BY FATOS MORINA

Python Programming

A QUICK AND EASY GUIDE

Python Programming

Fatos Morina

Contents

Introduction to Python

Python is one of the most popular programming languages that was created in 1991 by Guido van Rossum.

According to Guido van Rossum, Python is a:

> "high-level programming language, and its core design philosophy is all about code readability and a syntax which allows programmers to express concepts in a few lines of code."

It represents one of the easiest languages that you can learn and also work with. It can be used for software development, server side of web development, machine learning, mathematics, and any type of scripting that you can think of.

The good thing about it is that it is widely used and accepted through many companies and academic institutions, making it a really good choice especially if you are just starting your coding journey. Moreover, it has a large community of developers that use it and are willing to help. This community has already published many open source libraries that you can start using. They also actively keep improving them.

Its syntax is quite similar to that of the English language, making it easier for you to understand and use it quite intuitively.

Python runs on an interpreter system, meaning that you are not expected to wait for a compiler to compile the code and then execute it. You can instead build prototypes fast.

It works on different platforms, such as Windows, Linux, Mac, Rapsberry Pi, etc.

The Basics

In comparison to other languages, Python places a special importance on indentations.

In other programming languages, white spaces and indentations are only used to make the code more readable and prettier, whereas in Python they represent a sub-block of code.

The following code is not going to work, since the second line is not properly intended:

```
1  if 100 > 10:
2  print("100 is greater than 10")
```

For it to work properly, the indentation should look like the following:

```
1  if 100 > 10:
2      print("100 is greater than 10")
```

This may look hard to understand, but you can have code editors that highlight such syntax errors quite vividly. Moreover, the more Python code you write, the easier it gets for you to consider such indentations as second nature.

Comments

We use comments to specify parts that the program should be simply ignored and not executed by the Python interpreter. This means that everything written inside a comment is not taken into

consideration at all and you can write in any language that you want, including your own native language.

We can start comments with the symbol # in front of the line:

```
# This is a comment in Python
```

We can also include more extended comments in multiple lines using triple quotes:

```
"""
This is a comment in Python.
Here we are typing in another line and we are still insid\
e the same comment block.

In the previous line we have a space, because it is allow\
ed to have spaces inside comments.

Let us end this comment right here.
"""
```

print() method

We are going to use the method `print()` since it helps us see results in the console.

You do not need to know how it works behind the scenes or even know what a method is right now. Just think of it as a way for us to display results from our code in the console.

Operators

Arithmetic operators

Even though you may pull out your phone from your pocket and do some calculations, you should also get used from now on to implement some arithmetic operators inside Python.

When we want to add two numbers, we use the plus sign, just like in Math:

```
print(50 + 4)  # 54
```

Similarly, for subtraction, we use the minus sign:

```
print(50 - 4)  # 46
```

For multiplication, we use the star sign:

```
print(50 * 4)  # 200
```

When doing divisions, we use the forward slash sign:

```
print(50 / 4)  # 12.5
print(8 / 4)  # 2.0
```

This results in float numbers. If we want to only get the integer number when doing divisions also known as simply doing floor divisions, we should use the double fraction sign:

```
print(50 // 4)  # 12
print(8 / 4)  # 2
```

We can also find the remainder of a number divided by another number using the percent sign `%`.

```
print(50 % 4)  # 2
```

This operation can be helpful especially in cases when we want to check whether a number is odd or even. A number is odd if when divided by 2, the remainder is 1. Otherwise, it is even.

Here is an illustration:

```
print(50 % 2)  # 0
# Since the remainder is 0, this number is even

print(51 % 2)  # 1
#Since the remainder is 1, this number is odd
```

When we want to raise a number to a specific power, we should use the double star sign:

```
print(2 ** 3)  # 8
# This is a short way of writing 2 * 2 * 2

print(5 ** 4)  # 625
# This is a short way of writing 5 * 5 * 5 * 5
```

Assignment operators

These operators are used to assign values to variables.

When we declare a variable, we use the equal sign:

```
name = "Fatos"
age = 28
```

We can also declare multiple variables in the same line:

```
name, age, location = "Fatos", 28, "Europe"
```

We can use this way to also swap values between variables, for example, let us say that we have two variables `a` and `b` and want to switch their values.

One logical way to do that would be to introduce a third variable that serves as a temporary variable:

```
a, b = 1, 2

print(a)  # 1
print(b)  # 2

c = a
a = b
b = c

print(a)  # 2
print(b)  # 1
```

However, we can do that in a single line in the following way:

```
a, b = 1, 2

print(a)  # 1
print(b)  # 2

b, a = a, b

print(a)  # 2
print(b)  # 1
```

We can also have a merge of the assignment operator with arithmetic operators.

Let us see how we can do that for addition first.

Let us assume that we have the following variables:

```
total_sum = 20

current_sum = 10
```

Now we want to add the value of the `current_sum` to `total_sum`. To do that, we should write the following:

```
total_sum = total_sum + current_sum

print(total_sum)  # 30
```

This may look as not accurate, since the right hand side is not equal to the left hand side. However, in this case we are simply doing an assignment and not a comparison of both sides of the equation.

To do this quickly, we can use the following form:

```
total_sum += current_sum

print(total_sum)  # 30
```

This is equivalent to the previous statement.

Similarly, we can do the same with other arithmetic operators:

- Subtraction:

```
result = 3
number = 4

result -= number  # This is equal to result = result - nu\
mber

print(result)  # -1
```

- Multiplication:

```
product = 3
number = 4

product *= number  # This is equal to product = product *\
 number

print(product)  # 12
```

- Division:

```
result = 8
number = 4

result *= number  # This is equal to result = result / nu\
mber

print(result)  # 2.0
```

- Modulo operator:

```
result = 8
number = 4

result %= number  # This is equal to result = result % nu\
mber

print(result)  # 0
```

- Power operator:

```
result = 2
number = 4

result **= number  # This is equal to result = result ** \
number

print(result)  # 16
```

Comparison operators

We have learned in elementary school to do comparisons of numbers, such as checking whether a specific number is larger than another number, or whether they are equal.

We can use almost the same operators in Python to do such comparisons.

Let us see them in action.

Equalities

Checking whether two numbers are equal can be done using the `==`:

```
print(2 == 3)  # False
```

The last expression evaluates to False since `2` is not equal to `3`.

There is another operator that can be used to check whether `2` numbers are not equal. This is an operator that you may have not had the chance to see in your math classes written exactly in this way. This is the operator `!=`.

Let us do a comparison of whether `2` is not equal to `3`:

```
print(2 != 3)  # True
```

This expression evaluates to True since `2` is indeed not equal to `3`.

Inequalities

Now here we are going to see how to check whether a number is larger than another number:

```
print(2 > 3)  # False
```

This is something that you should already know from your math classes.

