open a file on a different location:

```
1 f = open("D:\\myfiles\welcome.txt", "r")
2 print(f.read())
```

```
Welcome to this text file!
This file is located in a folder named "myfiles", on the D drive.
Good Luck!
```

30.2.1 Read Only Parts of the File

By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

Example

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

```
Hello
```

30.2.2 Read Lines

You can return one line by using the `readline()` method:

Example

Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

```
Hello! Welcome to demofile.txt
```

Example

Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
```

By looping through the lines of the file, you can read the whole file, line by line:

Example

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
   print(x)
```

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

30.3 File close()

It is a good practice to always close the file when you are done with it.

30.3.1 Definition and Usage

The `close()` method closes an open file.

30.3.2 Syntax

```
file.close()
```

Example

Close a file after it has been opened:

```
f = open("demofile.txt", "r")
print(f.read())
f.close()
```

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

Note

You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

30.4 Python File Write

30.4.1 Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:

- `"a"` - Append - will append to the end of the file
- `"w"` - Write - will overwrite any existing content

Example

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

```
Hello! Welcome to demofile2.txt
This file is for testing purposes.
Good Luck!Now the file has more content!
```

Example

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

```
Woops! I have deleted the content!
```

Note

the `"w"` method will overwrite the entire file.

30.4.2 Create a New File

To create a new file in Python, use the `open()` method, with one of the following parameters:

- `"x"` - Create - Creates the specified file, returns an error if the file exists
- `"a"` - Append - Opens a file for appending, creates the file if it does not exist
- `"w"` - Write - Opens a file for writing, creates the file if it does not exist

Example

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

30.5 Delete File

To delete a file, you must import the OS module, and run its `os.remove()` function:

Example

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

30.6 Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
   os.remove("demofile.txt")
else:
   print("The file does not exist")
```

30.6.1 Delete Folder

To delete an entire folder, use the `os.rmdir()` method:

Example

Remove the folder “myfolder”:

