PYTHON

Programming 1

Source: https://github.com/ucll-programming/course-material

22. January 2024

Matej Vesel

Table of Contents

1. Introduction

1.1. Python shell

A shell is a piece of software that allows us to communicate with the operating system. For example, on Windows, the desktop, the taskbar and File Explorer all are part of Windows's shell. However, there's also the command line interface. This is a text-based way of interacting with your OS. For example, instead of using File Explorer to view your files, you use the ls command. F From now on, when we talk about the shell (or the terminal), we mean this command line interface.

1.1.1. Multiple Shells

A text-based shell is an application just like any other. Anyone can make their own shell, which is why there are many choices around:

- cmd.exe is severely outdated and should really be avoided.
- PowerShell
- Bourne-Again Shell (Bash)
- Z shell

We suggest you rely on Bash (Windows and Linux) and Z Shell (MacOS).

1.1.2. Launching the Shell

Visual Studio Code allows you to run the shell inside it:

- Open the Terminal menu.
- Select New Terminal.

Make sure to select the right shell. You can control which shell is created using the little arrow next to the + button. Python has a shell of its own. It allows you to type in Python commands, which are then executed on the spot. You can launch the Python shell from the "regular" shell using py, python or python3, depending on your OS.

```
$ py
Python 3.11.2 (tags/v3.11.2:878ead1, Feb  7 2023, 16:38:35) [MSC v.1934 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The OS shell and the Python shell are two completely different beasts. Do not confuse the two!

```
$ some OS shell command to type in
>>> some Python code
```

1.2. Python as Calculator

In essence, your machine is a calculator, albeit a very fancy one. Let's start with some simple things. Have a Python shell ready. Whenever we ask you to type something in the shell, make sure to leave out the >>> part. This is the prompt and indicates that what follows is what *you* have to enter. For example, >>> 5 means "please enter 5 followed by enter". Lines not starting with >>> are responses from the computer. Lines starting with # are merely comments. If you type them in, they will be ignored by the computer. Enter the following in the shell.

```
>>> 5
5
```

So, if you enter 5, Python will simply answer 5. Hardly impressive. Let's try something more daunting:

```
>>> 5 + 2
7
```

You can see that Python is not just a mere parrot: it actually evaluates your input and prints out the result. Let's see what else we can do...

```
# Multiplication (this line is merely a comment, you don't have to copy it)
>>> 8 * 7
56

# Exponentiation
>>> 2 ** 8
256

# Combination
>>> 479 * (1574 + 646)**3
5240761992000
```

Python can do a lot more than just calculations. In fact, there are very few limitations on what you can do: small scripts, games, server software, ... are all within reach. Admittedly, Python can be a bit slow compared to other languages, so don't get ready to create your AAA game yet. Python's power lies in the fact that it can do (almost) everything and that it allows you to do so with relative ease.

1.3. Scripts

While it is technically possible to develop your Python projects using the shell, it would be a rather awkward experience. Instead, Python code is typically written in a file (typically called a Python script) after which you tell Python to execute all commands written in that script. Using VSCode, create a file named my-script.py and add the following code to it:

```
1 + 2
```

1.3.1. Running your Script

Now that you've written a Python script, you'll want to tell Python to run it. Open a terminal with Git Bash (or another OS shell, as explained earlier) in the same directory as my-script.py and enter the following commands:

```
# If you're on Windows
$ py my-script.py
```

```
# If you're on MacOS
$ python3 my-script.py

# If you're on Linux
$ python my-script.py
```

Hmm, it does not appear to do anything. We kind of hoped it would output 3... When using the Python shell, Python will always print out the result of the last evaluation. This is not true for a script. This is also makes perfect sense: imagine playing a game while having Python print out the results of all millions of calculations performed. So, when using scripts, you have to explicitly ask Python to print out the result. Change the contents of my-script.py to

```
print(1 + 2)
```

Now run it again.

```
# Use the right command for your OS
$ py my-script.py
3
```

2. Arithmetic

2.1. Functions

An algorithm is a series of instructions that, when followed, achieve a certain goal. Cooking recipes are an example of algorithms: follow the recipe's steps and you'll end up with a (hopefully) delicious dish. A programming language, such as Python, allows you to write down instructions in a way that your computer can understand them. Recipes are given names: chili con carne, cacio e pepe, larb kai, etc. The same can be done for algorithms in programming languages: we bundle the instructions together and give them a name. Such a named group of instructions is called a function.

```
def my_function():
    instruction1
    instruction2
    instruction3
    ...
```

The code shown above defines a function named my_function.

2.1.1. Return Values

When following a recipe, you end up with some dish. Similarly, functions can also produce a result. This result is called the return value of the function.

```
def zero():
    return 0
```

Here we have defined a very simple function that produces 0 as return value.

2.1.2. Calling a Function

Once you have defined a function, you can call it. This means that you want all instructions contained within the function to be executed. Calling a function is done with function_name(). You can test this out in the Python shell:

```
>>> def zero():
...     return 0

>>> zero()
0
```

2.2. Parameters

Functions are everywhere:

- Google is in essence a function that returns a list of web pages.
- A satnav is a function that returns the shortest path to a certain destination.
- A chess AI is a function that returns the best move.
- And so on.

As of yet, we've been focusing on the *output* of functions, i.e., on their return value. Functions often also require inputs:

- Google needs a series of keywords.
- A satnav needs starting location and a destination.
- A chess AI needs the current state of the chess board and who it's playing as.

A function takes inputs in the form of parameters. This is what it looks like in Python form:

```
def function_name(parameter1, parameter2, ...):
    instruction1
    instruction2
    ...
    return return_value
```

If we were to translate our examples above into Python functions, we'd get

```
def google_search(keyword_list):
    ...
    return pages

def satnav(departure, arrival):
    ...
    return shortest_path

def chess_ai(chessboard, ai_color):
    ...
    return best_move
```

2.2.1. Calling a Function with Parameters

If you want to call a function with parameters, you have to provide values for each of these parameters.

```
satnav("Brussels", "Rome")
```

satnav has two parameters. As departure we specified Brussels, and arrival has been set to Rome. satnav will then compute and return the shortest path between those two cities.

The values (Brussels and Rome) we pass to a function as inputs are called arguments.

Below is an example of a very simple function:

```
def double(x):
    return 2 * x
```

2.3. Identifiers

Functions and parameters need names, otherwise you wouldn't be able to refer to them. These names are also called identifiers.

In this section, we discuss the rules about identifiers.

2.3.1. Python Rules

Python, like all programming languages, imposes strict rules on what makes a valid identifier. The actual rules are a bit complicated; we give you a simplified summary instead:

- There is no length limit on identifiers.
- An identifier can be any combination of letters (both upper and lowercase), digits and underscores, with the exception that the first character cannot be a digit.
- Identifiers are case sensitive: abc and ABC are completely unrelated identifiers as far as Python is concerned.

Not following these rules will cause Python to reject your code.

Conventions (običaj, navada)

Besides to the Python rules there are also naming conventions. Python will not complain if you do not follow these, but programmers are expected to adhere to them (držati se česa) nonetheless.

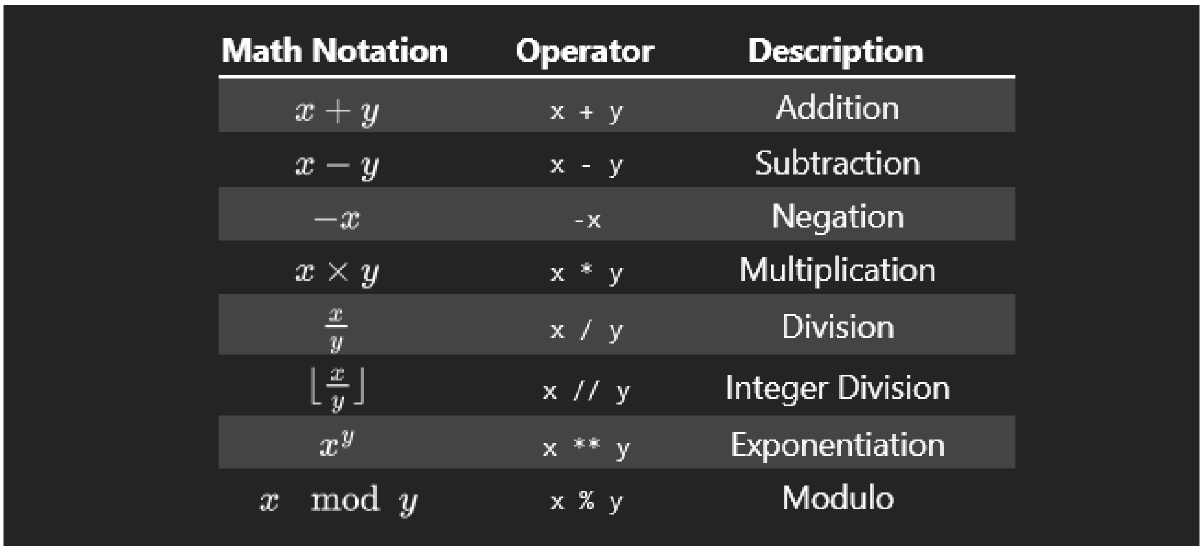
- Use descriptive names.
- All letters in an identifier should be lowercase.
- If an identifier consists of multiple words, these words should be separated using underscores (_).

Note that later we will see more Python concepts such as classes for which there will be a different naming convention.

Identifier	Description
determine_fee	Valid
DetermineFee	Violates naming conventions
Determine-Fee	Violates Python's syntax rules

2.4. Operators

Addition, subtraction, negation, multiplication, division, integer division, exponentiation, modulo.



Math Notation	Operator	Description
$x + y$	`x + y`	Addition
$x - y$	`x - y`	Subtraction
$-x$	`-x`	Negation
$x \times y$	`x * y`	Multiplication
$\frac{x}{y}$	`x / y`	Division
$\lfloor \frac{x}{y} \rfloor$	`x // y`	Integer Division
x^y	`x ** y`	Exponentiation
$x \mod y$	`x % y`	Modulo

2.5. Numbers

Your computer has two ways of representing numbers: integers and floating point numbers.

- Integers can represent arbitrarily large *whole* numbers such as 0, 1, 803980 and -34890840, but not decimal numbers like 1.5. However, integers are quite efficient.
- Floating point numbers are also able to represent whole numbers, but also support decimal numbers such as 0.1, 1.5, 3.1415. While it may seem floating point numbers can represent any number, they have their limitations. The rules are too complicated to go into detail here. Working with floating point numbers is typically also less efficient.

So, when it comes to whole numbers, such as 5, you can choose between two internal representations:

- The integer representation is simply written 5.
- The floating point representation is written 5.0.

For example, you can ask Python which type of number you are dealing with:

```
>>> type(5)
<class 'int'>

>>> type(5.0)
<class 'float'>
```

From now on, we'll use the Python names int and float to refer to the different representations.

2.5.1. How To Choose

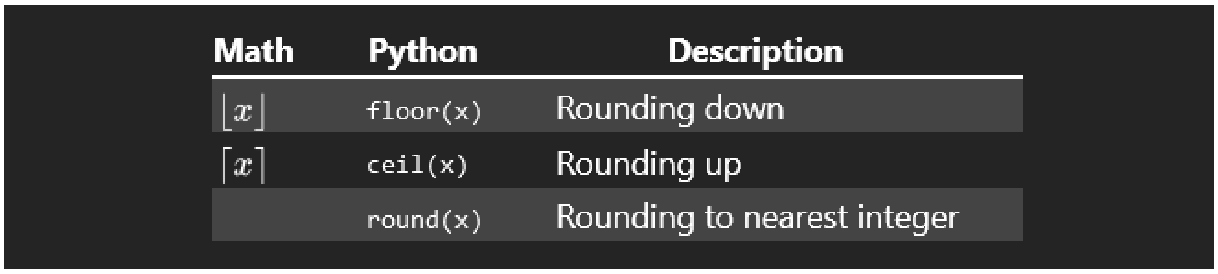
If you're wondering which type of number (int of float) to use, here's a rule of thumb: use int whenever you can, and only float when you actually need decimal numbers. An example of the limitations of floating point numbers:

```
# Everything okay here
>>> 2 * 0.1 == 0.2
True

# Uhh?
>>> 3 * 0.1 == 0.3
False

# Okay, things make sense again
>>> 4 * 0.1 == 0.4
True

#
>>> 5 * 0.1 == 0.5
True

>>> 6 * 0.1 == 0.6
False

>>> 7 * 0.1 == 0.7
False
```

There are plenty of sources that explain this behavior. For your convenience, here's one.

2.6. Rounding

You can turn a floating point number into an integer by rounding it. There are three ways to perform rounding:



Math	Python	Description
$\lfloor x \rfloor$	`floor(x)`	Rounding down
$\lceil x \rceil$	`ceil(x)`	Rounding up
	`round(x)`	Rounding to nearest integer

floor and ceil are not available by default: you first need to import them using the following code:

```
# Imports are generally placed at the top of the file
from math import floor, ceil
```

math is a module that contains all kinds of math-related functions. There are many more modules: Python itself comes with over 200 modules. Added to this, you can download many more from PyPI (more than 350k are available at the time of this writing).

Let's see the three functions in action:

```
>>> from math import floor, ceil
>>> floor(4.9)
4

>>> ceil(2.4)
3

>>> round(8.49)
8

>>> round(8.51)
9
```

2.7. Floor Division

2.7.1. Types of Division

Python provides two division operators:

- True division, written /.
- Floor division (also called integer division), written //.

2.7.2. True Division

True division is the one you're familiar with from mathematics:

```
>>> 4 / 2
2
```

```
>>> 5 / 2
2.5
```

As you can see, the result is an int if it can accurately represent the result, which is the case in the first example. If the result is not a whole number, Python will switch over to floats, as shown in the second example.

2.7.3. Floor Division

When using floor division, you will always end up with a whole number:

```
>>> 4 // 2

2


>>> 5 // 2

2
```

Whenever the division's "real result" is not a whole number, it is rounded down: in the case of 5 // 2, the "real" result 2.5 is being rounded down to 2.

You can interpret a // b as the answer to the question "how many times does b fit in a?". You'll find that floor division comes in handy more often than you might think.

If you're wondering if there's a difference between a // b and floor(a / b):

- The types of the results can be different: 5.0 // 3.0 evaluates to 1.0 whereas floor(5 / 3) evaluates to 1.
- When your operands are ints to begin with, // is the faster choice. You can run demo-benchmark-div.py to see for yourself.

2.8. Min/Max

2.8.1. Built-In Functions

Python comes with a number of built-in functions. We focus here on two of them: min and max.

min and max are functions that compute the minimum and maximum of their parameters, respectively. Most functions expect a specific number of arguments, but min and max are known as *variadic functions*, meaning they can take any number of parameters. We'll see later how to define your own variadic functions.