When trying to check whether a number is greater than or equal to another number, we need to use this operator `>=`:

```
print(2 >= 3)  # False
```

Similarly, for the other side, we have:

```
print(2 < 3)  # True
print(2 <= 3)  # True
```

Logical operators

In high school math, you have probably learned about logical operators such as `and` and `or`.

Briefly speaking, for an expression to evaluate to True when using `and`, both statements should be true. In Python, we implement it using `and`:

```
print(5 > 0 and 3 < 5)  # True
```

This example is going to evaluate to True since `5` is greater than `0`, which evaluates to True, and `3` is less than `5`, which also evaluates to True, from which we get, True and True, which evaluates to True.

Let us take an example when an `and` expression is going to evaluate to `False`:

```
print(2 > 5 and 0 > -1)  # False
```

`2` is not greater than `5`, so the statement in the left hand side is going to evaluate to False. No matter what's on the right hand, we are going to get the whole expression equal to `False`, since False and whatever value else is going to result to False.

When we want to have statements that at least one of them should evaluate to True, then we should use `or`:

```
print(2 > 5 or 0 > -1)  # True
```

This is going to evaluate to True since the statement in the right hand side evaluates to True.

If both statements are False, then `or` results with `False` in the end:

```
print(2 < 0 or 0 > 1)  # False
```

This is False since `0` is not greater than `2` and also `0` is not greater than `1`. Hence, the whole expression is False.

Data types

Variables

Variables can be considered as the building blocks of any computer program that you can think of.

They can be used to store values and then be reused as many times as you want. The moment you want to change their value, you can just change it in one place and that new value that you just changed is going to get reflected everywhere else where this variable is used.

Every variable in Python is an object.

A variable is created in the moment it is initialized with a value.

Here are the general rules for Python variables:

- A variable name must start with a letter or the underscore character. It cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
- Variable names are case-sensitive, meaning that `height`, `Height`, `HEIGHT` are all different variables.

Let us define our first variable:

```
1  age = 28
```

In this example, we are initializing a variable with the name age and assigning the value 28 to it.

We can define other variables along with it, like:

```
age = 28
salary = 10000
```

We can use pretty much any name that we want, but it is a better practice to use names that can be well understood both by you and other work colleagues who work with you.

We have other variable types in Python, such as float numbers, strings, and boolean values. We can even create our own custom types.

Let us see an example of a variable that holds a float number:

```
height = 3.5
```

As you can see, this initialization is quite similar to the ones when we had integer numbers. Here we are only changing the value on the right and Python interpreter is smart enough to know that we are dealing with another type of variable, namely a float type of variable.

Let us see an example of a string:

```
reader = "Fatos"
```

We put string values either in quotes or in apostrophes to specify the value that we want to store in string variables.

When we want to store boolean values, we need to use reserved words, namely, *True* and *False*.

```
text_visibile = False
```

We can also have a boolean value stored when we have expressions that result with a boolean value, for example, when we compare a number with another number, such as:

```
is_greater = 5 > 6
```

This variable is going to get initialized with the value False since 5 is lower than 6.

Numbers

We have three numeric types in Python: integers, floats and complex numbers.

Integers

Integers represent whole numbers that can be both positive and negative and do not contain any decimal part.

Here are a few examples of integers: `1`, `3000`, `-31234`, etc.

When adding, subtracting, or multiplying two integers, we get an integer as a result in the end.

```
print(3 + 5)  # 8
print(3 - 5)  # -2
print(3 * 5)  # 15
```

These are all integers.

This also includes cases when we raise a number to a power:

```
result = 3 ** 4  # This is similar to multiplying 3 * 3 *\
 3 * 3 in which case, we are multiplying integers togethe\
r
print(result)  # 81
```

When we want to divide two integers, we are going to get a float number.

Boolean

Boolean type represents truth values True and False. We include the explanation of this type inside this *Numbers* section, since booleans are indeed subtypes of the integer type.

More specifically, almost always a False value can be considered as a `0`, whereas a True value can be considered as a `1`.

As such, we can also do arithmetic operations with them:

```
print(True * 5)  # 5
print(False * 500)  # 0, since False is equal to 0
```

The exception for such integer representations of Boolean values is when we these values are strings such as “False” and “True”.

Floats

Float numbers are numbers that contain the decimal part, such as `-3.14`, `12.031`, `9,3124`, etc.

We can also convert float numbers using the `float()` function:

```
ten = float(10)

print(ten)  # 10.0
```

When adding, subtracting, or dividing two float numbers, or a float number and an integer, we get a float number as a result in the end:

```
print(3.4 * 2)  # 6.8
print(3.4 + 2)  # 5.4
print(3.4 - 2)  # 1.4
print(2.1 * 3.4)  # 7.14
```

Complex numbers

Complex numbers have both the real and the imaginary part that we write in the following way:

```
complex_number = 1 + 5j

print(complex_number)  # (1+5j)
```

Strings

Strings represent characters that are enclosed in either single quotes or double quotes and both of them are treated the same:

```
name = "Fatos"  # Double quotes

name = 'Fatos'  # Single quotes
```

If we want to include a quote inside a string, we need to let Python know that it should not close the string but simply escape that quote:

```
greeting = 'Hello. I\'m fine.'  # We escaped the apostrop\
he

double_quote_greeting = "Hello. I'm fine."  # When using \
double quotes, we do not need to escape the apostrophe
```

When we want to include a new line in a string, we can include the special character `\n`:

```
my_string = "I want to continue \n in the next line"

print(my_string)
# I want to continue
# in the next line
```

Since strings are arrays of characters, we can index specific characters using indexes. Indexes start from 0 and go all the way until the length of the string minus 1. We exclude the index that is equal to the length of the string since we start indexing from 0 and not from 1.

Here is an example:

```
string = "Word"
```

In this example, if we were to select individual characters of the string, they would unfold as follows:

```
string = "Word"

print(string[0])  # W
print(string[1])  # o
print(string[2])  # r
print(string[3])  # d
```

We can use negative indexes as well, which means that we start from the end of the string with `-1`. We cannot use `0` to index from the end of a string, since `-0 = 0`:

```
print(string[-1])  # d
print(string[-2])  # r
print(string[-3])  # o
print(string[-4])  # W
```

We can also do slicing and include only a portion of the string and not the entire string, for example, if we want to get characters that start from a specific index until a specific index, we should write it in the following way: `string[start_index:end_index]`, excluding the character at index `end_index`:

```
string = "Word"

print(string[0:3])  # Wor
```

If we want to start from a specific index and continue getting all the remaining characters of the string until the end, we can omit specifying the end index, as follows:

```
string = "Word"

print(string[2:])  # rd
```

If we want to start from `0` and go until a specific index, we can simply specify the ending index:

```
string = "Word"

print(string[:2])  # Wo
```

This means that the value of `string` is equal to `string[:2]` (excluding the character at position 2) + `string[2:]`.

