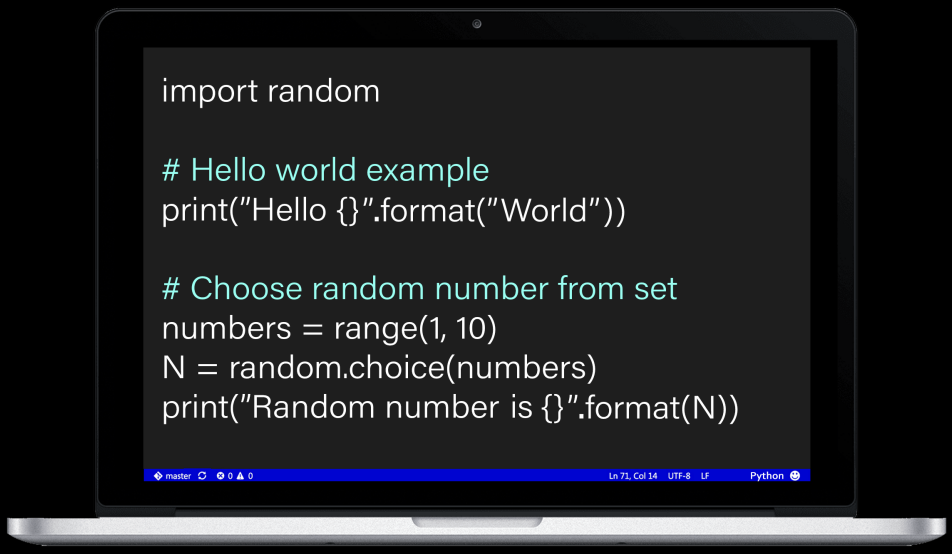
FIRST EDITION

PYTHON

GRAMMAR

Learn to speak Python



Learning Curve Books

DARK MODE

Title:	Python Grammar
Edition:	I – January 1, 2021
Genre:	Software Development
Imprint:	Independently published
ISBN:	9798646258084
Publisher:	Learning Curve Books
Author:	Learning Curve Books

Primary purpose of **Learning Curve Books** publishing company is to provide *effective education* for web designers, software engineers and all readers who are interested in being edified in the area of web development.

Python Grammar was written to help someone new to the Python Programming Language learn it from scratch without any previous programming experience.

For questions and comments about the book you may contact the author or send an email directly to our office at the email address mentioned below.

Email Us

learningcurvebook@gmail.com

—

Learning Curve Books™

Python Grammar

Chapter 1

Python Grammar

1.1 Preface

Every time a book is written it assumes some type of audience. This book, Python Grammar – *as the title suggests* – is about examining fundamentals of Python. It was written for someone new to Python language and learning it from scratch.

To avoid being a rudimentary guideline similar to Python tutorials that can be found online for free, some of the chapters will be augmented by providing slightly more complex examples. This is especially useful for those learners who seek *practical examples* or simply want to be challenged while still learning.

Python is a tool for problem solvers who appreciate writing clean, elegant code. It offers a vast array of simple language features. You might not need to know all of them – but learning them one at a time gets you closer to *Pythonic Thinking*.

Python is a type of language that provides common functionality from many packages that come with it by default. Most packages are congruent with the original feature set and play well together with existing data types.

Elegant code is not only code that looks clean, but the kind that also follows *logical aesthetic* – Python is perfect for writing logically elegant code without sacrificing performance – but only when its features are understood in context of their intended purpose. One goal of this book is to develop that awareness.

There is a proper and an improper way of writing code in any computer language.

This usually depends on the level of depth and familiarity with its feature set. As you continue to practice you will notice that Python, like many other languages, has its own "flavor" to writing code. This can be only learned from experience.

Many programmers that come to Python from other languages may experience a bit of *cognitive dissonance*. The goal of this book is to help readers develop *Pythonic Thinking* – a way of thinking about writing computer programs following intricate details of Python programming language.

1.1.1 Installation

Installing Python on either Mac or Windows is a relatively simple process. We'll briefly go into downloading the files, using Python shell and executing Python scripts with **python.exe** (or simply **python**) from the command line.

—

Note: installation chapter was moved to back of the book. This way we can dive right into Python examples. Find it there if you still need to install Python.

1.1.2 Python is a simple language that helps you deal with complex problems in an elegant way

Before getting good at Python's advanced features it's good to understand the fundamentals like printing text, creating variables, working with *data types*, *data structures* and *strings*, executing *file operations* for reading and writing data from|to your hard drive, command line output and a few other things.

Python is a simple language that helps you deal with complex problems in an elegant way. Many of its features have basic form. As a programmer you will use creativity and logic to put them together and write (hopefully) efficient code.

To master Python you will need to understand what those features are and learn how to use them in proper situation to avoid writing bad code.

1.1.3 Source Code

Source code will be presented in following format:

```
# print something
print("hello" * 2)
print(8 + 2)
```

Python source code will be followed by its output. To save space, the results will be printed right after >>> characters that follow source code examples:

```
>>>
hellohello
10
```

1.2 Thinking in Python

This book is a break-down of Python's most important features. But before we explore them, let's take a look at a practical Python program just to give the reader an idea of what Python code looks like.

We'll also learn how to "think in Python" by solving a simple bug.

This section demonstrates a simple program written in Python that converts kilometers to miles while expanding on some of its common features.

1.2.1 Convert Kilometers to Miles

```
# get user input in string format
kilometers = input("Enter number of kilometers:")

# convert kilometers to floating point number
kilometers = float(kilometers)

# km/mi conversion factor
factor = 0.621371

# determine miles from kilometers
miles = kilometers * factor

# print result to command line
print("%0.2f km = %0.2f mi" % (kilometers, miles))
```

```
>>>
Enter number of kilometers:10
10.00km = 6.21mi
```

Explanation: To convert kilometers to miles you have to multiply kilometers by the kilometers-to-miles conversion factor which is `0.621371`.

The built-in Python function `input()` will halt execution of your program to take user input directly from command line to produce a value. The input function returns whatever you typed in string format after you press Enter.

Multiplication operator `*` *can* work with strings, but that's not what we need here. This is why we converted kilometers variable to floating point (a decimal point number) using `float()` function, which can be thought of as a constructor function for floating point numbers.

Python provides several type constructor functions: `int()`, `float()`, `str()`, `complex()`, `tuple()`, `list()`, etc. Each creates a primitive or an object of that type. We'll take a closer look at them all in future chapters.

For now just know that values, like integers, can be converted to other types (decimal point numbers and strings, are some of the most common examples) by using the function that represents that type:

```
integer = 125
print(integer)

# convert integer to decimal number
decimal = float(integer)
print(decimal)

# convert decimal back to whole number
whole = int(decimal)
print(whole)

>>>
125
125.0
125
```

Integers are whole numbers. Converting or "casting" them to float produces a decimal point number. Here we converted an integer to a float and back.

One other common type you will often use is the *boolean* which has its own type constructor function `bool()`. It's a primitive value that can only have two possible states: `True` or `False`:

```
yes = True
no = False

print(yes)
print(no)

# initialize a boolean with one of its own values
print(bool(True))
print(bool(False))

# initialize a boolean with a number or string
print(bool(0))
print(bool(1))
print(bool(""))
```

```
>>>
True
False
True
False
False
True
False
```

Because *boolean* is limited to `True` or `False` the constructor function has to make some decisions when you try to initialize its value by using an *integer* or a *string*.

Empty string `""` will produce `False` when passed to `bool()` function. But a string containing some text will produce True. Similar to that, the integer `0` will be cast to `False` and integer `1` (or any positive *or* negative integer) to `True`:

```
print(bool(""))
print(bool("0"))
print(bool("Hello"))
print(bool(1))
print(bool(125))
```

```
print(bool(-1))
print(bool(0))
```

```
>>>
False
True
True
True
True
True
False
```

You can also check type of a value by using Python's built-in `type` function:

```
yes = True
no = False
n = 1
s = "hello there"

a = type(yes)
b = type(no)
c = type(n)
d = type(s)

print(a)
print(b)
print(c)
print(d)
```

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

Variables `a` and `b` are booleans, `n` is an integer, `s` is a string.

Why is type checking important?

Due to variety of types not all conversions are possible. In many cases Python will make a guess, but there are exceptions. Some conversions will fail.

What would it mean to convert a string of text like `"hello there"` to a decimal point number with `float()` function? It doesn't make much sense.

Solving the bug

The kilometers-to-miles conversion program we wrote earlier works only if you enter numeric values. But there is a bug. If you type a non-numeric value, like a random string of text: `asddasd` the program will fail with a nasty error:

```
Enter number of kilometers:asddasd

>>>
Traceback (most recent call last):
  File "km2mi.py", line 5, in <module>
    kilometers = float(kilometers)
ValueError: could not convert string to float: 'asddasd'
```

The code fails because the we tried to pass the string `"asddasd"` to the `float()` function which expects a numeric value.

By design the `input()` function always produces a string: if you enter `10`, it will return as `"10"`, `125.10` becomes `"125.10"`. Simply running `type()` function on it won't help us here, because we'll get `<class 'str'>` every time, regardless what value was entered.

Let's fix this bug by limiting the value produced by `input` function to numeric values. In Python primitive values have a method called `isnumeric()` that can be called directly on the variable name:

```
if kilometers.isnumeric():
    print("value is numeric!")
```

Knowing this we can update our original program to keep asking for input until a numeric value is entered:

```
# initialize to a non-numeric value
kilometers = ""

# while not loop to sift numeric values
while not kilometers.isnumeric():
    kilometers = input("Enter number of kilometers:")

kilometers = float(kilometers)
factor = 0.621371
miles = kilometers * factor

print("%0.2f km = %0.2f mi" % (kilometers, miles))

>>>
Enter number of kilometers:asdasd
Enter number of kilometers:sdfs
Enter number of kilometers:sdf
Enter number of kilometers:10
10.00 km = 6.21 mi
```

Only two lines were added to the original program to solve this bug.

The `while` loop executes a statement while a condition is met (that means it evaluates to boolean `True`). Our condition is "`kilometers` is not numeric" represented by the value returned from `kilometers.isnumeric()` function.

This means if you enter a string, the while loop will check if value entered was numeric. If not, the loop will execute the `input()` statement again. This goes on until a numeric value is recognized.

The definition of `kilometers` was moved to first line. It was initialized to a non-numeric value so that `input()` function inside the while loop has a chance to be executed for the first time at least once.

Now if user enters a gibberish string `"asdasd"`, the input function asks again until numeric value is entered. As you can see the bug is gone since by the time kilometers are converted to `float()` value is guaranteed to be numeric.

Chapter 2

Getting To Know Python

2.1 Zen of Python

Type **import this** into Python shell, or execute it as part of your program:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>> 
```

2.2 Basics

Simple syntax makes learning Python more fun than many other languages.

2.2.1 Statements and Expressions

Statements

Statements are building blocks of a computer program:

```
a = 1
```

Assigning a value to a variable is called an *assignment statement*.

Below is a for-*statement*, more commonly known simply as a for-*loop*. In this example it simply prints out each letter in a list:

```
for letter in ['a', 'b', 'c']: print(letter)
```

```
>>>
a
b
c
```

A for-loop becomes more powerful in combination with other statements.

For example in the next program we will print out and count only vowels in a title of a poem: "Say Not The Struggle Nought Availeth"

```
vowels = ('a', 'o', 'e', 'i', 'u')

title = "Say Not The Struggle Nought Availeth"

count = 0
for letter in title:
```

```
    if letter in vowels:
        print(letter)
        count += 1

print(f"There are {count} vowels in '{title}'")
```

```
>>>
a
o
e
u
e
o
u
a
i
e
There are 10 vowels in 'Say Not The Struggle Nought Availeth'
```

The following simple program consists of 4 statements:

```
a = 9
b = a
fstring = f"{a + b} messages"
print(fstring)
```

```
>>>
18 messages
```

This program defines variable `a` and assigns it value of 9. The second variable `b` is assigned the value of `a`. They are combined using in-line brackets `{a + b}` – a feature of formatted strings (or f-strings) which is a string that starts with `f`.

```
a = 1 + 2 + 3 +
    4 + 5 + 6
```

```
>>>
  File "multiline.py", line 1
    a = 1 + 2 + 3 +
                  ^
SyntaxError: invalid syntax
```

This multi-line statement produces an error because it's not escaped with backslash character. But it can be fixed as follows:

```
a = 1 + 2 + 3 + \
    4 + 5 + 6

print(a)

>>>
21
```

Alternatively, you can produce *line continuation* without using slash character by surrounding statements in parentheses:

```
a = (1 + 2 + 3 +
     4 + 5 + 6)

print(a)

>>>
21
```

Line continuation is preserved when defining lists on multiple lines:

```
colors = ['red',
          'green',
          'blue',
          'yellow']

>>>
```

No need for \ character here.

In Python [] brackets are used to define one of its special data structures called `list`. It's exactly what it sounds like: a list of values separated by comma. It's similar to arrays in other languages.

The statement above is an *assignment statement* because it uses an assignment operator. Three other common statements in Python are if-*statement*, for-*statement* and while-*statement*.

if-statements

Another special type of statement is the if-statement. It is often used with an *expression* and another statement. If that condition is met (in other words, if expression evaluates to boolean `True`) then the statement that follows will execute:

```
if True:
    print("Do something because it's true.")

if False:
    print("Don't do this because condition is false.")
    print("This text will never print to console.")

if 25 == 25:
    print("25 is 25")

if 5 * 5 == 25:
    print("5 * 5 is 25")

>>>
Do something because it's true.
25 is 25
5 * 5 is 25
```

You can use any valid Python expression with an if-statement:

```
if expression: statement
```

Expressions

An *expression* is a statement that produces a value (not all statements result in a value.) In other words, you can say that an expression *evaluates* to a value:

```
1 + 1
```

```
>>>
```

`1 + 1` is an expression that *evaluates* to 2. However, if you run this code nothing will be printed to the command prompt.

Even though expressions produce a value that value remains "silent" until it is printed out:

```
print(1 + 1)
print(4 + 3)
```

```
>>>
2
7
```

Expressions can be used in many different ways. They can be stored in variables or passed to functions as one of its arguments:

```
a = 12 + 1
b = 10 / 5

print(a, b)
```

```
>>>
13, 2.0
```

You can use expressions directly as function arguments:

```
print(1 + 4,
      10 * 2,
      "hello" + "there")

>>>
5 20 hellothere
```

A Python function that returns a value can also be thought of as an expression:

```
def func():
    return "hello"

msg = func()

print(msg)

>>>
hello
```

Here we defined a function called `func`. To call that function, use its name followed by parentheses `()`. All it does is return `"hello"` message, which is then stored in a variable named `msg`, and then sent as an argument to `print` function.

We'll see many examples of functions in this book and talk about them in more detail in a later chapter.

if-else

```
video_is_playing = False

if video_is_playing:
    print("video is playing")
else:
    print("video is not playing")
```

if and not

Sometimes it's more natural to inverse the logic of the expression in an if-statement. In this case you can use it together with the `not` keyword:

```
video_is_playing = False

if not video_is_playing:
    print("video is not playing")
else:
    print("video is playing")
```

The `not` keyword simply inverses the boolean condition. It's similar to `!` operator in other languages:

```
print(not True)     # False
print(not False)    # True
```

```
>>>
False
True
```

2.2.2 Semicolons

In most cases Python doesn't require semicolons. They can be added at the end of statements but mostly for optional aesthetic and readability (or lack thereof):

```
print("Hello");
```

Statement above is as valid as:

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

However, there is one case where you might need a semicolon to separate two or more statements on the same line:

```
print("a");print("b");print("c")
```

```
>>>
a
b
c
```

Note that the following wouldn't compile:

```
print("a") print("b") print("c")
```

```
>>>
File "sample.py", line 1
  print("a") print("b") print("c")
              ^
SyntaxError: invalid syntax
```

2.2.3 Simple Program

Here is example of what a simple Python program may look like:

```
c = 3
while c:
    print(c)
    c -= 1
```

Running this code will produce following output in the console:

```
3
2
1
```

How many spaces should be used to indent Python code?

```
c = 3
while c:
    print(c)
    c -= 1
```

```
c = 3
while c:
  print(c)
  c -= 1
```

```
c = 3
while c:
 print(c)
 c -= 1
```

One difference between Python and many other languages is the absence of curly brackets when defining a block of code. In this case, the body of the `while` loop is *indented* by four spaces. You can use any number of characters or tabs as long as they are consistent with the previous line in the same block scope.

Note: Python disallows combining different number of spaces on each consecutive line for indentation of the same block. Doing that would make it impossible to know where each block starts and ends.

The `while c:` statement is evaluated to `True` as long as `c` remains a positive number. Then `c` is decremented by `1` by executing `c -= 1` statement. When `c` reaches `0` `while` loop will interpret it as `False` and terminate. The result is 3 decreasing numbers printed in a row.

Here is another Python program that picks random numbers from a range:

```
import random

numbers = range(1, 15)
chosen = []

while len(chosen) < 6:
    number = random.choice(numbers)
    chosen.append(number)

print(chosen)
```

What's going on here?

Using **import** keyword the program includes a built-in *module* called `random`. This package contains methods for working with random numbers.

For example `random.choice` method returns a random element from a set. In Python *ordered* sets of items can be also referred to as *sequences*.

A Python sequence is an *iterable*, like a JavaScript `Array`.

The built-in **range** function generates a linear set of numbers between 1 and 15.

This sequence is then stored in the variable **numbers**.

In Python you don't have to type `int`, `float`, `var`, `let` or `const` to define a variable. Variable names are not even preceded by something like $ as you would see in PHP language. They are just names.

The script then creates a variable called **chosen** to store an empty array `[]`.

Inside the `while` loop a random number is chosen and stored in **number** variable.

numbers.remove(number) line removes that number from **numbers** sequence.

Then it will append the same number to **chosen** array.

This is equivalent to *moving* it from **numbers** array into **chosen** array.

Without a condition `while` loop iterates infinitely. But because it checks length of the array **chosen** using another built-in Python method **len**, when the length of chosen array reaches 6 the loop will be terminated.

Lastly, using built-in **print** function, we output the array **chosen**. It will contain 6 numbers randomly picked out from the original range that was stored in `numbers`.

If you run it, the result of this program will be printed to the console:

```
[14, 44, 42, 16, 5, 6]
```

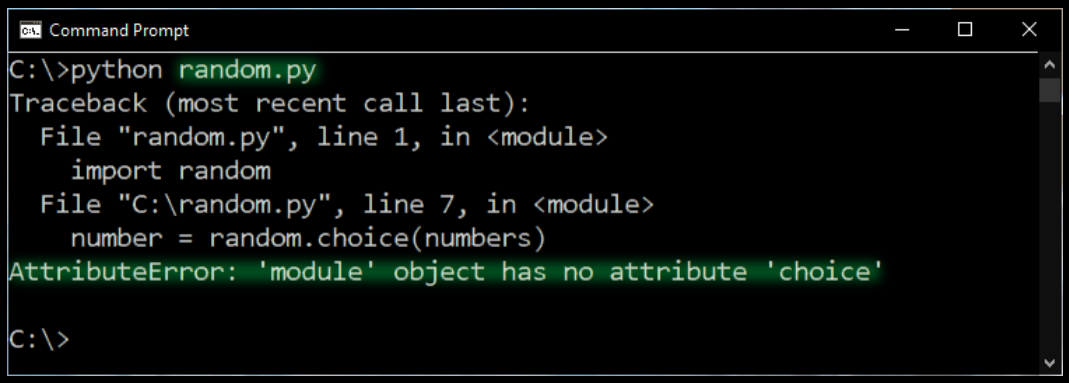
Or if you launch it from the Command Prompt:

```
C:>python hello.py
[7, 3, 41, 18, 20, 28]
```

Of course because numbers are chosen using a *Random Number Generator* if you run this script on your computer, the six numbers will be different every time.

2.2.4 A Common Pitfall

One mistake beginners can make here is to name the python script for this example **random.py**. Don't do it. You will generate an error:



Do not name your Python files the same name as built-in Python modules. Here, because we named our script **random.py** and inside imported the module called **random** (it is proper for modules to be specified without **.py**) we confused the Python interpreter.

Python is a High-level Language

In comparison to low-level programming languages designed to work close to the hardware-level, Python was designed to read more like a human language.

You can say that low-level languages like The Assembly Language created to make it easier to communicate with things like memory addresses. Their instruction sets resemble code that talks directly to the hardware.

Python is a high-level language. It's not as close to hardware as some others. But this makes Python easy to read and I hope you agree that it looks very clean.

Poetic Python

In comparison to languages that use excessive square and curly brackets Python may looks like a hand-written letter. Or perhaps even poetry.

In the book called **./code –poetry** by Daniel Holden and Chris Kerr the authors present 38 pages of poetry written in various computer languages.

Here is a Python poem from the book:

```
# This script should save lives.
import soul

for days in len(life)
    print "happiness"
```

Not many popular computer languages can claim the right to look and read as naturally as one would read poetry. Perhaps Python is one them.

There is something elegant about Python comments that begin with #. In contrast to many other languages that start comments with an ugly, repetitive sequence of two forward slashes `//` or something like `/* bonkers */`

If you're coming from C, C++, Java, PHP or JavaScript background and start writing Python code, you will notice that it *actually* feels lighter on the hands to write Python code due to absence of excessive brackets and optional semicolons.

2.3 Running Python Scripts

Python scripts are usually saved in files ending with **.py** extension. To execute the script type the following command into the Command Line:

```
python filename.py
```

If **filename.py** is located in the same directory from which you're calling this command the script will be executed and you will see its output directly in your **cmd.exe** on Windows, **Terminal** on macOS or your Linux **bash/cli**.

The **.py** extension is not a requirement. Your Python file could have been named anything. As long as the code is valid Python code, it would still be processed by the interpreter and executed. For example:

```
python test
```

This command would successfully execute Python code in **test** file.

If your Python script contains no errors the interpreter will convert your code into machine-readable code and execute it. After your program has executed all statements it will exit back to the command prompt.

There are exceptions. It is possible to write socket listeners in Python. Those types of programs will continue running and listening to incoming connections until execution is stopped manually or the server crashes.

2.3.1 Python comments

Every time you see # character in Python code it indicates a comment. Comments are useful for documenting your code:

```
# create variable a and assign 1 to it
a = 1
```

Comments can trail after the statement:

```
print(message) # Print my message
```

Comments are not Python code, they are free-style notes. As such they are ignored by the compiler. You can never generate a compiler error by a comment.

Comments are used to leave notes to self, reminding you what a block of code or a function below it does. They can be also notes left for other programmers or teammates who will be reading your code in the future.

Some tutorials and courses teach that using tripple quotes """ should be used as multi-line comments. That's not true. Python interprets """ as a stand-alone string value. And while the compiler will ignore them if they are not assigned to any variables they should never be used as comments:

```
"""This will actually
compile in Python, but is often mistaken
as a way to make multi-line comments.
Just. Don't."""
```

If you really need a multi-line comment, you can build a hash tag wall:

```
# Code below calculates number of raindrops
# required to fill a 20x10 pixel basin
# created by Voronoi Noise algorithm
```

Don't use """ to create multi-line comments. Use # instead. Python *allows* multi-line strings when using the tripple double quote """:

```
message = """Hello there from
a multi-line string!"""

print(message)
```

```
>>>
Hello there from
a multi-line string
```

You can also use triple single quotes to create strings spanning multiple lines:

```
message = '''Hello there from
a multi-line string!'''
```

Now let's intentionally try something that doesn't work:

```
msg = "Using single double quotes
to make multi-line strings
produces an error"
```

This code will produce `SyntaxError: EOL while scanning string literal`.

You can't use single double quotes " to span a string over multiple lines. Only triple double quotes """ or tripple single quotes '''.

2.3.2 An early exit

You can exit your Python script early and abandon executing any of the following commands after the `exit()` function:

```
print("This message will print.")
exit()
print("This message won't.")
```

Output:

```
This message will print.
```

Because we exited the script with built in `exit()` only the first statement was executed. You can use `exit()` anywhere in your program.

Chapter 3

Python Programming Language

3.1 Basic Operations

3.1.1 Learning To Speak In Python

When I was learning my first computer language initially I just wanted to learn how to do *basic operations*, like printing and formatting strings to console and adding numbers. Perhaps also making my own functions and calling them.

I wanted to learn how to **think in Python** at the level of its basic grammar before dealing with more complex ideas like closures, objects and static methods.

Primitive types, classes, objects, lists and other data types and data structures are also important and we will go over them one by one in detail after we cover basics.

As an introduction to the Python language the passage will first go over commonplace scenarios from the life of a Python programmer.

This part of the book will introduce you to the "flavor" of Python language. In other words – what it generally feels like to program in Python.

Like in many other languages, there are multiple ways of doing the same thing in Python. Examples throughout this book will give reader a preference as to which way they should use without bias.

All of the following examples will focus on everything that works in Python 3 and

above. Some examples of printing values may be more "outdated" than others, or rolled over as features from previous version.

Since so many Python programs have been written in prior versions of the language it was decided to include some of those examples as well. The book will make best effort to notify you about any deviations.

At the time when this book was written Python 3.9.0 was the latest version, awaiting 4.0 release. This means *most* of the examples will be based on the latest version of Python at this time.

Let's start with the very basics! Comments.

Comments

Comments start with # and placed at the beginning of a line or *after* a statement:

```
# Create a new variable
chickens = 10

egg = 1 # one egg
```

Printing Values

To print simple values use `print` function. In Python 3 or greater:

```
chickens = 10
print(chickens)        # prints 10 to command prompt
```

Printing Multiple Values

You can form a sentence by separating values by comma:

```
num = 5
what = "wolves"
print("there are",num,what)
```

This code will print the following string:

```
there are 5 wolves
```

Space characters were automatically inserted by `print` function. All you have to do is give the `print` function a comma-separated list of values. This is a great automatic feature for formatting natural text.

Using {} placeholders and string's `.format()` method

Objects of type string have a `.format()` method, making it possible to execute it directly on a *string literal*. Embedding empty curly braces {} into string creates a placeholder value specified in `.format()` method:

```
print("You have collected {} pine cones".format(15))
```

The number of {}'s must match the number of values passed into the `.format()` method as a comma separated list:

```
print("You have collected {} pine {}".format(15, "cones"))
```

Both of the statements above will produce the following output:

```
You have collected 15 pine cone
```

Formatting dynamic text values in proper English

Often when you have to write text output you may want to append the letter `"s"` to sentences containing numeric values. You might also swap the *linking verbs* `"are"` and `"is"` to make the text read in proper English.

What if after some operations your variable changed from 10 to 1 and you need to update your text value according to those changes?

For example: There "are" 10 apple(s). But there "is" 1 apple. Let's try this out:

```
apples = 10;
print("There are  apples.".format(apples))

>>>
There are 10 apples
```

Now let's change the value to 1 and run the same code:

```
apples = 1;
print("There are  apples.".format(apples))

>>>
There are 1 apples
```

Something is wrong. This doesn't read like proper English. It should say There *is* 1 apple. At first it might seem like we have to cover many different cases.

Luckily, these syntactic changes happen in all cases only if the value is or isn't equal 1. And it's the only test we need to write.

One way of solving this problem is to use Python's *inline* `if/else` statement:

```
cones = 15
letter = "s" if cones != 1 else ""
print("You have collected  pine cone".format(cones, letter));

cones = 1
letter = "s" if cones != 1 else ""
print("You have collected  pine cone".format(cones, letter))


>>>
You have collected 15 pine cones
You have collected 1 pine cone
```

Now let's go back to our apple example and swap linking verbs *is* and *are*:

```
apples = 1
letter = "s" if apples != 1 else ""
verb = "are" if apples != 1 else "is"
print("There   apple".format(verb, apples, letter));

apples = 200
```

```
letter = "s" if apples != 1 else ""
verb = "are" if apples != 1 else "is"
print("There   apple".format(verb, apples, letter))
```

```
>>>
There is 1 apple
There are 200 apples
```

Now it reads like proper English.

The inline `if/else` statement evaluates based on provided condition. In this case we were testing the numeric variable for not being equal to 1. And the linking verb for being either `are` or `is`.

The `.format()` method automatically interprets value by type

By default the `.format()` method will automatically interpret either numeric or string variables with respect to the type of variable's value.

Whole floating point numbers will be reduced to one trailing zero:

```
print("num = ".format(15.0000)) # prints 15.0
```

A proper number of trailing zeros will be displayed for non-0 decimal point:

```
print("num = ".format(15.0025)) # prints 15.0025
```

You can force the value to be interpreted by any type you want. See next section.

Floating point and `.format()` method

Using {:.f} will force the value to be interpreted as floating point number regardless of whether it actually is (in this case it's a whole number):

```
print("Your balance is $:f".format(125))
```

```
>>>
Your balance is $125.000000
```

By default the floating point number will display 6 trailing zeros. Perhaps this is not how you want to display a monetary value. But there is a solution.

Floating point with decimal point

Sometimes you want to choose number of digits to the right of . in a floating point number. For example, when displaying financial information like balance.

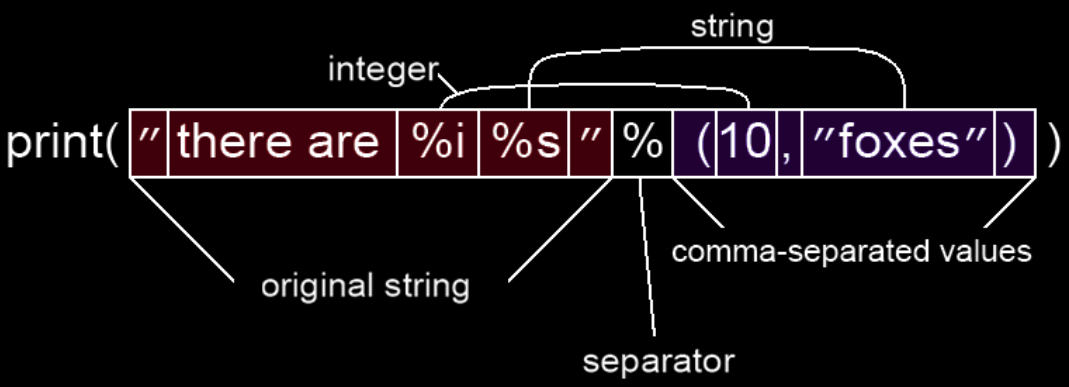
You can do that with {:.2f}

```
num = 125
print("Your balance is $:.2f".format(num))
```

```
>>>
Your balance is $125.00
```

Using % value interpretation character

You can also print one or multiple variables with `print` function by using % character embedded directly into the target string:



In this example `%i` placeholder will interpret the value as an *integer* (`int`)

The `%s` placeholder will interpret the value as a *string* (`str`)

Actual values are provided after `%` separator usually in parentheses.