```
>>> min(4, 9)
4

>>> max(2, 6, 4, 7, 5, 4)
7
```

2.9. Local Variables

Consider the following code:

```
def some_calculation(a, b, c):
    return min((-b-(b**2 - 4 * a * c)**0.5)/(2*a), (-b+(b**2 - 4 * a *
c)**0.5)/(2*a))
```

Such a monolithic mathematical expression is quite intimidating and hard to read. We can improve the readability by storing intermediate results in local variables as follows:

```
def some_calculation(a, b, c):
    d = b**2 - 4 * a * c
    sqrt_d = d ** 0.5
    x1 = (-b-sqrt_d)/(2*a)
    x2 = (-b+sqrt_d)/(2*a)
    return min(x1, x2)
```

Here, d, sqrt_d, x1 and x2 are local variables. They're called local because they only exist within the bounds of the function. No one outside the function can access them. The same holds true for parameters: these are also only accessibly from within the function. The region of code in which some variable x is visible is called the scope of x. Python is function scoped: functions form the boundaries of the scope of their variables.

```
def foo(a):
    b = 5
    return bar()


def bar():
    return a * b
```

In the code above, we have a function foo. It takes a parameter a and introduces a local variable b. It also calls bar, which returns the product of a and b. However, this code is invalid: a and b are variables that are only visible from within foo. If you wish bar to have access to a's and b's value, you have to pass them as parameters:

```
def foo(a):
    b = 5
    return bar(a, b)


def bar(a, b):
    return a * b
```

2.10. Assignment

In a previous section, you were shown how you could introduce local variables to store intermediate results:

```
def my_function():
    y = 2
    print(y)
```

Calling my_function() will print out 2.

You can also overwrite a variable's value using the same syntax:

```
def my_function():
    y = 2
    print(y)    # Prints 2
    y = 3
    print(y)    # Prints 3
```

2.10.1. Sequential Evaluation

Some students misinterpret how assignment works. Consider the following code:

```
def my_function():
    x = 1
    y = 2 * x
    print(x, y)
    x = 10
    print(x, y)
```

Take a good look at the code and predict what my_function() will print before continuing reading.

⋮

- The first line is straightforward: it introduces a new local variable `x` and sets it to `1`.
- The second line is where confusion can arise. The exact interpretation is "Evaluate 2 * x and store the result in y". At this point in time, `x` equals `1`, so `2 * x` evaluates to `2`, which is stored in `y`.
- The print on the third line writes `1 2` to the console.
- `x = 10` assigns a new value to `x`. Here, it is crucial to understand that `y` is left unchanged.
- The second print writes `10 2` to the console.

Some people, possibly inspired by Microsoft Excel, think that y = 2 * x introduces a "relationship" so that y is to be equal to 2 * x at all times. In their minds, x = 10 causes y to magically be updated to 20 as a way to preserve this relationship. While this is certainly a valid interpretation of how things *could* work, in Python this is *not* how it works!

If you are confused by this section, make sure to ask the lecturer for more explanations.

2.10.2. Augmented Assignment

Take a look at this line of code:

```
x = 4
x = x + 1
print(x)
```

Mathematical minds might be confused by the second assignment: how can x be equal to x + 1? That looks like circular reasoning!

However, as a programmer, you should know how to interpret this assignment:

- First, evaluate the right hand side: take the current value of x and add 1 to it, which yields 5.
- Next, assign this result to x.

In other words, executing these two lines of code will cause 5 to be printed out.

This pattern of reading and writing to the same variable (e.g., x = x + 1, y = y * 2, etc.) occurs quite often in code. For this reason, a shorthand notation has been introduced:

```
x = x + 1
# can also be written as
x += 1

y = y * 2
# can also be written as
y *= 2
```

For most operators, this shorthand x op= y notation exists.

In case you're already familiar with certain other programming languages, know that Python does not support the i++ notation. You'll have to write i += 1.

2.11. Modulo

The modulo operator % is probably new to you. In short, a % b computes the remainder when you divide a by b. We give an example to make the concept clearer.

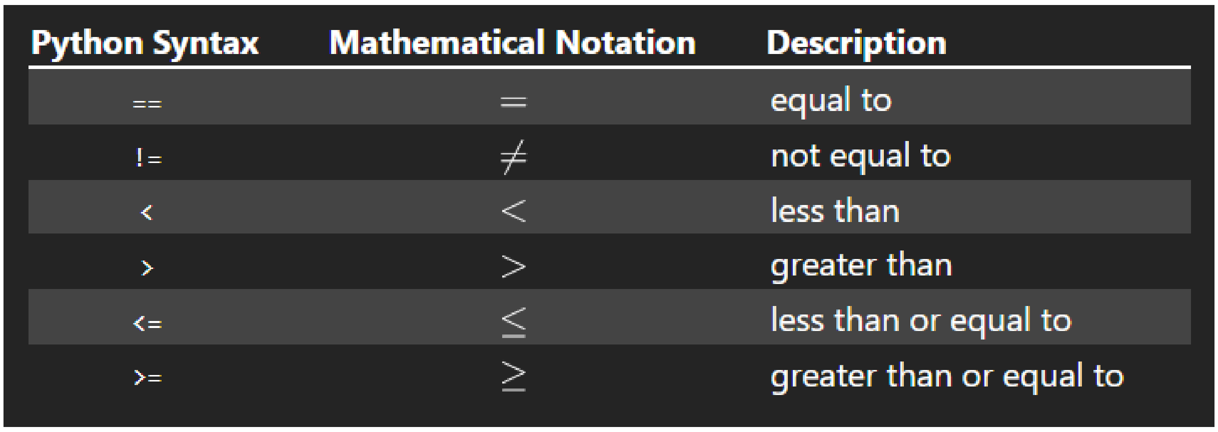
An web store shows 20 items per page. In total, there are 114 items. How many items are on the last page?

114 // 20 computes the number of times 20 fits in 114, i.e., the number of pages that are full. We need an extra page for the leftover items. 114 % 20 yields this number: 14.

3. Booleans

3.1. Comparisons

In the previous section, you were introduced to all kinds of operators and functions: +, -, *, min, max, etc. These all have one thing is common: they all operate on numbers and yield numbers as result. Let's take a look at a series of other operators, which should be very familiar to you:



Python Syntax	Mathematical Notation	Description
`==`	$=$	equal to
`!=`	$\neq$	not equal to
`<`	$<$	less than
`>`	$>$	greater than
`<=`	$\leq$	less than or equal to
`>=`	$\geq$	greater than or equal to

What would the result be of such a comparison? If you were to open a terminal and enter a comparison, you would get

```
>>> 1 == 1
True

>>> 1 == 2
False
```

True and False are a new type of value called booleans. As with integers, you can store them in variables, pass them as parameters and return them from functions. Numbers have all kinds of operations defined on them, such as addition, subtraction, multiplication, etc. Booleans have their very own set of operators: and, or and not. We will discuss them in detail later. Make sure to notice the difference between = and ==:

- a = b takes the value of b and *assigns* it to a.
- a == b compares the values of a and b, leaving both variables unchanged.

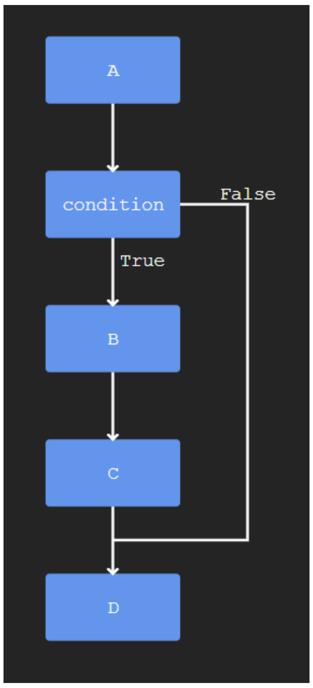
It is a common error to confuse the two operators. Confusingly, Python actually allows you to apply +, -, *, ... on boolean values. But we beg you, please resist the temptation to, for the following reasons:

- Python happens to be one of the few languages (e.g., JavaScript, C, C++) that allow this, but most others (e.g., C#, Java, Ruby, Go, Rust, Kotlin, Scala, Racket, ...) will consider it an error. We prefer you to learn to write code that doesn't rely on tricks that only work in very limited cases.
- To others, using +, -, etc. will automatically create the impression that you're working on numbers, thus confusing them. Writing readable code is your number two priority (your number one priority being that your code should behave correctly.)

So, in summary: we ask you to just assume +, -, *, etc. are not usable on boolean values.

4. Conditionals

4.1. If



Some instructions only need to be executed when certain conditions are satisfied.

- Award a point only *if* the answer to the question is correct.
- Give discount only *if* the order amounts to more than €100.
- Show emails only *if* password is correct.

Python provides the *if statement* to allow you to specify that certain instructions should be executed conditionally.

```
instruction A

if condition:
    instruction B
    instruction C
    # ...

instruction D
```

The code above can be visualized as the diagram to the right shows. When Python encounters an if statement, it will evaluate the condition. If it evaluates to True (or some truthy value, see later), the instructions inside the if (the indented lines) will be executed. If, however, condition evaluates to False (or some falsey value, see later), these instructions will be skipped and execution proceeds with whatever follows the if statement.

4.1.1. Return

Up until now, your functions have consisted for a series of instructions ending in a return statement. Consider the following code:

```
if condition:
    return 1
return 2
```

As you can see, it is possible to have return statements inside an if statement. It is important for you to know that a return statement cuts short the execution of a function. In other words, the moment execution encounters a return, all other instructions in that function are skipped. This means that in the code shown above, if the condition evaluates to True, 1 will be returned.

4.1.2. Truthy and Falsey Values

Ideally, the if's condition evaluates nicely to True or False. Some language insist on that and will report an error if you present the if with any other value. Python, however, is more flexible and allows any value, such as integers or any of the other types of values we'll encounter in the future. So, what happens were you to write this code?

```
if 5:
    instruction
```

Python makes the distinction between *truthy* and *falsey* values. A truthy value is any value that will be interpreted as True when used in an if. Likewise for falsey values, which will be considered False. This raises the question, is 5 a truthy value, or a falsey one? In the case of integers, all of them are truthy, except for 0, which is falsey. So in the example above, instruction will be executed.

4.2. Else

4.2.1. If Else

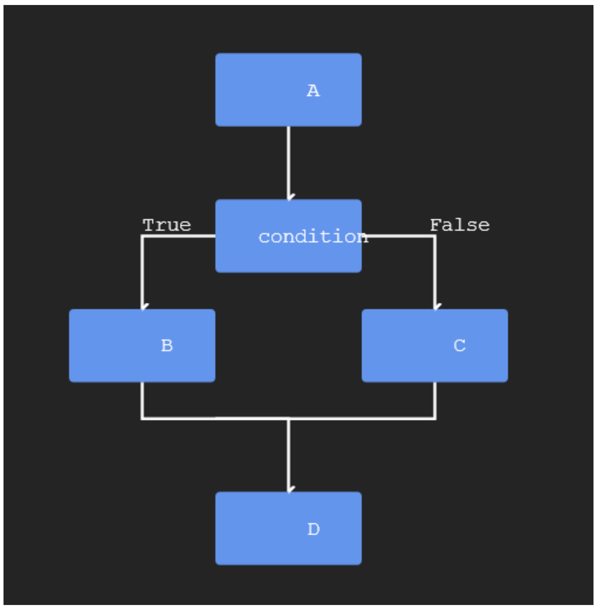
It often happens that you want some series of instructions to be executed when some condition holds, but some other series of instructions if the condition does not hold. You can write this as

```
instruction A
if condition:
    instruction B
if not condition:
    instruction C
instruction D
```

There is a shorter, more efficient way to achieve the same:

```
instruction A
if condition:
    instruction B
else:
    instruction C
instruction D
```

Visualized, this is equivalent with:



4.3. Elif

An if can be seen as a fork in the road: you can either go "left" or "right". Sometimes, however, there are more than two execution paths to choose from. Say, for example, you are writing a chess game and need a function that determines if a move is valid. Whether a move is valid depends on the piece: pawns, rooks, knights, bishops, queens and kings all have a different movesets. Your function will have to differentiate between all the different pieces.

```
if piece_type == pawn:
    # deal with pawn
else:
    if piece_type == rook:
        # deal with rook
    else:
        if piece_type == knight:
            # deal with knight
        else:
            if piece_type == bishop:
                # deal with bishop
            else:
                if piece_type == queen:
                    # deal with queen
                else:
                    # deal with king
```

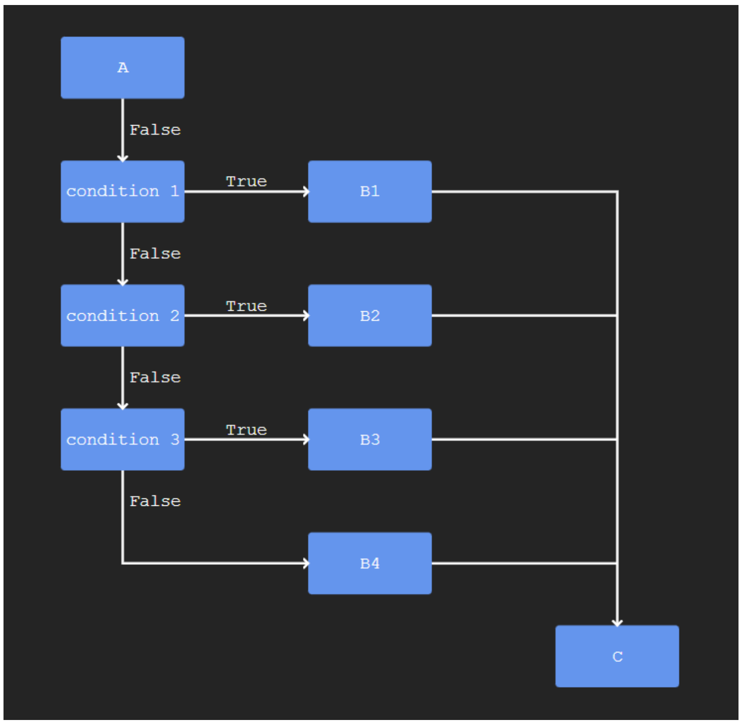
While technically correct, this is quite awkward to write. Luckily, there's a cleaner solution:

```
if piece_type == pawn:
    # deel with pawn
elif piece_type == rook:
    # deel with rook
elif piece_type == knight:
    # deel with knight
elif piece_type == bishop:
    # deel with bishop
elif piece_type == queen:
    # deel with queen
else:
    # deal with king
```

As you can see, using a if-elif-else, you can easily deal with different execution paths. More generally,

```
instruction A
if condition 1:
    instruction B1
elif condition 2:
    instruction B2
elif condition 3:
    instruction B3
else:
    instruction B4
instruction C
```

Can be visualized as:



Only one of the branches will be executed, namely the first one whose condition evaluates to a truthy value. For example:

```
x = 5

if x > 3:
    result = 1
elif x > 4:
    result = 2
else:
    result = 3
```

Here, result will end up with the value 1.

5. None

Python has a special value called None which is meant to represent "a missing value". For example, if a function requires the handle to your Twitter account as a parameter, but you have no account there, None can be used to indicate this.