Note that strings in Python are immutable, meaning that once a string object has been created, it cannot be modified, such as trying to change a character in a string.

As an illustration, let us say that we want to change the first character in the string, namely switching `W` with `A` in the following way:

```
string = "Word"
string[0] = "A"
```

Now, if we try to print `string`, we would get an error like the following:

```
# TypeError: 'str' object does not support item assignment
```

If we need a new string, we should simply create a new one.

String operators

We can concatenate strings using the + operator:

```
first = "First"
second = "Second"

concantenated_version = first + " " + second

print(concatenated_version)  # First Second
```

We can use the multiplication operator * with strings and numbers.

This can be used to repeat the string that many times. For example, if we want to repeat a string 5 times, but we do not want to write it manually five times, we can simply multiply it with the number 5:

```
string = "Abc"

repeated_version = string * 5

print(repeated_version)  # AbcAbcAbcAbcAbc
```

String built-in methods

There are a few built-in methods of strings that we can use that can make it easier for us to manipulate with them.

len()

`len()` is a method that we can use to get the length of a string:

```
sentence = "I am fine."

print(len(sentence))  # 10
```

replace()

`replace()` can be used to replace a character or a substring of a string with another character or substring:

```
string = "Abc"

modified_version = string.replace("A", "Z")

print(modified_version)  # Zbc
```

strip()

`strip()` removes white spaces that can be in the beginning or at the end of a string:

```
string = " Hi there "

print(string.strip())  # Hi there
```

split()

`split()` can be used to convert a string into an array of substrings based on a specific pattern that is mentioned as the separator. For example, let us assume that we want to save all words of a sentence in an array of words. These words are separated by white spaces, so we will need to do the splitting based on that:

```
sentence = "This is a sentence that is being declared her\
e"

print(string.split(" "))
# ['This', 'is', 'a', 'sentence', 'that', 'is', 'being', \
'declared', 'here']
```

join()

`join()` is the opposite of `split()`: from an array of strings, it returns a string. The process of concatenation is supposed to happen with a specified separator in between each element of the array that in the end results with a single concatenated string:

```
words = ["cat", "dog", "rabbit"]

print(" - ".join(words))  # cat - dog - rabbit
```

count()

`count()` can be used to count the number of times a character or a substring appears in a string:

```
string = "Hi there"

print(string.count("h"))  # 1, since, it is case sensitiv\
e and 'h' is not equal to 'H'

print(string.count("e"))  # 2

print(string.count("Hi"))  # 1

print(string.count("Hi there"))  # 1
```

find()

`find()` can be used to find a character or a substring in a string and returns the index of it. In case it does not find it, it will simply return `-1`:

```
string = "Hi there"

print(string.find("3"))  # -1

print(string.find("e"))  # 5

print(string.find("Hi"))  # 0

print(string.find("Hi there"))  # 0
```

lower()

`lower()` can be used to convert all characters of a string into lower case:

```
string = "Hi there"

print(string.lower())  # hi there
```

upper()

`upper()` can be used to convert all characters of a string into lower case:

```
string = "Hi there"

print(string.upper())  # HI THERE
```

capitalize()

`capitalize()` can be used to covert the first character of a string into uppercase:

```
string = "hi there"

print(string.capitalize())  # Hi there
```

title()

`title()` can be used to convert starting characters of each word (sequences that are separated by white spaces) of a string into uppercase:

```
string = "hi there"

print(string.title())  # Hi There
```

isupper()

`isupper()` is a method that can be used to check whether all characters in a string are upper case:

```
string = "are you HERE"
another_string = "YES"

print(string.isupper())  # False
print(another_string.isupper())  # True
```

islower()

`islower()` similarly can be used to check whether all characters are lower case:

```
string = "are you HERE"
another_string = "no"

print(string.islower())  # False
print(another_string.islower())  # True
```

isalpha()

`isalpha()` returns True if all characters in a string are letters of the alphabet:

```
string = "A1"
another_string = "aA"

print(string.isalpha())  # False, since it contains 1
print(another_string.isalpha())  # True since both `a` an\
d `A` are letters of the alphabet
```

isdecimal()

`isdecimal()` returns True if all characters in a string are numbers:

```
string = "A1"
another_string = "3.31"
yet_another_string = "3431"

print(string.isdecimal())  # False, since it contains 'A'
print(another_string.isdecimal())  # False, since it cont\
ains '.'
print(yet_another_string.isdecimal())  # True, since it c\
ontains only numbers
```

Formatting

Formatting a string can be quite useful since you may be using it quite frequently no matter the type of project or script that you are working on.

Let us first illustrate why do we need formatting and include string interpolation.

Imagine that I want to develop a software that greets people the moment they come in, such as:

```
greeting = "Good morning Fatos."
```

This now looks great, but I am not the only person who uses it, right?

I am just one of the people who gets to use it.

Now if someone comes and signs in, I will have to use their own names, such as:

```
greeting = "Good morning Besart."
```

This is just my first real user who is registering. I am not counting myself.

Now let us think that I get lucky and the second user also pops in on a Friday morning and our application should display:

```
greeting = "Good morning Betim."
```

As you can see, we are getting somewhere from a business perspective since two new users just showed up, but this is not a scalable implementation. We are writing a very static `greeting` expression.

You should already remember that we are about to mention something that was introduced just at the beginning.

Yes, we will need to use variables and include a variable next to the string, as follows:

```
greeting = "Good morning " + first_name
```

This is much more flexible.

This is one way of doing that.

Another way that I am referring to is using a method called `format()`.

We can use curly braces to specify places where we want to put dynamic values, such as the following:

```
greeting = "Good morning {}. Today is {}.".format("Fatos"\
, "Friday")
```

This is now going to put the first parameter of the method `format()` inside the first curly braces, which in our case is `Fatos`. Then, in the second occurrence of the curly braces, it is going to put the second parameter of the method `format()`.

If we try to print the value of the string, we should get the following:

```
print(greeting)
# Good morning Fatos. Today is Friday.
```

We can specify parameters with indexes inside curly braces like the following that can then be used:

```
greeting = "Today is {1}. Have a nice day {0}".format("Fa\
tos", "Friday")

print(greeting)
# greeting = "Today is {1}. Have a nice day {0}.".format(\
"Fatos", "Friday")
```

We can also specify parameters inside the `format()` method and use those specific words inside curly braces as a reference:

```
greeting = "Today is {day_of_the_week}. Have a nice day {\
first_name}.".format(first_name="Fatos",
                                                        \
                     day_of_the_week="Friday")
print(greeting)  # Today is Friday. Have a nice day Fatos.
```

We can combine both types of arguments in a single example, such as the following:

```
short_bio = 'My name is {name}. My last name is {0}. I lo\
ve {passion}. I like playing {1}.'.format(
    'Morina',
    'Basketball',
    name='Fatos',
    passion='Programming'
)

print(short_bio)
# My name is Fatos. My last name is Morina. I love Progra\
mming. I like playing Basketball.
```

As you can see, using named arguments as opposed to positional ones can be less error prone, since their ordering in the `format()` method does not matter.