```
num = 12
what = "foxes"
print("there are %i %s" % (num, what))
```

This code will print the following string to the command prompt:

```
there are 12 foxes
```

In Python 2.9 or less there are no parentheses:

```
chickens = 15
print chickens          # prints 15 to command prompt
```

(This is illegal in 3.0 and up, but still nice to know for legacy code.)

3.1.2 Conditions

Python provides a set of operators for comparing values based on logical conditions.

```
print(15 == 15)
print(15 != 10)
print(10 < 23)
print(10 == 10)
print(10 <= 10)
print(3 > 8)
print(16 >= 14)
```

```
>>>
True
True
True
True
True
False
True
```

Often you will want to use these conditions together with an `if...else` statement.

3.1.3 if and if...else statements

```
a = 100
b = 200

if b > a:
    print("b is greater than a")
else:
    print("b is lesser or equal to a")
```

```
>>>
b is greater than a
```

3.2 Operators

There are 7 different types of operators in Python: arithmetic, assignment, comparison, logical, identity, membership and bitwise.

In this section we will go over several diagrams that expose each operator from each category with a simple example. After that some of them will be demonstrated using source code examples.

3.2.1 Arithmetic Operators

Operator	Name	Example
+	Addition	`10 + 5`
−	Subtraction	`10 - 5`
*	Multiplication	`10 * 5`
/	Division	`10 / 5`
%	Modulus	`10 % 5`
**	Exponentiation	`10 ** 5`
//	Floor Division	`10 // 5`

As you can see Python offers a standard set of operators. But it also provides several useful operators of its own, not found in many other languages.

Let's take a look at some of the most common ones and how they work.

Assignment operator

Used to assign values to variables:

```
a = 10
```

It can be also used to assign replacement values to slice operation:

```
three = [0,0,0]

# slice out 2 first items, replace with [1,1]
three[:2:] = [1,1]

print(three)
```

```
>>>
[1, 1, 0]
```

We used `[:2:]` which is a slice operation with first and last optional arguments skipped. The second argument 2 tells us where to stop.

Don't worry if this doesn't make sense yet. Slicing is covered in more depth later in this book. For now just equality operator has multiple function.

Math operators

Common math operations do exactly what you expect:

```
print(1 + 1)          # 2
print(5 - 2)          # 3
print(3 * 3)          # 9
print(8 / 2)          # 4.0 (note conversion to floating point)
```

+, -, * and / are called operators. But they are not limited to mathematical operations. For example * can be used to multiply sequences and + to add strings (In Python strings are also treated as sequences.)

Later we will take a look at *overloading operators* – you will decide what happens to objects when an operator is applied to two object instances of same type.

** exponentiation operator

Use exponentiation operator to raise a number into a power of another value:

```
print(2 ** 2)
print(2 ** 3)
```

```
>>>
4
8
```

// floor division operator

Return integral part of the quotient:

```
print(6 // 2)
print(10 // 3)
print(12 // 3)
```

```
>>>
3
3
4
```

The result is how many times a whole number fits into another number. It works similar to applying `floor` function to a division.

% modulus operator

Return decimal part of quotient.

```
print(6 % 2.5)
print(10 % 3.5)
print(12 % 3.5)
```

```
>>>
1.0
3.0
1.5
```

Opposite to floor division, modulus returns the remainder component of a division. This is how much of the whole number on the right hand side of the operation *did not fit* into the number on the left hand side.

3.2.2 Assignment Operators

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
//=	a //= b	a = a // b
**=	a **= b	a = a ** b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b
>>=	a >>= b	a = a >> b
<<=	a <<= b	a = a << b

Most of the assignment operators are short hands to their original form.

+= increment assignment

Like many other operators, the increment assignment is not limited to integers. It can be used on sequences (string, tuple and list) but not on dictionaries or sets:

```
# integer
a = 1
a += 1
print(a)

>>>
2

# string
s = "ha"
s += "ha"
print(s)

>>>
haha

# tuple
t = (1,0)
t += (1,2)
print(t)

>>>
(1, 0, 1, 2)

# list
u = [1,0]
u += [1,2]
print(u)
```

```
>>>
[1, 0, 1, 2]
```

Trying to increment sets will produce TypeError:

```
# set
v = {1,0}
v += {1}
print(t)

>>>
Traceback (most recent call last):
  File "example.py", line 32, in <module>
    t += {1,2}
TypeError: unsupported operand type(s) for +=: 'set' and 'set'
```

A set is a self-contained collection of items: for this reason it doesn't share same qualities with the mutable type such as `list`.

Unlike `list` or `tuple`, `set` doesn't allow duplicates. Items are not guaranteed to appear in the same order they were defined.

Like `tuple`, the `set` data structure is immutable. This means once items are added to the set they cannot be changed.

Trying to apply access operator `[]` on a `set` will produce TypeError:

```
a = {1, 2}
a[0] = 'a'

print(a)

>>>
Traceback (most recent call last):
  File "example.py", line 3, in <module>
    a[0] = 'a'
TypeError: 'set' object does not support item assignment
```

Sets are not subscriptable which means they cannot be accessed with `[]` operator.

Items in a set are immutable – you cannot *change* them once they are added to the set. However, the elements of a set are mutable – which means you can still add or remove items (note that this still implies that no original value can be modified.)

For example `set.add(item)` method adds one item to the set:

```
a = {1, 2}
a.add(3)

print(a)
```

```
>>>
{1, 2, 3}
```

Since sets don't have an index, you can remove items by value:

```
a = {1, 2, 3, 4, 5}
a.remove(4)

print(a)
```

```
>>>
{1, 2, 3, 5}
```

Dictionaries don't support += operator. However if you have latest version of to Python you can use the | operator.

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

print(d|c)
```

```
>>>
{'a': 1, 'b': 2}
```

This won't work in Python 3.8 or earlier.

You can also add up two dictionaries using double star operator:

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

print({**d, **c})
```

```
>>>
{'a': 1, 'b': 2}
```

This works similar to JavaScript's `...spread` operator.

-= decrement assignment

This operator will decrease the number by another number.

```
num = 10
num -= 4

print(num)
```

```
>>>
6
```

Exactly the same result could have been achieved as follows:

```
num = 10
num = num - 4

print(num)
```

```
>>>
6
```

Other operators that follow *operator*= pattern work in exactly the same way.

***= multiplication assignment**

```
num = 10
num *= 20

print(num)
```

```
>>>
200
```

/= division assignment

```
num = 100
num /= 2

print(num)
```

```
>>>
50.0
```

****= power assignment**

```
num = 3
num **= 2

print(num)
```

```
>>>
9
```

Power assignment does the same thing as raising base of a number into a power exponent. Here raising 3 into power of 2 equals 9.

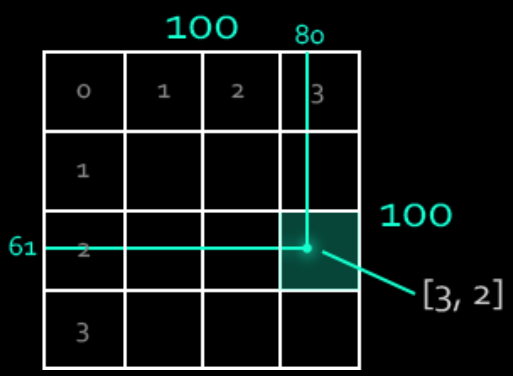
//= floor division assignment

Floor division assignment can tell you how many times a number fits into another number:

```
num = 10
num //= 3

print(num)

>>>
3
```

You can fit number 3 into 10 at most only 3 times. How can this be useful? This really depends on the type of a problem you're trying to solve.

For example this operation can be used to determine position of a point on a larger grid. In the following example we'll use the //= operator to determine location of a point on a 100 x 100 grid that is divided into 4 x 4 regions.




```
# grid dimensions
grid_x = 100
grid_y = 100

# location of a point
point_x = 80
```

```
point_y = 61

grid_size = 4

# scale grid relative to its size
rel_x = grid_x / grid_size
rel_y = grid_y / grid_size

# where is the point on a 4 x 4 grid?
point_x //= rel_x
point_y //= rel_y

# output result
print([point_x, point_y])
```

```
>>>
[3.0, 2.0]
```

Without floor division assignment you would have to call the `floor` function. But Python provides an operator to do the same thing.

String Multiplication

In Python when a sequence is multiplied by a number, it will be copied and appended to itself. Because strings are sequences you can do something as follows:

```
laugh = "ha" * 3
print(laugh)        # prints "hahaha"
```

```
>>>
hahaha
```

This script prints `"hahaha"` to command prompt.

Here's another example with an f-string and *arbitrary expressions*:

```
print(f"Fridays are s{'o'*8} sl{'o'*8}w.")
```

```
>>>
Fridays are soooooooo sloooooow.
```

Here {'o'*8} is an arbitrary expression that evaluates to 'oooooooo'. Python's f-strings work similar to how Template Strings work in JavaScript. You can call functions within them or evaluate values based on ternary operators.

Because f-strings are evaluated at runtime you can insert any valid Python expression enclosed in {} brackets into the f-string.

== Equality operator

Equality operator checks if two values or conditions are equal.

You can use it on numbers and strings:

```
if 10 == 10: print("10 equals 10")
```

```
>>>
10 equals 10
```

Python statements will first evaluate to a value and then be compared to the value on the other side of the equality operator:

```
if 10 == 5 + 5:
    print(True/colorblack)
```

```
>>>
True
```

You can also use the equality operator on other types, sets for example:

```
a = {1, 2, 3}
b = {1, 2, 3}
c = {5, 6, 7}

if a == b:
    print("sets are equal")
else:
    print("sets are not equal")

if a == c:
    print("sets are equal")
else:
    print("sets are not equal")

>>>
sets are equal
sets are not equal
```

Applying equality operator to values will often evaluate to a boolean.

```
print(1 == '1')  # False
print(1 == 1.0)  # True
print(1 == True) # True
```

Note in Python `True` and `False` values are defined with uppercase letter:

```
print(True) # True
print(true) # NameError: name 'true' is not defined
```

(Watch out if you come to Python from the world of JavaScript.)

It's possible to use == to compare functions:

```
def hello():
    print("hello there.")
```

```
print(hello == hello);    # True
```

```
>>>
True
```

Absence of ++ operator

C++ programmers will be thrown off by the absence of ++ operator:

```
chickens = 10
chickens++          # SyntaxError: invalid syntax
```

Incrementing a number should be done using the += operator:

```
num = 10
num += 1
print(num)          # 11
```

```
>>>
11
```

Or alternatively:

```
num = 5
num = num + 2
print(num)          # 7
```

```
>>>
7
```

Absence of === operator

If you're learning Python as a JavaScript developer you might find it surprising that Python does not have the *triple equality operator*. In JavaScript === is used to compare by value and type. Simply use == instead:

```
a = 1; b = 1
print(a == b)    # True
print(1 == "1")  # False
```

```
>>>
True
False
```

To imitate the === operator you can do something as follows:

```
a = 1; b = 1
print(a == b and type(a) == type(b))
```

```
>>>
True
```

Also note that the following statement:

```
1 == True
```

```
>>>
True
```

Evaluates to `True` because in Python boolean `True` is a subclass of the `int` (*integer*) type. The equality operator casts `True` to 1:

```
print(int(True))   # 1
```

```
>>>
1
```

3.2.3 Comparison Operators

Operator	Name	Example
==	Equality	a == b
!=	Not equal	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal	a >= b
<=	Less than or equal	a <= b

Comparison operators are straightforward. They help us determine relationship between string, boolean and numeric values depending on situation.

3.2.4 Logical Operators

Operator	Description	Example
and	Returns true if both statements are true	a == b and c < d
or	Returns True if one of statements is true	a == b or c == d
not	Inverse boolean result, if True, then False	not True

Unlike some other languages in Python logical operators are actual English words. This makes reading and writing statements more natural:

```
result = not(10 == 10)

>>>
False
```

Even though expression 10 == 10 evaluates to True, wrapping it in a not inverts the condition. This is helpful if you need to check if something is *not* true:

```
playing = True

if not playing:
    print("not playing")
else:
    print("playing")

>>>
playing
```

Use and if you require two (or more) conditions to be true and or if you need one of the two (or more) conditions to be true:

```
if 10 == 10 and 'a' == 'a':
    print("both statements evaluate to True")

if (5 == 3 or 1 == 1 or 'a' == 'b')
    print("at least one statement evaluates to True")

>>>
both statements evaluate to True
at least one statement evaluates to True
```

3.2.5 Identity Operators

Operator	Description	Example
is	Returns true if both variables are the same object	**a is b**
is not	Returns True if variables are not the same object	**a is not b**

```
a = 100
b = a
c = 2

if a is a:
    print("a is the same object as a")

if a is b:
    print("a is the same object as b")

if c is not a and c is not b:
    print("c is neither nor b")

>>>
a is the same object as a
a is the same object as b
c is neither nor b
```

3.2.6 Membership Operators

Memberships operators are helpful when you need to check if a value exists in a sequence object such as list, tuple, dictionary.

Operator	Description	Example
in	Returns True if value exists in a sequence	a in b
not in	Returns True if value does not exist in a sequence	a not in b

```
cars = ["tesla", "mercedes", "bmw"]

print("tesla" in cars)
print("honda" in cars)
print("bmw" not in cars)

>>>
True
False
False
```

You can also check lists, tuples and dictionaries:

```
list = [0, 1, 2, 3, 4]
print(1 in list)
print(125 in list)

tup = ('a', 'b', 'c')
print('a' in tup)
print('hello' in tup)

dictionary = "key": "door", "kitchen": "sink"
print("key" in dictionary)
print("kitchen" in dictionary)
print("door" in dictionary)
print("sink" in dictionary)
```

```
>>>
True
False
True
False
True
True
False
False
```

Note using `in` and `not in` operators on a dictionary tests for presence of a key with that name, not the value.

3.2.7 Bitwise Operators

On a 64-bit processor, each variable is 64 bits long, even if the value of that variable is set to 1. Bitwise operators help you work on the actual bits of the variable, rather than its primitive type value.

Operator	Name	Description
&	AND	Sets target bit to 1 if both bits are 1
|	OR	Sets target bit to 1 if one of two bits is 1
^	XOR	Sets target bit to 1 if only one of two bits are 1
~	NOT	Inverts all bits
<<	Zero fill left shift	Shift all bits left, discard left-most bit
>>	Signed right shift	Shift all bits right, discard right-most bit

Bitwise & operator

The & operator will produce a binary number in which bits remain 1 only if bits in both variables positioned in the same slot match.

For example `101 & 001` will result in `001` because only last bit to the right match. But `101 & 111` will produce `101` because only first and last bits match.

```
a = 1          # 00000001 in binary or 1 in decimal
b = 2          # 00000010 in binary or 2 in decimal
c = 4          # 00000100 in binary or 4 in decimal
d = 5          # 00000101 in binary or 5 in decimal

print(a & b)  # 00000000 or 0 in decimal
print(a & c)  # 00000000 or 0 in decimal
print(a & d)  # 00000001 or 1 in decimal
print(c & d)  # 00000100 or 4 in decimal
```

```
>>>
0
0
1
4
```

Bitwise | (OR) operator

You can "merge" individual bits using | operator.

```
a = 1          # 00000001
b = 2          # 00000010
c = 4          # 00000100

print(a | b)  # 00000011 or 3 in decimal
print(a | c)  # 00000101 or 5 in decimal
```

```
>>>
```

```
3
5
```

Bitwise XOR operator

If one bit is 1 and another 0, keep 1 in that slot.

```
a = 127       # 01111111
b = 128       # 10000000
c = 157       # 10111011

print(a ^ b)  # 11111111 or 255 in decimal
print(a ^ c)  # 11100010 or 226 in decimal
print(b ^ c)  # 00011101 or 29 in decimal

>>>
255
226
29
```

Bitwise NOT operator

Invert all bits.

```
a = 1         # 00000001
b = 4         # 00000100
c = 8         # 00001000

print(~a)     # -00000010 or -2 in decimal
print(~b)     # -00000101 or -5 in decimal
print(~c)     # -00001001 or -9 in decimal

print("0:b".format(~a)) # -10
print("0:b".format(~b)) # -101
print("0:b".format(~c)) # -1001
```

```
>>>
-2
-5
-9
-10
-101
-1001
```

Incidentally, due to how binary format stores negative numbers, by inverting all bits the number was incremented by 1 and negated.

Bitwise shift left operator

```
var = 1

print(var << 1)
print(var << 2)
print(var << 3)
print(var << 4)
print(var << 5)
print(var << 6)
```

```
>>>
2
4
8
16
32
64
128
```

In binary numeric system value 1 can be represented by 00000001.

Every time we shift the bites left, we produce the following pattern:

```
000000001    # 1
000000010    # 1 << 1 = 2 in decimal
000000100    # 1 << 2 = 4 in decimal
000001000    # 1 << 3 = 8 in decimal
000010000    # 1 << 4 = 16 in decimal
000100000    # 1 << 5 = 32 in decimal
001000000    # 1 << 6 = 64 in decimal
010000000    # 1 << 7 = 128 in decimal
```

This is equivalent of raising a number in second power. Because shift operation takes less computer cycles to complete, it used to be a favorite optimization to avoid multiplication.

Bitwise shift right operator

It works in the same way as shift left except in the opposite direction.

Chapter 4

Loops

Loops are useful when you need to iterate through a list of values. One of the most basics forms of a loop is the for loop. It's present in many other languages. Let's take a look at few examples of for loops first.

4.1 for loops

One of the most basic loops in Python is the `for-in` loop.

4.1.1 for-in loop syntax

The syntax of a for-in loop is as follows.

```
for x in iterable:
```

To create a loop that does nothing you can use `pass` keyword:

```
for x in iterable: pass
```

This loop has number of iterations equal to the number of items in *iterable* value. But no statements will be executed because `pass` simply hands execution over to next iteration step.

4.1.2 Iterating lists

Let's try something more meaningful. First we will create a list of alphabet letters which is an *iterable*:

```
alphabet = ["a", "b", "c"]
```

Now we can loop through each letter as follows:

```
for letter in alphabet:
    print(letter)

>>>
a
b
c
```

What is iteration? Iteration is repetition of a statement in a computer program which successively adds up to solving a larger problem. For example, to calculate total price of all items in a shopping cart you can "loop" or *iterate* through the list of all objects representing items currently stored in the shopping cart. The loop would add up or "reduce" their total values into a single value – the sum of all items:

```
prices = [19.99, 34.99, 44.99]
total = 0

for price in prices:
    total += price

print("total =", total)

>>>
total = 99.97
```

In mathematics this specific type of operation is so common it actually has a name – *reduction*. Reduction can be alternatively performed by function `reduce()` from `functools` package as will be shown below.

But in order to make it work we need to create our own `add()` function because its required as the first argument to `reduce()` function. Second argument is our iterable list:

`functools.reduce(`*`function, iterable`*`)`

Higher-order function. A function that takes another function as an argument is called a higher-order function. It abstracts the inner-workings of the operation to some function (`add()` in next example.)

Higher-order functions are not unique to Python and exist in many other languages.

```
import functools # gives us functools.reduce function

prices = [19.99, 34.99, 44.99]
total = 0

# define our own add() function
def add(a, b): return a + b

# "reduce" list to sum total of all items
total = functools.reduce(add, prices)

print("total =", total)

>>>
total = 99.97
```

The same operation and the same result as previous example where for-loop was used. Both examples use an *iterable* object: `prices`.

What is an iterable? An object is called *iterable* if it is possible to pass it to a for-loop for walking through each one of its items. Some of the built-in types in Python like `list`, `tuple` and `string` are *iterable*.

There is a distinction between an **iterable** and **iterator**. The `list` data type is an *iterable* but in itself it is *not* an iterator. It can be passed to the built-in `iter()` function which returns an *iterator* object. That object has *iterator*`.next()` method for stepping to next item.

What is enumerable? An enumerable is an iterable object in which all items are guaranteed to follow linear indexing.

In Python, like in many other languages, strings are iterable. String's items refer to its characters.

```
text = "hello"
```

In this variable named `text` and whose value is `"hello"`, the first letter `"h"` is accessed by `text[0]`, second letter is `"e"` can be accessed via `text[1]` and letter "o" is guaranteed to exist at `text[4]`.

```
for letter in "hello": print(letter)
```

```
>>>
h
e
l
l
o
```

4.1.3 Using iter() to iterate a list

Technically what takes place inside a for-loop can be thought of as iterating – we "walk" each item in the iterable data type until item list is exhausted.

In some instances you will want to iterate a list one item at a time. To do that Python provides a special function `iter()` that creates an iterator object.

To create an iterator pass a list or other iterable object to `iter()` function. Then every time you use `next()` function you'll progress to next item one at a time.

Next example uses an iterator to walk through a list of 4 items (out of 5 total.)

```
colors = ["red", "yellow", "green", "blue", "purple"]

# create iterator
```

```
stepper = iter(colors)

# print out the object
print(stepper)

>>>
<list_iterator object at 0x00E36178>

# take 4 steps
print(next(stepper))
print(next(stepper))
print(next(stepper))
print(next(stepper))

>>>
red
yellow
green
blue
```

The `next()` function steps into the data set one item at a time.

4.1.4 Iterating in reverse

You can use the built-in function `reversed()` to loop through letters backwards:

```
alphabet = ["a", "b", "c"]

for letter in reversed(alphabet):
    print(letter)

>>>
c
b
a
```

4.1.5 Nested iterables

Python makes it easy to access nested iterables. All you have to do is provide a list of variables separated by comma and they will map to the values in each iterable:

```
    for a, b in [[iterable], [iterable]]:
```

Example:

```
its = [[7,5],[3,1]]
  for a, b in its:
    print(a, b)

>>>
7 5
3 1
```

Number of values in the argument list `a, b` must match the number of values in each iterable. Here is an example with a different number of iterables:

```
its = [[7,5,0],[3,1,4]]
  for a, b, c in its:
    print(a, b, c)

>>>
7 5 0
3 1 4
```

4.1.6 Looping through numbers

One of the most common loop examples is iterating over a list of numbers. In Python a list of numbers can be created using the built-in `range()` function:

```
for value in range(0, 5):
    print(value)
```

```
>>>
1
2
3
```

You can also iterate through a range in reverse:

```
for value in reversed(range(0, 5)):
  print(value)
```

```
>>>
4
3
2
1
```

4.1.7 Iterating index and values at the same time

If you're coming to Python from languages like C++, PHP and JavaScript you might be used to working with index++ value in for loops.

In JavaScript and many other languages for-loops have syntax support for index iteration variable. Here it's specified in last statement of the for-loop: `i++`:

```
// An example of for-loop in JavaScript:
let nums = [9,5,5];

for (let i = 0; i < 10; i++)
  console.log(`Value at index $i is $nums[i]`);
```

```
>>>
Value at index 0 is 9
Value at index 1 is 5
Value at index 2 is 5
```

In Python basic `for-in` loop does not provide an index value. One way of imitating an indexed for-loop in Python is to create an extra variable. Alternatively you can also use `enumerate()` function.

See next examples.

One way of adding indexing to a Python for loop is to provide your own independent index variable:

```
index = 0
for value in [9,5,5]
  print("Value at index  is ".format(index, value))
  index += 1

>>>
Value at index 0 is 9
Value at index 1 is 5
Value at index 2 is 5
```

Alternatively you can use `enumerate()`:

```
colors = ["red",
       "yellow",
        "green",
         "blue",
       "purple"]

# print out enumerable object
print(enumerate(colors))

for (index, value) in enumerate(colors):
    print(f"index={index}, value = {value}")

>>>
<enumerate object at 0x017E88C8>
```

```
index=0, value = red
index=1, value = yellow
index=2, value = green
index=3, value = blue
index=4, value = purple
```

The `enumerate()` function produces an enumerable object.

4.2 while loops

4.2.1 while loop syntax

The while loop executes a statement until a logical condition is met:

```
while condition:  statement
```

Walk through a number of items:

```
num = 0
while num < 5:
    num += 1 < 5
    print(num)
```

```
>>>
1
2
3
4
5
```

Here's another simple example where a list is popped until all items are ejected:

```
fruit = ["guava", "apricot", "apple", "orange", "banana"]

while fruit and fruit.pop():
```

```
    print(fruit)

>>>
['guava', 'apricot', 'apple', 'orange']
['guava', 'apricot', 'apple']
['guava', 'apricot']
['guava']
[]
```

Note `"banana"` is missing. By the time we reach the statement, the last item is already removed ("popped") from the list because it's part of the `while` statement.

We also check if fruit is non-empty. The `list.pop` and other list methods are explained in *Chapter 6: Data Types and Structures*.

4.2.2 while loops and not operator

In this example `while` loop was used together with `not` operator to continue looping as long as the value is non-numeric:

```
# set kilometers to non-numeric value
# so while loop executes at least once
kilometers = ""

while not kilometers.isnumeric():
    kilometers = input("Enter number of kilometers:")

print(f"Value entered = {kilometers}")

>>>
Enter number of kilometers:sdfsdfs
Enter number of kilometers:nothingasd
Enter number of kilometers:dfhfghjfghfd
Enter number of kilometers:40
Value entered = 40
```

This while loop will continue to take user input until a numeric value is entered.

Chapter 5

Variables

5.1 Naming variables, variable types, memory and assigning objects by reference

Quantum theory suggests particles can exist in two different places at the same time. When it comes to variables – they are stored at a unique location in memory.

Every variable has 4 components: **name**, **value**, **type** and **address**.

In many languages including Python a *reference* is a variable that points to – or *refers* to a value – by location in RAM memory where it is physically stored.

Because variables are an essential part of any computer language we already practiced defining and using them in prior sections of this book.

This chapter goes in depth on variables by exploring how variables and values relate to each other and computer memory.

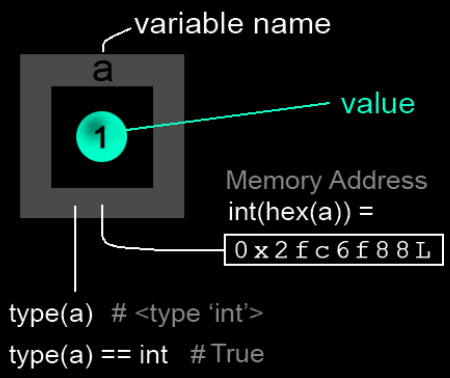
Python's compiler has a memory optimization trick. Two unique variables can share the same memory slot if their value is the same.

Shared memory for variables containing the same value is one of the following sections in this book that will explain how it works.

But first let's break down a variable into its basic components.

5.1.1 What is a variable?

Variables are *named placeholders* for values that reside at a memory address.



You can think of variables as values stored in placeholders. Here we have a variable named 'a' holding the value of the digit 1. In Python it's a primitive of type `int` (for *integer*.)

Variables are stored at a memory address which is assigned automatically in RAM at creation time also known as *run time*.

The variable name is a placeholder pointing to the address in RAM memory where the value is physically stored. When you create a new variable, essentially what you are doing is naming a memory address:

```
a = 1;
print(a)                # 1 (value of the variable)
print(type(a))          # <type 'int'>
print(id(a))            # 1473574832 (address as integer)
print(hex(id(a)))       # 0x57d4f7b0 (address as hexadecimal)
print(type(a) == int)   # True (this is a variable of type integer)
```

```
>>>
1
<type 'int'>
1473574832
0x57d4f7b0
True
```

On a 32-bit computer, this value of 1 is actually stored in a placeholder consisting of 32 bits that can be either 1 or 0. Only the last bit is switched on:

Why is this important? You need to understand this if you ever plan on writing or reading files in binary mode. Thinking in binary format can also help you optimize your data. Variable a only stores 1 value. If you have a set of flags, why create variable b? You can simply flip any of the 32 bits in a single variable to store 32 unique 0/1 flags.

For now just know that this is how computers represent values in memory. On 32 bit computers even small numbers like 1 are packed into 32 bit placeholders.

On a 64-bit computer there are 64 placeholders, etc. Understanding memory might be useful if you plan on optimizing your data. But it might not be necessary when writing simple Python scripts or just learning the language for the first time.

5.1.2 How to find out variable's memory address

When you create a variable a value representing that variable is physically created in memory. It is also placed there at an address.

Choosing exactly where in memory your value is stored is not something you have to worry about – the address is chosen automatically. Often, knowing where a variable is stored in memory can be useful. Python has built-in `id` and `hex` functions to help work with memory addresses.

5.1.3 Built-in `id()` function

```
a = 1
addr = id(a)
print(addr) # Prints 50098056 (or similar)
```

The built-in `id` function returns object identity – it's the variable's memory address in decimal format. If you run this on your computer, address will be different.

5.1.4 Built-in `hex()` function

Addresses are often displayed in `hex` format. To convert the object's decimal identity to actual memory address you can use the `hex()` function which converts any integer (not just an address) into *hexadecimal* format:

```
a = 1
addr = id(a)
hexaddr = hex(addr)
print(addr)    # Print decimal object identity
print(hexaddr) # Print memory address in hex format
```

```
>>>
50098056
0x2fc6f88L
```

Every time you run this program, it *may* (or may not) output a different address. This is normal behavior. The reason for it is because variables are stored in RAM (Random Access Memory.) This doesn't mean the address is chosen completely at random, however. It's just called that way.

In this context *random* simply means address won't follow an easily predictable pattern. Internally, it's decided by your computer processor in a way that's convenient at that time given various factors that depend on your processor's architecture.

When determining address the goal of your computer processor is to choose next available address slot as fast as possible – this gives addresses random-like nature.

There is certain logic to how your processor will choose a storage address based on various factors. It can be influenced relative to where in memory your script was running from at the time of execution.

One more thing about how Python manages memory

You should be aware of a quirk unique to Python language when it comes to memory addresses – it's explained in the next subsection.

5.1.5 Shared memory for variables containing the same value

Let's take a careful look at this example.