```
def register_user(name, email_address, twitter_handle):
    # ...
# John has no Twitter account
register_user("John Jackson", "john.jackson@gmail.com", None)
```

None is quite a boring value. The only useful operation you can use on it is comparison:

```
if variable == None:
    # ...
```

Technically, if you want to check if a variable contains None, it is better to use the is operator:

```
if variable is None:
    # ...
```

What is does exactly will be discussed later. Using == will also work, but it is less efficient.

5.1.1. Return Value

Consider the following function:

```
def say_hello():
    print("Hello")
```

There is no return statement, which is perfectly valid. However, functions always return a value in Python. In this case, because no return value was specified in the code, the function simply returns None. In other words, the code above is equivalent to

```
def say_hello():
    print("Hello")
    return None
```

6. Strings

6.1. Introduction

Besides numbers, programs also need to be able to deal with text: passwords, messages, HTML pages, ... are all forms of text. Programming languages' answer to this need are strings.

```
message = "Hello world!"
```

Here, the string "Hello world!" is assigned to the variable message. As you can see, strings are delimited by double quotes "...". The quotes themselves are not part of the string, they merely indicate where it starts and end. Python also allows the usage of single quotes: 'Hello world!' is the exact same string. Strings have all kinds of operations defined on them. We'll introduce them in the next series of exercises.

6.2. String Interpolation

Programs often need to construct strings at runtime. Python offers an easy way to do this.

```
item_count = 5
day = 23
month = 5
message = f"Your order containing {item_count} items will be delivered on
{day}/{month}"
```

After executing this code, message will be set to

```
Your order containing 5 items will be delivered 23/5
```

Building a string by "injecting" values is called *string interpolation*. In order to activate string interpolation, you must prefix the string literal with f. Without the f, no "injections" will take place and you will end up with literally "Your order containing {item_count} items will be delivered on {day}/{month}". You're not limited to putting variable names between the { }: expressions will also work. f"{x} + {y} is equal to {x + y}" will produce the expected result.

6.2.1. Formatting Options (Format Specification Mini-Language)

It is possible to tell Python how the injected values should be formatted using the `{expr:formatting}` syntax. There are many options; you can find an overview here. We give a few examples below. In practice, you should simply be aware that it is possible to specify a format. When you need to format things a certain way, you should be aware you can check if there's a format specifier doing exactly what you need instead of trying to do the formatting manually. A time of day is typically formatted 09:07:03 and not 9:7:3, i.e., the numbers should always count two digits. This can be achieved by adding :02:

- The 2 means the number should at least be 2 characters long.
- The preceding 0 indicates we want extra 0's to be added in case the number is too short.

```
>>> h = 1
>>> m = 2
>>> s = 3
>>> f'{h:02}:{m:02}:{s:02}'
"01:02:03"
```

It is possible to have floating point numbers show a specific number of decimals:

```
>>> pi = 3.141592
>>> f'{pi:.2f}'
"3.14"
```

You can left align, right align and center:

```
>>> text = 'abc'
>>> f"{text:<10}|{text:>10}|{text:^10}|"
'abc       |       abc|   abc    |'
```

6.3. Console Input

print allows you to output data to the terminal. You can also *receive* data using input, making it possible to interact with the user.

```
name = input("Enter your name: ")
```

Input will always return a string. If the user enters 15, your program will receive "15". Soon we will see how to convert this string to an actual integer 15.

6.4. Runnable Scripts

As of yet, you've been writing function after function and storing them in a .py file. But how does one make an actual application, like a game? Surely, launching a game can't involve having to open a Python shell and calling some function? There has to be a better way. Can we run a Python program just like we can start Chrome, Visual Studio Code, Netflix, etc.? Good news, everyone: the answer to this question is yes.

6.4.1. Top Level Statement

First, you need to know about top level statements. These are pieces of code you put outside functions, like this:

```
# This is a top level statement
print('Hello from the top')

def foo():
    # This is NOT a top level statement as it is inside a function
    print('Hello from inside foo')
```

If you store this within a file (e.g., script.py), you can run its contents from the shell. Python will then execute all top level statements:

```
$ py script.py
Hello from the top
```

While it is technically possible to put all your code at the top level, there are serious disadvantages to do so:

- It quickly becomes an unreadable mess. Functions add structure, improve readability and maintainability, etc.
- You want to be able to test targeted pieces of your code. Functions are nice little testable packages. Code on the top level is monolithic and much harder to test.

A typical approach is to define a main function:

```
def main():
    print('Hello!')

main()
```

main is a clear starting point, from which other functions will be called, that will call other functions, etc.

6.4.2. __name__

Top level statements are quite a hassle when it comes to testing. If you look at the tests we provide, they generally look like this:

```
import pytest
import student


def test_some_function():
    # ...
```

The second line, import student, tells Python we will be referencing the code written in student.py. However, importing will also cause the top level statements to be executed. This is problematic: first your application will run, and only when it finishes will the tests run.

We need a way to distinguish between these two scenarios:

- A Python file is being executed from the shell and we want the top level statements to be executed.
- A Python file is being imported from another Python file in which case we *don't* want the top level statements to be executed.

This can be achieved using the following trick:

```
def main():
    print('Hello!')


if __name__ == '__main__':
    main()
```

How it works exactly is not important, what matters is its effect: main will only be called if the script is directly ran from the shell, which is exactly what we need. Because running a python script from shell makes the script have environment _name_==’_main_’

We refer those interested in what __name__ does to this page.

6.4.3. Shebang

Running our script still requires going to the shell and entering

```
$ py app.py
```

You have to remember that the OS does not know anything about Python. As far as the OS knows, app.py is just a text file like any other, and py is just some executable. *We* know that py is actually the Python interpreter and app.py contains Python code, so we know to tell the OS to combine both: "launch py and have it run app.py." There is another way to tell the OS that app.py should be fed to py: if the first line of a file contains a shebang (#!), the OS will take this as a hint about how to run that file:

```
#!/usr/bin/env py

def main():
    print('Hello!')


if __name__ == '__main__':
    main()
```

Here, the first line informs the OS that if the user wishes to launch app.py, it should actually launch py instead and feed it (have it run) app.py. Those using Linux or MacOS need to tell the OS that it's okay to execute app.py. This is achieved as follows:

```
$ chmod +x app.py
```

This marks the file as executable and needs only to be done once. After adding a shebang (and making the file executable on OSses that require it), you should be able to run your script by simply writing ./app.py instead of py app.py. On MacOS and Linux, you should also be able to double click it from a graphical user manager. Windows works differently: if you want to be able to run your Python application from File Explorer or using a shortcut, you'll have to associate .py files with python.exe. Normally, this association should already have been established for you by the Python installer.

6.4.4. Summary

If you'd like your script to be runnable, use the following pattern:

```
#!/usr/bin/env py

def main():
    # Startup code

if __name__ == '__main__':
    main()
```

Running your script by double clicking on it from the GUI will cause a new window to pop up. Depending on your machine, this window might disappear immediately after the script is done. Therefore, in order to see the output, you may want to add some code that causes the window to stick around for a little while more. A few possible approaches:

- Rely on input()
- time.sleep

- os.system('pause')

6.5. Conversion

Before discussing conversions, it is crucial that you understand there is a difference between the integer 15 and the string "15". It is the same difference as a painting of a pipe and an actual pipe. Internally, the computer represents both in very different ways. Mathematical operations can only be done if the value is stored as an actual int. Fortunately, it is possible to switch between representations.

Syntax	Description
int(x)	Converts to an integer
float(x)	Converts to a floating point number
bool(x)	Converts to True/False
str(x)	Converts to string

```
>>> int("15")
15

>>> str(845)
"845"
```

Watch out with bool:

```
>>> bool('True')
True

>>> bool('False')
True                # !!!
```

The reason for this surprising result has to do with truthy vs falsey values. Any nonempty string is considered truthy, so bool(nonempty_string) will always return True, regardless of the contents of the string. Only bool("") will give you False.

6.6. Operators

Just like integers and booleans, strings have all kinds of operations defined on them. We'll discuss a number of them here. We start with an overview, after which we shortly discuss them in turn.

Syntax	Result	Description
"ab" + "cd"	"abcd"	Concatenation
"ab" * 3	"ababab"	Repetition
"a" in "abc"	True	Membership
"a" == "a"	True	Equality

Syntax	Result	Description
"a" != "a"	False	Inequality
"a" < "b"	True	Lexicographic comparison
"a" <= "b"	True	Lexicographic comparison
"a" > "b"	False	Lexicographic comparison
"a" >= "b"	False	Lexicographic comparison
len("abc")	3	Length

6.6.1. String Concatenation

Concatenation is a fancy name for joining strings together.

```
>>> "Hello" + "World"
"HelloWorld"
```

As you can see, you can add strings together using the + operator. Some code that exemplifies the difference between integers and strings:

```
>>> 1 + 2
3

>>> "1" + "2"
"12"
```

The operator + is reused in many places:

- You can add integers: 5 + 8.
- You can add floats: 1.2 + 3.4.
- You can add strings: "abc" + "def".

Assigning multiple meanings to the same thing is called overloading, or, in our case, more specifically, operator overloading. At a later point, we will discuss how you can overload operators yourself.

6.6.2. Repetition

The * operator allows you to repeat a string an arbitrary number of times:

```
>>> "x" * 10
'xxxxxxxxxx'
```

6.6.3. Membership Checking

The in (and its negation not in) can be used to see if a string contains another.

```
# Can't spell slaughter without laughter
>>> "laughter" in "slaughter"
True
```

```
# There's no I in team
>>> "i" not in "team"
True
```

6.6.4. Lexicographic Comparison

Lexicographic comparison actually does what you intuitively would expect s1 < s2 would do: it checks whether s1 would come before s2 in a dictionary.

```
>>> "Aaronson" < "Zukowski"
True
```

Remark, however, the following details:

- Comparison is case sensitive: "B" < "a" evaluates to True, because uppercase letters are considered "smaller" than lowercase letters.
- Every character (5, @, space, ...) has a specific place in the "alphabet". For example, ' ' < 'A'. The actual order is determined by The Unicode Standard.

6.6.5. String Length

Admittedly, len is not an operator but just a function. But do you really want a separate page just for len?

```
len hides no surprises: it returns the length of the string.
>>> len('schnackenpfefferhausen')
22
```

6.7. Escape Characters

Imagine we need to add a string in our code which contains a " , for example he said "hello". Using double quotes will give us trouble:

```
"he said "hello""
```

Python gets very confused by this: it sees this as two strings "he said " and "", separated by hello, which is grammatically incorrect. We got one unhappy Python on our hands. But we hear you ask, why not single quotes then?

```
'he said "hello"'
```

Indeed, this works perfectly. But, aha!, what if the string also contains apostrophes?

```
'Jack's mother said "hello"'
```

This again leads to a very grouchy Python. Similar issues actually arise quite often.HTML relies on tags to assign meaning to text: <emph>some important text</emph> emphasizes text. But what if we want the text itself to contain a <?x < y will be flagged down as an error, since it looks like you're starting a tag. HTML's answer to this problem is to provide &lt; as an alternative to <.This document is written using Markdown. Emphasizing text is done using *emphasized text*. However, if we write 5*7 equals 7*5, this results in "57 *equals* 75" instead of "5*7 equals 7*5". In order to prevent the Markdown processor from interpreting the multiplication sign * as the start of an emphasized block, we need to escape the * character which we can do using a backslash: 5*7 equals 7*5. Python allows you to escape characters using the backslash \ in strings. For example, Jack's mother said "hello" can be written

- "Jack's mother said \"hello\"", or
- 'Jack\'s mother said "Hello"'.

In other word, \" and \' allow you to insert " and ' characters in strings. The backslash \ is called an escape character because it changes the meaning of the next character. " means "end of string" whereas \" means "a quote character". There are more combinations possible:

Combination (7) Meaning

\"	"
\'	'
\n	Newline
\r	Carriage return
\t	Tab
\b	Backspace
\\	\

6.7.1. Raw Strings

Escape characters can sometimes be bothersome. Say you want to add a file path in your code: "C:\nodejs\training" would be interpreted by Python as

```
C:
odejs    raining
```

This is due to \n being transformed to a newline and \t to a tab. To prevent this from happening, you have to escape the \: "C:\\nodejs\\training". This quickly becomes bothersome. Prefixing a string literal with an r turns it into a raw string. Escape characters are ignored in such strings: r"C:\nodejs\training" works as intended. Raw strings will come in handy later when you learn about regular expressions.

6.8. Indexing

A string can be seen as a sequence of characters. You can access each of these characters individually using indexing:

```
>>> string = "abc"
>>> string[0]
"a"

>>> string[1]
"b"
```

```
>>> string[2]
"c"
```

Note that the indexing is zero-based, which means that the first element has index 0 and not 1.

6.9. Negative Indices

For your convenience, Python allows you to use negative indices. This equivalence holds:

```
string[-index] == string[len(string) - index]
```

In other words, a negative index starts counting from the end.