We can also use another way of formatting strings which consists of beginning a string with `f` or `F` before the opening quotation mark or triple quotation mark and include names of the variables that we want to be included in the end:

```
first_name = "Fatos"
day_of_the_week = "Friday"
continent = "Europe"

greeting = f'Good morning {first_name}. Today is {day_of_\
the_week}'

print(greeting)  # Good morning Fatos. Today is Friday.
```

Here is another example where we are using a triple quotient after the `F`:

```
continent = "Europe"

i_am_here = F'''I am in {continent}'''

print(i_am_here)  # I am in Europe
```

Lists

If you take a look at a bookshelf, you can see that they are stacked and put closely together. You can see that there are many examples of collecting, and structuring elements in some way. This is also quite important in computer programming. We cannot just continue declaring countless variables and manage them that easily.

Let us say that we have a class of students and want to save their names. We can start saving their names according to the way they are positioned in the classroom:

```
first = "Albert"
second = "Besart"
third = "Fisnik"
fourth = "Festim"
fifth = "Gazmend"
```

The list can keep on going and it is also quite hard for us to keep track of all of them.

There is fortunately an easier way for us to put these in a collection in Python called list.

Let us create a list called students and store in them all the names declared in the previous code block:

```
students = ["Albert", "Besart", "Fisnik", "Festim", "Gazm\
end"]
```

This is prettier, right?

Moreover, this way, it is easier for us to manage and also manipulate with the elements.

You may think that, "Well it was easier for me to just call first and get the value stored in there. Now it is impossible to get a value from this new list, called students".

If we would not be able to read and use those elements that we just stored in a list, that would make it less helpful.

Fortunately, lists have indexes, which start from 0, meaning that if we want to get the first element in a list, we need to use index 0 and not index 1 as you may think.

In the example above, the list items have these corresponding indexes:

```
students = ["Albert", "Besart", "Fisnik", "Festim", "Gazm\
end"]
# Indexes               0                1                   2
```

Now, if we want to get the first element, we simply write:

```
students[0]
```

If we want to get the second element, we just write:

```
students[1]
```

As you can probably get the gist, we simply need to write the name of the list and also the corresponding index of the element that we want to get in the square brackets.

This list is of course not static. We can add elements to it, like when a new student joins the class.

Let us add a new element in the list students with the value Besfort:

```
students.append("Besfort")
```

We can also change the value of an existing element. To do that, we need to simply reinitialize that specific element of the list with a new value.

For example, let us change the name of the first student:

```
students[0] = "Besim"
```

Lists can contain contain different types of variables, for example, we can have a string that contains integers, float numbers, and strings:

```
combined_list = [3.14, "An element", 1, "Another element \
here"]
```

Slicing

Similar to strings, lists can also be sliced, which as a result returns new lists, meaning that the original lists remain unchanged.

Let us see how we can get the first three elements of a list using slicing:

```
my_list = [1, 2, 3, 4, 5]

print(my_list[0:3])  # [1, 2, 3]
```

As you can see, we have specified `0` as the starting index and `3` as the index where the slicing should stop, excluding the element at the ending index.

If we want to simply start from an index and get all the remaining elements in the list, meaning that the end_index should be the last index, then we can omit and not have to write the last index at all:

```
my_list = [1, 2, 3, 4, 5]

print(my_list[3:])  # [4, 5]
```

Similarly, if we want to start from the beginning of the list and do the slicing until a specific index, then we can omit writing the `0` index entirely, since Python is smart enough to infer that:

```
my_list = [1, 2, 3, 4, 5]

print(my_list[:3])  # [1, 2, 3]
```

Strings in Python are immutable, whereas lists are mutable, meaning that we can modify lists' content after we declare them.

As an illustration, let us say that we want to change the first character in the string, namely switching `S` with `B` in the following way:

```
string = "String"
string[0] = "B"
```

Now, if we try to print `string`, we would get an error like the following:

```
# TypeError: 'str' object does not support item assignment
```

Now if we have a list and want to modify its first element, then we can successfully do so:

```
my_list = ["a", "b", "c", "d", "e"]

my_list[0] = 50

print(my_list)  # [50, 'b', 'c', 'd', 'e']
```

We can expand a list by concatenating it with another list using the + operator:

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

second_list = [4, 5]

first_list = first_list + second_list

print(first_list)  # [1, 2, 3, 4, 5]
```

Nesting a list inside a list

We can nest a list inside another list:

```
math_points = [30, "Math"]

physics_points = [53, "Phyiscs"]

subjects = [math_points, physics_points]

print(subjects)  # [[30, 'Math'], [53, 'Phyiscs']]
```

These lists do not need even have to have the same length.

To access elements of a list which is inside a list we need to use double indexes.

Let us see how we can access the element `math_points` inside the `subjects` list. Since `math_points` is an element in the `subjects` list positioned at index 0, we simply need to do the following:

```
print(subjects[0])  # [30, 'Math']
```

Now let us assume that we want to access `Math` inside `subjects` list. Since `Math` is at index `1`, we are going to need to use the following double indexes:

```
print(subjects[0][1])  # 'Math'
```

List methods

`len()` is a method that can be used to find the length of a list:

```
my_list = ["a", "b", 1, 3]

print(len(my_list))  # 4
```

Adding elements

We can also expand lists by adding new elements, or we can also delete elements.

Adding new elements at the end of a list can be done by using the `append()` method:

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

my_list.append("New element")

my_list.append("Yet another new element")

print(my_list)
# ['a', 'b', 'c', 'New element', 'Yet another new element\
']
```

If we want to add elements at specific indexes in a list, we can use the `insert()` method, where we specify the index in the first argument, and the element that we want to add in the list as a second argument:

```
my_list = ["a", "b"]

my_list.insert(1, "z")

print(my_list)  # ['a', 'z', 'b']
```

Deleting elements from lists

We can delete elements from lists using the `pop()` method, which removes the last element in the list:

```
my_list = [1, 2, 3, 4, 5]

my_list.pop()  # removes 5 from the list

print(my_list)  # [1, 2, 3, 4]

my_list.pop()  # removes 4 from the list

print(my_list)  # [1, 2, 3]
```