```
a = 1
print("a =", hex(id(a))) # print address of a

b = 2
print("b =", hex(id(b))) # print address of b

c = 1
print("c =", hex(id(c))) # same address as a

c += 1
print("c =", hex(id(c))) # changed to same as address of b
```

Running this example you will notice something peculiar in the output:

```
>>>
a = 0x3166f88L
b = 0x3166f70L
c = 0x3166f88L  # same address as a
c = 0x3166f70L  # after += 1 it's same address as b
```

How can two variables be stored at the same address? This happens only when two unique variables start to share the same value.

The address of `a` and `c` – two different variables – is exactly the same. This optimization occurs when different variables contain the same value (`1`).

Then, when we incremented `c` by `1` in the last step, the address of `c` changed and it now matches address of `b` because now both `c` and `b` share the same value (`2`).

This is not limited to same code blocks or current execution context. Address can be shared across global scope and function scope:

```
u = 1
print(id(u))
```

```
def func():
u = 1
print(id(u))

func()


>>>
41709448
41709448
```

Even in function scope variable address is the same if value is shared.

Python tries to re-use objects in memory that have the same value, which also makes comparing objects very fast.

As you can see this behavior is different from C-language, for example, where all 3 variables would have stored at 3 unique addresses in memory.

It's not uncommon for a JavaScript developer to learn Python. Moreover, by learning one language you learn another. Some of the following examples with contain brief JavaScript examples mirroring Python explanations that cover the same functionality.

5.1.6 Assignment by reference vs. assignment by value

When learning *any* computer language one of the first things you want to figure out is how values are assigned – by reference or by value?

You may already be familiar with **assignment by value** in JavaScript:

```
let a = 1;
let x = a;      // a copy of a was made and stored in x
a = a + 1;      // increment a by 1; now a is 2
console.log(a); // 2
console.log(x); // 1 (x is still 1)
```

When we assigned a to x on line 2 a *copy* of value stored in variable a was made.

Now x is a completely unique variable holding its own copy of number 1. When we increment a, the copy in x stays untouched – they are two separate variables.

Assignment by value in Python is exactly the same:

```
a = 1
x = a     # make a copy of a and store it in x
a = a + 1 # increment a
print(a)  # 2
print(x)  # 1 (x is still 1 because it's an original copy)
```

When a was assigned to x, a was actually *copied* to x. This is called assignment by value – both values will occupy a similar but unique place in memory:

Of course, this means we are using twice as much space in memory.

Now two variables a and x hold a unique value occupied by two different places in computer memory. That's what you would normally expect to happen.

Assignment By Reference in JavaScript

In JavaScript variables pointing to objects are assigned by reference. It's the same in Python. Below *Object Literal* and Python's *dictionary* examples are shown.

To demonstrate it let's take a look at variable named object pointing to an object containing x property, which is also a variable pointing to value of 1:

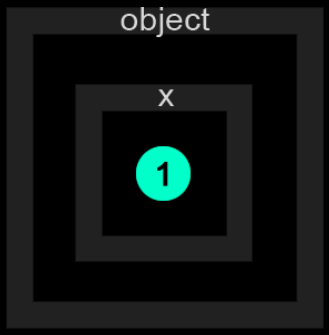


Figure 5.1: An object `object` encapsulates property `{x:1}`

In JavaScript variable type determines how equality operator = will assign it. For example, primitive values are assigned by value but objects by reference:

```
let object = x: 1;
let reference = object; // Assignment by reference
console.log(object.x, reference.x);

>>>
1, 1

reference.x++;
console.log(object.x, reference.x);

>>>
2, 2
```

Only `reference.x` was incremented. But `object.x` now also returns 2.

Unlike assignment by value from previous example nothing was actually copied to `reference`. The variable `reference` simply *points* to same place in memory as `object` because it was assigned by reference.

Because variable `reference` is a *reference* to `object`, `reference.x` points to the `x` value in `object.x`. This is why they both evaluate to 2.

Let's add something just to the end of the previous code listing:

```
object.x++;
```

And now let's print out both values again:

```
console.log(`object.x = ${object.x}`);
console.log(`reference.x = ${reference.x}`);
```

```
>>>
object.x = 3
reference.x = 3
```

This shows that changing value by using either variable changes it in both.

Assign By Reference In Python

```
# Create dictionary
original = {"x":1}
print(original)

# Make copy a reference to original
copy = original
print(copy)

# Change x in copy
copy["x"] = 2

# original["x"] now also contains 2
print(original)

>>>
{'x': 1}
{'x': 1}
{'x': 2}
```

Here is what actually happens:

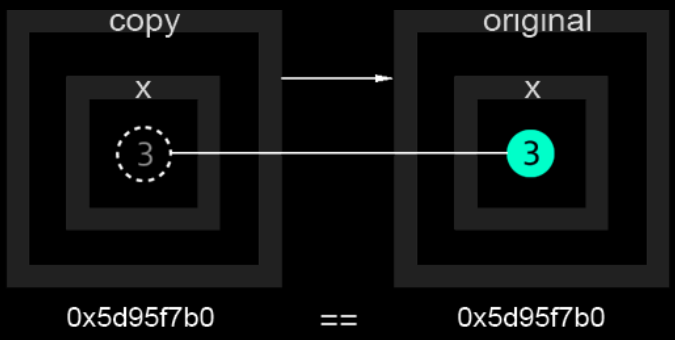


Figure 5.2: `thing` and `object` point to the same address in memory.

Both values *appear* to be incremented even though we only incremented one of them. They point to the same object in memory.

Assignment by reference is based on the same principle across many different languages. However, not all computer languages provide exactly the same set of features to deal with it.

Python doesn't allow assignment of primitive values (`string`, `number`, etc) by reference. This only happens when the assigned variable is object-like (dictionary, list, sequence.)

In some languages like C++ it is possible to assign variables as a reference using a leading ampersand (`&`) character:

```
int original = 9;     // create variable
int &ref = original;  // assign by reference
ref++;                // increment reference by 1
cout << original;     // prints 10
```

In this example `original` also returns `10` after `ref` is incremented. The C++ language relies heavily on references and memory pointers which often leads to crashing your computer. JavaScript and Python are more strict in this regard.

If *assignment by reference* is part of so many computer languages, it must be an important feature! But it seems pointless without a practical example.

5.1.7 Why assign by reference?

It is often mentioned that you can assign variables or objects by value or reference. But it is rarely explained why it is done this way.

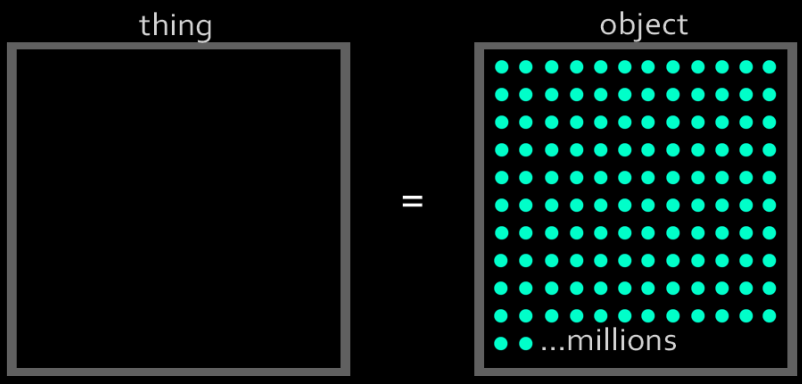
Computer memory is a limited resource. The larger the data sets you are generating or working with, the more memory you will require to store them.

Reference assignment solves two problems. One is that we save space in computer memory. We also avoid physically copying values from one place to another.

Copying is an computationally expensive operation for large data sets:

```
# assign by reference
object1 = object2
```

Imagine that an object contains 1 million properties. Copying the entire contents of an object is a lot of work.



Python's = operator has no way of knowing how large your object actually is. Had the = operator triggered a physical memory copy between objects, you'd have to wait a while until the operation would complete.

A simple assignment should never take that much time. This is the reason why most languages implement object assignment *by reference*.

If your object contains millions of properties it would be too much overhead for just one line of code. Now imagine if some of those properties are also objects containing thousands of other properties. This can bring execution to a halt.

A reference, on the other hand, is only assigned once as an alias to original value.

Note, however, this doesn't mean that you should always avoid copying objects. In fact, some algorithms require copying data. Assignment by reference simply increases efficiency in many cases.

5.1.8 Python Timers

How long would it actually take a Python program to copy 1 million values to another object? Let's find out while exploring Python timer functions.

```
# Include time module
import time

# Display monotonic time now
print(time.monotonic())

# Display monotonic time in nanoseconds
print(time.monotonic_ns())
```

As of the time when this book was written my results were:

```
1834844.703
1834844703000000
```

A monotonic clock is a clock that cannot go backwards. The method `time.monotonic()` returns the value of a monotonic clock in fractional seconds. Monotonic timers are often used when you need to measure elapsed time between two points in time.

Let's see monotonic timer in action by counting how long it takes to loop through 10 million numbers.

```
import time

# Track monotonic before for loop
start = time.monotonic()
print("start=", start)

# Create a range of 10 million numbers
number = range(0, 10000000)
```

```
print("Looping through 10 million numbers...")

# Loop 10 million times
for index in number:
pass

# Track time after for loop
end = time.monotonic()
print("end=".format(end))
print("Difference between two points in time:")
print("end-start=", end-start
```

Python for loops cannot be empty. But you can use pass if you simply want to loop through numbers without doing anything with them.

Now let's run the script to see its outcome in the Command Prompt:

```
C:\Python38-32>python book.py

start= 1836354.906
Looping through 10 million numbers...
end=1836355.656

Difference between two points in time:
end-start= 0.75
```

According to our script it took **0.75** seconds to loop through 10 million numbers in the range sequence. If you run the script again you will see slightly different results, but they will be close to the same number.

Your value might be different. This depends on many factors such as your processor type, whether other programs are running concurrently and eating up CPU cycles and so on. But in general you will get similar results every time you run this script.

Remember how in previous section we looked at assigning values by reference and not by value? This is **0.75** seconds to simply loop through 10 million numbers.

To actually copy 10 million properties from one object to another would take a lot longer. Let's take a look:

Let's add a new variable to this script:

copy = [];

And then within our for loop copy a value from range at current index:

for index in number: copy[index] = number[index]

This will copy 10 million values into copy array.

Now let's see how long it takes to assign one object to another by reference:

```
array = [1,2,3,4,5,6,7,8,9,10]
start = time.monotonic()
print("start=", start)
ref = array;
end = time.monotonic()
print("end=".format(end))
print("Difference between two points in time:")
print("end-start=%f" % (end-start))
```

The result is `0.000074` seconds. If you keep refreshing the script you will notice it is not a constant. Sometimes I also get `0.000010` or similar small numbers.

This is not a big surprise. All that was done under the hood was an exchange of a single address in memory. This operation doesn't take many computer processor cycles at all.

You can now see why assignment to objects by reference is implemented automatically in many languages.

But there is a problem – in many cases, you actually *do* want to physically copy an object from one to another. To solve it, your natural way of thinking will be to write your own copy method that loops through all properties and copies them one by one to properties of a separate object.

In languages like JavaScript, normally you have to write your own functions to do either a shallow or deep copy.

There is no such thing as "one solution fits all" for doing this. That's because your object can contain virtually any combination of properties, methods, other objects and arrays. Within those arrays, it is possible to have even more objects or arrays. It depends on the design of your object.

If you design a class efficiently, you may be able to reduce computational overhead by writing a copy function tailored specifically for your object structure.

With enough time you will start to notice that the most efficient object copy functions are recursive in nature. A recursive function is a function that calls itself until a resource such as a number of properties in an object or items in an array is depleted. A deep copy does this for each property, object and array within any other property, object or array.

5.1.9 Shallow Copy

Python's dictionary provides a special built-in function `copy` that can be called directly on dictionary object. The `copy` function also exists on lists, but not on tuples (because tuple is treated as an *immutable* object in Python.)

```
a = {"message": "hello"}
b = a.copy(a) # create copy of object a

print(f"Address of A is {hex(id(a))}")
print(f"Address of B is {hex(id(b))}")

>>>
Address of A is 0x2f15cd0
Address of B is 0x2f15d20
```

Both `a` and `b` are located at a unique memory address.

Let's change `"message"` property in object `b`:

```
b["message"] = "goodbye"
```

```
print(a)
print(b)
```

```
>>>
{'message': 'hello'}
{'message': 'goodbye'}
```

In contrast to our previous example `b` is a physical copy of `a` and not *a reference* to `a`. That's why the value changed only in object `b` – as it should.

Because both objects are stored at a unique memory address this also means that they occupy twice as much physical space.

To find out how much space *in bytes* is occupied by an object you can use the `getsizeof` function. In order to start using it you must include `sys` package:

```
import sys
print(sys.getsizeof(a))
print(sys.getsizeof(b))
```

```
>>>
128
128
```

At first sight it seems both objects occupy 128 bytes in computer memory. But it's not that simple. This is only if you're running your code from Python 3.8

You might get another number like 272 – in Python 2.7 – different versions of Python don't allocate the amount of memory in the same way.

Python's `getsizeof` function shows how much memory was allocated to store a given set of values. This doesn't really mean sum of all items stored in a dictionary.

But the size of a dictionary will noticeably expand based on number of attributes. In Python 3.8, for example, lists with 4 and 5 items will both allocate 128 bytes:

```
a = { "1":0, "2":0, "3":0, "4":0, }
b = { "1":0, "2":0, "3":0, "4":0, "5":0, }
```

```
print(sys.getsizeof(a), sys.getsizeof(b))
```

```
>>>
128, 128
```

But Python's memory manager will gracefully allocate an extra chunk of memory (68 bytes to be exact) soon as 6th item is added:

```
c = { "1":0, "2":0, "3":0, "4":0, "5":0, "6":0, }

print(sys.getsizeof(c))
```

```
>>>
196
```

If you have background in C or C++ be careful not to confuse Python's `getsizeof` function with `sizeof` operator in C. They don't work in the same way.

Moreover, `getsizeof` won't calculate size of nested properties in a dictionary.

You may want to think twice about using `getsizeof` method if calculating size of all elements in a list matters. That's because `getsizeof` is *shallow*. We will continue to explore what this means in next section.

What is a shallow copy?

To demonstrate the idea behind shallow copy, let's take a look at another example:

```
import copy # import copy.copy() and copy.deepcopy()

tree1 = { # create compound object
    "type" : "birch",
    "prop" : {"class" : {"kingdom" : "plantae"}}
}
```

```
# create a copy of tree1 using object's copy() function
tree2 = tree1.copy()
# create a copy of tree1 using copy() function from copy package
tree3 = copy.copy(tree1)
# create a copy of tree1 using deepcopy() function
tree4 = copy.deepcopy(tree1)

print(tree1)
print(tree2)
print(tree3)
print(tree4)

print(hex(id(tree1)))
print(hex(id(tree2)))
print(hex(id(tree3)))
print(hex(id(tree4)))
```

```
>>>
{'type': 'birch', 'prop': {'class': {'kingdom': 'plantae'}}}
{'type': 'birch', 'prop': {'class': {'kingdom': 'plantae'}}}
{'type': 'birch', 'prop': {'class': {'kingdom': 'plantae'}}}
{'type': 'birch', 'prop': {'class': {'kingdom': 'plantae'}}}
0x297e268L
0x297e9d8L
0x297ebf8L
0x297ee18L
```

All objects reside at a unique address in memory.

Each new object was created by copying `tree1` using a different copy function.

Remember that `tree1` is a *compound object* – it contains more than one level of nested objects. The `'prop'` attribute provides an extra level of nested dictionaries.

At this point it looks like there is no difference, especially between shallow copy with `copy.copy` method and the deep copy with copy.deepcopymethod.

If we change the first-level attribute "type" in any of the objects, and print them out again, they will change individually for each object:

```
tree1["type"] = "oak"
print(tree1["type"],
      tree2["type"],
      tree3["type"],
      tree4["type"])

>>>
('oak', 'birch, 'birch, 'birch')
```

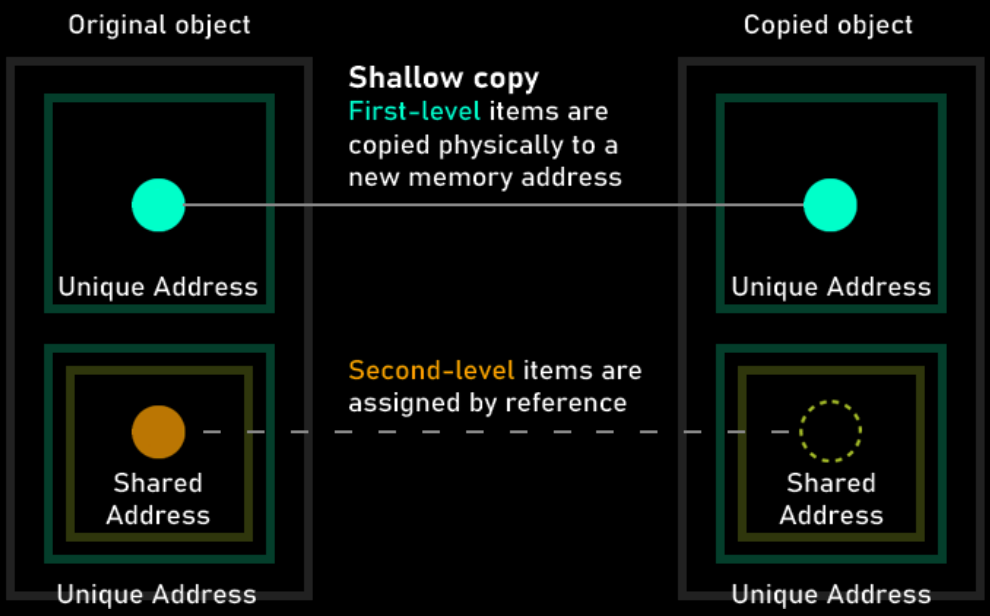
Now let's print out all trees again:

```
print(tree1)
print(tree2)
print(tree3)
print(tree4)


>>>
'type': 'oak', 'prop': 'class': 'kingdom': 'plantae'
'type': 'birch', 'prop': 'class': 'kingdom': 'plantae'
'type': 'birch', 'prop': 'class': 'kingdom': 'plantae'
'type': 'birch', 'prop': 'class': 'kingdom': 'plantae'
```

a Trying to change ["type"] attribute in any of the four objects won't change value of "type"in any of the other objects: it only affects the object being modified because "type"is a *first-level* item in the copied object stored at unique address.

None of *first-level* attributes are stored by reference – they all change separately for each object – even for objects that were copied using *shallow* `copy` functions.

The difference is in how *second-level* items are stored:



Let's change a *second-level* property `["prop"]["class"]` of *any* of the first 3 objects. Let's say `tree2` , in this example:

```
# change second-level attribute
tree2["prop"]["class"] = "Quercus"

print(tree1)
print(tree2)
print(tree3)
print(tree4)

>>>
{'type': 'birch', 'prop': {'class': 'Quercus'}}
{'type': 'birch', 'prop': {'class': 'Quercus'}}
{'type': 'birch', 'prop': {'class': 'Quercus'}}
{'type': 'birch', 'prop': {'class': {'kingdom': 'plantae'}}}
```

By changing *second-level* property `["class"]` of just the `tree2` object it was also changed in all of the first 3 objects.

That's because each of them was copied using a *shallow* copy function – where all *second-level* attributes are now stored by reference to the original.

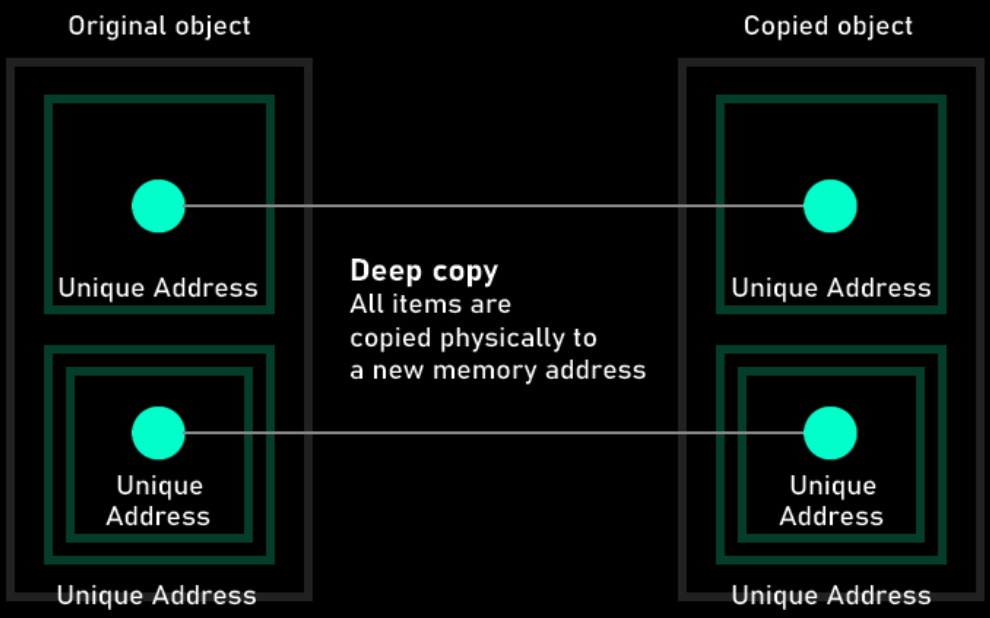
```
import copy

tree1 = { "name": "birch", "type": {"kingdom": "plantae"}}
tree2 = copy.copy(tree1) # shallow copy

# print address of objects
print(id(tree1))
print(id(tree2))
# print address of second-level attributes
print(id(tree1["type"]["kingdom"]))
print(id(tree2["type"]["kingdom"]))

>>>
43377256
43379160
43695240 <--- shared address
43695240 <--- shared address
```

Second-level, third-level and n*th*-level...after that will be also assigned by reference.

Deep copy

In our first example the `tree4` property was created using `copy.deepcopy` function – it recreated all attributes on the new object `tree4` by making a physical copy of all nested attributes it found in object `tree1` – no items were assigned by reference.

You can think of it visually as follows:



As you can imagine choosing deep copy on a large data set can potentially reduce performance of your application because all nested attributes are physically copied.

Some objects will have many nested attributes within other nested attributes. The `copy.deepcopy` method will iterate all nested attributes recursively using a potentially computationally expensive internal loop.

Python's `tuple` data type doesn't have a `copy` method for a reason – because it's *immutable*. Immutable objects don't support item reassignment after definition:

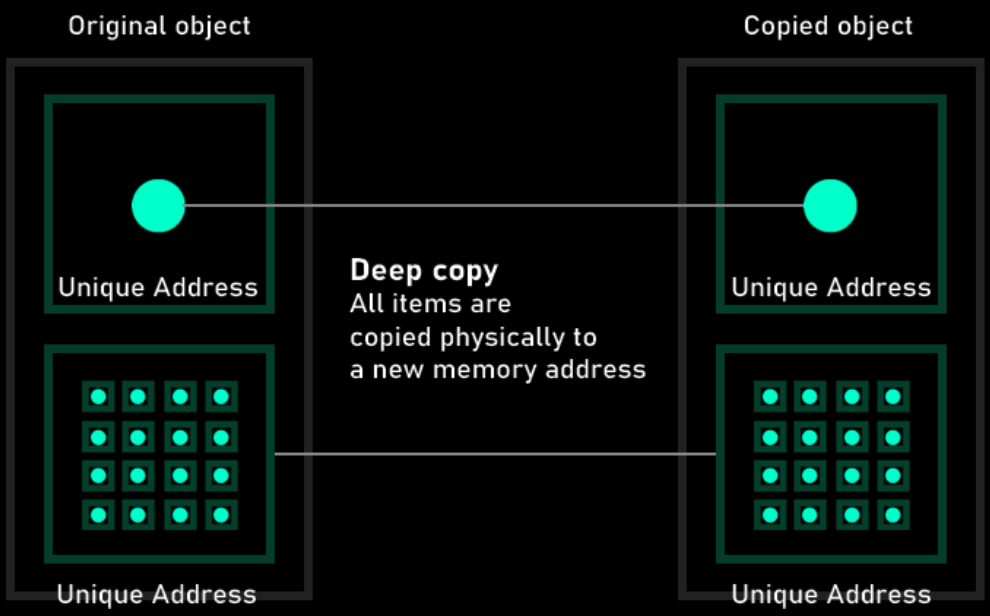
```
# create a tuple (immutable object)
tup = (1,2,3,4,5)
```

```
# attempt to change tuple's value at index 2:
tup[2] = 10

>>>
Traceback (most recent call last):
  File "tuple.py", line 5, in <module>
    tup[2] = 10
TypeError: 'tuple' object does not support item assignment
```

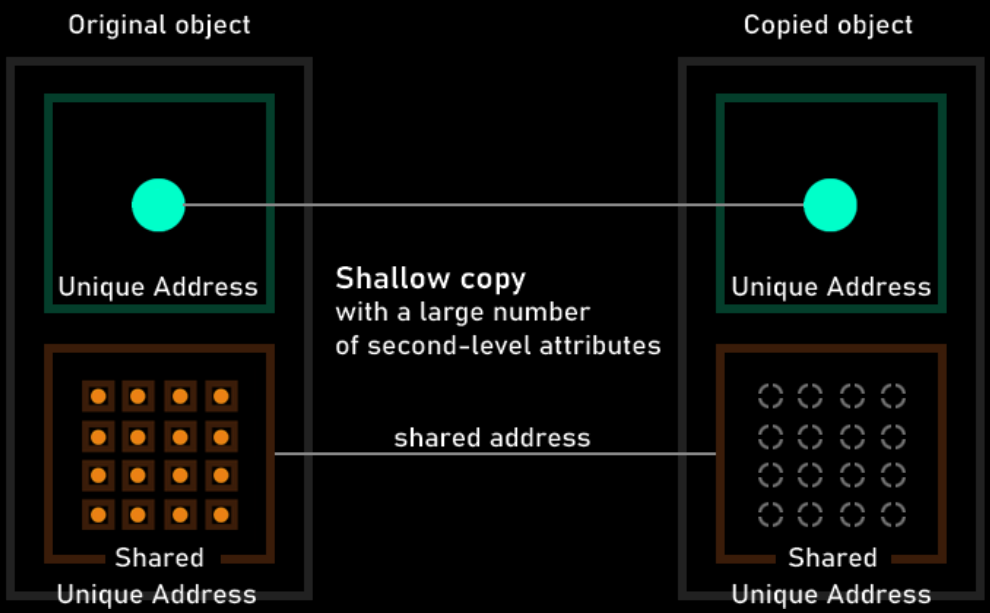
In many cases a copy of an object is made so that it can be modified. You'll probably want to use `list` or `dictionary` instead.

Be careful if using `copy.deepcopy` when you have a lot of data to crunch or when real-time optimizations matter. Use a deep copy only when it's 100% necessary and with proper data structures.

Why is it important? If you have an object with hundreds or thousands of *second-level* attributes copying becomes a computationally expensive operation:



And finally this is what a shallow copy with many attributes might look like.



You can see performance advantages here. And this is the reason why common copy functions are shallow.

Here objects stored in *second-level* attributes (note that those attributes can also contain hundreds of other objects) still exist physically only in the original object.

Note: You can create a recursive loop if you use a deep copy on an object that contains a reference to itself in one of its items. Python deals with it by tracking already copied objects in a `memo` dictionary.

The `memo` dictionary maps object id to the object. After making a deep copy of a complex object your memo might look as follows:

```
tree1 = {
    "type"    : "birch",
    "kingdom" : "plantae"
}
```

```
memo = {};
tree5 = copy.deepcopy(tree1, memo)

print(memo)
```

```
>>>
{47316864L: 'birch',
47309416L: {'kingdom': 'plantae', 'type': 'birch'},
47316944L: 'plantae',
47316904L: 'kingdom',
44916408L: 'type',
47311048L: ['plantae',
'kingdom',
'birch',
'type',
{'kingdom': 'plantae',
'type'   : 'birch'}]
}
```

Python keeps a mapping of objects to avoid copying objects residing at the same memory address more than once. This also prevents your deep copy from entering into an endless recursion loop or copying the same objects "too many" times.

Notice that none of the objects in the memo repeat. They are all stored at a unique memory address.

5.1.10 Stack, Heap and Garbage Collection

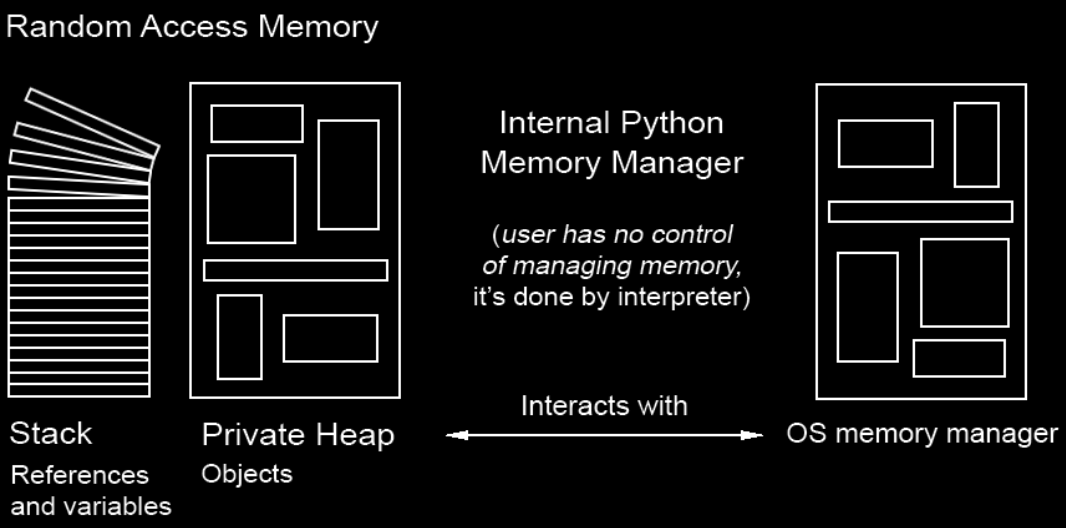


Figure 5.3: Simplified diagram of Python memory management model.

Python methods and variables are created on the Stack. A stack frame is created every time a method or a variable is created. Frames are destroyed automatically whenever a method is returned – a process called *garbage collection*. Objects and instance variables are created in Heap memory.

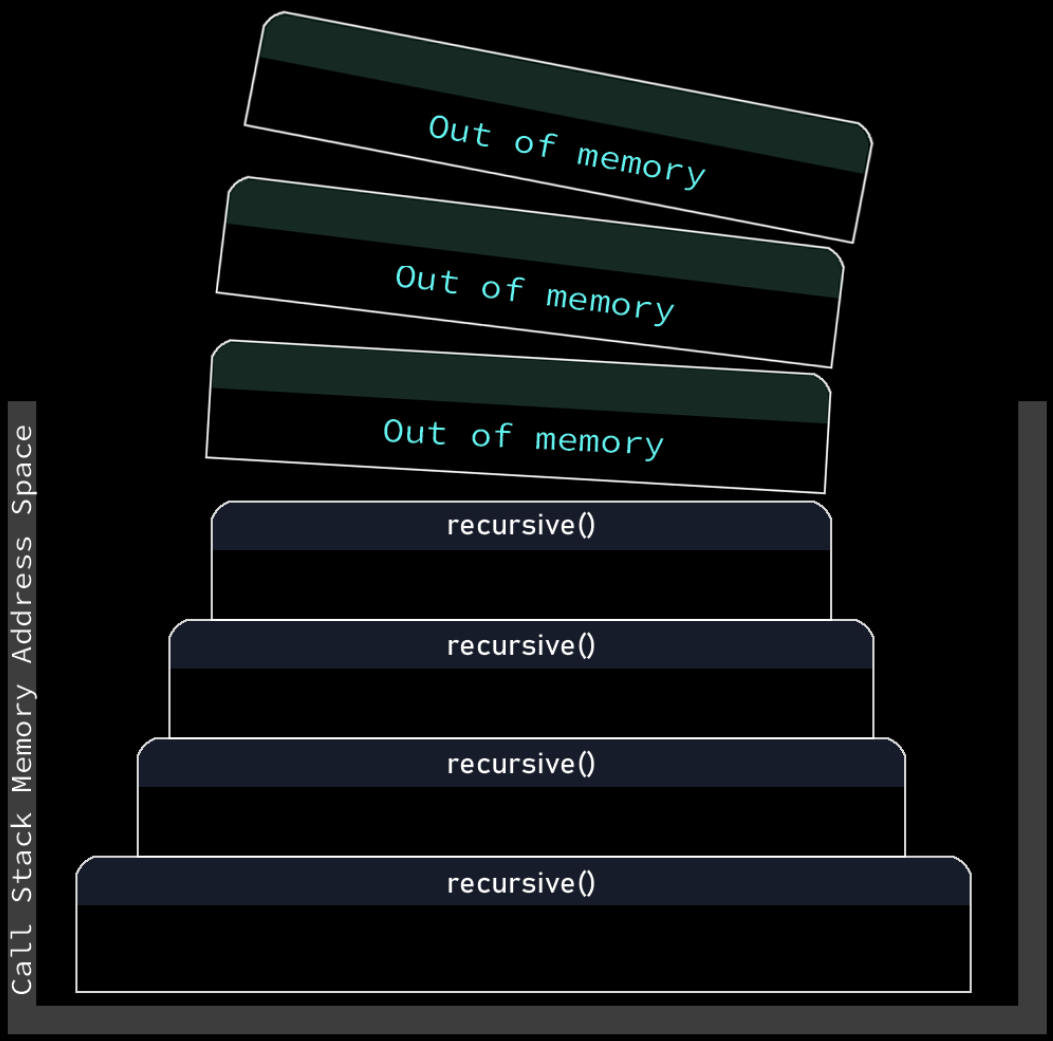
5.1.11 Stack Overflow

You can easily cause Stack Overflow in Python by executing a *recursive function*.