```
>>> string = "abcd"
>>> string[-1]
"d"

>>> string[-2]
"c"
```

6.10. Slicing

Say you want the first three characters in a string:

```
string = "abcdef"
first_three = string[0] + string[1] + string[2]
```

While this code works, it is very cumbersome. *Slicing* will make this process much simpler:

```
string = "abcdef"
first_three = string[0:3]
```

When indexing using something of the form i:j, you will get the substring that ranges from index i up to index j. Note that the slice string[i:j] does not include the character at index j. There are a few variations on this syntax:

Syntax	Description
s[:j]	Same as s[0:j]
s[i:]	Same as s[i:len(s)]
s[:]	Same as s[0:len(s)], i.e., returns the whole string

6.10.1. Steps

It is also possible to specify a step: s[start:end:step].

```
>>> "abcdefg"[::2]
"aceg"
```

Negative steps allow you to reverse the substring:

```
>>> "abcd"[::-1]
"dcba"
>>> "abcdefg"[4:2:-1]
'ed'
```

6.11. Methods

6.11.1. Extra String Operations

Programs are instructions that apply operations on all kinds of objects (integers, booleans, strings, …). As of yet, these operations have taken two forms:

- Operators like a + b, a - b, a < b,a == b, a and b, not a, etc.
- Functions like len(a), min(a, b), etc. Functions are more flexible than operators thanks to their ability to have any number of operands/parameters.

There's also a third form: methods. Their syntax is `a.method(b, c, d, ...)`. You can interpret this as "I ask object a to perform its operation called method with arguments b, c, d, ...” A method is in essence a function that is a part of an object, and the syntax a.method asks a for that part. Don't worry if this does not make sense yet, right now we only need to focus on the syntax. Why there is this third form of syntax will become apparent later. The idea of functions being part of objects is central to the object oriented paradigm, a topic which will be discussed later.

6.11.2. String Methods

Strings come with a bunch of methods. We have given an overview of a selection of them. A full list of string methods can be found online.

Method	Description
s.lower()	Returns a lowercase copy of s
s.upper()	Returns an uppercase copy of s
s.find(subs)	Returns the index of subs in s
s.startswith(prefix)	Checks if s starts with prefix
s.endswith(suffix)	Checks if s ends on suffix
s.strip()	Returns a copy where whitespace at both ends have been removed
s.lstrip()	Returns a copy where whitespace at the start have been removed
s.rstrip()	Returns a copy where whitespace at the end have been removed
s.ljust(width, fill)	Returns a left justified copy of size width padded with fill

Method	Description
s.rjust(width, fill)	Returns a right justified copy of size width padded with fill
s.capitalize()	Returns a string that has only the first letter capital
s.replace(find_char, replace_char)	Returns a string with all find_char replaced by replace_char. Note find_char can also be a substring!

lower() and upper() work as you would expect.

```
>>> "ABC".lower()
"abc"

>>> "Test".upper()
"TEST"
```

string.find(s) looks for s within string. If s occurs somewhere inside string, the starting index is returned, otherwise -1.

```
>>> "some string".find("tri")
6

# -1 in case substring was not found
>>> "some string".find("abc")
-1
ljust, rjust and center help with formatting strings.
>>> 'abc'.rjust(10)
'       abc'

# We can choose the padding character
>>> 'abc'.rjust(10, '.')
'.......abc'

>>> ' title '.center(20, '=')
'====== title ======='
```

6.12. Docstrings

7. Loops

7.1. While

Algorithms are built out of three kinds of building blocks. You've already met two of them:

- Sequencing, i.e., having steps be executed one after the other, in order.
- Selection, i.e., being able to specify whether certain steps should be either executed or skipped based on a condition. In Python, selection takes the form of an if.

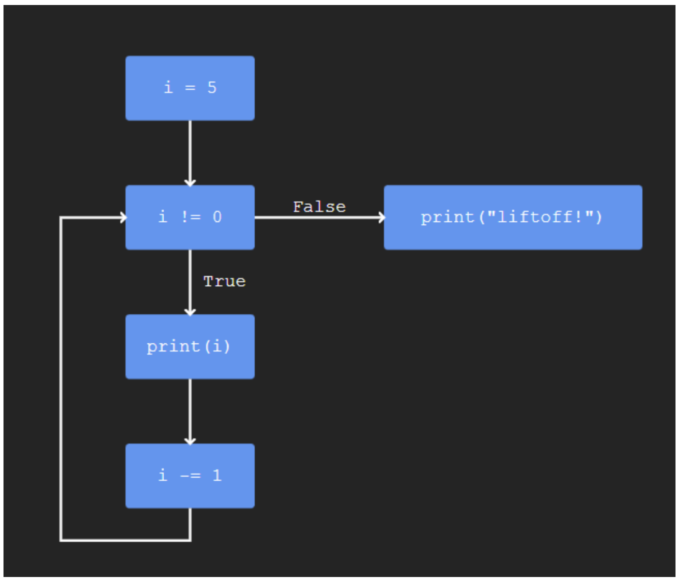
The last building block is iteration. This construct allows us to *repeat* certain steps for as long as we want.

Here's an example:

```
i = 5
while i != 0:
    print(i)
    i -= 1
print("liftoff!")
```

Python has multiple ways to perform iteration. In the code above, we are using a *while loop*. The body of the while loop, which consists of the print(i) and i -= 1 statements, is repeated for as long as the condition i != 0 is true.

We can visualize this code using a flowchart:



Let's go through the code step by step:

- We initialize i to 5.
- We arrive at the while loop. The condition i != 0 is evaluated; this yields True. Because the condition is true, the loop's body gets evaluated.
  - The value of i is printed, i.e., 5.
  - i gets decremented by 1 and is now equal to 4.
- Execution jumps back to the while line.
- The condition is re-evaluated: is i still not equal to 0? The answer is yes, causing the loop body to be executed a second time.
  - The value if i (4) is printed.
  - i gets decremented from 4 to 3.
- We jump back to the while line. This process keeps getting repeated:
  - 3 is printed and i goes from 3 to 2.
  - 2 is printed and i goes from 2 to 1.
  - 1 is printed and i goes from 1 to 0.
- i has reached 0. The condition is, once again, evaluated, but this time it yields False.
- Because the condition is not true anymore, the iteration ends, i.e., the loop body won't get executed again.
- We end up at the last print, which outputs "liftoff!".

Some students are under the impression that the condition is reevaluated after every step in the loop.

```
i = 0
while i != 1:
  i = 1
  i = 0
```

This is an infinite loop. Every time the condition is checked, i equals 0. Even though i gets set to 1 inside the loop's body, it doesn't matter: the condition is only checked after a *full* iteration. Intermediate values for i have no bearing on the loop.

7.1.1. return

As mentioned a while back, return immediately exits the function. This is also true for loops: if you have a return inside your loop, it will interrupt the loop.

```
def loop():
    i = 0
    while i < 10:
        print(i)
        if i == 5:
          return
        i += 1
```

will only print the numbers from 0 to 5.

7.2. For

A very common loop pattern is

```
i = 0
while i < i_max:
    # ...
    i += 1
```

In other words, the loop body is executed for every value from 0 to i_max. Such loops occur so often that many programming language provide a separate looping construct for it. Python is no exception:

```
for i in range(0, i_max):
    # ...
```

Meet the for loop. The loop shown above makes i go through all values of the range(0, i_max), i.e., 0, 1, 2, … i_max - 1. Note: the variable name i can be freely chosen. Note that range(start, stop) does not include stop itself.

```
>>> for i in range(0, 3):
...    print(i)
0
1
2
```

7.2.1. range

range exhibits a lot of similarities with slicing (4).

- range(start, stop) represents all integers from start to stop (exluding stop itself).
- range(start, stop, step) goes from start to stop with increments of step.
- range(stop) is the same as range(0, stop).
- Negative step allows you to count backwards.

7.3. Nesting

It is possible to nest your conditionals and loops as deeply as you want:

```
for y in range(height):
    for x in range(width):
        count = 0
        for dy in range(-1, 2):
            for dx in range(-1, 2):
                if check(x + dx, y + dy):
                    count += 1
        register(x, y, count)
```

Having the ability to do so does not necessarily mean it's a good idea:

- Nesting loops can lead to very inefficient code.
- If the nested looping is actually necessary, consider extracting part of the code into a separate function.

Below we rearranged the code so as to make it more manageable:

```
for y in range(height):
    process_row(y)


def process_row(y):
    for x in range(width):
        process_square(x, y)


def process_square(x, y):
    count = 0
    for dy in range(-1, 2):
        count += process_neighbors(x, y + dy)
    register(x, y, count)


def process_neighbors(x, y):
    count = 0
    for dx in range(-1, 2):
        if check(x + dx, y):
            count += 1
    return count
```

7.4. String Iteration

In our previous explanation, we claimed the for loop took the form

```
for variable in range(start, stop):
    # body
```

In reality, the for loop is a bit more flexible than this: range is just a specific case. The actual general shape is

```
for variable in iterable_object:
    # body
```

range is just one function that returns an iterable object. There are many iterable objects, strings being one of them:

```
>>> for char in "abcd":
...     print(char)
"a"
"b"
"c"
"d"
```

Later you will see lists, tuples and dictionaries, all of which are also iterable.

7.5. Walrus Operator

The walrus operator is an operator that can help write more readable code. It is a slightly more advanced topic and we don't expect you to use it, but for those students who are willing to go the extra mile, it's good to know it exists.

```
value = int(input("Enter an integer: "))
while value != 0:
    print(value)
    value = int(input("Enter an integer: "))
```

The code above reads integers from the keyboard and prints them out. It stops as soon as the user inputs 0. The code above contains some duplication: int(input("Enter an integer: ")) occurs twice. We should get rid of this, as duplication can easily lead to bugs.

7.5.1. Using Helper Function

One solution consists of introducing a helper function:

```
def input_integer():
    return int(input("Enter an integer: "))


value = input_integer()
while value != 0:
    print(value)
    value = input_integer()
```

We expect you to be able to perform this kind of code cleanup.

7.5.2. Using Walrus Operator

Another approach would be to rely on the walrus operator. a = b and a := b do exactly the same thing: they evaluate b and assign the result to a. The difference between both lies in where you are allowed to use them. a = b is a *statement*, which means you cannot put it inside something else. For example print(a = b) is invalid. An assignment always belongs on a line of its own. a := b is an *expression* and belongs "inside" something else. Writing a := b on a separate line will cause an error. However, print(a := b) is okay. But this raises the question, what would print(a := b) output? All expressions evaluate to a value. For example, 3 + 5 is also an expression and it evaluates to 8. So what does a := b evaluate to? Simple: it evaluates to the value of b.

```
print(a := 2 * 5)
```

- Before we can call print, we need to evaluate its argument a := 2 * 5.
- a := 2 * 5 first evaluates its right hand side.
- 2 * 5 yields 10 as result.
- This result is then assigned to a.
- a := 2 * 5 as a whole evaluates to 10.
- This is used as argument for print.
- Ultimately, 10 is written to the screen.

Let's apply the walrus operator to our loop.

```
while (value := int(input("Enter an integer: "))) != 0:
    print(value)
```

Here, we made the input functionality part of the condition. Whenever the condition of the while is evaluated, the user will be asked for an integer. This integer will be assigned to value and then be compared to 0. In the end, the walrus operator is never needed: it did not even exist prior to Python 3.8 and was purely added for convenience. Very few languages make the distinction between = and :=. a = b is generally considered an expression and can be used anywhere. Why does Python make it so complicated? A common mistake is to confuse = with ==:

```
if a = b:    # Should be a == b
    # ...
```

To catch this mistake, Python made it an error to use = in an expression, i.e., = must be used on its own line. This constraint, in turn, caused some need for duplication (see example at the top.) To counter this, := was added to the language.

7.5.3. Removing backspaces (exercise)

Internally, letters and symbols are represented by numbers, because numbers is really all a computers knows about. While any mapping between symbols and numbers will do, it's always nice if people agreed on one specific mapping. So, in 1991, some brave people set out to define the Unicode standard which defines a number (called a codepoint) for every character of any writing system in existence.

A few examples of codepoints:

Codepoint	Character
65	A

Codepoint	Character
66	B
952	θ
995	ϣ
3091	ఓ
129428	🦔

At the moment of this writing, Unicode supports 149,186 different characters. It also defines encodings for what some weirder "symbols", such as the bell signal (codepoint 7) and backspace (codepoint 8). In Python, we can insert a backspace in our strings using \b. For this exercise, you receive a string containing backspace characters. Your task would be to process them so that they actually remove the preceding character. For example, "abc\b" should be simplified to "ab". A backspace in the beginning of a string has no effect: "a\b\b\b" becomes the empty string.

8. Tuples

Integers, floating point numbers, booleans and strings can be seen as basic building blocks of every program. They can be said to be the atoms out of which everything is built. Let's examine a file containing a movie.

- The movie probably has a title, which is represented by a string. Other metadata such as director, actors, etc. are also strings.
- The visual part of the movie consists of a long series of images. A typical movie runs at 24 frames per second for 2 hours, so around 172,800 images.
- One image is a matrix of colors of size 4096×2160.
- A color is defined as a mix of red, blue and green, each of these represented as an int.
- The audio consists of multiple channels, e.g., mono (1 channel), stereo (2 channels), Dolby Surround 7.1 (8 channels).
- Each channel is a sound wave that's been sampled at a rate of (probably) 48,000 samples per second.
- Each sample is an integer.

As you can see, a movie is really just around 4.5 trillion numbers, and a few strings. Apart from having our building blocks, we also need a way of "glueing" them together, build molecules if you wish. That's where collections come in. We'll start with *tuples*. A tuple is an object that can hold an arbitrary number of other objects. The syntax is (a, b, c, d, ...) where a, b, c, d, ... are its content. A tuple can contain other tuples.

- A 2D-point has two coordinates. We can store these in a tuple: (4, 9).
- A playing card has a value and a suit: (8, 'hearts').
- A hand of playing cards can be represented using nested tuples: ((3, 'clubs'), (10, 'diamonds'), (12, 'spades), ...).
- A color has three components: R, G and B. These can be stored in a tuple of size three: (192, 128, 32).
- A row of colors can be represented by a tuple of color-tuples: ((0, 0, 0), (255, 255, 255), (192, 64, 64), ...).
- An image can be represented by a tuple of such rows: (((0, 0, 0), (255, 255, 255), (192, 64, 64)), ((128, 128, 128), (64, 64, 64)), ...).

The empty tuple is written (). However, the tuple with one element is a bit problematic: (5) is interpreted as simply 5 with redundant parentheses. In order to make Python understand you mean to create a tuple with one item, you need to write (5,).

8.1. Indexing

When we were discussing strings, we explained how you could determine the string's length using len and how to get access to individual characters. Tuples are very much like strings, except more flexible: a string can only contain characters, whereas a tuple can contain any kind of data. They also offer very similar functionality.

Syntax	Meaning
len(t)	Length of the tuple t

Syntax	Meaning
t[0]	First item in the tuple t
t[-1]	Last item in the tuple t
t[1:4]	Slicing the tuple

If you're wondering whether you can modify a tuple, the answer is no. We'll discuss lists soon, and these are in essence modifiable tuples.

8.2. Membership

To determine whether a tuple contains an element, you can use the in operator.