We can also specify the index of an element in the list that indicates which element in the list we should delete:

```
my_list = [1, 2, 3, 4, 5]

my_list.pop(0)  # Delete the element at index 0

print(my_list)  # [2, 3, 4, 5]
```

We can also delete elements from lists using `del` statement and then specifying the value of the element that we want to delete:

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

del my_list[0]  # Delete element my_list[0]

print(my_list)  # [2, 3, 4, 5]
```

We can also delete slices of lists using `del`:

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

del my_list[0:3]  # Delete elements: my_list[0], my_list[\
1], my_list[2]

print(my_list)  # [4, 1]
```

This can also be done using `remove()`:

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

my_list.remove(3)

print(my_list)  # [1, 2, 4]
```

`reverse()` can be used to reverse the elements in a list. This is quite easy and straightforward:

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

my_list.reverse()

print(my_list)  # [4, 3, 2, 1]
```

Index search

Getting elements of a list using indexes is simple. Finding indexes of elements of a list is also easy. We simply need to use the method `index()` and mention the element that we want to find inside a list:

```
my_list = ["Fatos", "Morina", "Python", "Software"]

print(my_list.index("Python"))  # 2
```

Membership

This is quite intuitive and related to real life: We get to ask ourselves whether something is part of something or not.

- Is my phone in my pocket or bag?

- Is my coworker's email included in the CC?
- Is my friend in this coffee shop?

In Python, if we want to check whether a value is part of something, that we can use the operator `in`:

```
my_list = [1, 2, 3]  # This is a list

print(1 in my_list)  # True
```

Since 1 is included in the array `[1, 2, 3]`, the expression evaluates to True.

We can also use it not only with arrays of numbers, but with arrays of characters as well:

```
vowels = ['a', 'i', 'o', 'u']
print('y' in vowels)  # False
```

Since `y` is not a vowel and not included in the declared array, the expression in the second line of the previous code snippet is going to result to False.

Similarly, we can also check whether something is not included using `not in`:

```
odd_numbers = [1, 3, 5, 7]
print(2 not in odd_numbers)  # True
```

Since `2` is not included in the array, the expression is going to evaluate to True.

Sorting

Sorting elements in a list is something that you may need to do from time to time. `sort()` is a built-in method that makes it possible for you to sort elements in a list in an ascending order alphabetically or numerically:

```
my_list = [3, 1, 2, 4, 5, 0]

my_list.sort()

print(my_list)  # [0, 1, 2, 3, 4, 5]

alphabetical_list = ['a', 'c', 'b', 'z', 'e', 'd']

alphabetical_list.sort()

print(alphabetical_list)  # ['a', 'b', 'c', 'd', 'e', 'z']
```

There are other methods of lists that we have not included in here.

List comprehension

List comprehension represents a concise way in which we use a `for` loop to create a new list from an existing one. The result is always a new list in the end.

Let us start with an example where we want to multiply each number of a list with 10 and save that result in a new list. First, let us do this without using list comprehensions:

```
numbers = [2, 4, 6, 8]  # Complete list

numbers_tenfold = []  # Empty list

for number in numbers:
    number = number * 10  # Multiply each number with 10
    numbers_tenfold.append(number)  # Add that new number\
 in the new list

print(numbers_tenfold)  # [20, 40, 60, 80]
```

We can implement that using list comprehensions in the following way:

```
numbers = [2, 4, 6, 8]  # Complete list

numbers_tenfold = [number * 10 for number in numbers]  # \
List comprehension

print(numbers_tenfold)  # [20, 40, 60, 80]
```

We can also include conditions when doing these list comprehensions.

Let us assume that we want to save a list of positive numbers.

Before we write the way we would implement this using list comprehensions, let us write a way in which we would create a list of only numbers that are greater than 0 in another list and increase those positive numbers by 100:

```
positive_numbers = []  # Empty list

numbers = [-1, 0, 1, -2, -3, -4, 3, 2]  # Complete list

for number in numbers:
    if number > 0:  # If the current number is greater th\
an 0
        positive_numbers.append(number + 100)  # add that\
 number inside the list positive_numbers


print(positive_numbers)  # [101, 103, 102]
```

We can do the same using list comprehension:

```
numbers = [-1, 0, 1, -2, -3, -4, 3, 2]  # Compelete list

positive_numbers = [number + 100 for number in numbers if\
 number > 0]  # List comprehension

print(positive_numbers)  # [101, 103, 102]
```

As you can see, this is much shorter and should take less time to write.

We can also use list comprehensions with multiple lists as well.

Let us take an example where we want to add each element of a list with each element in another list:

```
first_list = [1, 2, 3]
second_list = [50]

double_lists = [first_element +
                second_element for first_element in first\
_list for second_element in second_list]

print(double_lists)  # [51, 52, 53]
```

In the end, we are going to get a resulting list that has the same number of elements as the list with the longest length.

Tuples

Tuples are collections that are ordered and immutable, meaning that their content cannot be changed. They are ordered and their elements can be accessed using indexes.

Let us start with our first tuple:

```
vehicles = ("Computer", "Smartphone", "Smart watch", "Tab\
let")

print(vehicles)

# ('Computer', 'Smartphone', 'Smart watch', 'Tablet')
```

All the indexing and slicing operations that we have seen in the chapter of lists apply for tuples as well:

```
print(len(vehicles))  # 4

print(vehicles[3])  # Tablet

print(vehicles[:3])  # ('Computer', 'Smartphone', 'Smart \
watch')
```

Finding the index of an element inside a tuple can be done using the `index()` method:

```
print(vehicles.index('tablet'))  # 3
```

We can also concatenate or merge two tuples using the `+` operator:

```
natural_sciences = ('Chemistry', 'Astronomy',
                    'Earth science', 'Physics', 'Biology')

social_sciences = ('Anthropology', 'Archaeology', 'Econom\
ics', 'Geography',
                   'History', 'Law', 'Linguistics', 'Poli\
tics', 'Psychology', 'Sociology')

sciences = natural_sciences + social_sciences

print(sciences)
# ('Chemistry', 'Astronomy', 'Earth science', 'Physics', \
'Biology', 'Anthropology', 'Archaeology', 'Economics', 'G\
eography', 'History', 'Law', 'Linguistics', 'Politics', '\
Psychology', 'Sociology')
```

Membership check

We can check whether an element is part of a tuple using operators `in` and `not in` just like with lists:

```
vehicles = ('Car', 'Bike', 'Airplane')

print('Motorcycle' in vehicles)  # False, since Motorcycl\
e is not included in vehicles

print('Train' not in vehicles)  # True, since Train is no\
t included in vehicles
```

Nesting 2 tuples

Instead of merging, we can also nest tuples into a single tuple by using tuples that we want to nest inside the parenthesis:

```
natural_sciences = ('Chemistry', 'Astronomy',
                    'Earth science', 'Physics', 'Biology')

social_sciences = ('Anthropology', 'Archaeology', 'Econom\
ics', 'Geography',
                   'History', 'Law', 'Linguistics', 'Poli\
tics', 'Psychology', 'Sociology')

sciences = (natural_sciences, social_sciences)

print(sciences)
# (('Chemistry', 'Astronomy', 'Earth science', 'Physics',\
 'Biology'), ('Anthropology', 'Archaeology', 'Economics',\
 'Geography', 'History', 'Law', 'Linguistics', 'Politics'\
, 'Psychology', 'Sociology'))
```

Immutability

Since tuples are immutable, they cannot be changed after they are created. This means that we cannot add, or delete elements in them, or append a tuple to another tuple.