```
def recursive():
    return recursive()

# call it
recursive()
```

A *recursive function* is a function that calls itself from its own body. Calling a

function puts it on the stack until it returns. But a recursive function returns itself just to be called again, never leaving the stack:



The stack doesn't have an infinite memory reserve. With each function call more memory is allocated on the stack until it finally runs out. You should plan your

programs carefully so that something like this doesn't happen.

Running this code will result in the following output:

```
Traceback (most recent call last):
  File "addr.py", line 4, in <module>
    recursive()
  File "addr.py", line 2, in recursive
    return recursive()
  File "addr.py", line 2, in recursive
    return recursive()
  File "addr.py", line 2, in recursive
    return recursive()
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Stack memory is a limited resource. And eventually you run out of memory.

Chapter 6

Data Types and Structures

When I started learning data structures I would often get stuck trying to understand why Python has 4 relatively similar structures: `list`, `dictionary`, `tuple` and `set`.

As you dive deeper into working with these data types you will begin to form this *mental model* of their differences which should help you determine which data structure to choose for solving a particular problem:

LIST **Sequence** **mutable** **allows** duplicate values has **11** methods (*more may be added in the future*)	**DICTIONARY** **Not sequence** **mutable** **disallows** duplicates because it stores items as **{ key: value, }** pairs has **11** methods (*more may be added in the future*)
TUPLE **Sequence** **immutable** **allows** duplicate values has **2** methods: only has methods index() and count()	**SET** **Not sequence** **immutable** **disallows** duplicate values has **17** methods (*more may be added in the future*)

These are not only structures in Python. But they are the most important ones.

When learning data types and structures three main concepts to watch out for are whether data structure is a *sequence* (same as enumerable), whether it supports *immutability* and whether it *allows duplicate values*.

Data Types and Data Structures

Common data types include *boolean* (`bool`), *string* (`str`), *integer* (`int`), *complex number* (`complex`) and *decimal point numbers* (`float`). They can be interpreted as primitive values for using in statements such as: `1 + 2` and much more complex.

Data structures `list`, `dict` (as in *dictionary*,) `set`, `frozenset` and `tuple` are unique to Python. However, some of them resemble the same exact data structures from other languages that share similar functionality.

Difference between Data Types and Data Structures

A data structure is an abstract way of organizing complex data in a way that allows certain operations to be efficiently performed on it: an array or list, for example.

In computer science, Binary Tree can be thought of as a data structure . It consists of the abstract idea of nodes and possibly some methods for operating on them.

The Binary Tree data structure can also contain several methods defined for working with it, for example, `add_node()` and `remove_node()`.

While there is no such thing as a built-in Binary Tree data structure in Python, you can create one yourself by using `class` keyword. Classes will be covered later.

The Python's built-in data structures that can be categorized as *sequences* are *string*, *list*, *tuple*, *bytearray* and *range objects*. A sequence is exactly what it sounds like. It's an array-like set of values that follows some type of sequential (linear) order. Whereas dictionary, set and frozenset assume spontaneous order.

—

A data type is more primitive than data structure . For example an integer or `'int'` is a data type that stores whole numbers. A string or `'str'` is a data type that contains text. One way of looking at data types is to think of them as *units* for storing values in category represented by that type.

Numbers, strings, and booleans (True or False) can be thought of as primitive data types. Unlike data structures they usually don't have many methods attached to them because they represent basic form of a value which can be manipulated by

common operators.

Mutability

When it comes to data structures programmers need to understand the distinction between *mutable* and *immutable* data structures. The concept of immutability doesn't always work in the same way in different programming languages.

Generally, an immutable type disallows changing its values once they have already been set. However, you are still allowed to add new items or remove existing ones using that type's method `add()` or use any of its other built-in methods.

But manipulating the set by using its methods still follows one important rule: it assumes that none of the values will ever be changed once they are defined: in other words at all times the set remains immutable.

A `frozenset` cannot be updated after being initialized with a set of values. You can still use some of the set's methods on it like `union()`, `intersection()`, `copy()`, etc. Practical use cases will be covered later in this book.

Lists, Dictionaries and Ranges

The `list` in Python is a mutable data structure similar to `Array` in JavaScript. A `dict` is a dictionary – it's similar to JavaScript's *object literal* which is defined by curly brackets (and resembles JavaScript's JSON format.)

The `range()` function can be mistaken for a data type by beginners. There is no such thing as range data type – this built-in function produces a *sequence* of numeric values which are returned as `list` data structure.

Sequences

In Python a sequence is a general term for referring to an ordered set. Sequences can be thought of as being synonymous with enumerable data types (in that they can be placed into a one-to-one correspondence with the natural numbers.)

There is no such thing as sequence data structure in Python but you will hear the

word sequence often – `string`, `list` and `tuple` are considered to be *sequences*.

A sequence assumes there is linear order to stored values. Neither `set` nor `dict` (dictionary) are sequences, but `string`, `list` and `tuple` are.

Dictionaries refer to its data by using attributes – also known as properties or *keys*. The order in which values are stored cannot be relied on – it can change spontaneously at any moment throughout lifetime of your program. Dictionaries are iterable but they are not enumerable and therefore are not sequences.

You can think of a sequence as *enumerable* object because it assumes each item is stored at a numeric index. When learning data types it's important to draw distinction between *enumerable* and *iterable* objects.

Iterable objects can be *iterated* with a for-loop or the built-in iterator function **iter** and its sidekick function **next** which gives you control over stepping through the entire set – one item at a time.

Iterators walk through all object properties when order is not important. But it isn't uncommon for *iterable* types to be converted to *enumerable*.

Items in an *enumerable* object are looped through in the same order in which they were defined, using an indexing system where first item is stored at index 0, second item is stored at index 1, and so on. It assumes unchanging index slots.

You may have already used iterable and enumerable objects to write code without knowing what they are. Not many people learn computer languages in depth before they start using them to write code. However this awareness can only deepen your understanding of programming and will usually lead to greater code efficiency.

—

Data types help us think about how to organize and manage primitive data (like strings and numbers) in a computer program. **Data structures** are also types but they usually consist of more complex objects. Python has several built-in data types and data structures. Others can be imported from packages.

While there are no imposed rules on which data type or data structure you should use to solve any given problem it's important to know how to choose the right one for a given task.

The goal of this chapter is to demonstrate how each data type works in isolation

and then show some practical examples.

6.0.1 Built-in `type()` function

Once a variable is assigned to a value you can find out its type by using the built-in function `type()` . It only takes one argument – the value – and returns its type.

```
a = 1
b = 15.75
c = "hello"

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

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

Note in Python 2.7 `type(1)` evaluates to `<type 'int'>` and not `<class 'int'>`. Classes can be thought of as blueprints for creating objects of the type they represent. We will cover classes in more detail at a later chapter.

Each type has its own built-in class definition. The (sometimes) abbreviated name is what we're looking for. Here `1` is an *integer* or `'int'` and `"hello"` is a *string* or `'str'`. Decimals are floating point numbers or `'float'` for short.

The `type()` function can be also used to identify *lists*, *sets*, *tuples* and *dictionaries*:

```
tuple = ("a", "b")
list = ["a", "b"]
set = {"a", "b"}
dict = {"a": 1, "b": 2}
```

```
print(type(tuple))
print(type(list))
print(type(set))
print(type(dict))
```

```
>>>
<class 'tuple'>
<class 'list'>
<class 'list'>
<class 'dict'>
```

Whenever you need to find out the type of a value call `type(value)`.

Understanding the purpose of checking for type of a value might not make sense right away while learning Python. It is parallel to your growing skill as a programmer. At one point you might even want to know how to identify a function:

```
def fun():
  pass

print(type(fun))
```

```
>>>
<class 'function'>
```

Checking if a variable is a function is not uncommon and in some instances becomes an important part of writing Python code.

In previous examples we printed out variable type. That's not enough for doing something useful with it. To execute a statement based on whether type function evaluates to a specific type we can use its class name with `is` and `is not`:

```
t = ("a", "b", "c")
l = ["a", "b", "c"]
s = {"a", "b", "c"}
```

```
print(type(t) is tuple)
print(type(l) is list)
print(type(s) is set)
print(type(s) is not set)
```

```
>>>
True
True
True
False
```

To branch out you can write this `if` statement:

```
if type(t) is tuple:
    print("This is a tuple.")
else:
    print("This is not a tuple.")
```

```
>>>
This is a tuple.
```

What about functions?

Technically, functions are *callable* and don't belong to the same type of objects as primitive data types. They are not data structures either. Therefore there is no equivalent to be used together with `is` keyword:

```
print(type(fun) is function)
```

```
>>>
NameError: name 'function' is not defined
```

To identify a function use the built-in `callable` method instead:

```
def fun():
    print("fun")
```

```
print(callable(fun))
```

```
>>>
True
```

However keep in mind that running `callable` on classes will also return `True` because class names can be called to make an object instance of that class:

```
class cl:
    print("inside class")
print(callable(cl))
```

```
>>>
True
```

6.0.2 Duck Typing

Examples above present a small problem. If both function and class return `True` when passed to `callable` how do we make a distinction between function and a class? The answer is we don't.

When writing Python code it's best to think in terms of callable or not callable, rather than class or function. This concept is called Duck Typing. It's a common way of thinking about types in Python (and some other languages).

One simple way of thinking about Duck Typing is with a phrase *"If it looks and quacks like a duck, then it's a duck!"* What this means is – if something can be called – it is a *callable* regardless of whether it's a function or a class.

In practice the distinction between classes and functions is rarely important – when writing code that tests for either, that code is usually specialized to explicitly handle that particular type. It is proper to simply think of both of them as *callable*.

The concept of Duck Typing can be vaguely extended to Python data structures as well. For example `list` and `dictionary` are both considered to be *sequences*.

6.0.3 Data Types and Data Structures

This table shows *examples of values* (left column) and data types or structures they represents (right column).

Value	Data Type
`True`	`bool`
`False`	`bool`
`150`	`int`
`11.43`	`float`
`1j`	`complex`
`"hello"`	`str`
`f"dandelion"`	`str` (f-string)
`f"{3} dandelion"`	`str` (f-string with template value)

Value	Data Structure
`{"name": "Steve", "age": "27"}`	`dict`
`["wood", "steel", "iron"]`	`list`
`("wood", "steel", "iron")`	`tuple`
`{"wood", "steel", "iron"}`	`set`
`frozenset({"a", "b", "c"})`	`frozenset`
`b"Message"`	`bytes`
`bytearray(3)`	`bytearray`
`memoryview(bytes(10))`	`memoryview`
`memoryview(bytes(10))`	`memoryview`

Classes and `type()`

The `type()` function can be applied to any value to find out its type. We've looked at common data types in previous section.

We also identified a function but what about classes?

```
class Word:
  def __init__(self, text):
    pass

# create an object instance of type Word
light = Word("light")

print(type(Word))  # class
print(type(light)) # object instance
```

```
>>>
<class 'type'>
<class '__main__.Word'>
```

Essentially classes define a unique type of an object. This is why Python is saying that the type of class Word is `<class 'type'>`. This hints on circular logic of classes and types that can be also observed in the design of many other languages.

Because light is an instance of an object of type Word, here Python will tell us that this object was created by constructor function `__main__` that belongs to Word.

This is interesting because in our definition of the class Word the constructor function was named `__main__` and not `__init__` . That's because Python classes actually have two constructor functions. We'll take a closer look at how exactly this works in a later section about classes and objects.

6.0.4 Working with data types

Up until this point we've only identified Python data types by name and output from `type()` function. But what situations are they used in?

At first sight *list*, *tuple* and *set* differ only by type of brackets around the data set.

When should you choose a *list* over *tuple*? Or a *set* over a *list*? How are Python data types different or similar to data types in other languages? The following section will try to answer all of these questions with a few brief practical examples.

6.0.5 bool

`True` and `False` are primitive boolean values in Python. They can be used in many ways. One of them is to indicate whether a feature or a setting is currently in "on" or "off" state.

Examples below demonstrate booleans used together with an `if` statement and two different styles of Pythonic *ternary* operators.

```
dark  = True    # dark mode enabled
large = True    # large font
show  = False   # private email

if dark:
    print("Dark mode is on!")
else:
    print("Dark mode is off!")

# ternary if else operator
status = "on" if large else "off"
print(f"Use large font: status")

# ternary statement with tuples
public = False
status = ("private", "public")[public]
print(f"Email is {status}")
```

```
>>>
Dark mode is on!
Use large font: on
Email is private
```

Booleans can be used to check if an object like a `dictionary`, `list` or a `set` is empty by initializing the value with `bool` function.

```
print(bool(False))      # False
print(bool(True))       # True
print(bool(1))          # Numeric value 1 is cast to True
print(bool(0))          # Numeric value 0 is cast to False
print(bool([]))         # False because list is empty
print(bool([1, 2]))     # True because list is not empty
print(bool({})          # False because dictionary is empty
print(bool({"a": 1})    # True because dictionary is not empty
```

```
>>>
False
True
True
False
False
True
False
True
```

Do something if a list is empty, and something else if it isn't:

```
A = [5,7,3]

if bool(A) == True:
    first = A[0]
    print(f"List is not empty, first item is = first")
    print(first)
else:
    print("List is empty")
```

```
>>>
List is not empty, first item is = 5
```

6.1 str

Strings are straightforward. They are used to print text.

```
string = "Hello there."
print(string)
```

```
>>>
Hello there.
```

You just have to watch out for the many different ways in which strings can be defined in different versions of Python.

For example the *f-string* doesn't work in earlier versions of Python:

```
num = 10
fstr = f"There are {num} apples in the basket."
print(fstr)
```

```
>>>
There are 10 apples in the basket.
```

This code would fail in Python 2.7.16 but works in Python 3.8.3

6.2 int

Generally the integer represents a whole number and whole number operations:

```
number = 100
print(type(number))
print(number)
print(100 + 25)
```

```
>>>
```

```
<class 'int'>
100
125
```

But when two whole numbers are multiplied by a fractional number (or divided) they are automatically cast to `float`:

```
print(10 / 2)
print(10 * 0.1)
```

```
>>>
5.0
1.0
```

6.3 float

Floating point numbers are helpful for calculations that require decimal point.

```
float = 200.14
print(type(float))
print(float)
print(float + 81.33)
```

```
>>>
<class 'float'>
200.14
281.46999999999997
```

Popular Python math library `cmath` can be used to produce the value of PI:

```
import cmath
print('PI =', cmath.pi)
print(type(cmath.pi))
```

```
>>>
('PI =', 3.141592653589793)
<type 'float'>
```

Floating point numbers can be used for calculating any value that requires decimal point precision such as shopping cart balance or tax rate.

6.4 complex

A *complex number* is a number that can be expressed in the form of `a + bi` where `i` represents imaginary unit.

Complex numbers consist of a Real and an Imaginary Number added together.

In Python append `j` to create the imaginary part of the number:

```
num = 3 + 1j

print(num)
print(type(num))
```

```
>>>
(3 + 1j)
<type 'complex'>
```

To access each part of the number individually use `real` and `imag` attributes:

```
print(num.real)
print(num.imag)
```

```
>>>
3.0
1.0
```

Complex numbers can be added, subtracted, multiplied and divided:

```
a = (1 + 2j)
b = (2 + 5j)

print(a + b)
print(a - b)
print(a * b)
print(a / b)
```

```
>>>
(3+7j)
(-1-3j)
(-8+9j)
(0.41379310344827586-0.03448275862068965j)
```

PythonAlternatively, to create a complex number you can also use Python's built-in function `complex(`*`real, imaginary`*`)`:

```
c = complex(1, 5)
print(c)
```

```
>>>
(1+5j)
```

6.5 dict

A dictionary (or `dict`) stores data using comma-separated *`key:value`* pairs. It is like an *associative array* – where each item has a unique name or "key":

```
alphabet = {
    "a": 1,
    "b": 2,
    "c": 3,
}
```

Note the *trailing* comma in `alphabet` dictionary after the last item `"c"` is optional. This is legal syntax that won't generate an error, but the comma can be omitted.

```
print(alphabet)
print(type(alphabet))

>>>
{'a': 1, 'c': 3, 'b': 2}
<type 'dict'>
```

Combining dictionaries

If you're familiar with JavaScript you may know how to combine two JSON-style objects (that look exactly like Python's dictionaries) using something called spread operator that looks like 3 leading dots: `...`

```
let a = {'a': 1}
let b = {'b': 2}
let c = {...a, ...b}

console.log(c)

>>>
{'a': 1, 'b': 2}
```

(Do not try this in Python interpreter, this is JavaScript!)

As of Python 3.5 you can perform a similar operation to combine dictionaries by using the leading `**` operator as follows:

```
a = {'a': 1}
b = {'b': 2}

c = {**a, **b}
```

```
print(c)
```

```
>>>
{'a': 1, 'b': 2}
```

Working with dictionaries

It's tempting to write your own for-loops to perform many common operations on items in a dictionary – but instead you should use built-in dictionary methods.

Each data structure provides a built-in set of functions that can be called directly from the instance of the object. Each of the following sections will demonstrate examples of dictionary functions.

dict.clear()

To remove all items use `clear()` method on the dictionary object:

```
alphabet = { "a": 1, "b": 2, "c": 3, }
print(alphabet.clear())
```

```
>>>
None
```

dict.copy()

Creates a copy of the original dictionary.

```
alphabet = { "a": 1, "b": 2, "c": 3, }

# make a shallow copy of alphabet
c = alphabet.copy()
print(c)
```

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

The copy method will produce a shallow copy in contrast to a deep copy. This difference between the two is explained in another section of this book.

dict.fromkeys()

Helps create a dictionary from a set of keys stored in a list or tuple:

```
# keys stored in a tuple
keys = ("a", "b", "c")
value = 0

d = dict.fromkeys(keys, value)
print(d)
```

```
>>>
'a': 0, 'c': 0, 'b': 0
```

Note that the original order is not guaranteed:

```
# loop through it
for v in d:
    print(v)
```

```
>>>
a
c
b
```

You can't rely on dictionaries for data to remain linear. If you need a linear set of values you should use a `list` instead.

dict.get()

You can access dictionary attributes directly with brackets `[]`:

```
cats = {
  "one": "luna",
  "two": "felix",
}

print(cats["one"])
print(cats["two"])
```

```
>>>
luna
felix
```

But the safest way to get value of an attribute is by using `get` function:

```
print(cats.get("one"))
print(cats.get("two"))
```

```
>>>
luna
felix
```

Trying to access an attribute that doesn't exist will produce `None`:

```
print(cats.get("three"))
```

```
>>>
None
```

The `get` function has another form that takes second argument. If item with the key is not found (in this example `"three"` doesn't currently exist in `cats` dictionary) then the get function will return whatever you pass as second argument:

```
name = cats.get("three", "undefined")

print(name)
```

```
>>>
undefined
```

This is useful when the key is not found. You can assign it an alternative value of your choice, for example: `"undefined"` or `"not available"`.

dict.items()

Return dictionary's key and value pairs as a list of tuples.

```
notebook = {
  "size"      : "11 x 8.5",
  "type"      : "notebook",
  "genre"     : "sheet music",
  "published" : 2021
}

result = notebook.items()

print(result)
```

```
>>>
dict_items([('size', '11 x 8.5'), ('type', 'notebook'),
('genre', 'sheet music'), ('published', 2021)])
```

Why not then iterate over the results?

```
for item in result:
    print(item)
```

```
>>>
```

```
('size', '11 x 8.5')
('type', 'notebook')
('genre', 'sheet music')
('published', 2021)
```

To access keys and values separately:

```
for item in result:
    key = item[0]
    value = item[1]
    print(f"key={key}, value={value}")
```

```
>>>
key=size, value=11 x 8.5
key=type, value=notebook
key=genre, value=sheet music
key=published, value=2021
```

dict.keys()

If you only need keys:

```
notebook = {
  "size": "11 x 8.5",
  "type": "notebook",
  "genre": "sheet music",
  "published": 2021
}

for key in notebook.keys():
    print(f"key = key")
```

```
>>>
key = size
key = type
```

```
key = genre
key = published
```

dict.values()

To access values:

```
notebook = {
    "size": "11 x 8.5",
    "type": "notebook",
    "genre": "sheet music",
    "published": 2021
}

for value in notebook.values():
    print(f"value = {value}")
```

```
>>>
value = 11 x 8.5
value = notebook
value = sheet music
value = 2021
```

dict.pop()

Remove item from dictionary by key name:

```
vehicle = {
    "make": "Tesla",
    "model": "3",
    "year": 2021
}

vehicle.pop("model")
```

```
print(vehicle)
```

```
>>>
{'make': 'Tesla', 'year': 2021}
```

dict.popitem()

Remove last item from dictionary:

```
vehicle = {
    "make": "Tesla",
    "model": "X",
    "year": 2021
}

vehicle.popitem("model")

print(vehicle)
```

```
>>>
{'make': 'Tesla', 'model': 'X'}
```

dict.setdefault()

Return value of key specified in first argument. If key doesn't exist returns `None` or optional alternative value specified in second argument.

```
vehicle = {
    "make": "Tesla",
    "model": "X",
    "year": 2021
}

a = vehicle.setdefault("model", "undefined")
b = vehicle.setdefault("modex", "undefined")
```

```
c = vehicle.setdefault("modec")

print(a)  # X
print(b)  # undefined
print(c)  # None
```

```
>>>
X
undefined
None
```

dict.update()

Insert item into dictionary:

```
vehicle = {
    "make": "Tesla",
    "model": "X",
    "year": 2021
}

vehicle.update({"price":"$79,990"})
print(vehicle)
```

```
>>>
{'make': 'Tesla', 'model': 'X', 'year': 2021, 'price': '$79,990'}
```

6.6 list

In contrast to dictionary the `list` data structure provides a way to store linear data. Each item is assumed to be stored at an index: 0, 1, 2...etc.

Lists items are defined as a comma-separated list inside parenthesis `()`:

```
trees = ("oak", "birch", "pine")

print(trees)
```

```
>>>
("oak", "birch", "pine")
```

First item "oak" is stored at index 0. "birch" is stored at index 1, and so on.

Because items are assumed to be stored at a numeric index starting from 0, to modify the second value in the list you can access it directly as follows:

```
trees[1] = "sassafras"

# print the list with altered item
print(trees)
```

```
>>>
("oak", "sassafras", "pine")
```

The value "birch" in second slot was successfully replaced by "sassafras".

Combining lists

If you're familiar with JavaScript you know how to merge arrays (they resemble Python's lists) using spread operator that looks like 3 leading dots: ...

```
let a = [1, 2]
let b = [3, 4]
let c = [...a, ...b]

console.log(c)
```

```
>>>
[1, 2, 3, 4]
```

As of Python 3.5 you can perform a similar operation to combine lists by using the leading * operator as follows:

```
a = [1, 2]
b = [3, 4]
```

```
c = [*a, *b]

print(c)
```

```
>>>
[1, 2, 3, 4]
```

Like dictionaries, lists also have several built-in helper functions. Use them whenever possible before trying to invent your own code to do the same thing.

list.append()

The `list.append` method adds 1 element to the list.

Appending another list will nest it:

```
a = [1, 2]
b = [6, 7, 8]

a.append(3)
a.append(4)
a.append(5)
a.append(b)

print(a)
```

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

list.extend()

In previous example we tried to append another list to a list. But append simply nests the original list with a new one. This isn't always what you want to happen.

In order to *extend* a list by another list you can use `extend` operation:

```
a = [1,2,3]
b = [4,5,6]

a.extend(b)

print(a)

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

list.clear()

Removes all items from the list:

```
a = [1,2]

a.clear()

print(a)

>>>
[]
```

list.copy()

To create a copy of a list:

```
a = [1,2]
b = a.copy()

print(b)

>>>
[1, 2]
```

list.count()

Count the number of times a value appears on a list:

```
message = ['H', 'e', 'l', 'l', 'o']

num = message.count('l')

print(f"Letter 'l' was found num times in 'message'")
```

```
>>>
Letter 'l' was found 2 times in '['H', 'e', 'l', 'l', 'o']'
```

list.index()

Find the index position of a value in a list. Note, lists follow 0-index enumeration of values. The first letter 'a' is located at index 0, 'o' is at 1, etc.

```
vowels = ['a','o','i','u','e']

num = vowels.index('u')

print(num)
```

```
>>>
3
```

list.insert()

What if you don't want to either append to a list nor extend it...but rather, you want to insert new items at a specific index? That's what `insert` method is for:

```
a = ['a', 'b', 'd', 'e']
```

```
# something doesn't look right
print(a)

# let's insert 'c' where it belongs:
a.insert(2, 'c')

print(a)

>>>
['a', 'b', 'd', 'e']
['a', 'b', 'c', 'd', 'e']
```

list.remove()

And to remove 'c' from the list again:

```
a = ['a', 'b', 'c', 'd', 'e']

a.remove('c')

print(a)

>>>
['a', 'b', 'd', 'e']
```

Important: only one matching item will be removed:

```
a = ['a', 'b', 'c', 'd', 'c', 'e']

a.remove('c')

print(a)

>>>
['a', 'b', 'd', 'c', 'e']
```

Note the second c is still there after the remove operation. To remove all 'c's from the list you can first count them, and then call remove the number of times it appears in the list:

```
alpha = ['a', 'b', 'c', 'd', 'c', 'e']

for item in range(alpha.count('c')):
    alpha.remove('c')

print(alpha)

>>>
['a', 'b', 'd', 'e']
```

The range function will create a range of numbers in the amount of 'c' letters found in the list. This is why our for loop will iterate exactly the number of times 'c' appears on the list.

list.pop()

The pop method ejects the last item from the array:

```
a = ['a', 'b', 'c', 'd', 'e']

a.pop()
print(a)

a.pop()
print(a)

a.pop()
print(a)

a.pop()
print(a)
```

```
a.pop()
print(a)
```

```
>>>
['a', 'b', 'c', 'd']
['a', 'b', 'c']
['a', 'b']
['a']
[]
```

If you try to pop an empty array you will get an IndexError:

```
a = []
a.pop()
```

```
>>>
Traceback (most recent call last):
  File "sample.py", line 18, in <module>
    a.pop()
colorerrorIndexError: pop from empty list
```

list.reverse()

```
alpha = ['a', 'b', 'c', 'd', 'e', 'f']

alpha.reverse()

print(alpha)
```

```
>>>
['f', 'e', 'd', 'c', 'b', 'a']
```

list.sort()

By default `sort` method rearranges numbers and letters in ascending order.

```
alpha = ['b', 'a', 'c', 'f', 'e', 'd']

alpha.sort()

print(alpha)
```

```
>>>
['a', 'b', 'c', 'd', 'e', 'f']
```

Because this is how characters appear in the ASCII table, we get a natural order for letters. Numeric values will also be sorted by their actual amount:

```
alpha = [4,5,11,12,0,1,6,7,8,9,10,2,3]

alpha.sort()

print(alpha)
```

```
>>>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

6.7 range(*start*, *end*) or range(*total*)

`range()` is a built-in function that produces a `list` containing only numeric values in amount specified by the range defined by its argument(s).

To define a range between numbers `4` and `5` (inclusive):

```
ran = range(4, 6)
```

```
for item in ran:
    print(item)

>>>
4
5
```

Using range function with only one argument will produce a range of numbers starting from 0 in the amount specified by the argument:

```
ran = range(5)

for item in ran:
    print(item)

>>>
0
1
2
3
4
```

The range function is primarily used for generating lists of numbers.