```
>>> 3 in (1, 2, 3, 4, 5)
True
>>> 6 in (1, 2, 3, 4, 5)
False
```

x in t does not work recursively, by which we mean that if t contains another tuple, in will not look inside that nested tuple.

```
>>> 3 in (1, (2, 3))
False
```

The in operator could check for substrings when applied on strings:

```
>>> 'relax' in 'rancho relaxo'
True
```

This will not work with tuples:

```
>>> (2, 3) in (1, 2, 3, 4)
False
```

The in operator is stricter when dealing with tuples: x in t only returns True if there's an item of t that equals x.

8.3. Iteration

Tuples are iterable, which means they're compatible with the for loop.

```
for item in (1, 2, 3, 4, 5):
    print(item)
```

prints all items in the given tuple.

8.4. Destructuring

When using tuples to represent coordinates, colors, etc. you know exactly how many items a tuple will have. Often you will need to get access to each item of the tuple:

```
def process(color):
    r = color[0]
    g = color[1]
    b = color[2]
    # ...
```

Such code can be simplified using destructuring:

```
def process(color):
    r, g, b = color
    # ...
```

Destructuring is also possible in for loops:

```
colors = (
    (50, 128, 240),
    (0, 0, 0),
    (255, 200, 140),
    (142, 20, 185)
)

for r, g, b in colors:
    # ...
```

8.5. Functions

8.5.1. min and max

Both min and max know how to handle tuples. When you pass either function a single argument, they expect it to be an iterable. They then proceed to find the lowest/highest value.

```
>>> min((5, 2, 3, 9))
2

>>> max((5, 2, 3, 9))
9
```

8.5.2. sum

The built-in function sum allows you to compute the sum of all items in an iterable collection, such as a tuple. (Later we will also discuss lists, sets and generators; sum will also work on them.)

```
>>> sum((5, 2, 3, 9))
19
```

8.5.3. sorted

The function sorted(t) returns a sorted copy of t.

```
>>> sorted((4, 2, 9, 1, 3))
```

```
(1, 2, 3, 4, 9)
```

There's a caveat though: the values in t must be comparable to each other. If you mix different types of values, the sorting will lead to an error. When sorting, it is best to stick to tuples where all elements have the same type,

```
>>> sorted((4, "5", 9, 1, 3))
TypeError: '<' not supported between instances of 'str' and 'int'
```

9. Named Tuples

We can distinguish two uses for tuples:

- On the one hand, we can simply see them as a readonly list. Typically, the items all have the same type.
- On the other hand, tuples can be used to bundle data together. For example, a playing card has a value (integer) and a suit (string).

In this chapter, we focus on the latter usage. Say we want to keep track of users. We need the following data:

- The user's name. We want the user's first name and last name as two separate strings.
- The user's email address.
- The user's address, which is composed of
    - Street
    - House number
    - ZIP code
    - City
    - Country

We can represent this using tuples:

```
user_data = (
    ("Sherlock", "Holmes"),
    "sherlock.holmes@aol.com",
    (
        "Baker Street",
        "221b",
        "NW1 6XE",
        "London",
        "United Kingdom"
    )
)
```

In order to fetch specific information, we need to know the exact index, e.g., user_data[2][4] retrieves the user's country. This approach, however, is extremely error prone. Nothing about user_data[2][4] expresses that it fetches the country. You can also imagine how messy things get if we need to add or removes fields and index updates are required across the entire code base. A much more readable solution would have us assign *names* to the different parts of the tuple, so that we could write user_data.address.country instead of user_data[4][2]. This is exactly what named tuples allow use to do:

```
from collections import namedtuple


UserData = namedtuple("UserData", [
    "name",
    "email_address",
    "address"
])

Name = namedtuple("Name", [
    "first_name",
    "last_name"
])
```

```
Address = namedtuple("Address", [
    "street_name",
    "house_number",
    "zip_code",
    "city",
    "country"
])


user_data = UserData(
    Name("Sherlock", "Holmes"),
    "sherlock.holmes@aol.com",
    Address("Baker Street", "221b", "NW1 6XE", "London", "United Kingdom")
)

user_data.name.first_name   # Sherlock
user_data.address.country   # United Kingdom
```

9.1.1. Defining a Named Tuple

First, you need to define the structure of the new named tuple. This includes the *name* (e.g., Address) and the fields (e.g., street_name, house_number, …)

```
Name = namedtuple("Name", ["field1", "field2", "field3", ...])
```

The name has to be repeated twice. It is inconvenient but unavoidable. The name of the tuple defines a new type: it becomes a "colleague" of int, str, bool, etc. Custom types have their own naming convention (CapWords convention): the words in the name are capitalized and joined together without separator. For example, UserData is correct, user_data, User_Data, User_Data are wrong.

9.1.2. Creating Tuple Objects

Once you have defined the new type, you can create objects of that type.

```
obj = Name(field1_value, field2_value, field3_value, ...)
```

Getting the argument order right is still tricky when creating an large tuple such as Address. Keyword arguments (discussed elsewhere) can help out:

```
user_data = UserData(
        name=Name(
             first_name="Sherlock",
             last_name="Holmes"),

        email_address="sherlock.holmes@aol.com",

        address=Address(
             street_name="Baker Street",
             house_number="221b",
             zip_code="NW1 6XE",
             city="London",
             country="United Kingdom")
)
```

9.1.3. Accessing Fields

Once you have created a new tuple object, you can access its field with the . operator:

```
obj.field1    # Evaluates to field1_value
obj.field2    # Evaluates to field2_value
obj.field3    # Evaluates to field3_value
...
```

10. Lists

10.1. Introduction

Tuples are *immutable*: once created, they cannot be changed: we cannot remove, add or overwrite elements. This immutability can be a strength, but in other cases it can be quite an impediment. *Lists* are very much like tuples, except they can be modified.

10.1.1. Literals

A list literal has the following syntax:

```
>>> lst = [1, 2, 3, 4]
```

As you can see, the only difference lies in the brackets: square brackets for lists, round brackets for tuples. The list with one item also requires no trickery: [1] works just fine.

10.1.2. List Functionality

All functionality available on tuples also works on lists: len, indexing, slicing, sum, min, max, iteration, destructuring, etc. all work without change. Actually, all code you write that interacts with tuples will behave the same if given lists. Tuples have a *subset* of the functionality of lists.

```
>>> sum([1, 2, 3, 4])
10

>>> max([4, 1, 7, 5, 2, 3])
7
```

10.2. Updating

Accessing the items of a list is done the same way as with tuples, i.e., by indexing.

```
>>> lst = ["a", "b", "c", "d"]

# Fetch first item, which has index 0
>>> lst[0]
"a"

# Fetch last item, which has index -1
>>> lst[-1]
"d"
```

Using the syntax lst[index] = x you can overwrite the item at the given index.

```
>>> lst = [7, 4, 1, 9, 3]

# Overwrite first item
>>> lst[0] = 100
>>> lst
[100, 4, 1, 9, 3]

# Double the last item
```

```
>>> lst[-1] *= 2
>>> lst
[100, 4, 1, 9, 6]
```

10.3. Adding Items

Lists provide two ways to add items to them. We discuss each in turn.

10.3.1. append

lst.append(x) adds an extra element to the end of the list.

```
>>> xs = [1, 2, 3]
>>> xs.append(4)
>>> xs
[1, 2, 3, 4]
```

10.3.2. insert

If you need to add an element somewhere else than at the end, you can rely on the insert method. lst.insert(index, x) adds an element x so that its index equals index. All subsequent elements are shifted one position.

```
>>> xs = [1, 2, 3, 4]
>>> xs.insert(0, 9)
>>> xs
[9, 1, 2, 3, 4]
```

10.4. Removing Items

There are two ways to remove items from lists:

- Either you specify the *index* of the item to be removed, or
- You specify the value of the item to be removed.

10.4.1. pop(index)

lst.pop(index) removes the item at the given index. You can omit index in which case the last element will be removed.

```
>>> xs = [1, 2, 3, 4, 5]
>>> xs.pop(1)
>>> xs
[1, 3, 4, 5]

>>> xs.pop()
>>> xs
[1, 3, 4]
```

10.4.2. del (index or slice)

del is a keyword and has a syntax of its own. Functionality-wise it is a more powerful version of pop. del lst[index] does the same as lst.pop(index): it removes the item with the given index. However, del also works with slices: del lst[start:stop] removes all items from start to stop.

```
>>> xs = [1, 2, 3, 4, 5]
>>> del xs[-1]
>>> xs
[1, 2, 3, 4]

>>> del xs[:2]
>>> xs
[3, 4]
```

10.4.3. remove(value)

Lastly, we have lst.remove(item) which removes the first occurrence of item.

```
>>> xs = [6, 4, 2, 3, 4, 9, 4]
>>> xs.remove(2)
>>> xs
[6, 4, 3, 4, 9, 4]

>>> xs.remove(4)
>>> xs
[6, 3, 4, 9, 4]
```

10.5. References

By now, we discussed many different kinds of values or objects (in Python these terms are synonymous):

- Integers, e.g., 0, 1, 2
- Floating point numbers, e.g., 0.1, 3.1415
- Booleans, e.g., True and False
- Strings, e.g., "hello"
- Tuples, e.g., (1, 2, 3)

All these objects are immutable, or stateless, meaning you cannot change them. Lists, however, are the first type of object we've encountered that we can *modify*. We say that lists are mutable, or stateful. This brings us to object identity, a concept that becomes relevant when dealing with state. The distinction between working with stateful objects and stateless objects is an important one, important enough to assign names to the two styles:

- Imperative programming involves mostly stateful objects, but stateless objects are allowed.
- Functional programming allows only stateless objects.

Recently, the world has moved towards functional programming, as it simplifies things a lot.

10.5.1. Example: Lists

Consider the following code:

```
xs = [1, 2, 3]
ys = [1, 2, 3]
```

Are xs and ys the the same list?

- On the one hand, one could claim that xs and ys are the same: they're both lists, and they contain the same elements. How more "the same" can you get?
- On the other hand, we could say they're different lists. We can add an element to xs and it will not appear in ys.

They're two different answers to the same question. For this reason, Python offers two different ways to check for equality.

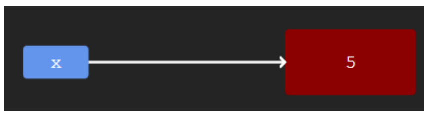
- xs == ys checks that xs and ys have the same kind of objects and if they have the same contents. In our case, xs == ys yields True.
- xs is ys checks whether they're actually the same object. In our case, xs is ys yields False: they're distinct lists.

10.5.2. Memory Layout

Looking at how things are stored in your computer's memory may make things clearer. Everything has to be stored somewhere in memory (RAM): every int, boolean, string, functions, ... In order to access something stored in memory, you need to know where it is located. This is possible using memory addresses. Without the address, you don't know where the object resides in memory, and therefore you cannot reach it. The term "memory address" is seldomly used when using high level programming languages like Python, JavaScript, Java, etc. Instead, the term "reference" is used, but in essence it is the same. From now on, we'll be using "reference" exclusively. So what happens (2) when we write x = 5?

- Some memory is allocated, enough to store the value 5 in.
- x is a variable, which can be thought of as a box. A *reference* to 5 is put inside this box.

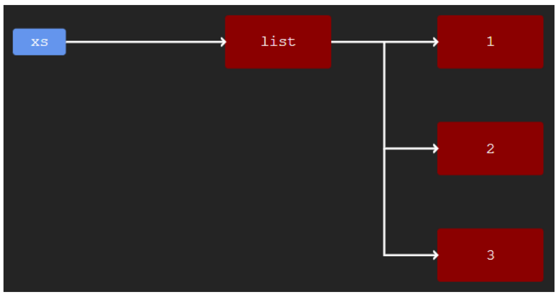
We can visualize this as follows:



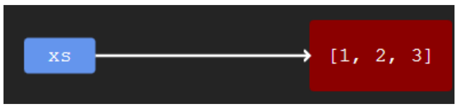
Now let's create a list:

```
xs = [1, 2, 3]
```

If we visualize the memory layout, we get:



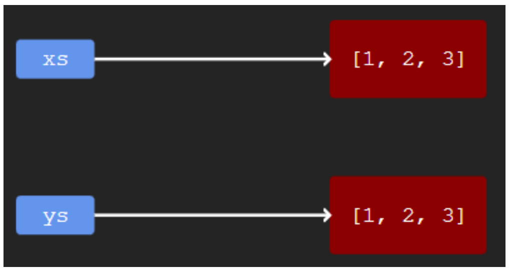
As you can see, the list also contains references. To make the diagrams more manageable, we can simplify this visualization to:



It is important, however, that you realize that this representation is just a simplification and that the first diagram shows the actual memory layout.

Let's introduce a second list:

```
xs = [1, 2, 3]
ys = [1, 2, 3]
```

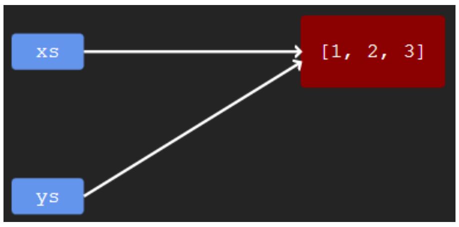


This diagram makes it clear that although xs and ys refer to "the same" list, these lists are actually distinct entities in memory. xs == ys is True, but xs is ys is False.

10.5.3. Assignment

We now examine the following code:

```
xs = [1, 2, 3]
ys = xs
```



When you assign one variable (xs) to another (ys), as shown above, the contents of xs is copied to ys. As mentioned before, the contents of xs is a *reference* to a list [1, 2, 3], so it is the *reference* that is being copied. As the diagram shows, both xs and ys refer to the same list object.