We cannot even modify existing elements in it. If we try to modify an element in a tuple, we are going to face a problem like the following:

```
vehicles = ('Car', 'Bike', 'Airplane')

vehicles[0] = 'Truck'

print(vehicles)
# TypeError: 'tuple' object does not support item assignm\
ent
```

Dictionaries: Key-Value Data Structures

As we saw previously, elements in lists are associated with indexes that can be used to reference those elements. There is another data structure in Python that allows us to specify our own custom indexes and not just numbers. These are called dictionaries and they are similar to dictionaries that we use to find the meaning of foreign words we do not understand.

Let us assume that you are trying to learn German and there is a new word that you have not had the chance to learn it before that you just saw at a market: Wasser.

Now, you can pick up your phone and check its corresponding meaning in English using Google Translate or any other application of your choice, but if you were to use a physical dictionary, you would need to find this word by going to that specific page and check its meaning sitting right beside there. The reference or the key for the meaning of this word would be the term Wasser.

Now, if we want to implement this in Python, we should not use lists that have indexes only as numbers. We should use dictionaries

instead.

For dictionaries, we use curly braces and have each element that has two parts: the key and the value. In our previous example, the key was the German word, whereas the value was its translation in English, as can be seen in the following example:

```
german_to_english_dictionary = {
    "Wasser": "Water",
    "Brot": "Bread",
    "Milch": "Milk"
}
```

Now, when we want to access specific elements in the dictionary, we simply use keys, for example, let us assume that we want to get the meaning of the word Brot in English. To do that, we can simply reference that element using that key:

```
brot_translation = german_to_english_dictionary["Brot"]
print(translation)  # Bread
```

When we print the value that we get, we are going to get the translation in English.

Similarly, we can get the English translation of the word Milch by getting the value of the element that has Milch as the key:

```
milch_translation = german_to_english_dictionary["Milch"]
print(milch_translation)  # Milk
```

We can also get the value of an element in a dictionary using `get()` and specifying the key of the item that we want to get:

```
german_to_english_dictionary.get("Wasser")
```

We can create dictionaries using `dict()`:

```
words = dict([
    ('abandon', 'to give up to someone or something on th\
e ground'),
    ('abase', 'to lower in rank, office, or esteem'),
    ('abash', 'to destroy the self-possession or self-con\
fidence of')
])

print(words)
# {'abandon': 'to give up to someone or something on the \
ground', 'abase': 'to lower in rank, office, or esteem', \
'abash': 'to destroy the self-possession or self-confiden\
ce of'}
```

Both keys and values can be of any data type.

We can have more than one key with the same value, but keys should be unique.

Adding new values

We can add new values inside dictionaries by specifying a new key and a corresponding value and then Python is going to create a new element inside that dictionary:

```
words = {
    'a': 'alfa',
    'b': 'beta',
    'd': 'delta',
}

words['g'] = 'gama'

print(words)
# {'a': 'alfa', 'b': 'beta', 'd': 'delta', 'g': 'gama'}
```

If we specify the key of an element that is already part of the dictionary, that element is going to be modified:

```
words = {
    'a': 'alfa',
    'b': 'beta',
    'd': 'delta',
}

words['b'] = 'bravo'


print(words)
# {'a': 'alfa', 'b': 'bravo', 'd': 'delta'}
```

Removing elements

If we want to remove elements from a dictionary, we can use the method `pop()` and also specify the key of the element that we want to delete:

```
words = {
    'a': 'alfa',
    'b': 'beta',
    'd': 'delta',
}

words.pop('a')

print(words)  # {'b': 'beta', 'd': 'delta'}
```

We can also delete values using `popitem()` which removes the last inserted key-value pair starting from Python 3.7. In earlier versions, it deletes a random pair:

```
words = {
    'a': 'alfa',
    'b': 'beta',
    'd': 'delta',
}

words['g'] = 'gamma'

words.popitem()

print(words)
# {'a': 'alfa', 'b': 'beta', 'd': 'delta'}
```

There is another way that we can delete elements, namely by using `del` statement:

```
words = {
    'a': 'alfa',
    'b': 'beta',
    'd': 'delta',
}

del words['b']

print(words)  # {'a': 'alfa', 'd': 'delta'}
```

Length

We can get the length of a dictionary using `len()` just like with lists and tuples:

```
words = {
    'a': 'alfa',
    'b': 'beta',
    'd': 'delta',
}

print(len(words))  # 3
```

Membership

If we want to check whether a key is already part of a dictionary so that we avoid overriding it, we can use the operator `in` and `not in` just like with lists and tuples:

```
words = {
    'a': 'alfa',
    'b': 'beta',
    'd': 'delta',
}

print('a' in words)  # True
print('z' not in words)  # True
```

Comprehension

We can use comprehension just like with lists to create dictionaries in a quick way.

To help us with that, we are going to need to use a method called `items()` that converts a dictionary into a list of tuples where the element in index `0` is a key, whereas in position with index `1`, we have a value.

Let us first see the method `items()` in action:

```
points = {
    'Festim': 50,
    'Zgjim': 89,
    'Durim': 73
}

elements = points.items()

print(elements) # dict_items([('Festim', 50), ('Zgjim', 8\
9), ('Durim', 73)])
```

Now let us create a new dictionary from this existing dictionary `points` using comprehension. We can assume that a professor is in a good mood and generous enough to reward each student with a bonus of `10` points. We want to add these new points to each student by saving these new points in a new dictionary:

```
points = {
    'Festim': 50,
    'Zgjim': 89,
    'Durim': 73
}

elements = points.items()

points_modified = {key: value + 10 for (key, value) in el\
ements}

print(points_modified)  # {'Festim': 60, 'Zgjim': 99, 'Du\
rim': 83}
```

Sets

Sets are unordered and unindexed collections of data. Since elements in sets are not ordered, we cannot access elements using indexes or using the method `get()`.

We can add tuples, but we cannot add dictionaries or lists in a set.

We cannot add duplicate elements in sets. This means that when we want to remove duplicate elements from another type of collection, we can make use of this uniqueness in sets.