```
# create a list of 10 numbers 0-9
r = range(10)

print(r)

r[0] = 'start'
r[1] = 5000
r[9] = 'end'

print(r)
print(type(r))
```

```
>>>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
['start', 5000, 2, 3, 4, 5, 6, 7, 8, 'end']
<class 'list'>
```

A range is still just a `list` and values are not required to be numeric. (But range is primarily used for working with numbers.)

6.8 tuple

The `tuple` data structure represents a mixed list of items. Tuples are similar to lists, but main difference is that tuples are *immutable*. Lists are *mutable*.

Mutable types allow you to change item values after they're defined. Trying to change values in an immutable object will produce an error.

A list of vowels should not change once defined. It wouldn't make sense to add any more items to a complete set of vowels. Therefore, they should be probably stored in a `tuple`, `set` or `frozenset`.

In Python it's common to use some of the data structures to initialize data structures of another type. You can initialize a `tuple` using a `list` by passing the `list` to `tuple` constructor function:

```
list = ['apples', 'oranges', 'pineapples']

tup = tuple(list)

print(tup)
```

```
>>>
('apples', 'oranges', 'pineapples')
```

It's possible to create both a list of tuples, and a tuple of lists:

```
# list of tuples
a = [(1, 2), (3, 4)]

# tuple of lists
b = ([1, 2], [3, 4])

print(a)
print(b)
```

```
>>>
[(1, 2), (3, 4)]
([1, 2], [3, 4])
```

So what is the difference? This section will try to answer that question.

Just like lists, tuples may contain values of different types:

```
mixed = ("almonds", "oranges", 15, 7, 0, ['a','b','c'])
print(tup)
```

```
>>>
('almonds', 'oranges', 15, 7, 0, ['a','b','c'])
```

You can also initialize a tuple from a list:

```
employees = ["mary", "robert", "david"]

tup = tuple(employees)

print(tup)
```

```
>>>
('mary', 'robert', 'david')
```

So far tuple works just like a list. You can even access items by using square brackets [] with an index:

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

print(tup[0])
print(tup[1])
```

```
>>>
mary
robert
mary
robert
```

Again, no difference here.

tuple is immutable

Finally, the key difference between a list and a tuple is that tuple is immutable – its items cannot be changed once defined:

```
# ok
employees[0] = "david"
print(employees)

# not ok
tup[0] = "david"
print(tup)
```

```
>>>
['david', 'robert', 'david']
Traceback (most recent call last):
  File "example.py", line 22, in <module>
    tup[0] = "david"
TypeError: 'tuple' object does not support item assignment
```

Immutable types don't respond to access operators and usually don't have any methods for changing values.

In fact `tuple` only has two methods: `index()` and `count()`. Whereas a list has `append()`, `clear()`, `copy()`, `count()`, `reverse()`, `sort()` and some others.

As you can see `tuple` can be thought of as an *immutable* `list`.

This also means tuples can be only defined once. After that none of its items can be changed. However, unlike `set` or `frozenset` (we will see the difference between those shortly,) `tuple` *can* contain duplicate items.

Nested tuples

Like lists, tuples can be nested and iterated:

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

print(nested)
```

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

To iterate the same tuple:

```
for var in nested:
  print(var)
```

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

In contrast to `list` which is an *iterable* data structure, the values of `tuple` can also be *unpacked* using the following syntax:

```
(var1, var2, ...)  = tuple
```

```
mixed = ("almonds", "oranges", 15, ['a', '1'])

# unpack tuple items into variable names
(nuts, fruit, number, list) = mixed

print(nuts)
print(fruit)
print(number)
print(list)

>>>
almonds
oranges
15
['a', '1']
```

JavaScript programmers will find this similar to Destructuring Assignment feature.

Tuple data structure only has 2 methods:

tuple.count()

Count the number of times a value appears in tuple:

```
word = ('h','e','l','l','o')

# count number of times letter 'l' appears in the tuple
num = word.count('l')

print(num)

>>>
2
```

Python makes it possible to count complex values:

```
tup = ([1,2],[1,2],[1,2],[1,3])

a = tup.count([1,2])
b = tup.count([1,3])

print(a)
print(b)
```

```
>>>
3
1
```

However, count is shallow and won't count nested items:

```
tup = (1,1,1,(1,1))

num = tup.count(1)

print(num)
```

```
>>>
3
```

tuple.index()

Find out index of requested value.

```
fruit = ('r','e','d','a','p','p','l','e')

i = fruit.index('p')

print(i)
```

```
>>>
4
```

Like lists, tuple indices are 0-index based and start count from 0, not 1. For example the first item `'r'` is located at index 0.

In this example the return value is 4 because the first occurrence of letter `'p'` in the tuple is located at index 4. The `index` function returns soon as it finds first occurrence of the value without iterating the rest of the list.

Tuples are immutable but can contain items of other types, including lists

We already know that by design tuples are *immutable*. This means once items are assigned for the first time you can't change them again.

However, lists are mutable and a list can be one of the items in a tuple. In this case you can actually change values in a list that appears in a tuple:

```
tup = (0, ['a'])

tup[1][0] = 'b'

print(tup)

>>>
(0, ['b'])
```

Changing a list in a tuple is completely legal.

```
tup = (1, 2)
print(tup)

tup[0] = 5
print(tup)

>>>
(1, 2)
Traceback (most recent call last):
  File "tuple.py", line 5, in <module>
    tup[0] = 5
TypeError: 'tuple' object does not support item assignment
```

Trying to change items in an immutable data type such as tuple will produce a TypeError. And that's exactly what happened here. Use lists if you need ability to change values in your data set.

There is one gotcha when it comes to working with tuples. And it's about tuple definitions that contain only one item. They are treated as strings. See below.

Constructing tuples containing 0 or 1 items.

Tuple containing only 1 item will be treated as a string:

```
tup = ('a')

print(type(tup))
print(tup)
```

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

If you *need* the data set to be treated as tuple, add a comma:

```
tup = ('a',)

print(type(tup))
print(tup)
```

```
>>>
<class 'tuple'>
('a',)
```

Note that empty tuples are still treated as tuple type:

```
tup = ()
```

```
print(type(tup))
print(tup)
```

```
>>>
<class 'tuple'>
()
```

Accessing tuples by index or slice

To access data in tuple you must use either an integer or a slice.

The following examples demonstrate this:

```
tup = (10,20,30,40,50)

# access individual items
print( tup[0] )
print( tup[1] )

# slice out items between 1 and 3
print( tup[1:3] )

# reverse a tuple with slicing operation
print( tup[::-1] )
```

```
>>>
10
20
(20, 30)
(50, 40, 30, 20, 10)
```

The last two examples use slicing operation. We will take a closer look at how they can help us slice sequences in one of the next sections.

A *slice* is a built-in operation that works on sequences. This means it works on a `tuple` because tuples are sequences. We'll learn more about **slicing sequences** in one of the next sections. But first...what are sequences?

Working with sequences

Before we move on I think this is a good place to talk about sequences.

The `tuple` is considered to be a *sequence* just like a `list` or `string`.

A sequence is defined by *particular order in which things follow each other.* When we think of Python sequences we think of the order in which items are arranged.

Sequences can be thought of as *ordered sets*. However, the `set` (as we will see later) structure is not a sequence in Python. In a `set` order is not guaranteed.

Sequences guarantee that items in a sequence will be accessible and available at the index at which they were originally defined:

```
str = "hello";

# access second character
print(str[1])

>>>
e
```

Python is an evolving language and that means more sequence types *may* be added in the future. Sequences usually share a similar set of operations. Mutable sequences like `list` and `string` can use + operator to the same effect.

For example the concatenation operator + can be used on two sequences to combine them. Or you can repeat the content of a sequence by a number of times:

```
ha = "ha"
print(ha * 3)

checkered = ("black", "white")
print(checkered * 2)

zebra = [0, 1]
print(4 * zebra)
```

```
>>>
hahaha
('black', 'white', 'black', 'white')
[0, 1, 0, 1, 0, 1, 0, 1]
```

Slicing sequences with `[start:stop:step]`

Sequences can be sliced using the built-in `[start:stop:step]` operator where *start* and *stop* indicate *starting* and *ending* index of the span in the sequence.

The value at the index indicated by *stop* will not be included in the result.

start, *stop* and *step* are all optional arguments. Using `[:]` operator without parameters returns a copy of the entire sequence:

```
tup = ('cat', 'dog', 'spider')

print(tup[:])
print(tup[::])
```

```
>>>
('cat', 'dog', 'spider')
('cat', 'dog', 'spider')
```

Because all arguments are optional, using either `[:]` or `[::]` will produce the same result using default arguments. Default arguments are:

```
[0, len(sequence), 1]
```

Note the *step* argument cannot be 0. It should be at least 1 or negative.

In previous example we produced a copy of the entire sequence. Most of the time, however, you will use slicing operation to cut out a section from a sequence.

For example, here's how you would extract the word `'slice'` from a longer string:

```
message = "Can you find slice in this sequence?"
```

```
# extract the word 'slice' from the string
cut = message[13:18]

print(cut)

>>>
slice
```

Strings are sequences of characters. The word `'slice'` is located between index 13 and 18. And that's exactly what was returned.

The same `[:]` operation can be applied to other sequence types: lists or tuples. In next example we will extract a series of numbers from a tuple:

```
numbers = (0,1,2,3,4,5,6,7,8,9,10)

# extract (3,4,5,6)
cut = numbers[3:7]

print(cut)

>>>
(3, 4, 5, 6)
```

Note the ending index 7 will not be included in the result. It's only a stop point.

Slicing is not indexing

One common mistake made by beginners is thinking that slicing works in the same way as indexing does. This is not exactly true! For example:

```
seq = [0,1,2,3]
res = seq[0:3]
print(res)
```

```
>>>
[0, 1, 2]
```

Here the last item in the result is 2, but slicing stop is 3. Proper thinking about slicing can be explained by the following diagram:

```
                    +---+---+---+---+---+---+
                    | P | y | t | h | o | n |
                    +---+---+---+---+---+---+
Slice :             0   1   2   3   4   5   6
Index :               0   1   2   3   4   5
```

This way of thinking might help clear some things up when slicing.

This also explains why following slice operation produces an empty list:

```
python = ['P','y','t','h','o','n']

print(python[4:4])
```

```
>>>
[]
```

This is simply because our slice start and stop values are equal.

Slice examples

Here are several examples of the slice operation applied to a sequence.

```
seq[start:stop]      # items from start to stop-1
seq[start:]          # items from start to last item
seq[:stop]           # items from start to stop-1
seq[:]               # copy everything
seq[start:stop:step] # items from start to stop-1, skip steps
seq[-1]              # only last item
```

```
seq[-2:]            # last two items
seq[:-3]            # everything except last 3 items
seq[::-1]           # all items, in reverse
seq[1::-1]          # first two item,s in reverse
seq[:-4:-1]         # last three items, in reverse
seq[-4::-1]         # everything except last 3 items, in reverse
seq[None:None]      # everything
```

Other use cases

The slice operation can be used in several ways. The following examples will demonstrate some of those cases.

You can apply it directly to a sequence in a statement:

```
print("Hello"[1:3])
print(("a", "b", "c")[1:2])
print((1, 2, 3, 4, 5)[2:5])
```

```
>>>
el
('b',)
(3, 4, 5)
```

You can use negative indexes to slice backwards:

```
tup = ('cat', 'dog', 'spider')

print(tup[-1:])     # ('spider',)
print(tup[-2:])     # ('dog', 'spider')
print(tup[-3:])     # ('cat', 'dog', 'spider')
```

```
>>>
('spider',)
('dog', 'spider')
('cat', 'dog', 'spider')
```

Recall that all parameters of the slice operation are optional. This is why it's also possible to only specify the ending index. Using negative values here will slice sequence from the other end:

```
print(tup[:-1])     # ('cat', 'dog')
print(tup[:-2])     # ('cat',)
print(tup[:-3])     # ()
```

```
>>>
('cat', 'dog')
('cat',)
()
```

Third argument

You can use the third *step* argument to jump over entries.

In this example we will create a tuple `zebra` with repeating colors. By using *step* argument we can "step over them" to collect only the colors we want:

```
zebra = ('black', 'white', 'black', 'white')

black = zebra[::2]
white = zebra[1::2]

print(black)
print(white)
```

```
>>>
('black', 'black')
('white', 'white')
```

This is similar to using a for loop and incrementing index by 2 instead of 1.

Reversing a sequence

Negative *step* can be used to reverse a tuple (or any sequence):

```
forwards = [1,2,3,4,5]
backwards = forwards[::-1]
print(backwards)
```

```
>>>
[5, 4, 3, 2, 1]
```

Or a string (string is a sequence):

```
print("Hello"[::-1])
```

```
>>>
olleH
```

Replacing items

Sometimes you will want to simultaneously replace a sliced set of items with other values. To do that you can use `[::]` together with equality operator to provide a set of replacement values:

```
nums = [1, 2, 3]
nums[0:2] = ('a', 'b')
print(nums)
```

```
>>>
['a', 'b', 3]
```

This operation can be interpreted as "Replace the items between index 0 and 2 with values ('a', 'b') and return result as a copy of original list.

Replacing with step

We already used step argument in previous examples. It works in a similar way when using together with replacement values (you can use tuples or lists.)

This feature is great for mapping values to a larger set that follows a pattern.

```
ten = [0,0,0,0,0,0,0,0,0,0]
ten[::2] = [1,1,1,1,1]
print(ten)

ten = [0,0,0,0,0,0,0,0,0,0]
ten[1:6:2] = (3,3,3)
print(ten)

ten = [0,0,0,0,0,0,0,0,0,0]
ten[1:10:2] = (1,2,3,4,5)
print(ten)

>>>
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
[0, 3, 0, 3, 0, 3, 0, 0, 0, 0]
[0, 1, 0, 2, 0, 3, 0, 4, 0, 5]
```

Python will automatically expand the list if the number of items on the replacement list exceeds the length of the slice:

```
nums = [1,2,3]
nums[0:2] = ('a','b','c','d','e','f')
print(nums)

>>>
['a', 'b', 'c', 'd', 'e', 'f', 3]
```

Important: the set of assigned values must be *no less* in length than the length of the slice. Not following this rule will produce an error.

Using del and slices to remove items

Slicing can be used together with `del` keyword to delete items:

```
items = [0,1,2,3,4,5]
del items[1:5]

print(items)
```

Items 1, 2, 3 and 4 were removed directly from the original list.

6.9 set

A `set` is an *unordered*, *immutable* collection of unique elements – it cannot contain duplicate values. Sets are not sequences, because they are unordered.

Like tuples, sets are immutable – their values cannot be modified once defined.

The difference is that sets cannot contain duplicate values. Each value in a set is unique and appears only once even if duplicates were provided in the assignment:

```
s = {3, 3, 3, 5}
print(s)
```

```
>>>
{3, 5}
```

Duplicate values were automatically removed. One use case could be storing email lists when you need to make sure that no email appears on the list more than once.

The set data structure provides a number of methods you can and should use to manipulate sets (but not in a way that changes its values). These methods will be explored in the following section of this book.

set.add()

Use `add(`*`element`*`)` method to add a new item to the set:

```
followers = "@pythonista"
followers.add("@pythonic129")

print(followers)
```

```
>>>
'@pythonista', '@pythonic129'
```

If the element already exists, nothing happens.

set.discard()

Discard item from set by value.

```
followers = "@pythonista", "@javascript", "@pythonic129"
followers.discard("@javascript")

print(followers)
```

```
>>>
'@pythonista', '@pythonic129'
```

set.isdisjoint()

This method returns `True` if none of the items are present in either set:

```
users1 = "@christine", "@stanley", "@luna"
users2 = "@david", "@grace", "@john"
users3 = "@roger", "@christopher", "@grace"

a = users1.isdisjoint(users2)
b = users2.isdisjoint(users3)

print(a)
print(b)
```

```
>>>
True
False
```

First result is `True` because none of the users in `user1` variable match `user2`.

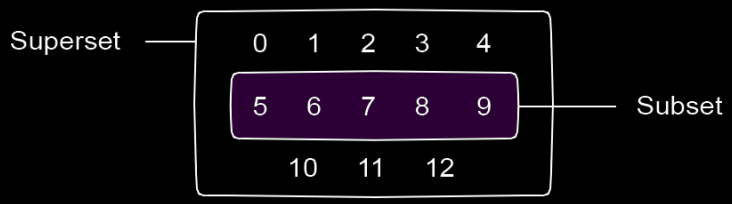
Second result is `False` because `@grace` was found in both sets.

set.issubset()

Return `True` if one set is a subset of another set. A subset is a set all of whose values match

```
ids1 = {5,6,7,8,9}
ids2 = {1,2,3,4,5,6,7,8,9,10,11,12}

if ids1.issubset(ids2):
    print("ids1 is a subset of ids2")
else:
    print("ids1 is not a subset of ids2")

if ids2.issubset(ids1):
    print("ids2 is subset of ids1")
else:
    print("ids2 is not a subset of ids1")

>>>
ids1 is a subset of ids2
ids2 is not a subset of ids1
```

The number of matching items in subset must be equal or less than number of items in superset. A subset cannot contain more items than the superset, or contain any different values from superset.



Here's one way of how it could be visualized.

set.clear()

To remove all items from the set:

```
followers = "@pythonista", "@pythonic129"
followers.clear()

print(followers)

>>>
set()
```

set.copy()

To create another copy of the same set:

```
followers = "@pythonista", "@pythonic129"
copy = followers.copy()

print(copy)

>>>
'@pythonista', '@pythonic129'
```

set.update()

Sets have an explicit `update()` method which will merge the set with values from another set:

```
e = {0,1,2}
f = {3,3,3,4,5}
f.update(e)
print(e)
print(f)
```

```
>>>
{0, 1, 2}
{0, 1, 2, 3, 4, 5}
```

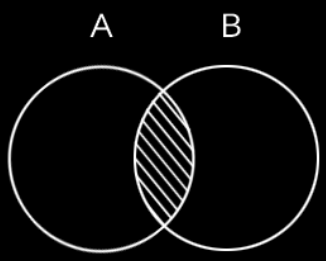
Because sets cannot contain duplicate values, the set `f` was limited to values {3,4,5} and then it was added to the original set `e`.

As you can see it would be a mistake to think of sets as strictly immutable. But you can think of sets as being explicitly mutable. You have to use a specialized `update()` method to update its values.

This prevents accidental updates with common operators but still gives the programmer explicit ability to modify sets as necessary.

set.intersection()

You can compare two sets with `intersection()` method. Visually it might look something like this. The method returns items that match in two different sets:



Here is an example:

```
A = {1,2,3}
B = {5,6,2}
C = A.intersection(B)

print(C)
```

```
>>>
{2}
```

One practical use case of this operation could be comparing social media followers between two users:

```
# list of followers followed by user 1
userlist1 = {"@pythonic129", "@jsdev", "@michael"}

# list of followers followed by user 2
userlist2 = {"@dave01, "@pythonic129", "@luna"}

shared = userlist1.intersection(userlist2)

print(shared)
```

```
>>>
{'@pythonic129'}
```

Two different users share the same follower: `@pythonic129`

set.intersection_update()

The `intersection_update()` method is different from `intersection()` because instead of returning a new set with matching values, it removes unwanted items from the original set. Otherwise, it performs exactly the same operation.

```
A = {1,2,3}
B = {5,6,2}

C = A.intersection_update(B)

print(C)    # None
print(A)
```

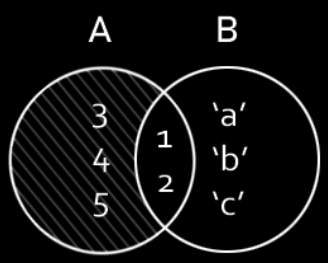
```
>>>
None
{2}
```

Notice that original set A was actually changed, or updated – all non-matching items were removed.

The function `intersection_update()` has no return value and simply returns `None`, not a new set like `intersection()`.

set.difference()

The `difference()` method can be visualized as follows:



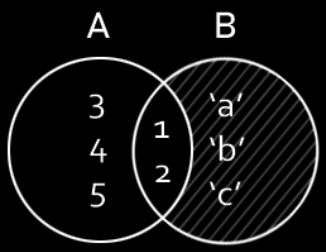
Only items {3, 4, 5} are returned. The items 'a', 'b' and 'c' don't appear in set A at all, so they will also be ignored.

```
A = {1, 2, 3, 4, 5}
B = {1, 2, 'a', 'b', 'c'}

result = A.difference(B)

print( result )
```

```
>>>
{3, 4, 5}
```

Note that in this case items in the sets are not combined but rather excluded. The second set acts as *exclusion filter* for the values in first set.

This operation can be reversed when called on set B:




```
A = {1, 2, 3, 4, 5}
B = {1, 2, 'a', 'b', 'c'}

result = B.difference(C)

print( result )

>>>
{'b', 'a', 'c'}
```

Note that because sets don't guarantee order of items, the items in the return value appear to be shuffled, which shouldn't matter when working with sets.

How can `difference()` be used in a real life situation? One possible practical example of this would be to exclude blocked users from your social media timeline:

```
followers = {"@pythonista", "@pythonic129", "@jsdev"}
blocked = {"@jsdev", "@painter1"}

timeline = followers.difference(blocked)

print(timeline)
```

```
>>>
{'@pythonic129', '@pythonista'}
```

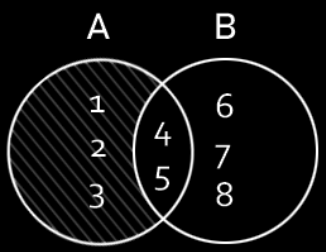
User `@jsdev` was excluded from the timeline.

set.difference_update()

Identify difference between two sets and remove matching items from the set on which the method was called:

```
s1 = {1, 2, 3, 4, 5}
s2 = {4, 5, 6, 7, 8}

res = s1.difference_update(s2)
print(s1)
print(s2)
```

```
>>>
{1, 2, 3}
{4, 5, 6, 7, 8}
```

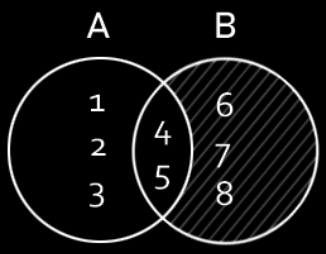


Values 4 and 5 were removed from first set.

You can reverse this operation by calling the method on second set:

```
s1 = {1, 2, 3, 4, 5}
s2 = {4, 5, 6, 7, 8}
```

```
res = s2.difference_update(s1)

print(s1)
print(s2)

>>>
{1, 2, 3, 4, 5}
{6, 7, 8}
```



In this case matching values 4 and 5 were removed from second set.

There is one more thing! Sets cannot be sliced:

```
s = {1, 2, 3}

print([:])

>>>
Traceback (most recent call last):
  File "example.py", line 32, in <module>
    print(e[:])
TypeError: 'set' object is not subscriptable
```

The `frozenset` data structure (see next) cannot be sliced either.

6.9.1 frozenset

Unlike `set`, none of the items in a `frozenset` can be changed (neither by operators nor methods.) In other words `frozenset` is completely *immutable*.

Let's define a simple frozen set and print it out:

```
fs = frozenset({'a', 'b', 'c'})
print(fs)

>>>
frozenset('c', 'a', 'b')
```

The `frozenset` data structure becomes useful for locking a number of items that belong to a group that should not be modified:

```
vowels = ('a', 'e', 'o', 'i', 'u')
semivowels = ('w', 'y')

freeze = frozenset(vowels)

# print out the frozen set
print(freeze)

# attempt to modify wovels
freeze.add('m')

>>>
frozenset('i', 'a', 'o', 'u', 'e')
Traceback (most recent call last):
  File "example.py", line 19, in <module>
    freeze.add('m')
AttributeError: 'frozenset' object has no attribute 'add'
```

There are only 5 wovels in English – with 'y' and 'w' letters used as semi-vowels. The frozen set is complete. We can be sure that no matter what's added to it cannot be a vowel. For this reason `frozenset` doesn't allow `add()` method.

Trying to use a method from `set` structure that modifies an item will fail, usually because frozenset does not provide it.

Apart from that detail `set` and `frozenset` are the same.

Variable order of items in set and frozenset

Neither `set` nor `frozenset` guarantee that the items will appear in the same order in which they were defined when trying to print or iterate them.

Sets are used for solving the type of problems where it isn't required for each value to appear in linear order on the list.

Every time you output a frozen set you may notice items are rearranged again:

```
print(frozenset({'a', 'b', 'c'}))
```

```
>>>
frozenset({'a', 'c', 'b'})

>>>
frozenset({'c', 'a', 'b'})

>>>
frozenset({'b', 'c', 'a'})
```

A `set` (or `frozenset`) doesn't imply linear index order like lists.

The same will happen if you try to iterate it with a for-loop:

```
for item in fs: print(item)
```

```
>>>
c
a
b
```

Almost every other time the items will appear in a different order.

This is completely normal behavior for a set. They simply don't rely on indexing.

The key difference between set and frozenset

Contrary to `set`, `frozenset` does not have an `update()` method. Once defined, a frozen set is completely immutable. Let's take a look!

```
fs = frozenset('a', 'b', 'c')
fs.update('d')
print(fs)
```

```
>>>
Traceback (most recent call last):
  File "example.py", line 38, in <module>
    fs.update('d')
AttributeError: 'frozenset' object has no attribute 'update'
```

Attempting to update a frozen set results in `AttributeError`.

At first it might not make any sense as to why you would want to use a data structure which does not allow changing its items.

Chapter 7

Functions

In previous examples we used the built-in function `print` (and many others!) The print function is part of Python's specification. It's available to use out of the box.

But you can also create your own functions. In this chapter we'll explore the basic anatomy of Pythonic functions.

In Python functions are defined using `def` keyword.

```
def message():
    print("My message.")
```

Python allows defining functions on a single-line without indents:

```
def message(): print("My message.")
```

If the function does not require any arguments you can execute it by typing its name with empty parenthesis at the end:

```
message()
```

```
>>>
My message.
```

You can make a function execute as many statements as you want:

```
# Define a function that executes multiple statements:
def messages():
    print("My message.")
    print("My other message.")
    print("Something else.")

# Call this function:
messages()
```

```
>>>
My message.
My other message.
Something else.
```

7.1 Using `return` keyword to return values

Functions help us print messages, execute a number of any Python statements in a row or do calculations in isolation. But they can also *return a value* or multiple values that can be later stored in a variable or printed.

To return a value use the `return` statement and the value:

```
def thousand():
    return 1000

# Execute the function
# and assign its return value to variable num:
num = thousand()

print(num) # prints 1000
```

Here we used the `return` keyword to return a value from the function and assign it to a new variable. We then printed it out using `print` function.

7.2 Function Parameters and Arguments

One property of a function is the ability to receive information passed to it as parameters. Function *parameters* are optional but if present, they become part of the function's definition. Parameters are defined as a list of variable names.

Arguments are the real values passed to the function that will be assigned to its parameter names. Inside scope of the function you will use its parameter names to refer to the values that were passed into the function.

Python has 4 different ways in which it allows specifying function parameters: *positional*, *default*, *keyword* and *arbitrary*.

7.2.1 Positional arguments

In this section we'll explore *positional arguments*. Positional arguments define the *required* arguments a function expects to receive. Not providing them when calling the function will result in an error.

Let's create a function called `add` that takes two arguments: `a` and `b`.

It can be defined as follows:

```
def add(a, b):
    return a + b
```

Here the a and b variable names are the *parameters* of the function.

This function takes two numeric values and returns their sum. Let's execute it two times in a row by passing two different sets of *arguments*:

```
r = add(10, 5) # 10 and 5 are the function's arguments
s = add(12, 8) # 12 and 8 are the function's arguments

print(r) # Prints 15
print(s) # Prints 20
```

The values 10, 5 and 12, 8 are the arguments passed to the function.

7.2.2 Default arguments

You can define *default arguments* in function's parameter list. Default arguments will be used as a fall back if no other values are provided:

```
def say(name = "John", msg = "Hello"):
    print(name, ":", msg)
```

Let's call this function to see what happens if we skip all arguments:

```
say()
```

```
>>>
John : Hello
```

The function falls back on default values. You can also pass partial arguments:

```
say("Steve")
```

Outputs:

```
Steve : Hello
```

Or you can overwrite both arguments:

```
say("Steve", "Goodbye")
```

Outputs:

```
Steve : Goodbye
```

Arguments are usually passed to the function in the same order they appear in the function's parameter list in function's definition. But not always:

In Python you can also use something called *keyword arguments*.

7.2.3 Keyword arguments

As you will shortly see *default arguments* and *keyword arguments* share the same syntax. But the difference is in whether they are assigned in the parameter list itself or passed as arguments.

In previous example we used *default arguments* to define a function. That's when default values are used in case if some or all arguments are missing. But they are still assigned in the same order as they appear on the parameter list.

In contrast to *default arguments* – which simply *define* default values for any or all missing arguments – *keyword arguments* allow you to *change the order* in which values are *passed* to the function:

```
def say(name, message):
    print(f"{name} : {message}")

say(message = "Hello", name = "John")
```

We redefined the order in which arguments were passed. Here's the output:

```
John : Hello
```

As you can see the arguments were passed in an alternative order but the function still produced proper output.

7.2.4 Mixing keyword and default arguments

It's possible to mix default and keyword arguments together. The only proper way of doing that is to prioritize non-keyword parameters in the function definition:

```
def say(name, message = "Greetings"):
    print(f"{name} : {message}")
```

Here positional non-keyword parameter `name` is listed first without a default value.

This is acceptable and it works as expected.

```
say(name = "Steve")
```

After setting `name` to `"Steve"` parameter `message` is automatically assigned value of `"Greetings"` by default:

```
>>>
Steve : Greetings
```

But you may run into two issues with partial definitions.

First, by simply defining a default parameter before positional one:

```
def say(name = "John", message):
    print(f"{name} : {message}")
```

This function definition will produce the following error:

```
SyntaxError: non-default argument follows default argument
```

But why using positional arguments *after* default arguments is not allowed?

```
def say(name = "John", message):
    print(f"{name} : {message}")

# Overwrite default argument?
# Or assign value to  message?
say("Steve")
```

This would be simply too confusing for the compiler. After all, what does it really mean? Should the first argument be replaced by the default argument? Or is the user of this function trying to pass second parameter?

7.2.5 Arbitrary arguments

Arbitrary parameters are the assumed parameters of a function. They are not all explicitly defined within the function definition.

Arbitrary parameters are defined by using the star * (asterisk) character.