```
>>> xs = [1, 2, 3]
>>> ys = xs
>>> xs.append(4)

>>> xs
[1, 2, 3, 4]
>>> ys
[1, 2, 3, 4]

>>> xs == ys
```

```
True
>>> xs is ys
True
```

xs.append(4) starts at xs, follows the arrow and ends up at the list [1, 2, 3]. It then adds 4 to this list. Since ys refers to the exact same list, it should come as no surprise that printing out the contents of ys yields [1, 2, 3, 4].

10.5.4. Terminology

In order to be able to express ourselves unambiguously, we introduce the following terminology:

- Two objects x and y are equal if x == y is True (same content).
- Two objects x and y are the same if x is y is True (same content & same memory address)

Two same objects are always equal, but not all equal objects are the same.

10.5.5. Relationship with Mutability

We started this explanation about mutability. But what does this have to do with references? When working with immutable objects, the distinction between equal objects and same objects does not matter: the results will always be the same. However, when mutable objects are involved, it can be crucial to keep the distinction in mind.

10.6. Conversion

Strings, tuples and lists are all very similar:

- They represent a sequence of items.
- They can be indexed: xs[index].
- They have a length: len(xs).
- Many operations (min, max, etc.) can be applied on them.

You might wonder why Python bothers having all three. Why not just lists?

- Strings are in essence tuples that contain characters. They exist as a separate type for efficiency reasons: they have a very specialized, optimized implementation.
- Tuples are useful because they're immutable. This may seem like a drawback, but in reality, this property can simplify things *a lot*.

Using str(x), tuple(x) and list(x) you can convert (3) x to a string, tuple or list, respectively.

10.6.1. str(x)

We already discussed str. It can be applied on (as good as) any value x to build a string representation of x.

```
>>> str([1, 2, 3])
"[1, 2, 3]"

>>> str((1, 2, 3))
'(1, 2, 3)'
```

10.6.2. tuple(x)

tuple(x) requires x to be iterable. What this means exactly will be discussed later, but for now you can imagine it refers to any kind of value that contains multiple values, which includes strings, tuples and lists. Given an iterable value x, tuple(x) will take all values stored in x and put them in a tuple.

```
# Converts string to tuple
>>> tuple("abcd")
("a", "b", "c", "d")

# Converts tuple to tuple, not very useful
>>> tuple((1, 2, 3, 4))
(1, 2, 3, 4)

# Converts list to tuple
>>> tuple([1, 2, 3, 4])
(1, 2, 3, 4)
```

10.6.3. list(x)

Just like tuple, list(x) needs x to be iterable. It works like you would expect: it takes all items in x and puts them in a list.

```
# Converts string to list
>>> list("abcd")
["a", "b", "c", "d"]

# Converts tuple to list
>>> list((1, 2, 3, 4))
[1, 2, 3, 4]

# Converts list to list, can be used to make a copy
>>> list([1, 2, 3, 4])
[1, 2, 3, 4]
```

10.7. Strings

10.7.1. Splitting Strings

Converting a string to a list works on the level of single characters:

```
>>> list("abcd")
["a", "b", "c", "d"]
```

Typically, we want something "smarter". Say we have the string "Blondie,Angel Eyes,Tuco" and want to extract the three names. More specifically, we want to construct the list ["Blondie", "Angel Eyes", "Tuco"]. The string method split does exactly that: string.split(separator) will cut string into pieces, where each piece is delimited by separator.

```
>>> "a,b,c".split(",")
["a", "b", "c"]

# Be careful with spaces
>>> "one, two, three".split(",")
```

```
["one", " two", " three"]

>>> "one, two, three".split(", ")
["one", "two", "three"]
```

10.7.2. Joining Strings

The opposite operation separator.join(strings) can also come in handy: given a tuple/list of strings strings, we want to join them together into a single string. The different strings are "glued" together using separator.

```
>>> ", ".join(["a", "b", "c"])
"a, b, c"

>>> " and ".join(["Mac", "Charlie", "Dennis", "Dee"])
'Mac and Charlie and Dennis and Dee'
```

11. Sets

11.1.1. Benchmarking List Operations

Let's have fun with a little benchmarking. The file benchmark-lists.py measures the time of several list-related operations. You can run it on your own machine if you'd like. These are the results when we run it:

Operation	Time
Add element at beginning of list	2.38s
Add element at end of list	0.004s
Remove element at beginning of list	0.59s
Remove element at end of list	0.007s

Note that the absolute times do not mean much: each operation is repeated thousands of times. What matters is how the times relate to each other. As you can see, operations at the beginning of the list are dramatically slower compared to those taking place at the end of the list. What could be the reason behind this phenomenon? It's actually very simple: when adding/removing an item at the beginning of a list, all subsequent elements have to be shifted right or left. This overhead does not exist when adding/removing at the end of the list.

11.1.2. Data Structures

A list is a kind of *data structure*. A data structure provides a number of operation*s*. For example, a list allows indexing, adding and removing elements, looking for elements, etc. As shown above, these operations can differ in speed. The time complexity of an operation describes how much time each operation takes. This is actually quite a complicated subject; we won't go into the details here. Instead, we'll simply make the distinction between *slow* operations and *fast* operation s. For example, insertion and deletion are *slow* when done at the front of the list, but *fast* when done at the end.

11.1.3. Membership Testing

Let's examine the operation `x in lst`, i.e., checking whether x occurs in lst. Is this a slow or a fast operation? There is no magic or special trickery involved in the operation. In essence, it performs the following algorithm:

```
for item in lst:
    if item == x:
        return True
return False
```

In other words, it goes through the list, compares x with each item, and if some item turns out to be equal to x, `x in lst` returns True and False otherwise. This is a slow operation: think of having to find a term in an encyclopedia that isn't alphabetically sorted. Sadly, we cannot do better than this: lists are inherently slow when it comes to membership checking.

11.1.4. Sorted Lists

Looking up something in an encyclopedia is efficient because it is alphabetically sorted, which allows us to perform a binary search. Maybe we can apply a similar trick to lists? There is nothing that keeps us from doing exactly that. If it is *guaranteed* that the list is sorted, then `x in lst` can rely on the binary search algorithm, making it a fast operation. However, in order to provide that guarantee, we have to make sure that every other operation keeps the list sorted. This means we have to make some sacrifices. For example, append(x) has to be dropped, as it adds an item to the *end* of the list, which could jeopardize the sorted nature of the list. We can replace it with an add(x) operation that adds x to the list, but at the "right" position. In short, in order to have a sorted list, we'll have to give up the ability to choose the actual location of items in the list. We cannot make [3, 1, 2] anymore, only [1, 2, 3]. In return, we get a fast in operation. This sorted list is a separate kind of data structure, as it provides a different set of operations, with different time complexities. The in on regular lists is *slow*, but in on sorted lists is *fast*.

11.1.5. Sets

A *set* is yet another data structure. It is very similar to a sorted list, except it's even faster. But, just like with sorted lists, it comes at a price. The good news about sets is that the basic operations are very fast. benchmark-sets.py contains some benchmarking code:

Operation	Set Time
Adding an item	0.006s
Removing an item	0.009s
Membership	0.007s

The bad news (2):

- Sets are unordered: you have no control over the order of the items. You can therefore also not index a set, i.e., my_set[0] is invalid.
- No duplicates allowed: every item can appear only once in a set. Adding an item a second time has no effect.

Choosing to use a set is typically done for purposes of efficiency: a list can do everything a set can do, just more slowly.

11.1.6. Python Sets

Sets are deemed important enough for Python to have them built-in. Here is a quick overview:

Syntax	Description
{1, 2, 3}	Set literal
len(my_set)	Number of items in set
my_set.add(x)	Add x to set my_set
my_set.remove(x)	Remove x from set my_set
x in my_set	Check if x is an element of my_set

Because sets are unordered, {1, 2, 3} and {3, 2, 1} are equal sets. Because sets "swallow" duplicates, {1, 1, 1} and {1} are also equal. Note that the empty set cannot be written {}, you have to use set(). The reason for this is that dictionaries use the same curly braces. If there's at least one element, the syntax gives away if the literal is meant to denote a set or a dictionary:

- {value} is a set.
- {key: value} is a dictionary.

However, when no items are listed, both set and dictionary syntax reduce to {}. To resolve this ambiguity, Python chooses to see {} as a dictionary and lets you use set() for an empty set.

11.2. More Operations

We give a quick overview of what other set operations are provided. (4)

Syntax	Description
s1.intersection(s2, s3, ...)	Returns common elements
s1 & s2	Same as intersection
s1.union(s2, s3, ...)	Returns union of all sets
s1 | s2	Same as union
s1.isdisjoint(s2)	Checks if the intersection is empty
s1.issuperset(s2)	Checks if s1 contains all elements of s2
s1 >= s2	Same as issuperset
s1 > s2	Same as s1 >= s2 and s1 != s2
s1.issubset(s2)	Checks if s2 contains all elements of s1
s1 <= s2	Same as issubset
s1 < s2	Same as s1 <= s2 and s1 != s2

12. Dictionaries

12.1. Literals

Consider the following situations:

- You need to translate a text and would like having a dictionary that, given a word in English gives the corresponding word in Dutch.

English	Dutch
cat	kat
dog	hond

- You have found a recipe for a delicious chocolate cake. It comes with a list of ingredients and the required quantities for each.

Ingredient	Quantity
chocolate	250g
eggs	5
sugar	125g
flour	75g
butter	175g

- Given a student number, you wish to know the name of the student associated with that number.

Student Number	Student Name
r0123456	Karen Eiffel
r0654321	Harold Crick
r0121212	Ana Pascal
r0954321	Jules Hilbert

In each cases, we can represent this data as a table. Each row associated some data in the left column with some other data in the right column:

- cat is associated with kat.

- chocolate is associated with 250g.
- r0123465 is associated with Karen Eiffel.

We call the data in the left columns the keys and the corresponding data in the right column the values. Python allows us to easily model such data using *dictionaries*, or dicts for short. Here's what the translation dictionary would look like:

```
translating_dictionary = {'cat': 'kat', 'dog': 'hond'}
```

Dictionaries are not limited to strings: both keys and values can have any type. Dictionaries can even contain other dictionaries recursively. For example, this is a perfectly valid dict:

```
{
    1: True,
    'a': [1, 2, 3],
    False: {1: 2, 3: 4}
}
```

Actually, there *is* a limitation on the types of keys: the objects have to be hashable and comparable using ==. We'll delve further into this later. For now, you can imagine there are no constraints on key types.

12.2. Lookup

In the previous explanation, you were shown how to define a new dict:

```
translating_dictionary = {"cat": "kat", "dog": "hond"}
```

We now turn our attention to actually making use of this dict. The most common operation is to, given a key, look up the associated value. For example, say we want the translation of cat. Here, cat is the key, and kat is the value we wish to look up in our dictionary. We can do this as follows:

```
translation = translating_dictionary["cat"]
```

After executing this statement, translation is equal to "kat".

12.3. Insertion

dicts are mutable, i.e., they can be modified. For example, we can add new key/value pairs to an existing dict as follows:

```
# We start with empty dict
my_dict = {}

my_dict['a'] = 1
my_dict['b'] = 2
my_dict['c'] = 3
```

After executing this code, my_dict equals {'a': 1, 'b': 2, 'c': 3}. It is also possible to overwrite existing associations:

```
# We start with empty dict
```

```
my_dict['a'] = 5
my_dict['b'] *= 2
```

Now my_dict equals {'a': 5, 'b': 4, 'c': 3}.

12.4. Deletion

Apart from adding and overwriting key/value pairs, it is also possible to delete them:

```
del dictionary[key]
```

After this del statement, dictionary will no longer associate key with a value. If key did not appear in dictionary at the moment of the del operation, you will get an error.

12.5. Membership Testing

Given a dictionary d, you can look up the value associated with some key using d[key]. However, it is an error to look up a key that is not included in the dictionary:

```
>>> d = {'a': 1, 'b': 2}
>>> d['c']
KeyError: 'c'
```

In order to prevent this error from occurring, we might want to first check if the dictionary does indeed contain an entry with key. You can do this with the in operator:

```
>>> d = {'a': 1, 'b': 2}
>>> 'a' in d
True

>>> 'b' in d
True

>>> 'c' in d
False
```

Note that in only works on keys. You cannot look for values.

```
>>> d = {'a': 1, 'b': 2}
>>> 1 in d
False
```

12.6. Enumerating

As of yet, we've seen two ways to retrieve information (2) from a dictionary:

- d[key] looks up the value associated with key.
- key in d tells you whether d contains a certain key.

Remark how you always need a key to start with. If you have no key handy, you can't do much with the dict, apart from trying to guess which keys it contains.

12.6.1. keys()

Luckily, dicts allow you to ask for a list of keys:

```
>>> d = {'a': 1, 'b': 2}
>>> d.keys()
dict_keys(['a', 'b'])
```

While keys() does not really return a list, you can think of the result as one. Why does keys() not simply return a list? Consider the following code:

```
>>> d = {'a': 1, 'b': 2}
>>> keys = d.keys()
>>> keys
dict_keys(['a', 'b'])

>>> d['c'] = 3
>>> keys
dict_keys(['a', 'b', 'c'])
```

If keys() were to return list, it would contain a copy of the keys that are present at the moment you call keys(). Adding new keys to the dict later would not affect the list. The list is "dumb": it is not aware that its contents come from a dict. A dict_keys object is very much *like* a list, but it is "smarter": it *knows* its items are keys of a dict and will keep itself synchronized. In other words, if you were to add or remove keys after calling keys(), this change would be reflected in the dict_keys object.

12.6.2. values()

dicts also provide you with a values() method, which returns a "list" of values:

```
>>> d = {'a': 1, 'b': 2}
>>> d.values()
dict_values([1, 2])
```

12.6.3. items()

Say you want to iterate over all key/value pairs. You can achieve this using

```
for key in d.keys():
    value = d[key]
    # ...
```

However, you achieve the same more efficiently using items():

```
for key, value in d.items():
    # ...
```

This way, you don't have to manually look up every key. In essence, items() returns a list of pairs, i.e., tuples of size 2. The loop above relies on automatic destructuring. It is equivalent with

```
for pair in d.items():
    key = pair[0]
    value = pair[1]
    # ...
```

12.6.4. len()

Finally, if you simply need to know how many key/value pairs are present in a dict, you can rely on len:

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> len(d)
3
```

12.7. Lookup Default

Say you have a dict that represents the contents of your pantry:

```
pantry = {
    'sugar': 200,
    'eggs': 5,
    'flour': 750
}
```

You intend to bake some dessert and need to know if you have enough eggs. The dessert requires four eggs, so you perform the following check:

```
if pantry['eggs'] >= 4:
    # Enough eggs
```

But what if you check for vanilla beans?

```
if pantry['vanilla beans'] >= 1:
    # ...
```

This will lead to a crash: you can only look up using an existing key. To prevent a crash, you would need to write

```
if 'vanilla beans' in pantry and pantry['vanilla beans'] >= 1:
    # ...
```

That's a lot of typing for something relatively trivial. Luckily, there exist better solutions.