```
1 import os
2 os.rmdir("myfolder")
```

Note

You can only remove *empty* folders.

31 Methods

31.1 String Methods

Note

All string methods returns new values. They do not change the original string.

Method	Description
`capitalize()`	Converts the first character to upper case
`casefold()`	Converts string into lower case
`center()`	Returns a centered string
`count()`	Returns the number of times a specified value occurs in a string
`encode()`	Returns an encoded version of the string
`endswith()`	Returns true if the string ends with the specified value
`expandtabs()`	Sets the tab size of the string
`find()`	Searches the string for a specified value and returns the position of where it was found
`format()`	Formats specified values in a string
`format_map()`	Formats specified values in a string
`index()`	Searches the string for a specified value and returns the position of where it was found
`isalnum()`	Returns True if all characters in the string are alphanumeric
`isalpha()`	Returns True if all characters in the string are in the alphabet
`isdecimal()`	Returns True if all characters in the string are decimals
`isdigit()`	Returns True if all characters in the string are digits
`isidentifier()`	Returns True if the string is an identifier
`islower()`	Returns True if all characters in the string are lower case
`isnumeric()`	Returns True if all characters in the string are numeric
`isprintable()`	Returns True if all characters in the string are printable
`isspace()`	Returns True if all characters in the string are whitespaces
`istitle()`	Returns True if the string follows the rules of a title
`isupper()`	Returns True if all characters in the string are upper case
`join()`	Joins the elements of an iterable to the end of the string
`ljust()`	Returns a left justified version of the string
`lower()`	Converts a string into lower case
`lstrip()`	Returns a left trim version of the string
`maketrans()`	Returns a translation table to be used in translations
`partition()`	Returns a tuple where the string is parted into three parts
`replace()`	Returns a string where a specified value is replaced with a specified value

`rfind()`	Searches the string for a specified value and returns the last position of where it was found
`rindex()`	Searches the string for a specified value and returns the last position of where it was found
`rjust()`	Returns a right justified version of the string
`rpartition()`	Returns a tuple where the string is parted into three parts
`rsplit()`	Splits the string at the specified separator, and returns a list
`rstrip()`	Returns a right trim version of the string
`split()`	Splits the string at the specified separator, and returns a list
`splitlines()`	Splits the string at line breaks and returns a list
`startswith()`	Returns true if the string starts with the specified value
`strip()`	Returns a trimmed version of the string
`swapcase()`	Swaps cases, lower case becomes upper case and vice versa
`title()`	Converts the first character of each word to upper case
`translate()`	Returns a translated string
`upper()`	Converts a string into upper case
`zfill()`	Fills the string with a specified number of 0 values at the beginning

31.2 List Methods

Method	Description
`append()`	Adds an element at the end of the list
`clear()`	Removes all the elements from the list
`copy()`	Returns a copy of the list
`count()`	Returns the number of elements with the specified value
`extend()`	Add the elements of a list (or any iterable), to the end of the current list
`index()`	Returns the index of the first element with the specified value
`insert()`	Adds an element at the specified position
`pop()`	Removes the element at the specified position
`remove()`	Removes the item with the specified value
`reverse()`	Reverses the order of the list
`sort()`	Sorts the list

31.3 Tuple Methods

Method	Description
`count()`	Returns the number of times a specified value occurs in a tuple
`index()`	Searches the tuple for a specified value and returns the position of where it was found

31.4 Set Methods

Method	Description
`add()`	Adds an element to the set
`clear()`	Removes all the elements from the set
`copy()`	Returns a copy of the set
`difference()`	Returns a set containing the difference between two or more sets
`difference_update()`	Removes the items in this set that are also included in another, specified set
`discard()`	Remove the specified item
`intersection()`	Returns a set, that is the intersection of two other sets
`intersection_update()`	Removes the items in this set that are not present in other, specified set(s)
`isdisjoint()`	Returns whether two sets have a intersection or not
`issubset()`	Returns whether another set contains this set or not
`issuperset()`	Returns whether this set contains another set or not
`pop()`	Removes an element from the set
`remove()`	Removes the specified element
`symmetric_difference()`	Returns a set with the symmetric differences of two sets
`symmetric_difference_update()`	inserts the symmetric differences from this set and another
`union()`	Return a set containing the union of sets
`update()`	Update the set with the union of this set and others

31.5 Dictionary Methods

Method	Description
`clear()`	Removes all the elements from the dictionary
`copy()`	Returns a copy of the dictionary
`fromkeys()`	Returns a dictionary with the specified keys and value
`get()`	Returns the value of the specified key
`items()`	Returns a list containing a tuple for each key value pair
`keys()`	Returns a list containing the dictionary's keys
`pop()`	Removes the element with the specified key
`popitem()`	Removes the last inserted key-value pair
`setdefault()`	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
`update()`	Updates the dictionary with the specified key-value pairs
`values()`	Returns a list of all the values in the dictionary

31.6 Array Methods

Method	Description
`append()`	Adds an element at the end of the list
`clear()`	Removes all the elements from the list
`copy()`	Returns a copy of the list
`count()`	Returns the number of elements with the specified value
`extend()`	Add the elements of a list (or any iterable), to the end of the current list
`index()`	Returns the index of the first element with the specified value
`insert()`	Adds an element at the specified position
`pop()`	Removes the element at the specified position
`remove()`	Removes the first item with the specified value
`reverse()`	Reverses the order of the list
`sort()`	Sorts the list

31.7 *statistics* Module Methods

Python has a built-in module that you can use to calculate mathematical statistics of numeric data. The `statistics` module was new in Python 3.4.

Method	Description
`statistics.harmonic_mean()`	Calculates the harmonic mean (central location) of the given data
`statistics.mean()`	Calculates the mean (average) of the given data
`statistics.median()`	Calculates the median (middle value) of the given data