```
def list(*arguments):
    for value in arguments:
        print(value)
```

Nothing happens if none of the arguments are passed:

```
list()
```

```
>>>
()
```

An empty tuple is the result.

Now let's pass some common animal names:

```
list("Fox",
     "Wolf",
     "Coyote",
     "Bear",
     "Rabbit")
```

```
>>>
Fox
Wolf
Coyote
Bear
Rabbit
```

The `arguments` parameter is a tuple.

Because tuples are iterable you can use a for loop here.

7.2.6 Built-in functions

Python has a number of built-in functions. For example the function `sum` does roughly the same thing our custom function `add` does from previous example.

```
a = sum([10, 5])
b = sum([12, 8])

print(a) # Prints 15
print(b) # Prints 20
```

```
>>>
15
20
```

You can quickly sum up a range:

```
r = range(0, 5)
a = sum(r)

print(a) # Prints 10
```

The last example produced `10` after adding numbers `0 1 2 3 4` generated by `range(0, 5)` function.

7.2.7 Overwriting built-in functions

Python won't generate an error if you create a custom function with the name `sum`. However this means you won't be able to use the built-in function `sum` in your program. Watch out for empty function definitions:

```
def a():
```

This function definition will produce an error. Functions require at least one statement. You will probably never have to create or execute an empty function. But just to be complete here is a proper syntax for creating an empty function:

```
def b(): return
def c(): pass

b() # nothing happened
c() # nothing happened
```

Use either `return` without value or `pass` keywords.

7.2.8 Practical example

Previous examples were intentionally basic in order to demonstrate how functions are defined and executed. Let's try to build something practical.

What are we calculating?

You often see social media apps display time in "elapsed" format. This is done by calculating the difference between two numeric time stamp values in seconds: the one from when the event occurred and the current time now.

The interval between the two dates is then converted to something like "32s" for 32 seconds, "1h" for 1 hour, "2d" for days or "3mo" for 3 months. Alternatively, in cases where interval is too long, simply display a date. For example it could be: `Wed Nov 18 2020` or `Jan 1 2021`.

In this section we will take a look at one way of creating this function.

Preparing the data

Before writing the actual function let's define some helper data sets.

We know that a minute is 60 seconds, an hour is 3600 seconds and so on. Let's create a table to store our intervals in one variable:

```
# create a table measuring key intervals in seconds
table = [1,             # seconds (1 to avoid division by 0)
         60,            # 1 minute
         3600,          # 1 hour
         3600*24,       # 1 day
         3600*24*30,    # 1 month
         3600*24*30*12] # 1 year
```

Each value represents a "checkpoint" to test the time interval (in seconds) against. Let's also create a list of endings in string format:

```
endings = ["s", "m", "h", "d", "mo", "y"]
```

Writing `seconds2elapsed` function

This function converts a time interval in seconds stored in `t` parameter and returns a value that describes that period of time in human-readable form.

```
import time

def seconds2elapsed(t):
  difference = abs(time.time() - t)

  # convert to integer (truncate decimal point)
  difference = int(difference)

  if (difference == 0)
    return "now"

  i = 5

  # iterate over table backwards
  for interval in reversed(table):

    # if difference is greater
```

```
    if difference > interval:
      t = table[i]
      v = (difference - (difference % t)) / t

      # early exit from function
      return f"{int(v)}{endings[i]}"

  # python's for-in loops don't have indexing
  # so we will have to implement our own
  i -= 1
```

The **time.time()** method returns time now in seconds with a decimal point. In order to start using it you must include **time** package at the top of your program.

First we calculate the difference between time now and time in seconds passed to the function in the `t` parameter. Using `abs` function makes sure the interval between two dates is never negative.

The core functionality is in the for loop.

This for loop implements **reversed** method – it flips the order of all items on the **table** list. We could have easily reworked our table variable to simply start with the year first. But here **reversed** was used to demonstrate one possible way of iterating in reverse which wouldn't always be obvious to someone new to the language.

We start with greatest interval which is equal to 1 year and compare how many times 1 year can fit into the value stored in `difference` variable calculated in first step. If value is less than 1 year it then checks for months, etc.

The following line of code does all the number crunching:

```
v = difference - (difference % t) / t
```

The modulo operator (%) returns the remainder of a division operation. This helps us perform a clean division by `t` which as you remember refers to the amount of 1 year, 1 month, 1 day, 1 hour or 1 minute depending on table index.

Remember that `difference` variable is stored in seconds. But we want 61 seconds to evaluate to 1 minute (or `"1m"`.) That's what subtracting from the modulo operation will help us determine - how many times a value fits into the interval.

The logic goes something like this: if the value in `difference` variable is greater than 1 year, make proper divisions to calculate 1 in the "1y" string that will be returned. If the interval is larger than 2 years it will be calculated as "2y".

If difference is lesser than a year, check if it can be represented by months. If difference is shorter than at least one month it will be returned in hours.

If smaller than hours return in minutes. If it's equal to or less than 60 seconds, return value in seconds: "37s" for example. If value is 0, return it as `"now"`.

Using `seconds2elapsed` Function

To see the function in action we need to find a way to call it every few seconds.

There is more than one way of executing a function at a time interval. For the sake of simplicity this section will demonstrate how to do it with **threading** package.

The **threading** package has **Timer** property. It works similar to **setInterval** in JavaScript if you're familiar with that function.

The **Timer** object has **start()** method that executes the function after a period of time specified in seconds. In this case it was set to `3.0` seconds.

Note: This example assumes that **seconds2elapsed** function we wrote earlier is also included at some point after importing **time** and **threading** packages.

```
import time
import threading
# === add definition of seconds2elapsed() function here ===
since = time.time()
def every3seconds():
  # execute this function every 3.0 seconds
  threading.Timer(3.0, every3seconds).start()
  print(f"{seconds2elapsed(since)}")
```

```
# trigger the timer
every3seconds()
```

Let's run this script to see what happens:

```
>>>
3s
6s
9s
12s
15s
```

First we generate a value representing the time the program started and store it in variable **since**. Threading package **Timer.start()** function will execute our function **every3seconds**.

The interval is measured against our time table and outputs a string that tells us how much time elapsed since the program start every 3 seconds.

Chapter 8

Built-in Functions

Python mastery depends on understanding the built-in functions.

8.0.1 User input

In Python you can use `input()` function to take user input directly from the command line. To demonstrate how it works at its basic level let's use `input()` to query for a value in combination with *f*-strings to print out entered values.

The following script will create a question prompt asking to enter your name:

```
input("Enter your name: ")
```

You'll be greeted with the prompt message and blinking cursor:

```
>>>
Enter your name: _
```

The input function returns the entered value in string format. This means you can assign that entered value to a variable:

```
first = input("Enter your first name: ")
last = input("Enter your last name: ")
```

```
print(f"You entered: {first} {last}")
```

```
>>>
Enter your first name: Stanley
Enter your last name: Kubrick
You entered: Stanley Kubrick
```

If we verify the type of entered value we'll see that `input` function always returns the entered value in string format, even if a numeric value was entered:

```
value = input("Enter first value: ")
print(f"You entered {value} and its type is {type(value)}")
value = input("Enter second value: ")
print(f"You entered {value} and its type is {type(value)}")
```

```
>>>
Enter first value: 5
You entered 5 and its type is <class 'str'>
Enter second value: hello
You entered hello and its type is <class 'str'>
```

We just used the Python's built-in `type()` function to find out the type of value that was entered. Looks like they are both strings. But what if we want numbers?

To extract a numeric value from input you can cast it to `int` type. Python offers a function for constructing each type: `bool()`, `int()`, `str()`, `dict()`, `float()`, `tuple()`, `list()`, `set()`, `frozenset()` and so on.

```
grains = 1000
print(f"You have {grains} grains of wheat.")

add = input("Enter number of grains to add: ")

# cast the value to int type and then add up the result
grains = grains + int(add)
```

```
# print final value
print(f"You now have {grains} pounds of wheat.")
```

```
>>>
You have 1000 grains of wheat.
Enter number of grains to add: 145
```

Note that by itself the value of 5 is considered to be an integer:

```
digit = 5
print(type(digit))
```

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

In order to change one type to another you have to use the built-in function that matches the name of the data type:

```
# cast digit 5 to string format
string = str(digit)
print(f"value string is of type {type(string)}")

# cast digit 5 to floating point format:
fl = float(digit)
print(f"value fl is of type {type(fl)}")
```

```
>>>
value 5 is of type <class 'str'>
value 5.0 is of type <class 'float'>
```

But why would we want to convert one type of a value to another if it was already defined by using the type of our choice?

If casting doesn't make sense yet, don't worry – some of the things will start to become more obvious with practice. By design it is convenient for `input()` function to produce values in string format.

This process of converting one type of a value to another type is called *type-casting*.

The `input` function automatically converts (or "casts") user input from the prompt into string format, even if the value that was typed was an integer or a floating point number.

A similar problem arises if we wanted to treat user input as integer to perform a mathematical operation on two entered values:

```
one = input("Enter 1st numeric value: ")
two = input("Enter 2nd numeric value: ")
res = one + two
print(res)
```

```
>>>
Enter 1st numeric value: 100
Enter 2nd numeric value: 34
10034
```

Because `input()` automatically converted both values to strings the + operator added up two strings instead of treating them as a numeric operation.

The result is concatenation of two strings `"100"` and `"34"` into one: `"10034"`

In Python the + operator will add two strings together if values on both of sides are of type string. If both values are numbers, it will perform mathematical addition.

If we want the addition of two integers to take place we first need to cast both values to `int` type:

```
one = input("Enter 1st numeric value: ")
two = input("Enter 2nd numeric value: ")
res = int(one) + int(two)
print(res)
```

```
>>>
Enter 1st numeric value: 100
Enter 2nd numeric value: 34
134
```

Now the addition between numbers 100 and 34 were properly calculated.

Adding a string to an integer produce TypeError error:

```
print("username" + 221837600)
```

```
>>>
Traceback (most recent call last):
File "input.py", line 1, in <module>
print("username" + 221837600)
TypeError: can only concatenate str (not "int") to str
```

Likewise, to produce a username in `string` format you could cast integer to string with the built-in `str()` method:

```
print("username" + str(221837600))
```

```
>>>
username221837600
```

String (`str`) and Integer (`int`) are two of the most basic data types in Python but there are a few others.

We'll explore Python data types in one of the following sections. For now just know that numbers like 10 are treated like integers or `"int"` data type and text strings such as `"text"` are considered to be values of type string or `"str"`.

In the following sections we will go through some of the most important built-in functions and demonstrate simple examples of how they work.

eval()

The `eval` function is used to evaluate a Python statement in string format:

```
x = 5
expression = eval("x + 1")
```

```
print(expression)

>>>
6
```

Note, variables already defined in global scope will transition into eval function.

The eval function has two more arguments: globals and locals. They mirror Python's built-in functions globals() and locals()

At first it might not be obvious as to why you would use a function to evaluate a Python statement inside a Python program itself. As you continue writing Python code, you will eventually come across a case where you might need to do this.

Be careful of using eval function with user input. Since it can execute any function by following a statement written out in string format, the user can type literally any Python command, including file opening or writing operations on your platform.

print()

We've already discussed print function in many examples in this book. It's simply here to help us print messages to the console:

```
if type("message") == 'str'
    print("This is a string!")

>>>
This is a string!
```

callable()

Determine if an object is a function or a class (both are callable):

```
# Define function a
def a(): pass
```

```
# Define class b
class b: pass

print(callable(a))
print(callable(b))
```

```
>>>
True
True
```

Both function `a` and class `b` are callable.

open()

This function opens a file.

It's described in much more detail in **File System** chapter.

len()

Get length of a sequence (string, list, tuple, etc.)

```
print(len("python"))
print(len(["h", "e", "l", "l", "o"]))
print(len(1,2,3))
print(len((1,2)))
```

```
>>>
6
5
3
2
```

slice()

When used with one argument the slice function will slice off a chunk of a list at a specified slice position.

Create a slice object and use it to slice a tuple:

```
numbers = (1,2,3,4,5,6,7,8,9,10)

# create slicing object
knife = slice(5)

# slice the number at knife
print(numbers[knife])
```

`slice()` is similar to *slice operator* that was explained in-depth in section *Slicing sequences with [start:stop:step]* in *Chapter 6: Data Dypes and Structures*

The slice function has 3 arguments:

`slice(`*`start, end, step`*`)`

To slice out a segment from a list you can use `start` and `end` arguments:

```
numbers = (1,2,3,4,5,6,7,8,9,10)

# create slicing object
knife = slice(5)

# slice the number at knife
print(numbers[knife])
```

```
>>>
(1, 2, 3, 4, 5)
```

The third argument *`step`* will "step" over items at an interval. By default it's set to 1. Setting range to 0, 8 and step to 2 we can collect only the first four 1's from the tuple containing only 1's and 0's:

```
numbers = (1,0,1,0,1,0,1,0,1,0,1)

# create slicing object
knife = slice(0, 8, 2)

# slice the number at knife
print(numbers[knife])

>>>
(1, 1, 1, 1)
```

abs()

Produce an absolute number (converts negative numbers to positive)

```
a = -1640
b = 1640

print(abs(a))
print(abs(b))
print(abs(-403))

>>>
1634
1634
403
```

chr()

Get character that represents a value in unicode format:

```
char = chr(120)

print(char)
```

```
>>>
x
```

id()

Convert address of a variable to an integer.

hex()

Convert a number to hexadecimal format.

bool()

Constructor function for values of type `boolean` (True or False)

int()

Constructor function for values of type `int` (integer)

complex()

Constructor function for values of type `complex` (complex number)

str()

Constructor function for values of type `int` (integer)

dict()

Constructor function for item of type `dict` (dictionary)

float()

Constructor function for values of type `float` (decimal point number)

tuple()

Constructor function for values of type `tuple`

—

There are many other built-in functions in Python but they won't be listed here to save space for practical examples. They are easily available for look-up online.

Chapter 9

Overloading Operators

In Python you can overload common operators: +, -, *, / and few others.

Overloading an operator replaces its function. Overloaded operators are added to your custom classes. They cannot be used with regular types like *number* or *string*. Those objects already have the + operator overloaded natively.

```
print(1 + 1)
```

```
>>> 2
```

But...

```
class Word:
    def __init__(self, text):
        self.text = text

hello = Word()
there = Word()

hello + there # TypeError
```

```
>>>
TypeError: unsupported operand type(s) for +: 'Word' and 'Word'
```

Without overloading + operator trying to add two objects doesn't make any sense. What does it mean to add two objects of type `Word`? And what should happen?

If only this code worked, it would be a beautiful way for solving *abstract* problems with code. When adding two objects of type `Word` it seems natural that another object of type Word should be returned containing both strings concatenated.

In the following section we will overload Python's built-in *magic methods* in order to add this new functionality to all objects of type `Word`.

9.1 Magic Methods

Python uses *magic methods* to overload operators. The `__add__` magic method will overload `+` sign so that instead of adding numbers like `1 + 1` you will be able to add objects of a particular type. You can use `__mul__` and `__div__` objects to override `*` and division `\`operators respectively.

Let's overload magic method `__add__` of our simple `Word` class from the previous section. To keep things simple in this example we will only overload + operator:

```
class Word:
  def __init__(self, text):
    self.text = text
    print("Initialized word", self.text)

  # Overload + operator
  def __add__(self, other):
    return self.text + " " + other.text
```

We've just overloaded + operator for all object instances of type `Word`. When two objects of this type are added using + a string will be returned.

For each operator you overload you have to supply two arguments to the *magic method*: `self` and `other` where parameter `self` links to the object on left hand side of the operator and `other` is a link to object to the right of the operator.

Let's see our overloaded operator in action:

```
hello = Word("hello")
there = Word("there")

print(hello + there)
```

Instead of `TypeError` we will now see the following result in command prompt:

```
hello there
```

This is how operator overloading works at its basic. But this doesn't really explain it in context of a practical case. In the next section we will write a simple 3D vector class, often used in 3D vector math libraries for crunching 3D geometry.

9.1.1 3D Vector Class

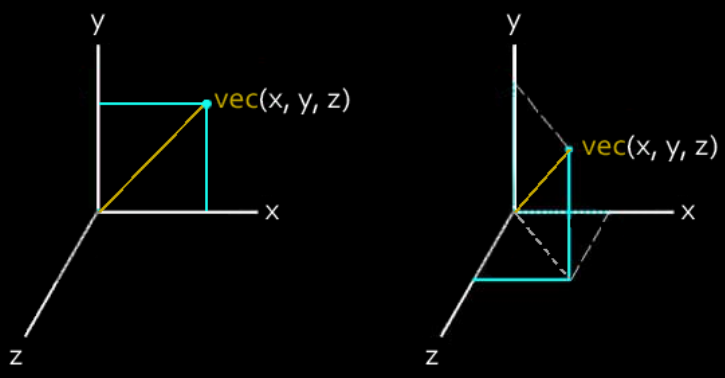
This abstract vector object can be used to *add* (+), *subtract* (−) or multiply (∗) 3D vector objects. The < and > symbols are overloaded to compare vector length.

The `length` method will calculate 3D vector's length. This is done by getting the square root of the sum of each vector's coordinates on each axis multiplied by itself (see the formula in body of the `def length(self)` function.)

Because we need `math.sqrt` function, don't forget to first import `math` module:

```
import math
```

A 3D vector can be represented using this Cartesian-like coordinate system in 3D space where positive `z` axis extends toward the viewer.



A 3D vector is defined by 3 points in space: `x`, `y` and `z` which are assumed to extend from the origin of the coordinate system at `[x=0, y=0, z=0]`. Therefore it can be represented by the following Python class:

```
class vector:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
        self.len = self.length()
```

Unlike JavaScript, in Python you *cannot* have class attribute and method share the same name. You cannot have an attribute called `length` and a method called `length`. Your code will still compile but one will overwrite the other without displaying any errors.

For this reason in this example attribute `len` will store length of the vector. And `length()` is the function that calculates it.

Python object constructor `__init__(self)` allows calling functions found later in the definition of that class. You can say Python's class methods are *hoisted* to the constructor which is how it also works in JavaScript and many other languages.

Our constructor calls `self.length()` method defined later in the class. Here is the definition:

```
    def length(self):
        self.length = math.sqrt(self.x * self.x +
                                self.y * self.y +
                                self.z * self.z)
        return self.length
```

This function uses `math.sqrt` to calculate the length of the vector given 3 coordinates on `x`, `y` and `z` axis.

And finally, let's overload several *magic methods*:

```
    def __add__(self, other):
        self.x += other.x
        self.y += other.y
        self.z += other.z
        return self
    def __sub__(self, other):
        self.x -= other.x
        self.y -= other.y
        self.z -= other.z
```

```
        return self
    def __mul__(self, other):
        self.x *= other.x
        self.y *= other.y
        self.z *= other.z
        return self
    def __div__(self, other):
        self.x /= other.x
        self.y /= other.y
        self.z /= other.z
        return self
    def __lt__(self, other):
        #print self.length()
        #print other.length()
        return self.length() < other.length()
    def __gt__(self, other):
        return self.length() > other.length()
    def __str__(self):
        return "[,,]".format(self.x, self.y, self.z)
```

Let's use this class to create two vectors:

```
vec_1 = vector(5, 3, 1)
vec_2 = vector(3, 4, 2)
```

When we pass vec_1 or vec_2 to print function __str__ method is activated:

```
print(vec_1)        # [5,3,1]
print(vec_1.length) # 5.9
print(vec_2)        # [3,4,2]
print(vec_2.length) # 5.3
```

And finally, let's see our overloaded operators in action:

We have just created a vector class that allows us to easily add, subtract, multiply, divide and compare 3D vectors by length. You can use the same logic to overload objects of any type.

Chapter 10

Classes

The Sun is a type of a star. Earth is a type of a planet. A car is a type of a vehicle. You can create your own custom classes to represent all kinds of objects.

A class is an abstract representation of an object. Classes define types of data. For example a class can represent anything from a number, a text string or something more complex like an abstract model of a car, train, airplane or an animal.

A class has a constructor function responsible for building an instance of the object of type it represents. A class can also have a number of *attributes* (known as *properties* in some other languages) and *method* definitions.

Let's create a class `Car` with one attribute and one method:

```
class Car:
    top_speed = "180mph"
    def describe():
        print("I am method of a Car class")

print(Car.top_speed) # prints "180mph"
Car.describe()       # prints "I am method of a Car class"
```

10.0.1 Class Constructors

There is nothing special about empty classes. Classes are mostly used as blueprints for creating objects of type they represent. The most common use of a class is to *instantiate* an object – *or many objects* – of that type.

A **constructor** is a function that gets called when you instantiate an object using that class – without it an object cannot be instantiated.

In Python, object construction happens in two stages. Therefore there are two *magic methods* responsible for construction of an object: `__new__` and `__init__`. You can use either one of them. Or you can use both.

The following examples will demonstrate how it all fits together.

```
class Car:
    def __init__(self):
        print("Hello from __init__")
```

Defining two classes with the same name overwrites the previous one:

```
class Car:
    def __init__(self):
        print("Hello from __init__")

class Car:
    def __init__(self):
        print("Hello from overwritten class Car")
```

No redefinition error is generated as in some other languages. The second definition simply overwrites the first one and will be used when object is instantiated. f

Instantiating an object

To instantiate an object of type `Car` let's call `Car()` constructor and assign it to a variable name `mazda`:

```
mazda = Car()
```

The variable `mazda` now holds an actual instance of an object of type `Car`. Parameter `self` is automatic reference to object instance being created. If it is printed out, it will display where in memory the object instance is located.

Let's print it out in the following example:

```
class Car:
    def __init__(self):
        print(self)

tesla = Car()
print(tesla)
```

This example also creates an object instance of type `Car`:

```
<__main__.Car instance at 0x0000000002CBF6C8>
```

Object attributes

In the beginning of this chapter we accessed `Car.top_speed` attribute.

You can think of it as a class-wide attribute that belongs to the class itself. Only attributes defined within body of that class inherit this behavior.

Adding attributes to the actual instance of an object is done through `self` property.

In this example constructor assigns two hard-coded values to `make` and `wheel` attributes via `self` attribute that points to the *instance* of the object:

```
class Car:
    def __init__(self):
        self.make = "Tesla"
        self.wheels = 4
        print(self.make, self.wheels, "wheels")

tesla = Car()       # Prints "Tesla 4 wheels" from constructor
```

Even though no arguments were passed to the constructor via `Car()` call the attributes were assigned to object instance from the constructor via `self`.

Making object constructors more flexible

We don't really want to hard-code attribute values in the constructor as in previous example. To make our class more flexible we want to let the user pass them via the constructor itself in order to create different types of cars:

```
class Car:
    def __init__(self, p1, p2):
        self.make = p1
        self.wheels = p2
        print(self.make, self.wheels, "wheels")

volvo = Car("Volvo", 18)      # prints "Volvo 18 wheels"
```

You can also access attributes individually from object instance:

```
print(volvo.make)   # Prints "Volvo"
print(volvo.wheels) # Prints 18
```

It's legal to use the same parameter names as instance attributes:

```
class Car:
    def __init__(self, make, wheels):
        self.make = make
        self.wheels = wheels
        print(self.make, self.wheels)
```

This is the most common form of Python class constructors.

10.0.2 Using __new__constructor

The magic method __new__ is optional. It is executed before __init__ as part of the same process of creating the instance of an object.

When using __new__ you *must* add (object) to classCar (object): definition.

```
class Car(object):
    def __new__(cls):
        print("Hello from __new__")
        return object.__new__(cls)
```

Instead of self arbitrary cls parameter is used.

It is required to return an object from __new__ method.

In the same way as __init__ you can supply additional parameters:

```
class Car(object):
    def __new__(cls, make, wheels):
        cls.make = make
        cls.wheels = wheels
        return object.__new__(cls)
```

Like __init__, __new__ also assigns attributes.

Let's check:

```
volvo = Car("volvo", 18)
print(volvo.make)
print(volvo.wheels)
```

Prints:

```
volvo
18
```

We had just seen how __new__ can be used to replace __init__ but it probably shouldn't except in advanced cases.

10.0.3 Using `__new__` and `__init__` together

If both exist, `__new__` method executes first:

```
class Car(object):
    def __new__(cls, make, wheels):
        cls.make = make
        cls.wheels = wheels
        return object.__new__(cls)

    def __init__(self, make, wheels):
        print(self.make, self.wheels)
```

Attributes `make` and `wheels` were assigned in `__new__` to `cls` object.

The object returned from `__new__` is passed on to `self` inside `__init__` – they both point to the same object instance. `__init__` requires at least one `self` parameter. Therefore `__new__` *must* return an object – otherwise init is never called.

```
volvo = Car("volvo", 10) # Prints "volvo 10"
```

When using `__new__` and `__init__` together, the same number of parameters *must* be specified in both magic methods.

`self` overrides `cls`

You can assign properties to `cls` object but assignment to `__init__`'s `self` will overwrite them:

```
class Car(object):

    def __new__(cls, make, wheels):
        cls.make = make
        cls.wheels = wheels
        return object.__new__(cls)
```

```
    def __init__(self, make, wheels):
        self.make = "over"
        self.wheels = "written"
        print(self.make, self.wheels)
```

The following code will print `"over written"`, not `"volvo 80"`:

```
volvo = Car("volvo", 80) # Prints "over written"
```

You can override both *magic methods* or just one to your heart's content or depending on the situation.

Just keep in mind single `__init__` constructor is the most common way to create objects in Python. In most cases you won't need to override `__new__` at all.

`__new__` and `__init__` methods are *magic methods*. They are executed under the hood anyway but by explicitly defining them we can take control over the method's body and return values in order to modify their behavior.

Memory optimization with `__slots__`

By default Python uses `dict` to store object instance's attributes. This can take a lot of RAM when dealing with thousands or millions of objects.

Using `__slots__` method you can reduce RAM usage by 40 to 50% in many cases.

```
class SomeClass(object):
  __slots__ = ['name', 'identifier']
  def __init__(self, name, identifier):
    self.name = name
    self.identifier = identifier
```

To make Python use slots create a list of argument names matching those passed to class constructor and assign it to `__slots__` attribute.

This should automatically reduce RAM usage, especially in programs that use small classes and many instances of the same object.

What's Next?

Classes are usually created to define object methods that you can call on an instance of an object. Class constructors have access to `self` as we have just seen where `self` and *object instance* are synonymous.

Static methods, on the other hand, can be called on class name itself – they are used as utility functions and don't have access to `self`.

But – uniquely to Python – there is also something called a *class method*, which is sort of a hybrid between a static method and a method that can access `self`.

In the following section we will discern the difference between regular class methods and static methods and how they work in Python.

10.1 Instance, Class and Static Methods

If you are coming to Python from an Object Oriented programming language such as C++, you may find Python's distinction between *class methods* and *static methods* confusing. Considering that even *instance methods* can be still called on Class itself in the same way only static methods can be called in C++.

In Python the object constructor has two attributes (properties) representing the instantiated object with their default names `cls` and `self`. The `cls` object is constructed in magic method `__main__` and then passed on as argument called `self` to magic method `__init__`. So the construction process is two-fold.

There are 3 types of methods you should be aware of when making classes in Python: instance methods, static methods and class methods. Which type of method should be used and when? The following sections will make an attempt to answer this question.

10.1.1 Instance methods

Instance methods are your regular object methods, they assume positional `self` parameter and are intended to be called on instance of an object created (after it is *instantiated*) using that class:

To create a custom class method use `def` keyword:

```
class Car:
   def describe():
       print("This is a car.")
```

Instance methods *can* behave as static methods.

```
Car.describe() # Prints: "This is a car."
```

By default `describe()` *can be called* as a *static method* – directly on `Car` class without having to instantiate an object.

But there is a catch. While this is *possible* it is suggested to use Python's `@staticmethod` decorator to explicitly define it as static method.

10.2 Static methods

A static method cannot and will be used only as a general purpose *utility* function.

Static methods can be called directly from class name like `Car.info()` but they can neither access nor modify class state.

To make static method definition explicit, use `@staticmethod` keyword directly above the method's definition:

```
class Car:
    @staticmethod
    def describe():
        print("This is a car.")