12.7.1. get()

dicts have a method named get. d.get(key) behaves exactly the same as d[key], but it takes an optional second parameter: a default value in case key does not appear in d.

```
>>> pantry = {
    'sugar': 200,
    'eggs': 5,
    'flour': 750
}

>>> pantry['vanilla beans']
KeyError: 'vanilla beans'

>>> pantry.get('vanilla beans')
KeyError: 'vanilla beans'

>>> pantry.get('vanilla beans', 0)
0
```

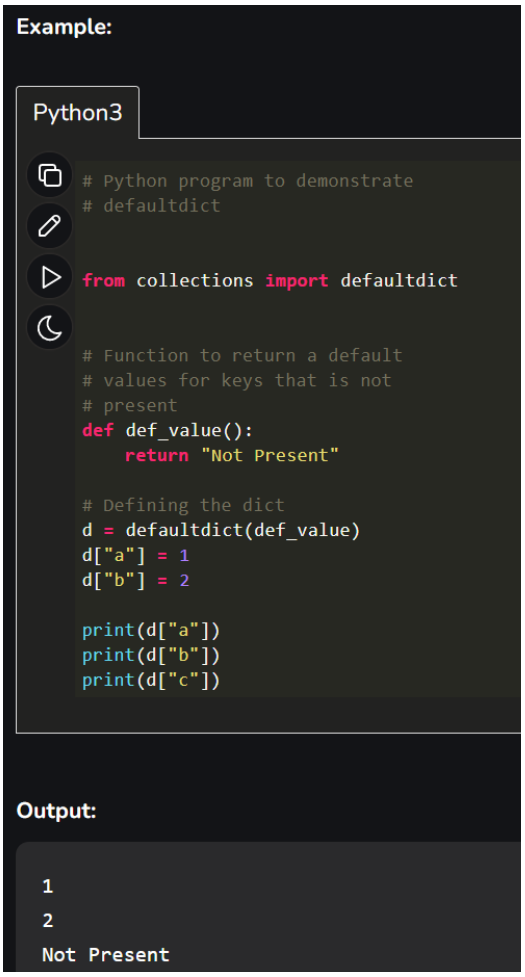
As you can see, get allows you to succinctly express that a missing ingredient in the ingredient is the same as having 0 of that item. Therefore, the check above can be rewritten as

```
if pantry.get('vanilla beans', 0) >= 1:
    # ...
```

12.7.2. setdefault()

A very similar method to get is setdefault. Contrary to what its name may lead you to believe, d.setdefault(key, if_missing) actually performs a lookup, just like d.get(key, if_missing). The difference is in what happens if key is missing: while get simply returns the if_missing value, setdefault will actually add it to the dict:

```
>>> pantry = {
    'sugar': 200,
    'eggs': 5,
    'flour': 750
}

>>> pantry.get('vanilla beans', 0)
0

# As you can see, pantry is still unchanged
>>> pantry
{
    'sugar': 200,
    'eggs': 5,
    'flour': 750
}

# setdefault also returns the default in case
the key is absent
>>> pantry.setdefault('vanilla beans', 0)
0

# setdefault added an extra key/value pair to
the dict
>>> pantry
{
    'sugar': 200,
    'eggs': 5,
    'flour': 750,
    'vanilla beans': 0
}
```


Example:

```
# Python program to demonstrate
# defaultdict

from collections import defaultdict


# Function to return a default
# values for keys that is not
# present
def def_value():
    return "Not Present"

# Defining the dict
d = defaultdict(def_value)
d["a"] = 1
d["b"] = 2

print(d["a"])
print(d["b"])
print(d["c"])
```

Output:

```
1
2
Not Present
```

More advanced students might like to know about defaultdict. This is a special kind of dict that allows you to define an if_missing value at creation. Every lookup d[key] is then equivalent to d.get(key, if_missing). The main advantage is that you only need to define the if_missing value once (i.e., at creation) instead at every lookup.

13. Objects

Programming languages provide us with a small set of basic data types. Typically, this is what you get:

- Integers
- Floating Point Numbers
- Booleans
- Strings

However, there are many other kinds of objects we would like to represent, depending on the software we're building:

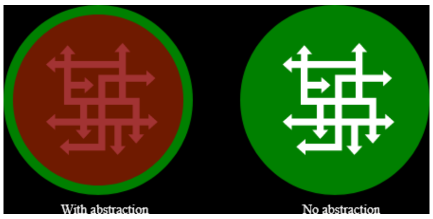
- A calendar app needs objects to represent dates, meetings, birthdays, …
- A strategy game needs a map, different kinds of units, resources, …
- A web shop involves an item catalog, a shopping basket, orders, customers, …

We've seen that we can group related data together, e.g., with tuples. Technically, this is more than sufficient to satisfy all of our software building needs: we can model arbitrarily complex data simply by introducing more tuples and putting these tuples in other tuples. Ultimately, we'll end up with tuples 20 deep, but hey, it works. However, we have to consider the human factor. While a machine has no trouble following instructions describing how to work with complex data, humans are the still the ones who have to write down those instructions. And humans are notoriously bad at dealing with such complexity. Fortunately for us, programming languages provide ways to reduce this complexity through the use of *abstraction*.

13.1.1. Abstraction

Abstraction is part of our daily lives: our brains use it to make interacting with the world manageable. When you work on your laptop, you interact with it on a very high level of abstraction: you type on your keyboard and letters "magically" appear, you use your mouse or trackpad to move the cursor around and click on different GUI elements. While doing so, you're not thinking of the billions of instructions that are being executed by your laptop to render fonts, process data, play music, etc. You think even less about the logic gates built out of transistors on your CPU that furiously manipulate electrical signals that represent bits. And you certainly don't think about changes in quantum fields as spinny muons and charming leptons (or whatever those things are called) buzz around. Abstraction is all about hiding details and only focusing on the things that matter. Back to the programming world. Functions are actually a basic form of abstraction (functions were even originally called abstractions.) Take print("Hello") for example: internally, printing to the screen is quite an elaborate process that involves millions of instructions. But that doesn't interest you. The only thing you care about is that Hello appears on the screen. There are only three things you need to know about print:

- Its name.
- Its parameters.
- Its return value.



This is called its public interface. A function also contains a set of instructions, but those are implementation details and are private. The problem with using tuples to model objects is that, contrary to functions, they fully expose their internal details. We can visualize this as follows:

What we need is to make the complexity private (red) and only wrap it inside a simple public interface (green).

13.1.2. Safety

Abstraction also provides safety. Take, for example, a television. It contains plenty of electronics, but you're not supposed to touch those. Instead, the television comes with "public interface", i.e., a remote control. All interactions with your television should go through this remote control. If the television is well made, there is no combination of remote control buttons you could press that would *break* the television. The remote control should be 100% safe to use. However, tinkering with the internals of the television is a lot riskier. Even though you could probably change the channel by prodding the right circuit, prodding the wrong circuit could break your television. The same is true with code. The public interface (2) should be made in such a way that

- it's easy to use; and
- it cannot break the object.

In later sections, we'll give you more concrete examples.

13.2. Classes

We gave you a long explanation about the virtues of abstraction. However, just telling you straight away how to achieve it will probably leave you utterly confused, as there are multiple concepts involved. Instead, we will explain the new concepts one by one. So, if at first it is not clear to you how these concepts aid in achieving abstraction, no need to worry, we'll get there eventually.

13.2.1. Box

Let's start very simple, with a rather silly example.

```
class Box:
    def __init__(self):   # Box.__init__
        self.contents = 0
```

Here, we defined a *class* named Box. Note that the class itself is not a box, it merely describes what boxes look like. Let's create a box:

```
box = Box()
```

As you can see, Box is callable as if it were a function. Box() creates and returns an actual Box object. A class contains functions, but because they're inside a class, they are called methods instead. Box defines just one ~~function~~ method, namely __init__. The name has a special meaning: it tells Python it is the constructor. A constructor's responsibility is to initialize newly made Box objects. When you create a new Box, Python creates an empty object in memory and passes it to __init__, i.e., the self parameter refers to the fresh object. __init__ then performs self.content = 0 which means "let this new object have a field named content and set it to 0". After execution box = Box(), the variable box references this initialized object. We can use our new Box as follows:

```
print(box.contents)
```

box.contents asks the box for the value of its contents field. This field has been set to 0 by the constructor, so box.contents will evaluate to 0. You can update this field as you see fit:

```
box.contents = "Hello"
```

A class is merely a description of what objects will look like. Once you have defined a class, you can instantiate it, i.e., creating objects that conform to what the class describes. In the example above, the Box class states that all Box objects shall have a contents field that's initially set to 0. You can then proceed to make as many Box objects as you wish:

```
box1 = Box()
box2 = Box()
box3 = Box()
```

Each of these boxes will have their own contents.

```
box1.contents = 1
box2.contents = "Hello"
box3.contents = True
```

We called the __init__ method the constructor. There seems to be some disagreement about whether this is actually the right term. In this course, we'll keep using the term constructor, but realize that you may find other sources disagreeing with this choice of terminology.

13.3. Constructor Parameters

The __init__ method creates and initializes fields. It is possible to pass it parameters so that you can choose what the field's values are set to.

```
class Box:
    def __init__(self, initial_contents):1
        self.contents = initial_contents
```

```
class Box:
    def __init__(self):
        self.contents = 0
```

Creating a Box object now requires a parameter:

```
>>> box = Box(5)
>>> box.contents
5
```

It is often easier to have the parameter have the same name as the field:

```
class Box:
    def __init__(self, contents):
        self.contents = contents
```

13.4. Methods

13.4.1. is_empty

Let's add a method to our Box class.

```
class Box:
    def __init__(self, contents):
        self.contents = contents

    def is_empty(self):
        return self.contents is None
```

box.is_empty() can now be used to check if our box is empty, i.e., contains None.

```
>>> box = Box(5)
>>> box.is_empty()
False

>>> box.contents = None
>>> box.is_empty()
True
```

Notice that although is_empty has a parameter self, the call to is_empty does not provide an argument. This is a peculiarity of object-oriented syntax: the self parameter is actually set to the Box object that receives the method call. In a way, you can think of box.is_empty() being the same as is_empty(box). The self parameter must appear as the first parameter of all methods of all classes. Due to this ubiquity, many other programming languages made the self parameter (or its equivalent) implicit, i.e., you don't need to mention it every time because it is added automatically for you behind the scenes.

13.4.2. put

We add another methods to Box.

```
class Box:
    def __init__(self, contents):
        self.contents = contents

    def is_empty(self):
        return self.contents is Nones

    def put(self, contents):
        self.contents = contents
```

We can use these methods as follows:

```
>>> box = Box(5)
>>> box.put("Gwyneth")
>>> box.contents
"Gwyneth"
```

Make sure to notice the call to put: the method expects two parameters, but the call only mentions one argument. As explained before, the first parameter is the Box object on which the method is called, so you can see box.put("data") is actually being put(box, "data"), which fits better with the definition. At this point, the methods we defined offer absolutely no added value: they don't contain any nontrivial logic and don't do anything that we couldn't do ourselves. This is due to the fact that contents is publicly available: the methods are nothing more than useless middle men. However, things are about to change.

13.5. Private Members

```
class Box:
    def __init__(self, contents):
        self.contents = contents

    def is_empty(self):
        return self.contents is None

    def put(self, contents):
        self.contents = contents
```

This class definition tells us that a Box object has three members:

- a field ‘contents’;
- three methods named is_empty, and put.

All these members are publicly available, i.e., anyone with access to a Box object can make use of these members. When we told you about abstraction, we explained how we should *hide* the internals and define a clean, easy-to-use public interface. So, let's hide something, namely the field contents.

```
class Box:
    def __init__(self, contents):
        self.__contents = contents

    def is_empty(self):
        return self.__contents is None

    def put(self, contents):
        self.__contents = contents
```

The only change we needed to make is rename contents to __contents (__contents is a private member). Python recognizes this naming pattern and understands that you mean to make that field private:

```
>>> box = Box(8)
>>> box.__contents
AttributeError: 'Box' object has no attribute '__contents'
```

Our Box class has become even more useless than before! We can't even access its contents anymore. Let's make it available again by defining a method:

```
class Box:
    def __init__(self, contents):
        self.__contents = contents

    def is_empty(self):
        return self.__contents is None

    def put(self, contents):
        self.__contents = contents

    def contents():
        return self.__contents
```

Let's put it to use:

```
>>> basket = Box("lotion")
>>> basket.contents()
"lotion"
```

Before:

```
class Box:
    def __init__(self,
contents):
        self.__contents =
contents

    def is_empty(self):
        return
self.__contents is None

    def put(self, contents):
        self.__contents =
contents
```

Let's summarize:

- __contents is a private field.
- There are three public methods: is_empty, put and contents. They give us access to __contents in an albeit indirect way.

We'll discuss the benefits of this in the next section.