Let us start creating our first set using curly brackets as follows:

```
first_set = {1, 2, 3}
```

We can also create sets using the `set()` constructor:

```
empty_set = set()  # Empty set

first_set = set((1, 2, 3))  # We are converting a tuple i\
nto a set
```

Like all data structures, we can find the length of a set using the method `len()`:

```
print(len(first_set))  # 3
```

Adding elements

Adding one element in a set can be done using the method `add()`:

```
my_set = {1, 2, 3}

my_set.add(4)

print(my_set)  # {1, 2, 3, 4}
```

If we want to add more than one element, then we need to use method `update()` and as an input for this method a list, tuple, string or another set:

```
my_set = {1, 2, 3}

my_set.update([4, 5, 6])

print(my_set)  # {1, 2, 3, 4, 5, 6}

my_set.update("ABC")

print(my_set)  # {1, 2, 3, 4, 5, 6, 'A', 'C', 'B'}
```

Deleting elements

If we want to delete elements from sets, we can use methods `discard()` or `remove()`:

```
my_set = {1, 2, 3}

my_set.remove(2)

print(my_set)  # {1, 3}
```

If we try to delete an element that is not part of the set using `remove()`, then we are going to get an error:

```
my_set = {1, 2, 3}

my_set.remove(4)

print(my_set)  # KeyError: 4
```

To avoid such errors when removing elements from sets, we can use the method `discard()`:

```
my_set = {1, 2, 3}

my_set.discard(4)

print(my_set)  # {1, 2, 3}
```

Set Theory Operations

If you remember high school math lessons, you should already know union, intersection, and the difference between two sets of elements. These operations are supported for sets in Python as well.

Union

Union represents the collection of all unique elements from both sets. We can find the union of two sets using the pipe operator `|` or the `union()` method:

```
first_set = {1, 2}
second_set = {2, 3, 4}

union_set = first_set.union(second_set)

print(union_set)  # {1, 2, 3, 4}
```

Intersection

Intersection represents the collection that contains elements that are in both sets. We can find it using operator `&` or the `intersection()` method:

```
first_set = {1, 2}
second_set = {2, 3, 4}

intersection_set = first_set.intersection(second_set)

print(union_set)  # {2}
```

Difference

The difference between two sets represents the collection that contains only the elements that are in the first set, but not in the second. We can find the difference between two sets using the - operator or the method `difference()`

```
first_set = {1, 2}
second_set = {2, 3, 4}

difference_set = first_set.difference(second_set)

print(difference_set)  # {1}
```

As you can probably remember from high school, ordering of sets when we find the difference of two sets matters, which is not the case with the union and intersection.

This is similar to arithmetic, where `3 - 4` is not equal to `4 - 3`:

```
first_set = {1, 2}
second_set = {2, 3, 4}

first_difference_set = first_set.difference(second_set)

print(first_difference_set)  # {1}

second_difference_set = second_set.difference(first_set)

print(second_difference_set)  # {3, 4}
```

Type Conversions

Conversions between primitive types

Python is an object oriented programming. That is why it uses constructor functions of classes to do conversions from one type into another.

int()

`int()` is a method that is used to do a conversion of an integer literal, float literal (rounding it to its previous integer number, i.e. 3.1 to 3), or a string literal (with the condition that the string represents an int or float literal): :

```
three = int(3)  # converting an integer literal into an i\
nteger
print(three)  # 3

four = int(4.8)  # converting a float number into its pre\
vious closest integer
print(four)  # 4

five = int('5')  # converting a string into an integer
print(five)  # 5
```

float()

`float()` is similarly used to create float numbers from an integer, float, or string literal (with the condition that the string represents an `int` or `float` literal):

```
int_literal = float(5)
print(int_literal)  # 5.0

float_literal = float(1.618)
print(float_literal)  # 1.618

string_int = float("40")
print(string_int)  # 40.0

string_float = float("37.2")
print(string_float)  # 37.2
```

str()

`str()` can be used to create strings from strings, integer literals, float literals, and many other data types:

```
int_to_string = str(3)
print(int_to_string)  # '3'

float_to_string = str(3.14)
print(float_to_string)  # '3.14'

string_to_string = str('hello')
print(string_to_string)  # 'hello'
```

Other Conversions

The way we can convert from one type of data structure into another type is the following:

```
destination_type(input_type)
```

Let us get started with specific types, so that it becomes much clearer.

Conversions to lists

We can convert a set, tuple, or dictionary into a list using the `list()` constructor.

```
books_tuple = ('Book 1', 'Book 2', 'Book 3')
tuple_to_list = list(books_tuple)  # Converting tuple to \
list
print(tuple_to_list)  # ['Book 1', 'Book 2', 'Book 3']


books_set = {'Book 1', 'Book 2', 'Book 3'}
set_to_list = list(books_set)  # Converting set to list
print(set_to_list)  # ['Book 1', 'Book 2', 'Book 3']
```

When converting a dictionary into a list, only its keys are going to make it into a list:

```
books_dict = {'1': 'Book 1', '2': 'Book 2', '3': 'Book 3'}
dict_to_list = list(books_dict)  # Converting dict to list
print(dict_to_list)  # ['1', '2', '3']
```

If we want to keep both keys and values of a dictionary, we need to use the method `items()` to first convert it into a list of tuples where each tuple is a key and a value:

```
books_dict = {'1': 'Book 1', '2': 'Book 2', '3': 'Book 3'}

dict_to_list = list(books_dict.items())  # Converting dic\
t to list

print(dict_to_list)
# [('1', 'Book 1'), ('2', 'Book 2'), ('3', 'Book 3')]
```

Conversions to tuples

All data structures can be converted to a tuple using the `tuple()` constructor method, including a dictionary in which case we get a tuple with the keys of the dictionary:

```
books_list = ['Book 1', 'Book 2', 'Book 3']
list_to_tuple = tuple(books_list)  # Converting tuple to \
tuple
print(list_to_tuple)  # ('Book 1', 'Book 2', 'Book 3')


books_set = {'Book 1', 'Book 2', 'Book 3'}
set_to_tuple = tuple(books_set)  # Converting set to tuple
print(set_to_tuple)  # ('Book 1', 'Book 2', 'Book 3')


books_dict = {'1': 'Book 1', '2': 'Book 2', '3': 'Book 3'}
dict_to_tuple = tuple(books_dict)  # Converting dict to t\
uple
print(dict_to_tuple)  # ('1', '2', '3')
```

Conversions to sets

Similarly, all data structures can be converted to a set using the `set()` constructor method, including a dictionary in which case we get a set with the keys of the dictionary:

```
books_list = ['Book 1', 'Book 2', 'Book 3']
list_to_set = set(books_list)  # Converting list to set
print(list_to_set)  # {'Book 2', 'Book 3', 'Book 1'}


books_tuple = ('Book 1', 'Book 2', 'Book 3')
tuple_to_set = set(books_tuple)  # Converting tuple to set
print(tuple_to_set)  # {'Book 2', 'Book 3', 'Book 1'}


books_dict = {'1': 'Book 1', '2': 'Book 2', '3': 'Book 3'}
dict_to_set = set(books_dict)  # Converting dict to set
print(dict_to_set)  # {'1', '3', '2'}
```

Conversions to dictionaries

Conversions into dictionaries cannot be done with any type of sets, lists, or tuples, since dictionaries represent data structures where each element contains both a key and a value.