Car.describe() # Prints: "This is a car."
```

This class behaves in exactly the same way as previous example. But this is a more proper way of defining a static method.

10.2.1 Class Methods

To specify `describe()` method as class method use `@classmethod` keyword:

```
class Car:
   @classmethod
   def describe():
      print("This is a car.")
```

Now calling `Car.describe()` will result in error:

```
Car.describe()
```

```
TypeError: describe() takes 0 positional arguments but 1 was given
```

In order to make this work the positional `self` parameter must be explicitly specified on a non-static class method:

```
class Car:
    @classmethod
    def describe(self):
        print("This is a car.")
```

Of course, when self is involved it is assumed that there will be a constructor:

```
class Car:
    def __init__(self, make):
        self.make = make
    @classmethod
    def describe(self):
        print("The make of this car is ", self.make)
```

Technically, non-static class methods such as the one above are not intended to be called on class name – `Car`. Non-static class methods are reserved for use on *instance* of an object.

However, soon as `self` parameter is added the `describe()` it makes `describe()` *callable* as a static method again:

```
Car.describe() # "This is a car."
```

It is possible. But this is probably not something you should be doing. The proper way to call a non-static class method is to first create an instance of an object:

```
mazda = Car() # Prints "This is a car."
```

10.2.2 Static methods

Static methods are methods that can be called directly on the class name:

Here `Car.count()` calls static method `count` if it's defined. This method is not attached to an *instance* of an object of type `Car`.

Static attributes are useful for tracking class-wide information. For example how many object instances of type `Car` were created:

To draw a distinction between a static method from object instance method, Python provides two keywords: `@classmethod` and `@staticmethod`:

```
class Car:
    @classmethod
    def describe():
        print("This is a car")

class Car:
    total_cars = 4
    @staticmethod
    def count():
        print("This is a car")

Car.describe()
```

All variables declared inside the class body are *static attributes*. They should contain values relating to entire class as a whole, not the object instances created with it. (For example, to keep track of the number of all instantiated cars.)

10.2.3 Static Attributes

At this point you can actually access `Car.make` and `Car.wheels` attributes directly from the class definition:

```
print(Car.total_cars)
```

```
>>>
4
```

Attributes defined at class-level are called *static* attributes.

10.3 Objects

An object can be thought of as an instance of a class. Let's add more code to the previous example to actually do something with this class. To create an object instance of the class `Car`:

```
class Car:
    def __init__(self):
        self.make = "Tesla"
        self.wheels = 4

    def describe(self):
        message = "This car is a %s and it has %d wheels"
        print(message % (self.make, self.wheels))

# Instantiate object tesla from class Car
tesla = Car()

# Print out some values
print(tesla)
print("My Tesla has %i" % tesla.wheels)
```

Output:

```
My Tesla has 4 wheels
```

Once instantiated, you can access properties from `Car` class definition via the object instance `tesla`. The `wheels` property can be accessed using dot operator:

```
tesla.wheels
```

And the make can be accessed as:

```
tesla.make
```

This is how property access of instantiated objects works. You can also access methods in the same way. But in order to execute them, add parentheses:

```
tesla.describe()
```

But design of our class can still be improved.

Classes are patterns created to help you reduce amount of code written. This principle of *code reuse* is very important in programming. In order to take advantage of it you can create a more flexible constructor method that takes user-defined values and sets them to the `self` object.

10.4 Constructors

In the previous section we created an object `tesla` from `Car` class. But every time we create a new object it still only allows us to create the same `"Tesla"` car with `4` wheels because the values are hard-coded inside the constructor.

Ideally, you want the class to be as flexible as possible. Let's modify the constructor by allowing whoever is using our class to enter their own values:

```
def __init__(self, make, wheels):
    self.make = make
    self.wheels = wheels
```

The only difference here is the addition of `make` and `wheels` arguments to the constructor. Inside the constructor we now assign values passed in those arguments to `self.make` and `self.wheels` object properties instead of hard-coding them.

Now we can create multiple objects of the same type `Car` using the same class but with drastically different properties:

```
tesla = Car("Tesla", 4)
tesla.describe()

truck = Car("Volvo", 18)
truck.describe()
```

Output will be:

```
This car is a Tesla and it has 4 wheels
This car is a Volvo and it has 18 wheels
```

We just used the same class definition to create a 4-wheel Tesla and a Volvo, the 18-wheeler truck. Two different cars with unique characteristics.

That's what you want to achieve with your class design – a model that can create similar objects, without changing the original design of the class.

By doing this you reduce the size of your computer program.

Another Class Example

Let's create a class to represent a web server:

```
class Server:
    # Object constructor
```

```
    def __init__(self, os, distro, ver, ram, cpu, cores, gpu):
        self.os = os
        self.distro = distro
        self.ver = ver
        self.ram = ram
        self.cpu = cpu
        self.cores = cores
        self.gpu = gpu

    # return list of attributes
    def attributes(self):
        return vars(self)

webserver = Server("Linux", "Ubuntu", "18", "32Gb", "Intel",
8, "Nvidia RTX")


# Call object's attributes() method
attrs = webserver.attributes()

# Print attributes
print(attrs)
```

Here is the output:

```
'os': 'Linux', 'distro': 'Ubuntu', 'ver': '18', 'ram': '32Gb','cpu': 'Intel'
```

We've just printed out all attributes of our instantiated object.

In Python object properties are called *attributes*.

Python's built-in function `vars` works with objects. It retrieves the list of attributes from the object passed to it. You don't have to add any modules to start using it.

When we passed `self` object to `vars` function in the body of the Server's class `attributes` method, it packaged the object's attributes into a JSON object and returned it using function's `return` keyword.

Note: You don't have to call `vars` inside a class method. It's a global scope function that works anywhere.

Prettifying JSON output

Working with JSON objects is very common.

Output becomes less readable the more properties there are in the JSON object. To prettify JSON output you can use `json.dumps` method from built-in `json` module. Let's add following code to previous example:

```
# Import json module (so we can use json.dumps method)
import json

# Format JSON object with 2 spaces
formatted = json.dumps(attrs, 2)
```

The output of running this line of code will be a list of all object's attributes as a JSON object but with line breaks and formatted with 2 spaces:

```
{
  "os": "Linux",
  "distro": "Ubuntu",
  "ver": "18",
  "ram": "32Gb",
  "cpu": "Intel",
  "cores": 8,
```

```
  "gpu": "Nvidia RTX"
}
```

The `json.dumps` function will also properly format a JSON object which by specification requires double quotes around property names. (Single-quotes are considered a malformed JSON format and shouldn't be used to define JSON.)

The `import json` line adds `json` module to our code. It has a bunch of json-related functions. In this case we only needed `json.dumps`. Usually importing modules is done at the very top of your file, but you can add it anywhere in the file as long as it's *above* the place in code where `json.dumps` is actually used.

Chapter 11

File System

Python has a number of functions for working with files.

11.0.1 File operations

Opening files with built-in open() function

Let's open `file.txt` for reading:

```
# which file?
filename = "file.txt"

print(f"Opening {filename} for reading...")

f = open(filename) # open file
```

```
>>>
Opening file.txt for reading...
```

The last line of code is equivalent to executing `open` function with `"rt"` flag:

```
f = open(filename, "rt")
```

`"rt"` is the default mode for opening files. It means opening the file for (`r`)eading in (`t`)ext format (as opposed to binary.) We'll take a look at all modes shortly.

The `open` function will make an attempt to open the file `"file.txt"` in the same directory from which script was executed. But you can also provide a full path to the file.

The default value for second argument in file function is `"r"`.

It's an optional parameter for specifying file operation mode.

Here `"r"` tells `open` function to open the file for *reading*.

For example, to open a file for writing you can use `"w"` instead:

```
f = open(filename, "w")
```

We will take a look at all seven modes in a moment. But first I think it's important to cover error handling.

Error handling

What will happen if we try to open a file that doesn't exist?

```
f = open(filename)
```

Because we didn't create this file yet, Python will complain to the command line:

```
Traceback (most recent call last):
  File "fileops.py", line 5, in <module>
    f = open(filename, "r")
FileNotFoundError: [Errno 2] No such file or directory: 'file.txt'
```

What just happened is called an exception. These types of errors are common to many other programming languages. What makes exceptions unique is that they are not generated by *your code* but by circumstance in which it was executed.

(For example if `file.txt` existed, no error would be generated.)

11.0.2 Handling Exceptions with try: except: block

We have just seen that if the file doesn't exist Python throws an error.

Because reading a file is an operation that can often fail for various reasons outside of your control (for example: *file was deleted or its permissions are set to "read-only" and you're trying to open it for writing*) it's important to catch these types of failures without interrupting execution flow of your program.

And that's what catching errors helps us accomplish in general – exceptions are not limited to file operations and can happen anywhere in your program.

For example one classic example is trying to divide by 0:

```
100/0
```

```
>>>
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    100/0
ZeroDivisionError: division by zero
```

Note that Python identified the error as `ZeroDivisionError`.

Adding two previously undefined variables:

```
hello = 1
world = 2
hello + world
```

```
>>>
Traceback (most recent call last):
  File "fileops.py", line 1, in <module>
    hello + world
NameError: name 'hello' is not defined
```

This erroneous code produced `NameError` error. If this exception is uncaught, your program will halt and exit. That's not something you want to ever happen.

try: except: block

Python exception handling is done with a **try:** and **except:** block.

```
try:
    print("Do something.")
except:
    print("Statement failed.")
```

There is nothing wrong with the code in **try:** block:

```
>>>
Do something.
```

In order to trigger the **except:** block the statement concealed in **try:** must fail.

To catch an exception you can write **try:** and **except:** blocks around any Python statement you want to test for errors. If a natural error occurs in the **try:** block the code in **except:** clause will be executed.

At this point all we are doing is displaying `"Statement failed."` message if the code fails. But it still doesn't give us control over handling the actual error code.

The solution to that is to tell **except:** what error we're trying to handle. Python error codes are displayed on the command line when the statement actually fails.

If file you're trying to open doesn't exist you get `FileNotFoundError` exception. This is the error code we need to pass to **except:** as follows:

```
try:
    file = open('filename.txt')
except FileNotFoundError:
    print("FileNotFoundError has occurred")
```

In addition you can interpret the error code by copying it into a variable called **error** (but you can use any other name) if you need to print it out:

```
try:
    file = open('filename.txt')
except FileNotFoundError as error:
    print(f"{error} has occurred")
```

We can now apply this to our file example as follows:

```
filename = "test.txt"

try:
    print(f"Opening {filename} for writing")
    f = open(filename, "w")
except FileNotFoundError as error:
    print(f"Error is = {error}")
```

This should be the standard way of opening files.

Now even if, for whatever reason, the open function fails we can gracefully display a message and move on to the next statement without breaking the program flow. Use **try:** and **except:** to handle errors outside of your control when dealing with files and similar operations.

Raising exceptions

It is possible to raise your own exceptions. This is useful when you are 100% sure the code relies on success of an operation. And if it fails, there is no way you want to perform any other future operations. This often indicates a serious error.

To raise your own exception use **raise** keyword:

```
try:
    print("About to raise an exception")
    raise ZeroDivisionError
finally:
    print("Goodbye!")
```

```
>>>
About to raise an exception
Goodbye!
Traceback (most recent call last):
  File "raise.py", line 3, in <module>
    raise ZeroDivisionError
ZeroDivisionError
```

The **raise** keyword must point to an existing exception error class. We have just risen the `ZeroDivisionError` even though we did not perform the division by 0 operation – the exception was forced. But ideally you want to create your own exception types which can be defined by extending `Error` class:

```
class Error(Exception):
# base error class
pass

class CustomError(Error):
# custom exeption
# that defines attributes
# with more information
# about this error
def __init__(self):
    print("The dough has not risen!")
    pass

try:
    raise CustomError
finally:
    print("Goodbye!")

>>>
The dough has not risen!
Goodbye!
  Traceback (most recent call last):
    File "raise.py", line 16, in <module>
```

```
raise CustomError
__main__.CustomError
```

Error names usually end with the word Error, so be sure to name yours in a way that follows that pattern to remain consistent.

```
f = open("file.txt", "w")
```

One more thing about `"w"` mode

Be careful not to open *important* files with `"w"` – even if file already exists a new blank file will be created overwriting content in the original file.

Force creating a blank file for writing if it doesn't exist:

```
f = open("file.txt", "w")
```

This mode will clear out contents of an existing `file.txt` or create a new one from scratch. At this point the file is ready to be written to.

Open file modes

Mode	Description
r	Open file for reading. (default)
t	Open in text mode. (default)
w	Open file for writing. (Creates a new file if file doesn't exist; truncate file if it exists.
x	Open file for exclusive creation. (If the file already exists this operation will fail.)
a	Open file to appending content to the end of file (Creates a new file if it does not exist.)
b	Open file in binary mode (requires another mode)
+	Open file for updating (reading and writing)

If you want to avoid overwriting (truncating) an important file with the `"w"` option, you can try to open the file for *exclusive creation* using `"x"` mode:

```
try:
  # Statement fails with an error if file already exists
  # or creates a new blank file if it doesn't exist
  f = open("file.txt", "x")
except:
  print("Can't open file.txt - file already exists")
```

Not all files are in plain text format. For example Photoshop files, PDF files, or video game 3D model data is stored in binary format. In that case, when you need to read file's binary data you can use `"b"` mode:

```
# open file in binary mode with
f = open("file.dat", "rb")
```

When opening a file with `"b"` you *must* also include one of the four file modes (`"r"`, `"w"` or `"a"`) and at most one plus (`"+"`). Trying to open file with just the single character `"b"` will fail.

To open file for *appending* data to its end:

```
f = open("file.txt", "a")
```

Open file for *reading* and *writing* (updating)

```
f = open("file.txt",  "r+")
```

Modes can be combined. To open file for both reading and writing in binary mode you can use "r+b":

```
f = open("file.txt", "r+b")
```

This mode allows you to use both **read** and **write** functions on open stream.

Reading data from file

This section will also introduce you to another way of handling exceptions. This time we will use **try-finally** block. The **finally** part executes regardless of whether operation failed or not. It's often used to do optional clean up.

Our file `the_tyger.txt` contains four lines of famous poem *"The Tyger"* by William Blake. First, let's open this file for reading and then simply output its contents. The file `the_tyger.txt` contains only the first four lines of the poem:

```
Tyger Tyger, burning bright,
In the forests of the night;
What immortal hand or eye,
Could frame thy fearful symmetry?
```

Let's write a function that reads the contents of this poem, prints it out and "finally" closes the file:

```
try:
  # Open file to read its contents
  poem = open('the_tyger.txt')
  contents = poem.read()
  print(contents)
except FileNotFoundError:
  print("FileNotFoundError has occurred")
finally:
  print("> closing the file...")
  poem.close()
  print("> file closed!")

>>>
Tyger Tyger, burning bright,
In the forests of the night;
What immortal hand or eye,
Could frame thy fearful symmetry?
```

Regardless whether there was an error or not **finally** keyword helps us close the file (which is something you should normally do after reading it.)

```
poem.close()
```

Reading data from file one line at a time

Often you will want to parse a text file one line at a time. This example demonstrates how to set variables based on what the file parser finds on each line.

Writing to a file with write() function

You can write data to a file using **write** function:

```
writer = open('new-file.txt', 'w')
writer.write("Some content")
writer.close()
```

This opens `new-file.txt` and writes a string `"Some content"` to it. A new blank file will be created. If a file with the same name already exists it will be overwritten and turned into a blank file. Be careful using "w" on important files.

Appending data to a file with "a" and write()

To append data to a file you need to open it in `"a"` mode. This means opening file for writing – **write()** function will *append* data to the very end:

```
# Open the poem file in append mode:
poem = open('the_tyger.txt', 'a')

# add a line break
poem.write("\n")
```

```
# add your own line to the very end of the file:
poem.write("My brilliant poetic line.")

# close file
poem.close()

# read the file again and print its new content
poem = open('the_tyger.txt', 'rt')
print(poem)
```

Every time you execute this script a new line will be added to the bottom of the original file. After running it 3 times in a row the file now contains:

```
Tyger Tyger, burning bright,
In the forests of the night;
What immortal hand or eye,
Could frame thy fearful symmetry
My brilliant poetic line.
My brilliant poetic line.
My brilliant poetic line.
```

Added lines are highlighted in purple.

Using os package

The **os** package provides several functions for working with files.

For example, to move file to another directory you can use the **os.rename** function. Here are some examples:

```
import os

# Rename file 'file.txt' to 'new.txt' in current directory
os.rename('./file.txt', './new.txt')
```

```
# It's equivalent to:
os.rename('file.txt', 'new.txt')
```

You can move your file to an upper directory:

```
# Move the file to an upper level directory and rename it:
os.rename('new.txt', './../moved.txt')
```

Or move it into an existing folder `./calendar` in current directory:

```
# Move a file into an existing directory:
os.rename('months.txt', './calendar/months.txt')
```

Deleting files

```
# Delete file:
os.remove('file.txt')
```

We just covered most of the basic file operations you might want to use in your program.

11.1 Desktop Salvation

Does your Desktop ever get cluttered with files? What if there was a way to quickly move all files on the Desktop to their respective folders by type.

Image files with extension `.jpg`, `.jpeg`, `.bmp`, `.psd` and `.png` could be moved to "Images" folder. Text documents like `.txt`, `.doc` and `.xls` can be moved to "Text" folder. Likewise, the development files such as `.py`, `.js`, `.html` `.css` could go into "Dev" folder.

You can specify your own rules by modifying the `folders` dictionary.

Knowing everything we know so far about loops and file operations, let's write a simple program that will scan all files on desktop, identify file type by their extension and move them to a folder.

We can create a relationship between the folder and a list of file extensions that should be moved there by using a dictionary of tuples.

```
import os

desktop = "C:Desktop" # replace with your desktop path

folders = {
  "Text" :
   (".txt",
    ".doc",
    ".xls"),
  "Images" :
   (".psd",
    ".jpg",
    ".jpeg",
    ".jpg_large",
    ".jfif",
    ".png",
    ".bmp",
    ".tga"),
  "Dev" :
   (".html",
    ".css",
    ".js",
    ".py",
    ".c",
    ".cpp",
    ".h"),
  "Video" :
   (".mpg",
    ".mov",
    ".mp4"),
  "Music" :
   (".mp3"),
}
```

To keep things simple the code consists of several nested for loops.

```
moved = 0
# list all files in desktop directory
for file in os.listdir(desktop):
    # for each destination folder from our dictionary
    for folder in folders:
        # for each destination folder
        for extension in folders[folder]:
        # if matches file extension
        if file.endswith(extension):
            # check if folder exists, if not create it
            directory = f"{desktop}{folder}f"
            if os.path.isdir(directory) == False:
                os.mkdir(directory)
            # move file to this directory
            original = f"{desktop}{file}"
            target = f"{directory}{file}"
            # make sure it's a file and not directory
            if os.path.isfile(original) == True:
                # make sure target file doesn't already exist
                if os.path.isfile(target) == False:
                    # finaly move file...
                    os.rename(original, target)
                    moved += 1
print(f"{moved} files moved")
```

Running this script moves all files with listed extensions to one of target directories (`"Images"`, `"Text"`, `"Video"`) etc. If folder doesn't exist it will be created.

```
python salvation.py
```

```
>>>
174 files moved
```

Chapter 12

MongoDB

Like with any language, Python requires a Mongo database driver to start using Mongo. One of the common Mongo drivers for Python is called `PyMongo`. To download and install this driver go to Command Prompt and type:

```
python -m pip install pymongo
```

Running Mongo Database Server

Before you can start executing commands on your Mongo server from your Python script, first it needs to run on your system as a stand alone application.

Inserting Items Into Collection

In your Python program import `pymongo` module at the top of the file. And now we are ready to connect to the Mongo database which will usually run on `localhost` at port 27017 by default.

```
# Import the pymongo module
import pymongo

# Create a client
client = pymongo.MongoClient("mongodb://localhost:27017/")

# Access user collection (table)
```

```
db = client["users"]

# Return a list of all existing databases in the system
print(myclient.list_database_names())
```

You can also check if the database exists in system:

```
databases = client.list_database_names()
if "users" in databases:
    print("Database users exists.")
```

Mongo doesn't consider a collection as created until you actually add some data to it. So before connecting to "user" collection we need to insert some entries.

12.0.1 Inserting Dictionaries Into Mongo Collections

In contrast to MySQL which is a relational database, Mongo is document-based. This means instead of writing queries, items are inserted into Mongo collections (think of them as being similar to tables in MySQL) using JSON-like object format.

12.0.2 `.insert_one()` method

In this example we will insert a simple user object containing an email and a password into "users" collection. This is something you would do on the back-end when registering a new user from a web form input.

```
import pymongo

# Connect to client
client = pymongo.MongoClient("mongodb://localhost:27017/")

# Grab a link to the database and users collection
database = client["database"]
users = database["users"]
```

```
# Create a user dictionary to insert into "users" collection
user = "email": "felix@gmail.com", "password":"catfood15"

# Finally insert it into Mongo users collection
users.insert_one(user)
```

First, we created a Mongo client, acquired live database object, and created a link to "users" collection via collection variable.

Remember that in Python JSON-like object structures are called *dictionaries*. Here we also created an arbitrary dictionary and inserted it into `users` collection (table) using Mongo's `insert_one` method.

Chapter 13

What can Python be used for?

Python's syntax makes it one of the easiest languages to learn. You are probably a front-end developer who writes code primarily in JavaScript. You may have only heard of Python without actually taking time to learn it. I can understand that:

Virtually all front-end work is done in JavaScript – the only language supported by modern browsers and Python is not even a client-side language.

Python is usually used for server-side scripting, DevOps and things like Machine Learning. Due to recent year's progress in these areas of computer technology Python has also become one of the fastest growing computer languages.

Recently Python has experienced unprecedented growth. What contributed to this interest in the language since 2010? Let's take a look.

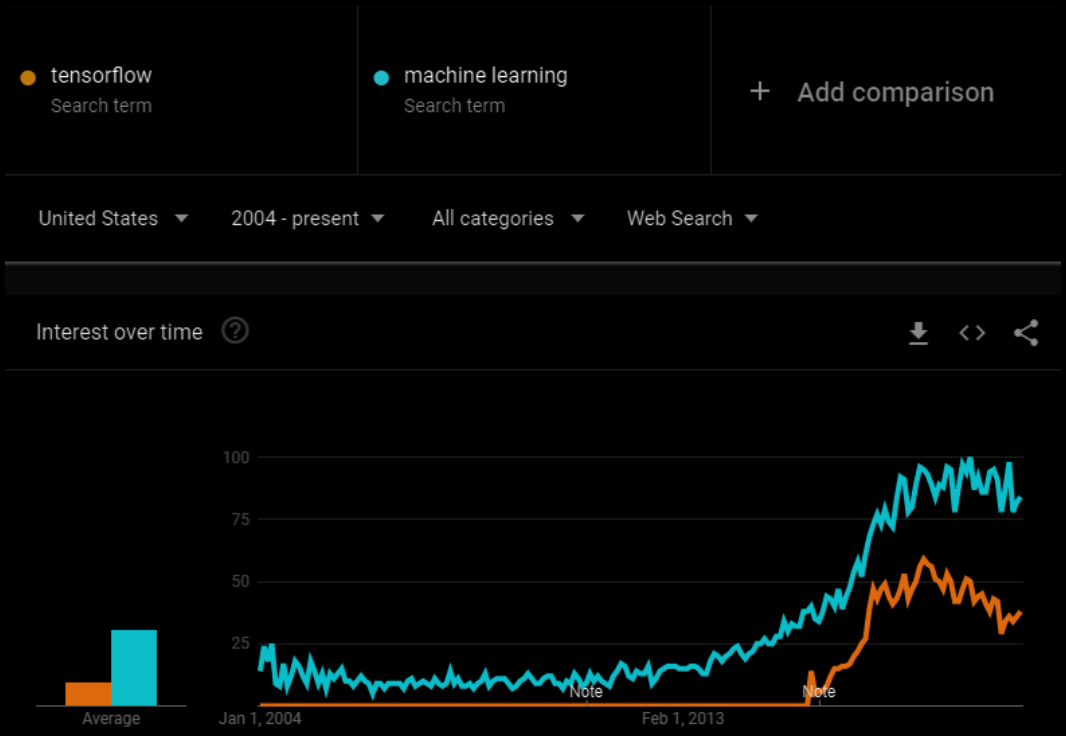
This growth can probably be attributed to a combination of many things.

We already know that due to its simple syntax Python is an easy language to learn. But what else makes it a popular language?

13.0.1 Machine Learning

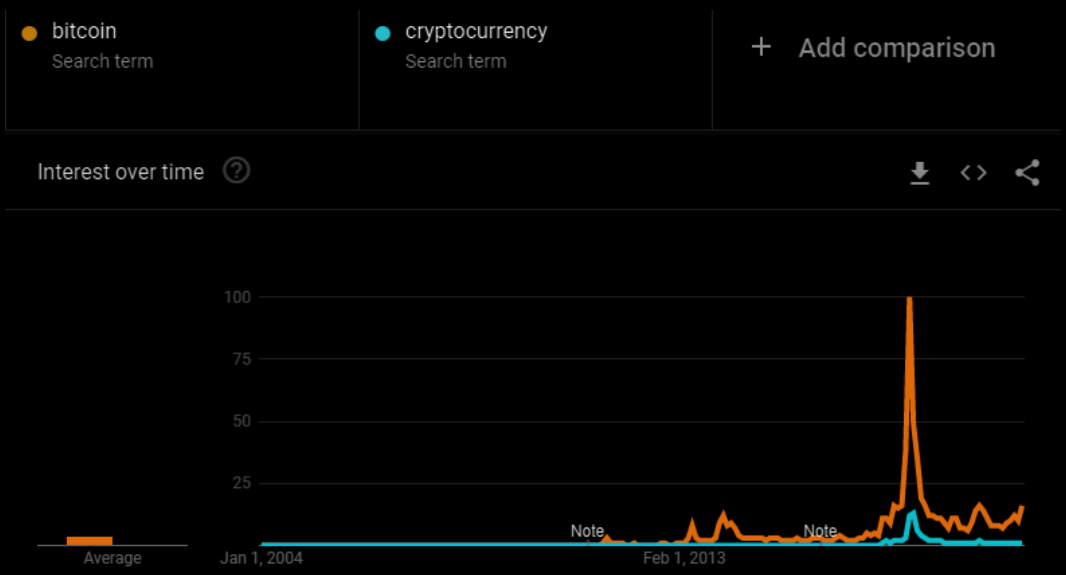
A great deal of Python's popularity can be attributed to the increased interest in Machine Learning. Google's TensorFlow library used for writing Machine Learning algorithms is written in Python (that also has a C++ counterpart.)

TensorFlow library can also work on Nvidia graphics card with CUDA and can be installed on Linux operating systems.



13.0.2 Blockchain

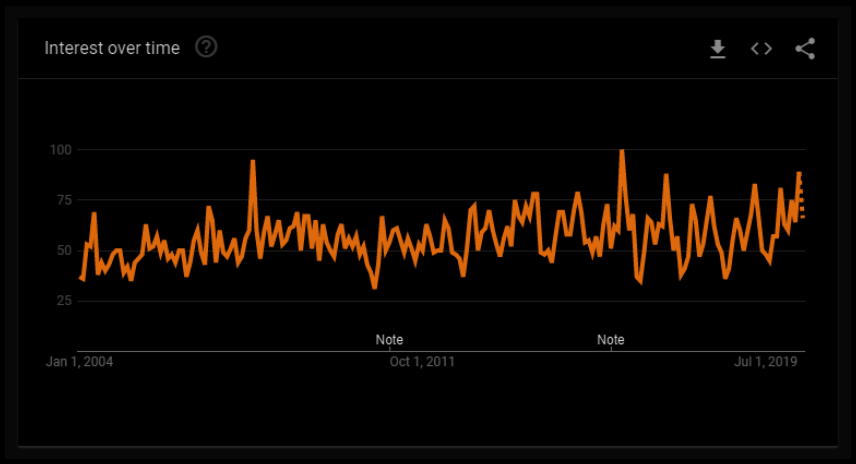
Perhaps you've also heard of **blockchain** technology and **cryptocurrency**. It's not uncommon for cryptocurrency miner software to be written in Python.



It's fairly simple to see that this and many other *relatively* new technologies have together contributed to the growth of Python language.

13.0.3 Pygame

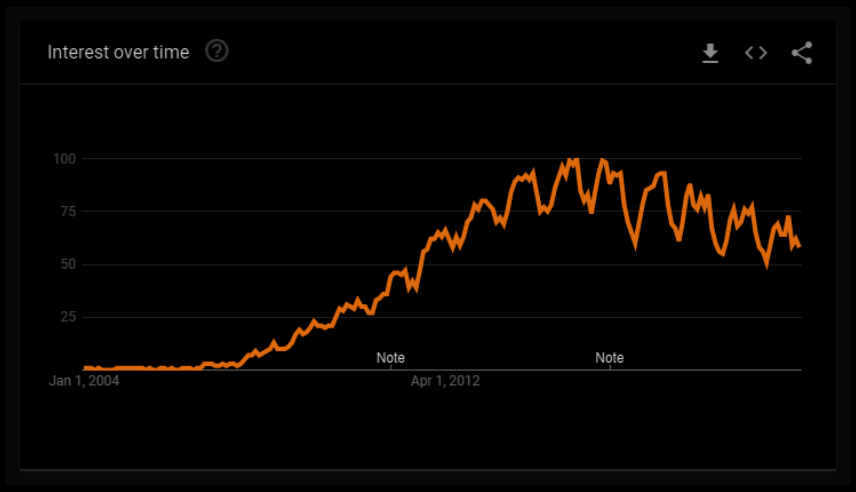
Pygame is a cross-platform set of Python modules designed for writing video games. It includes computer graphics and sound libraries designed to be used with the Python programming language.



The interest in Pygame has been pretty consistent over the last two decades.

13.0.4 Arduino

Arduino is an open-source electronics platform based on easy-to-use hardware and software. It's intended for anyone making interactive projects.



Though it may appear as though interest in Arduino is fading, a decline in Google

Trends can also be indicative of world-wide acceptance. Instead of searching for something people go directly to the site because it's saved in their bookmarks.

Python is not the only language you can use to program Arduino projects. It is possible to use C, C++ and JavaScript, but it's on the list of acceptable choices.

13.1 Amazon AWS

Amazon's Web Services (AWS) is a popular platform that provides on-demand cloud server resources for solving various computing problems. While you don't have to use Python with AWS, it is likely that at one point in your career as a web developer you may run into this combination.

13.2 Algebra, Matrices and Image Processing

Python has several image processing libraries which makes it a great tool for server-side image manipulation. Some of the popular libraries are `scikit`, and `NumPy`. `NumPy` is a core library in Python that provides tools for working with *arrays* (Arrays, or ordered lists in Python are called *lists*). But because images are basically arrays `NumPy` is often used for image manipulation. `Scipy` is another library that works on n-dimensional arrays, in particular the module `scipy.ndimage`.

13.3 MySQL, MongoDB or PostgreSQL

Python can be used with MySQL, MongoDB and PostgreSQL. That makes it possible to create your own API end-point server in Python for your web application.

This setup allows you to compete with Node.js (or Deno) servers. If not in speed, at least in server architecture. As you will shortly see writing Python API code can be a joy.

13.4 File System Operations

Python can read and write files from the file system. This makes it a great companion to scripts that format excel spreadsheets, read an email inbox and generally to any situation where you want to copy, edit, or save files on hard drive.

13.5 Math Library

Python has a great math library including built-in math functions that are not even available "out of the box" in some of other popular languages.

If you're someone who has an appreciation for math and physics, you will find Python to be a natural companion.

Python has a large scientific community based around it which often leads to writing optimized math-focused modules. Whereas languages like JavaScript are thought of being used primarily for web development.

It can handle large data sets and can be used exclusively to perform complex mathematics.

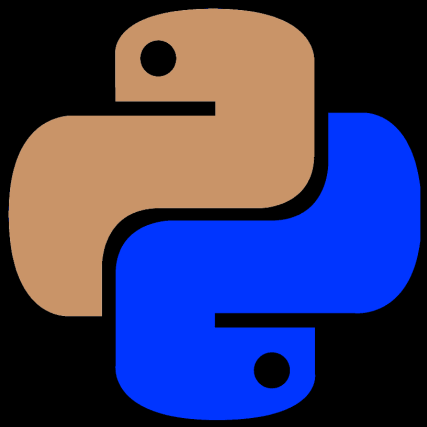
—

Finally, Python is available for almost any Operating System. This includes UNIX-based, Windows, macOS, iOS, and Android.

13.6 The Story Behind Python Logo

Python is a general-purpose language. Originally, its name was influenced by *Monty Python* - the British surreal comedy troupe, not the snake – but over time, it was inevitable for the symbol to be generally accepted.

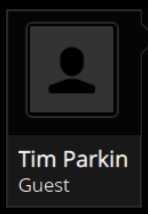
The Python Software Foundation is a non-profit corporation that holds intellectual rights to the Python programming language.



Tim Parkin designed what is now known as the Python logo with two snakes.

In the spirit of good humor, I will include Tim Parkin's answer to one of the critics of the Python logo who said it has various undertones and communicates balance (instead of true properties of Python languages such as abstraction, composition and "the whole is greater than the parts" principle.)

It was decided to include the following forum discussion because it quickly communicates the design of the Python logo from a reliable source – the logo designer himself. *Note: the print was edited with minor adjustments for this book-format, with only small additions like uppercase letters.*



Tim Parkin
Guest

> **Michael said:**
>
> *That said, and conceding that the first impression is positive, I don't see how it represents Python. More to the point, the longer I look at it the less I like it, and I would NOT wear it on a T-shirt.*

Over 25 people disagree with you so far and thats without any advertising whatsoever (it's an older version of the logo) because you can get T-Shirts from cafepress/pydotorg and any profits go to the PSF.

> *Both the cross and the yin-yang have religious associations, which will be positive for some and negative for others but will certainly be unrepresentative of what Python is. This would be a great logo for Taoist Christians, if such a group exists.*
>
> *How is Python about "balance"? It is about abstraction, composition, the whole greater than the parts, yes, but there's nothing there that really draws on duality. So the whole two-ness of the thing is one of the parts that disturbs me.*

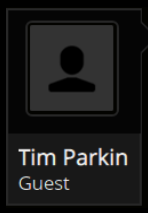
They're friendly snakes at a tadpole fancy dress competition having a 'cuddle'. Where do you think Python eggs come from...

Tim Parkin

P.S. The logo is actually based on mayan representations of snakes which often represent only the head and perhaps a short length of tail. The structure of the snake representations the natural coiling/nesting of a snake as seen side on.

The Mesoamerican calendar also represents snake heads in a similar manner.

The abstraction of the snake design used in mayan culture seemed non-denominational enough to only raise contrived objections. The shapes used (cross/spiral/yin-yang) are also primitive enough that there will always be connotations that can be derived.



The two headed snake was also an influence on the design which is also a common 'meme' in many continents, including Africa.

And I'd like to see you tell a civil war soldier that it looks like his trousers are held up by a two headed tadpole.

If you look carefully at the logo, you will also see an indian symbol of peace. (I'll leave this one alone as it can also mean something else.)

13.7 Python Grammar

What is this book about and who is it for?

1.) It's a practical introduction to Python language.
2.) It's a collection of examples relevant to learning Python for the first time.
3.) Reference for many practical features of Python 3.0 and beyond
4.) A book that sparsely uses visual diagrams to explain abstract principles.
5.) Guidebook for programmers already familiar with another language (such as JavaScript, C, C#, C++, PHP or Java) who now also want to learn Python.

13.8 Which Python?

Python 3 – also known as "Python 3000" or "Py3K" is the first ever intentionally backwards incompatible Python release.

Python 3.0 is a major version release and as such contains more changes than usual. However this doesn't change the language in a fundamental way and many of the things you're used to in previous versions are still there.

This book is for Python 3.0 and greater. Running some examples from this book will not work in Python <= 3 – in many ways due to changes to `print` function.

13.9 Book Format

There is no good reason for the presentation format of a book about a computer language to be any more complicated than the syntax of that language.

Source code will be presented in simple mono-spaced typewriter-like font without any decorations, line numbers or colors:

```
print("Hello from Python running on .".format("Ubuntu"))
```

What you learn matters only when knowledge is put into practice. It is recommended to actually type along with the code examples and run them from the command line on either Windows or in the Mac's Terminal:

```
python hello.py
```

Output:

```
Hello from Python running on Ubuntu.
```

If you have previous experience writing code in other languages, you've probably already guessed that Python replaced {} with value provided to `.format` method.

If not, and Python is your first language, don't worry. This and many other examples found throughout **Python Grammar** will be explained in complete detail.

Reminder: When learning how to code *in any language* simply memorizing keywords doesn't work. It's important to learn how to use the language as a problem-solving tool. While this book alone cannot teach you that skill, you are encouraged to improvise on examples shown in this book and write your own code.

Python Grammar Symbol

While reading, look out for this logo. Whenever you see a box like this it's usually a **Python Grammar Pro Tip!** It will often contain some practical wisdom. It might also offer help in guiding you to avoid common pitfalls on your journey, or point out mistakes you might run into while learning...which is usually not a fault of your own.

Diagrams will be used sparsely. Only if they become absolutely necessary to explain visual concepts. Sometimes connecting the dots is possible only by visualizing the relationship between composite ideas.

13.10 Learning Python

This book – Python Grammar – is exactly what it sounds like.

Using simple tutorial format we'll learn how to use Python to write short computer programs. We'll explore Python syntax by using many source code examples that will help you reach a level of confidence needed to write meaningful programs.

Like any programming language, Python gives you a set of problem-solving tools like *data structures*, conditional statements and operators. Using your creativity you will use these tools to write computer programs.

Mostly we will focus on the features of Python programming language. That makes it a book for beginners. But these simple rules mirror foundational patterns seen in many programming languages, not just Python.

This is a walk-through book. It is structured in a linear way. It's best to go through it one chapter at a time. But it can also be used as a desk reference in case you need to look up one particular feature in isolation.

My first language was C++. And then I picked up PHP and JavaScript, because both languages were similar to the syntax of C++.

Python is a very different language, but in a good way.

I like Python. It's the computer scientist's language. I don't think I am one. What I do think is that Python is actually physically a lot easier on the hands to write. There are no excessive brackets or semicolons. Python is dynamically-typed – so there are no variable type keywords, making definitions clean. The language is very simple. Coding in Python is a lot like writing an email to your computer.

In the following section we will install Python and then go over the basics!

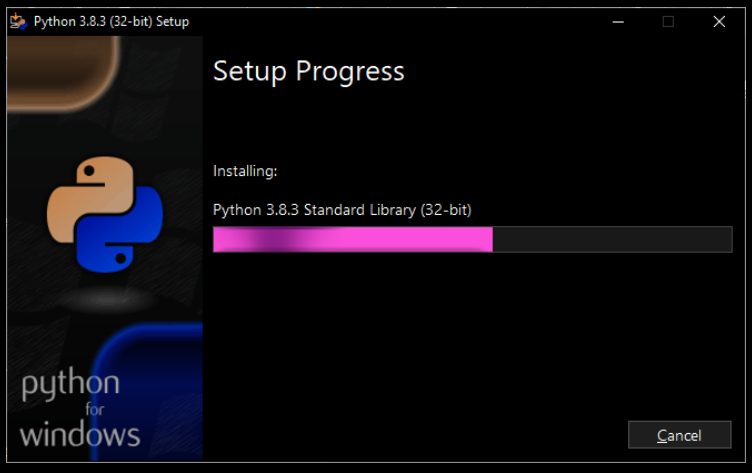
Chapter 14

Installing Python

Before starting programming in Python you need to install it on your system. I won't go into great detail on how to install packages of applications on your operating system because primarily this book is about Python the language. But this section will give you a few pointers in the right direction.

14.1 Installing on Windows

Head over to `https://www.python.org/downloads/` and download Python executable. Launch it and go through simple setup process:



At the time when I started writing this book latest version of Python was `3.8.3`. Head over to `python.org` and install the latest version.

If you already have Python installed, you can check what version you have by executing the following command:

```
C:\Python>python --version
Python 2.7.16

C:\Python>
```

Earlier versions of Python might not support some of the examples in this book. If you have an old version it is recommended to upgrade to latest.

14.2 Installing on Mac

Mac already comes prepackaged with Python. Which version of Python was originally installed with the OS will depend on the version of your operating system.

The Mac's system installation path is:

```
/System/Library/Frameworks/Python.framework
```

and:

```
/usr/bin/python
```

Earlier versions of macOS such as the OS X came with Python 2.7. While it's great for practice it is recommended to upgrade to latest version.

However, if you choose to upgrade keep in mind that you will end up with two different versions of Python on your system. They will both work but may not be able to run the same Python scripts – it's possible that some of the later Python versions such as Python 3.9 or Python 4.0+ wouldn't compile code written in some of the previous versions of the Python compiler because not all versions of Python are backward compatible.

14.2.1 Checking current Python version

To check if you have Python already installed open Terminal and type the following command (at the $ command line symbol.)

```
$ python --version
Python 2.7.5
```

For Python 3.0+ you can also try typing `python3 --version` command:

```
$ python3 --version
Python 3.8.3
```

If for some reason the commands typed above result in an error it means you don't have Python installed on your system.

This section assumes familiarity with Mac OSX command line installation tools. Depending on the OS version it may be `homebrew`:

```
brew install python3
```

Or use `apt` and `apt-get` installers if you're on Linux. Don't forget to `apt-get update` first:

```
sudo apt-get update
sudo apt-get install python
```

But this might install the latest version on your current Linux build. To install a specific version, for example 3.8, you can execute:

```
sudo apt-get update
sudo apt-get install python3.8
```

This book won't go into much more detail about the installation process on different versions of Mac or Linux operating systems. It's assumed the reader is already somewhat familiar with the process.

14.2.2 Installing GCC

One more thing. Python's original implementation is called CPython. It was written in the C language and it's what compiles your Python program into bytecode and later interprets that bytecode. CPython is both an interpreter and a compiler. Long story short, we need to install GCC to run Python.

If you already have Xcode installed chances are you already have GCC installed on your Mac. Installing Xcode is one of easiest options for getting GCC on a Mac.

Simply search for Xcode on Mac App Store page:

```
https://www.apple.com/app-store/
```

14.2.3 Installing XCode command line tools

If you're installing Xcode from scratch it's helpful to run the following command:

```
xcode-select --install
```

This will add the command line tools.

14.2.4 The OXC GCC installer

Without Xcode you can look into:

```
https://github.com/not-kennethreitz/osx-gcc-installer
```

For instructions on how to install the stand-alone GCC.

It's advised not to install both Xcode and the stand-alone GCC together. This combination might complicate your installation and result in configuration problems that are hard to resolve. For this reason installing both is often avoided by Python developers on the Mac.

14.3 Using `python` command

Now that Python is installed on your system you are ready to use `python` executable to run your Python scripts.

Navigate to the directory from your command prompt (Windows) or Terminal (Mac) or bash (Linux) where you installed Python. Create and save your Python script into a file `script.py` and then execute the following command:

```
python script.py
```

This will execute your python script.

14.4 Running `python` scripts system-wide

On Mac or Linux Python will likely be available for running from any location on your hard drive in Terminal or bash command line soon after the installation because generally it's installed globally. But it may not be the same on Windows.

New projects are likely to be stored in a unique directory.

On Windows for example it could be: **C:\develop\my_app\main.py**. In order to execute `main.py` from command line we must first navigate there (using the `cd` command) to let `python` know where the script is.

But in some cases we won't be able to run Python outside of its own installation folder. On Windows you might run into an error if the location of `python.exe` is not also present in the `PATH` environment variable.

Some Python installation software will automatically add `python.exe` to `PATH`. But sometimes it won't happen. So let's make sure **python.exe** exists in `PATH` variable so it can be executed from anywhere on the command line.

The solution is to include path to **python.exe** to your **Environment Variables**.

There are at least two different ways to access Environment Variables on Windows.

The easiest one is to **hold Windows key** and **press R**.

The following "Run" dialog will appear:

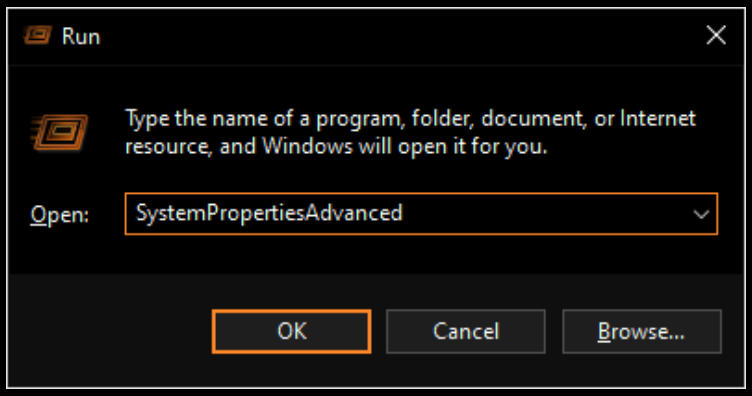
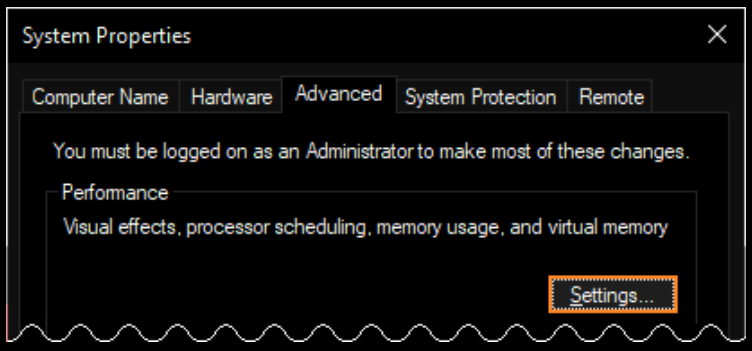
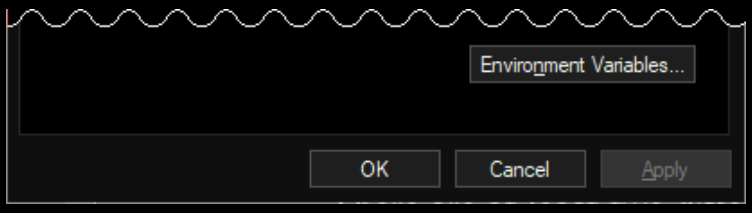


Figure 14.1: Hold the Windows key and press R.

Type "SystemPropertiesAdvanced" and press Enter. The Environment Variables editor window will appear:



Click on Environment Variables button in lower right corner...



You can access the same window from command line:

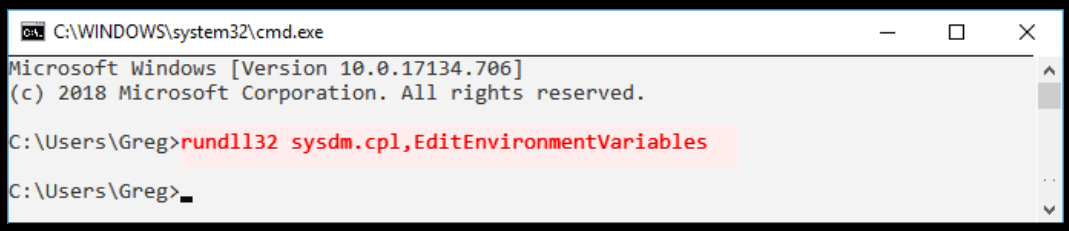


Figure 14.2: **Running:** rundll32 sysdm.cpl,EditEnvironmentVariables

Running the command **rundll32 sysdm.cpl,EditEnvironmentVariables**

Either way, you will arrive at the following window:

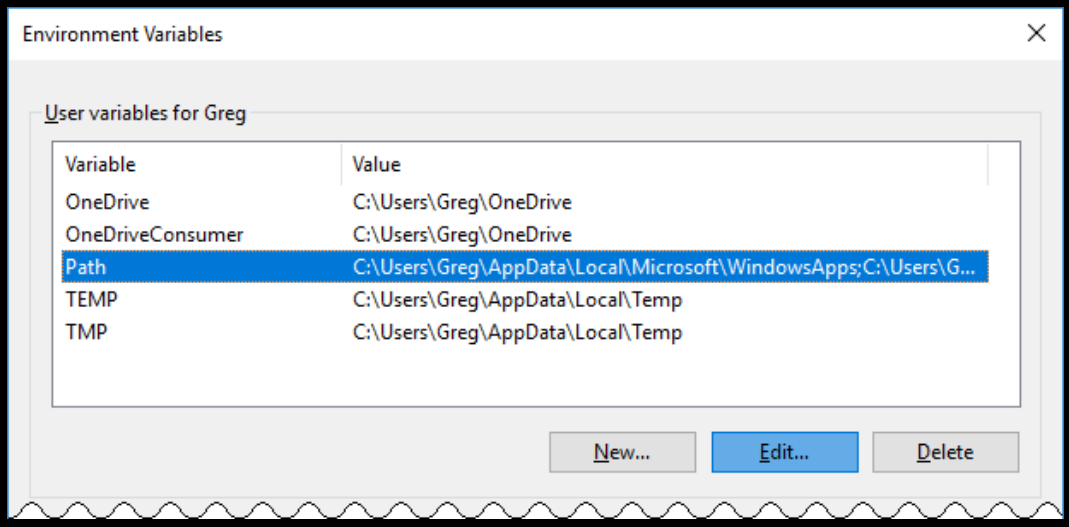


Figure 14.3: Upper half of **Environment Variables** window.

In the *upper* box of **Environment Variables** window (where it says user variables for **YourUserName**) click on **Path** and click **Edit...** button.

Find out the path where you installed Python (the place where **python.exe** is located). In this example I'll use mine which is simply `C:\Python38-32\`.

In next window, click **New** and type:

```
C:\Python38-32\
```

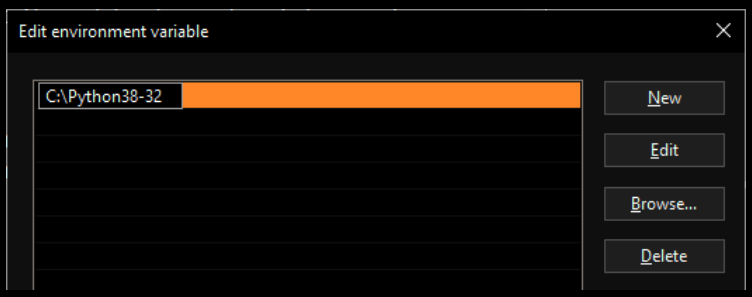


Figure 14.4: You will probably see other variables on the list. Just add path to your Python installation directory.

Click Ok button and you're all set. You can now run Python from anywhere on your hard drive, not just its installation folder.

14.5 Using Python Shell

Shell is a sandbox environment for executing Python statements.

To enter Python shell simply type `python` on command line and hit Enter:

```
C:\>python
```

You will be greeted by intro showing your current Python version and what type of process and operating system you're running on:

```
Python 3.8.0 (Nov 14 2019, 22:29:45) [GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more info...
>>>
```

The >>> is where you will enter your Python commands.

Exiting from Python Shell

To exit back to the Command Prompt, press `Ctrl-Z` and hit `Enter`.

Everything you type into the shell will be erased on exit. So it's only good for experimenting with Python commands just to see how they work. This is similar to the JavaScript developer's console in Chrome.

Executing Python Scripts

In order to actually execute a Python script program you need to first save your Python code into a file with `.py` extension.

After doing that, you can run `python filename.py` command on that file. Starting from next chapter it is assumed that all examples will be executed in this way.

14.6 `python` command usage

Type:

```
python -h
```

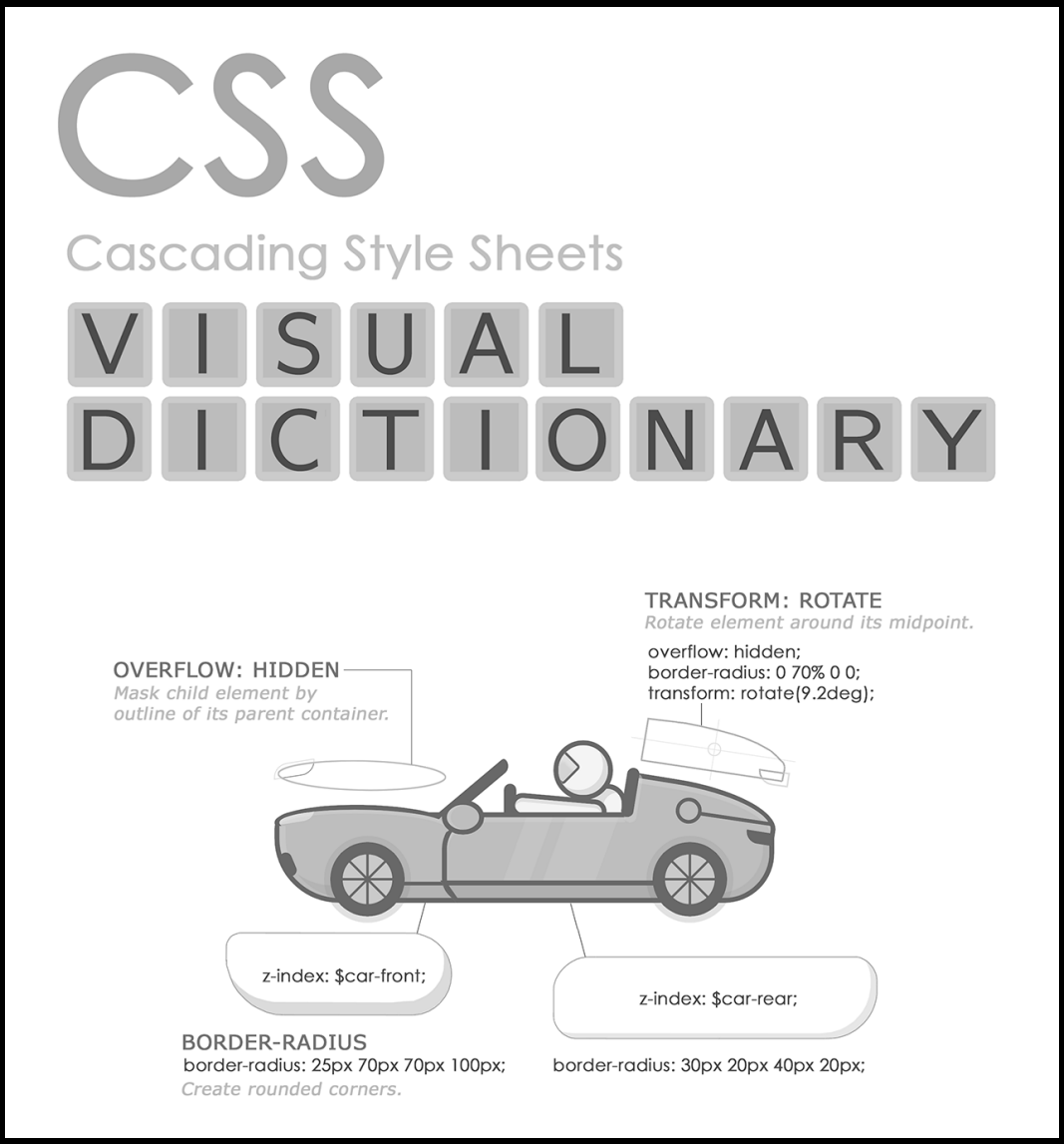
To see all available modes available for executing python scripts.

Chapter 15

Other Web & Software Development Books

15.1 CSS Visual Dictionary

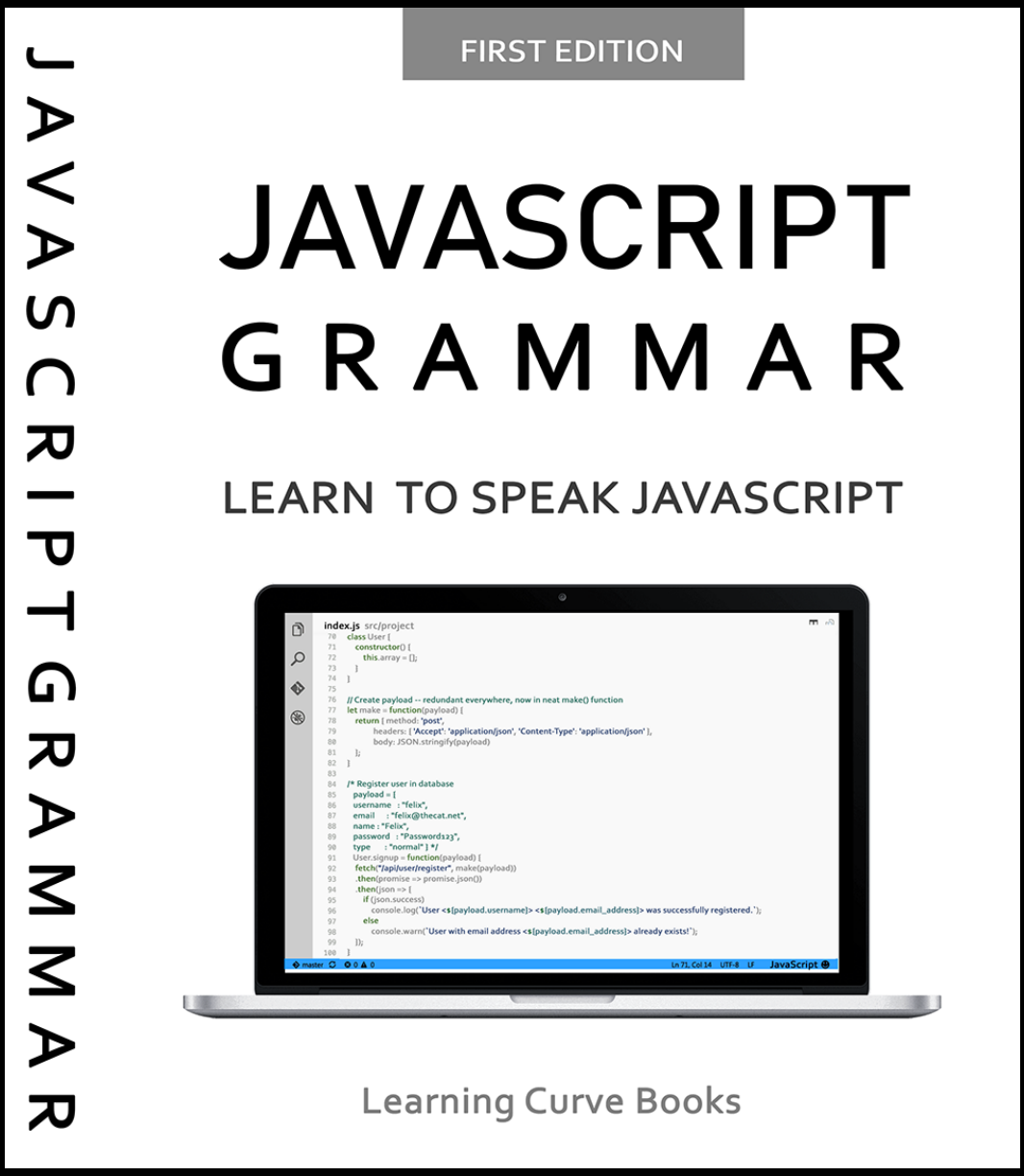
If you enjoyed Python Grammar, look for other books by Learning Curve Books.



Look for a copy of **CSS Visual Dictionary** on Amazon. (cover art may vary.)

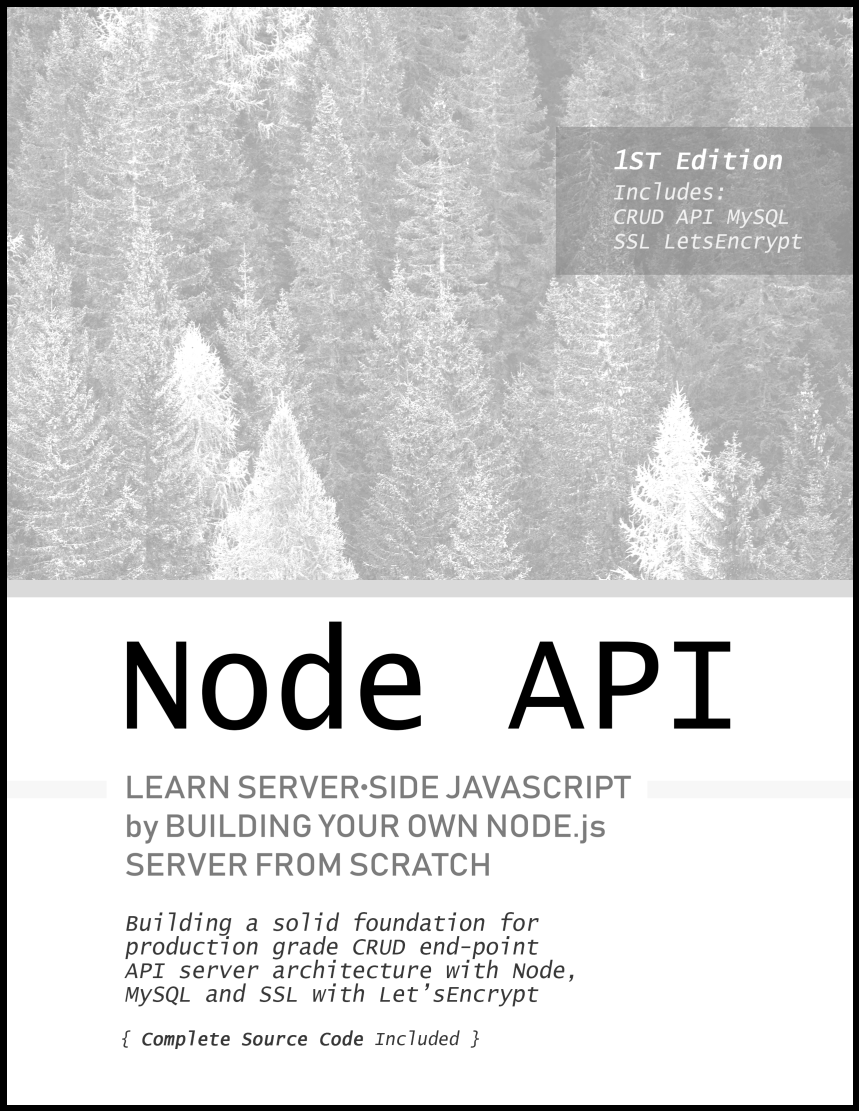
15.2 JavaScript Grammar

If you enjoyed Python Grammar, look for other books by Learning Curve Books.



Look for a copy of **JavaScript Grammar** on Amazon. (cover art may vary.)

15.3 Node API



Look for a copy of **Node API** on Amazon. (cover art may vary.)

Index