13.6. Hiding Data

We repeat the code of the Box class for your benefit:

```
class Box:
    def __init__(self, contents):
        self.__contents = contents

    def is_empty(self):
        return self.__contents is None

    def put(self, contents):
        self.__contents = contents

    def contents():
        return self.__contents
```

We made the field (= data) private and replaced it with public methods (= code). The idea of replacement of data by code is fundamental in the world of software engineering: it shows up time and again if you pay attention to it. But why is this such a powerful concept? Data is *dumb*. If contents were simply a public field, anyone can do whatever they want with it: anyone can read it or overwrite it with arbitrary values. Code, however, is smart: it allows you to specify any behavior you want. Say for example you want to restrict what kind of values are allowed in the box, e.g., only numbers. The only way to update the Box's contents is through put. If we add a check there, we can control what goes in the box:

```
class Box:
    def __init__(self, contents):
        self.__contents = contents

    def is_empty(self):
        return self.__contents is None

    def put(self, contents):
        # Checks whether parameter is an integer
        if isinstance(contents, int):
            self.__contents = contents
        else:
            raise ValueError()

    def contents():
        return self.__contents
```

A bit of explanation is in order:

- isinstance(value, type) checks whether value has type type. Examples of types are int, str, bool, list, tuple, dict, any class name, ...
- raise ValueError() raises an exception. We'll discuss these in another chapter. Suffice it to say, raising an exception is a way for a function or method to say "I refuse to do this".

```
>>> box = Box(5)
>>> box.put("invalid value")
ValueError

>>> box.contents()
5
```

13.6.1. Loophole

There is still a loophole though: the constructor does not check its parameter. Let's fix this:

```
class Box:
    def __init__(self, contents):
        if isinstance(contents, int):
            self.__contents = contents
        else:
            raise ValueError()

    def is_empty(self):
        return self.__contents is None

    def put(self, contents):
        if isinstance(contents, int):
            self.__contents = contents
        else:
            raise ValueError()

    def contents():
        return self.__contents
```

13.6.2. Getting Rid of Redundancy

The same checking code is repeated twice, which is generally a bad idea. We rewrite our code a bit:

```
class Box:
    def __init__(self, contents):
        self.put(contents)

    def is_empty(self):
        return self.__contents is None

    def put(self, contents):
        if isinstance(contents, int):
            self.__contents = contents
        else:
            raise ValueError()

    def contents():
        return self.__contents
```

As you can see, the constructor simply delegates to put. Remark that fields can be created by any method, not just the constructor.

13.6.3. Finishing Touches

We can improve on Box a little bit more:

```
class Box:
    def __init__(self, contents):
        self.put(contents)

    def is_empty(self):
        return self.__contents is None

    def put(self, contents):
        if self.valid_contents(contents):
            self.__contents = contents
        else:
            raise ValueError()

    def contents():
        return self.__contents

    def valid_contents(self, contents):
        return isinstance(contents, int)
```

The validation is now moved to a separate method named valid_contents. This change is in accordance with Curly's Law that states that everything should have exactly one purpose. In our experience, it is a good rule to live by and can avoid many a headache.

13.7. Properties

Say you have the following class:

```
class Box:
    def __init__(self, contents):
        self.contents = contents# Box().contents                # Not
Box().contents()!!
```

As you can see, the contents field is public. Say you rely on this Box class all over your code base, so that it is riddled with stuff like

```
print(box.contents)
box.contents = "800"
box.contents *= 2
```

Now it turns out that you need to make contents a bit more intelligent, like maybe you need all changes to the contents to be logged somewhere. In order to accomplish this, you need to upgrade contents from dumb data to smart code:

```
class Box:
    def __init__(self, contents):
        self.put(contents)

    def put(self, contents):
        log_change()
        self.__contents = contents

    def contents(self):
        return self.__contents
```

However, this change makes it necessary to update all other code that relies on Box:

```
print(box.contents())
box.contents.put("800")
box.contents.put(box.contents() * 2)
```

This is a *serious* problem: a small change in one class ripples throughout the entire code base. One of the highest priorities in software engineering is that changes can remain localized, i.e., that one component can be modified locally without it having any impact on other components. One could claim we could have avoided this problem by showing some foresight: we should've known better than to directly expose contents and go straight for the method-approach, even though at the time we didn't need to. This is a dangerous mentality, however: one could easily foresee all kinds of potential changes and start designing their classes to try to take into account all possible future needs. This inevitably leeds to an overengineered mess. Luckily, Python provides solutions that allow you to start with a simple solution first (You aren't gonna need it YAGNI) and, if need be, gracefully improve it without needing to break other code. Python's properties is one such solution. Here's what a Box relying on properties would look like:

```
class Box:
    def __init__(self,
contents):
        self.put(contents)

    def put(self, contents):
        log_change()
        self.__contents =
contents

    def contents(self):
        return self.__contents


What do we have to change in
other places?
```

```
class Box:
    def __init__(self, contents):
        self.contents = contents

    # This is the getter
    @property
    def contents(self):
        return self.__contents

    # This is the setter
    @contents.setter
    def contents(self, value):
        log_change()
        self.__contents = value
```

And this is how you would use it:

```
>>> box = Box(5)
>>> print(box.contents)
>>> 5
>>> box.contents *= 2           # box.contents = box.contents * 2
```

As you can see from the usage, interacting with a property has the exact same syntax as interacting with a field. This aspect is very important: it means that you can upgrade a field to a property without the outside world noticing. box.content causes the getter to be called. In other words, you can define yourself what actually happens when the property (field) is read. Writing to box.content, e.g., box.content = 10, will call the setter. Similarly, you can fully customize what happens when writing to a property (field): you can perform validation, logging, notify others of the change, etc. Defining a setter is optional: if you leave it out, the property will be considered readonly, i.e., attempts to write to it will result in an error. A property only consists of a getter and optional setter. In general, you'll still need a separate (private) field to actually have some memory to store the value in. The @property defines a getter method, while @contents.setter defines a setter method. Using prints, we can demonstrate when the getter and setter are being called.

```
class Box:
    def __init__(self, contents):
        self.contents = contents

    # This is the getter
    @property
    def contents(self):
        print('Reading')
        return self.__contents

    # This is the setter
    @contents.setter
    def contents(self, value):
        print('Writing')
        self.__contents = value

>>> box = Box(5)
Writing      # Because constructor sets property

>>> box.contents
Reading      # Because property's getter was called
5            # Property returned 5
```

```
>>> box.contents = 10
Writing      # Because property's setter was called

>>> box.contents *= 2
Reading      # Because the property's value needs to be read first
Writing      # Writing the property's value after it has been doubled
```

14. File IO

In this chapter, we'll deal with files. Files can be used to store any kind of data: images, video, audio, text, documents, fonts, etc. Generally, an extension is used to indicate what kind of data is stored:

- .jpg, .gif and .png are popular image formats.
- .mp3, .m4a, .ogg, .wav. .mid are typical audio formats.
- .pdf, .docx, .html are used for documents.
- and so on.

14.1.1. Binary vs Text Files

We distinguish between two kinds of files: binary files and text files. This terminology, even though it's standard, can be a bit confusing: *all* files ultimately contain a long sequence of 0 and 1s, so in that sense, *all* files are binary. So, let's clarify the exact difference between these two types of file.

- A *text file* stores data in human-readable form. All bytes are to be interpreted as characters (ASCII/Unicode). Examples of human-readable file formats are html, md, csv, xml, json and yaml. Each line ends with a newline character “\n”
- A *binary file* does not try to be human-readable and can therefore store data in much more compact form. Most file formats are binary: .jpg, .gif, .png, .mp3, docx, etc.

To actually see the difference, use Visual Studio Code (or any other text editor) to open an .html file and a .jpg. Consider numbers: in a text file, a number will be written in decimal form, each digit written out as a separate byte. For example, the number 10000 would take up five bytes, as we need five digits to represent it. In a binary file, we would wonder how many bytes we actually need to represent 10000. The answer is two, as two bytes allow us to represent integers up to 65,535. We could even go further and think in terms of bits: we really only need 14 bits. In reality it's a little more complicated, but we don't want to fill pages on that topic here. We have to leave something to the other courses :-)

14.2. Reading files

In this section we discuss how we can read data from text files.

14.2.1. Opening Files

Files can be read from and written to. That is really their purpose. However, multiple programs could try to access the same file at the same time. This would lead to chaos: the two programs would overwrite each other's data, and the file would become corrupt, i.e., the data inside it makes no sense. To prevent this, the operating system demands that you open a file first. You can then interact with the file. After you're done, you close the file. Handle is a variable that provides access to a file. A handle represents a list of lines same as .readlines(). Durgin changes, the information is stored in a buffer. In case of a crash, a buffer still contains the information. Example with a bank account: You are making a transaction, it crashes, where does the money go? At the end you can flush the buffer to clear it for the new informatin. Python locks a file when opened. The OS will make sure no two programs have the same file open at the same time. If you try to open a file that's already in use by another program, you will get an error. It's also important that you close a file, hereby giving others a chance to use the file. As always, reality is a bit more complex. For example, multiple programs reading from the same file at the same time is perfectly safe. However, once one program needs to write it, it needs exclusive access. This is why when opening a file, you need to mention whether you intend to read or write to it. Based on that, the OS can decide whether to grant you access or not. Opening and closing a file in Python can be done using

```
file = open("my-file.txt")
# interact with file
file.close()
```

This approach is a bit risky however: what if something bad happens while interacting with the file? The file.close() statement would then be skipped, making it impossible for other programs to access the file. For this reason, Python offers a special construct:

```
with open("my-file.txt") as file:
    # interact with file
```

Here, at the end of the with block, the file will automatically be closed. No matter what happens while interacting with the file, it is guaranteed that it will be closed at the end. In other words, you should always rely on with when dealing with files. If you write code that opens a file, then crashes before closing it again, there's no problem: the OS will clean up after you. However, a script can recover from crashes (see exceptions). In this case, the close operation will be skipped yet your program keeps running. As long as it runs, the file will remain open, making it inaccessible to others. This can become a big problem in the case of long running programs.

14.2.2. Encoding

There are many ways texts can be encoded: ASCII, UTF-8, EBCDIC and so on. These days most files are encoded using UTF-8. It is strongly recommended that you make the encoding explicit when opening the file. This is done as follows:

```
with open("my-file.txt", encoding='utf-8') as file:
    # interact with file
```

The encoding= parameter is known as a keyword argument. Suffice it to say that Python allows you to explicitly mention the parameter name, for the sake of clarity. Keyword arguments have a few more advantages, but we won't discuss them in detail right now.

14.2.3. Reading from Files (3)

Say you open a file using

```
with open("my-file.txt", encoding='utf-8') as file:
    # interact with file
```

Inside the with block, the variable file is set to a special object representing the opened file. Note that file is just an identifier (called handle): you can pick any name you want. You can invoke methods on it, just like you could with strings, using the syntax file.method_name(arguments). We discuss some of these methods, specifically those that let you read from the file.

- `file.read()` reads the entire contents of the file and returns it as a *single string*. You probably shouldn't use this approach when the file is large.

- `file.readlines()` reads (almost) the entire contents of the file and returns it as an *array of strings*, where each string correspond to a line.
- `file.readline()` reads the next line in the file and returns it as string. If no more lines are left in the file, an empty string is returned.

.readlines() and .read() cannot be used if the file is too big, because they cannot fit into a working memory. An opened file keeps track of a "current stream position", i.e., it remembers to what point you have read from the file. Initially this position is set to the beginning of the file. file.readline() advances this position to the beginning of the next line, so that the next time you invoke file.readline() it knows where to look for the next line. file.read() and file.readlines() both read the entire contents in one go, therefore the position is moved to the end of the file. Whenever you read lines (i.e., using readline or readlines), know that each string will contain the newline character \n. In other words, a line abc in the file will be represented by "abc\n".

.rstrip() removes \n, \t, whitespace, …

.exist() does a path exist

.isfile() checks if it's a file

.isdir() checks if it's a directory

Say we have a file input.txt with the following contents:

```
First line
Second line
Third line
Fourth line
Fifth line
# We open it the "wrong" way (= not using with) so that
# we can work step by step for the sake of this example

>>> file = open('input.txt', encoding='utf-8')

>>> file.readline()
"First line\n"

>>> file.readline()
"Second line\n"

# Also starts reading from the current stream position
>>> file.readlines()
["Third line\n", "Fourth line\n", "Fifth line"]

# Current stream position is at end of file, no more lines left
>>> file.readline()
""
```

14.3. Writing files

All data a Python program generates disappear as soon as it ends. If you want your program to "remember" data from the previous time it was run, you have to store this data somewhere. Typically this is done using files.

14.3.1. Opening a File with Write Access

As discussed earlier, files need to be opened prior to interacting with them. By default, we only get read access. In order to get write access, we have to request this explicitly:

```
with open(filename, 'w', encoding='utf-8') as file:
    # Interact with file
```

The 'w' tells the OS we intend to write. Note that opening a file in mode w will **empty that file**: all data that was in it is lost. The possible modes for opening a file are

Mode	Description
'r'	Read only
'w'	Write only (**empties** file first!), (creates a file if it doesn't exists)
'r+'	Reading and writing
'a'	Appending (writing at end)

14.3.2. Writing to a File (3)

The following methods can be used to write to a file:

- `file.write(string)` writes the given string to the file. The current stream position moves with it: you can call write multiple times and each string will be added to the end of the file. Appends.
- `file.writelines(strings)` writes all strings to the file.

Note that neither of these methods add newlines.

```
with open('output.txt', 'w', encoding='utf-8') as file:
    file.writelines(['a', 'b', 'c'])
```

The file `output.txt' now contains

```
abc
```

Note how, even though the method is called writelines, all strings are written to the same line. In order to put the strings on separate lines, you have to add the newlines yourself:

```
with open('output.txt', 'w', encoding='utf-8') as file:
    file.writelines(['a\n', 'b\n', 'c\n'])
```

An alternative approach to writing to a file is to rely on print:

```
with open(output_file, 'w') as output:
    print(string, file=output_file)
```

Here, we named the file output to avoid confusion: print has a parameter named file which can be used to express where to send the string to. By default, this is STDOUT, i.e., to the screen. In our case, we want the data to be sent to a file. Note that print does automatically add a newline, so each string you feed to print will end up on a separate line. Handle is a variable that provides access to a file. A handle represents a list of lines same as .readlines(). Durgin changes, the information is stored in a buffer. In case of a crash, a buffer still contains the information. Example with a bank account: You are making a transaction, it crashes, where does the money go? At the end you can flush the buffer to clear it for the new informatin.

`.rstrip()` removes \n, \t, whitespace, …

`.exist()` does a path exist

`.isfile()` checks if it’s a file

`.isdir()` checks if it’s a directory