`statistics.median_grouped()`	Calculates the median of grouped continuous data
`statistics.median_high()`	Calculates the high median of the given data
`statistics.median_low()`	Calculates the low median of the given data
`statistics.mode()`	Calculates the mode (central tendency) of the given numeric or nominal data
`statistics.pstdev()`	Calculates the standard deviation from an entire population
`statistics.stdev()`	Calculates the standard deviation from a sample of data
`statistics.pvariance()`	Calculates the variance of an entire population
`statistics.variance()`	Calculates the variance from a sample of data

31.8 *math* Module

Method	Description
`acos()`	Returns the arc cosine of a number
`acosh()`	Returns the inverse hyperbolic cosine of a number
`asin()`	Returns the arc sine of a number
`asinh()`	Returns the inverse hyperbolic sine of a number
`atan()`	Returns the arc tangent of a number in radians
`atan2()`	Returns the arc tangent of y/x in radians
`atanh()`	Returns the inverse hyperbolic tangent of a number
`ceil()`	Rounds a number up to the nearest integer
`comb()`	Returns the number of ways to choose k items from n items without repetition and order
`copysign()`	Returns a float consisting of the value of the first parameter and the sign of the second parameter
`cos()`	Returns the cosine of a number
`cosh()`	Returns the hyperbolic cosine of a number

`degrees()`	Converts an angle from radians to degrees
`dist()`	Returns the Euclidean distance between two points (p and q), where p and q are the coordinates of that point
`erf()`	Returns the error function of a number
`erfc()`	Returns the complementary error function of a number
`exp()`	Returns E raised to the power of x
`expm1()`	Returns Ex - 1
`fabs()`	Returns the absolute value of a number
`factorial()`	Returns the factorial of a number
`floor()`	Rounds a number down to the nearest integer
`fmod()`	Returns the remainder of x/y
`frexp()`	Returns the mantissa and the exponent, of a specified number
`fsum()`	Returns the sum of all items in any iterable (tuples, arrays, lists, etc.)
`gamma()`	Returns the gamma function at x
`gcd()`	Returns the greatest common divisor of two integers
`hypot()`	Returns the Euclidean norm
`isclose()`	Checks whether two values are close to each other, or not
`isfinite()`	Checks whether a number is finite or not
`isinf()`	Checks whether a number is infinite or not
`isnan()`	Checks whether a value is NaN (not a number) or not
`isqrt()`	Rounds a square root number downwards to the nearest integer
`ldexp()`	Returns the inverse of math.frexp() which is x * (2**i) of the given numbers x and i
`lgamma()`	Returns the log gamma value of x
`log()`	Returns the natural logarithm of a number, or the logarithm of number to base
`log10()`	Returns the base-10 logarithm of x
`log1p()`	Returns the natural logarithm of 1+x
`log2()`	Returns the base-2 logarithm of x
`perm()`	Returns the number of ways to choose k items from n items with order and without repetition
`pow()`	Returns the value of x to the power of y
`prod()`	Returns the product of all the elements in an iterable
`radians()`	Converts a degree value into radians
`remainder()`	Returns the closest value that can make numerator completely divisible by the denominator
`sin()`	Returns the sine of a number
`sinh()`	Returns the hyperbolic sine of a number
`sqrt()`	Returns the square root of a number
`tan()`	Returns the tangent of a number
`tanh()`	Returns the hyperbolic tangent of a number
`trunc()`	Returns the truncated integer parts of a number

31.8.1 Math Constants

Constant	Description
`math.e`	Returns Euler's number (2.7182...)
`math.inf`	Returns a floating-point positive infinity
`math.nan`	Returns a floating-point NaN (Not a Number) value
`math.pi`	Returns PI (3.1415...)
`math.tau`	Returns tau (6.2831...)

31.9 *random* Module

Method	Description
`seed()`	Initialize the random number generator
`getstate()`	Returns the current internal state of the random number generator
`setstate()`	Restores the internal state of the random number generator
`getrandbits()`	Returns a number representing the random bits
`randrange()`	Returns a random number between the given range
`randint()`	Returns a random number between the given range
`choice()`	Returns a random element from the given sequence
`choices()`	Returns a list with a random selection from the given sequence
`shuffle()`	Takes a sequence and returns the sequence in a random order
`sample()`	Returns a given sample of a sequence
`random()`	Returns a random float number between 0 and 1
`uniform()`	Returns a random float number between two given parameters
`triangular()`	Returns a random float number between two given parameters, you can also set a mode parameter to specify the midpoint between the two other parameters
`betavariate()`	Returns a random float number between 0 and 1 based on the Beta distribution (used in statistics)
`expovariate()`	Returns a random float number based on the Exponential distribution (used in statistics)
`gammavariate()`	Returns a random float number based on the Gamma distribution (used in statistics)
`gauss()`	Returns a random float number based on the Gaussian distribution (used in probability theories)

`lognormvariate()`	Returns a random float number based on a log-normal distribution (used in probability theories)
`normalvariate()`	Returns a random float number based on the normal distribution (used in probability theories)
`vonmisesvariate()`	Returns a random float number based on the von Mises distribution (used in directional statistics)
`paretovariate()`	Returns a random float number based on the Pareto distribution (used in probability theories)
`weibullvariate()`	Returns a random float number based on the Weibull distribution (used in statistics)

31.10 *requests* Module Methods

Example

Make a request to a web page, and print the response text:

```
import requests

x = requests.get('https://w3schools.com/python/demopage.htm')

print(x.text)
```

```
<!DOCTYPE html>
<html>
<body>

<h1>This is a Test Page</h1>

</body>
</html>
```

31.10.1 Definition and Usage

The `requests` module allows you to send HTTP requests using Python.
The HTTP request returns a Response Object with all the response data (content, encoding, status, etc).

31.10.2 Download and Install the Requests Module

Navigate your command line to the location of PIP, and type the following:

```
$ pip install requests
```

31.10.3 Syntax

```
requests.methodname(params)
```

31.10.4 Methods

Method	Description
`delete(url, args)`	Sends a DELETE request to the specified url
`get(url, params, args)`	Sends a GET request to the specified url
`head(url, args)`	Sends a HEAD request to the specified url
`patch(url, data, args)`	Sends a PATCH request to the specified url
`post(url, data, json, args)`	Sends a POST request to the specified url
`put(url, data, args)`	Sends a PUT request to the specified url
`request(method, url, args)`	Sends a request of the specified method to the specified url