Converting a list, or a tuple into a dictionary can be done if each element in a list is also a list with two elements, or a tuple with two elements.

```
books_tuple_list = [(1, 'Book 1'), (2, 'Book 2'), (3, 'Bo\
ok 3')]
tuple_list_to_dictionary = dict(books_tuple_list)  # Conv\
erting list to set
print(tuple_list_to_dictionary)  # {1: 'Book 1', 2: 'Book\
 2', 3: 'Book 3'}

books_list_list = [[1, 'Book 1'], [2, 'Book 2'], [3, 'Boo\
k 3']]
tuple_list_to_dictionary = dict(books_list_list)  # Conve\
rting list to set
```

```
print(tuple_list_to_dictionary)  # {1: 'Book 1', 2: 'Book\
 2', 3: 'Book 3'}


books_tuple_list = ([1, 'Book 1'], [2, 'Book 2'], [3, 'Bo\
ok 3'])
tuple_list_to_set = dict(books_tuple_list)  # Converting \
tuple to set
print(tuple_list_to_set)  # {'Book 2', 'Book 3', 'Book 1'}

books_list_list = ([1, 'Book 1'], [2, 'Book 2'], [3, 'Boo\
k 3'])
list_list_to_set = dict(books_list_list)  # Converting tu\
ple to set
print(list_list_to_set)  # {'Book 2', 'Book 3', 'Book 1'}
```

In case when we want to convert a set into a dictionary, we need to have each element as a tuple of length 2.

```
books_tuple_set = {('1', 'Book 1'), ('2', 'Book 2'), ('3'\
, 'Book 3')}
tuple_set_to_dict = dict(books_tuple_set)  # Converting d\
ict to set
print(tuple_set_to_dict)  # {'1', '3', '2'}
```

If we try to do a conversion of a set that has each element as a list of length 2 into a dictionary, we are going to get an error:

```
books_list_set = {['1', 'Book 1'], ['2', 'Book 2'], ['3',\
 'Book 3']}
list_set_to_dict = dict(books_list_set)  # Converting dic\
t to set
print(list_set_to_dict)  # {'1', '3', '2'}
```

After we run the last code block, we are going to get an error:

```
TypeError: unhashable type: 'list'
```

Wrap Up

In conclusion, Python has a variety of data types that you can use to store data. These data types are important to know so that you can choose the right one for your needs. Be sure to use the right data type for the task that is in front of you to avoid errors and optimize performance.

Control Flow

Conditional statements

When you think about ways we think and also communicate with each other, you may get the impression that we are indeed always using conditions.

- If it's 8 am, I take the bus and go to work.
- If I am hungry, I eat.
- If this item is cheap, I can afford it.

This is also something that you can do in programming. We can use conditions to control the flow of the execution.

To do that, we use the reserved term if and an expression that evaluates to a True or False value. We can then also use an *else* statement where we want the flow to continue in cases when the if condition is not met.

To make it easier to understand, let us assume that we have an example where we want to check whether a number is positive:

```
1  if number > 0:
2      print("The given number is positive")
3  else:
4      print("The given number is not positive")
```

If we were to have number = 2: we would enter into the if branch and execute the command that is used to print the following text in the console:

```
The given number is positive
```

If we would have another number, such as -1, we would see in the console the following message being printed:

```
The given number is not positive
```

We can also add additional conditions and not just 2 like above by using elif which is evaluated when the if expression is not evaluated.

Let us see an example to make it easier for you to understand it:

```
if number > 0:
    print("The given number is positive")
elif number == 0:
    print("The given number is 0")
else:
    print("The given number is negative")
```

Now if we were to have number = 0, the first condition is not going to be met, since the value is not greater than 0. As you can guess, since the given number is equal to 0, we are going to see the following message being printed in the console:

```
The given number is 0
```

In cases when the value is negative, our program is going to pass the first two conditions since they are not satisfied and then jump into the else branch and print the following message in the console:

```
The given number is negative
```

Looping / Iterator

Looping represents the ability of the program to execute a set of instructions over and over again until a certain condition is met. We can do it with both while and for.

Let us first see the iteration with for.

for loop

This looping is simple and quite straightforward. All you have to do is specify a starting state and mention the range in which it should iterate, as can be seen in the following example:

```
1  for number in range(1, 7):
2      print(number)
```

In this example, we are iterating from 1 to 7 and printing each number (from 1 up to 7 excluding 7) in the console.

We can change both the starting and the ending numbers in the range as we want. This way, we can be quite flexible depending on our specific scenarios.

while loop

Let us now describe iterations with while. This is also another way of doing iterations that is also quite straightforward and intuitive. Here we need to specify a starting condition before the while block and also update the condition accordingly. The **while** loop needs a "**loop condition.**" If it stays True, it continues iterating. In this example, when `num` is `11` the **loop condition** equals `False`.

```
number = 1

while number < 7:
    print(number)
    number += 1  # This part is necessary for us to add s\
o that the iteration does not last forever
```

This while block is going to print the same statements as the code we used with the for block.

Iteration: Looping Through Data Structures

Now that we have covered both iteration and lists, we can jump into ways of iterating through lists.

We do not just store things into data structures and leave them there for ages. We are supposed to be able to use those elements in different scenarios.

Let us take our list of students from before:

```
students = ["Albert", "Besart", "Fisnik", "Festim", "Gazm\
end"]
```

Now, to iterate through the list, we can simply type:

```
for student in students:
    print(student)
```

Yes, it's that simple. We are iterating through each element in the list and printing their values.

We can do this for dictionaries as well. However, since elements in dictionaries have 2 parts (key and the value), we need to specify both the key and the value as follows:

```
german_to_english_dictionary = {
    "Wasser": "Water",
    "Brot": "Bread",
    "Milch": "Milk"
}

for key, value in german_to_english_dictionary:
    print("The German word " + key + " means " + value + \
" in English")
```

We can also get only keys from the elements of the dictionary:

```
for key in german_to_english_dictionary:
    print(key)
```

Note that key and value are simply variable names that we have chosen to illustrate the iteration, but we can use any name that we want for our variables, such as the following example:

```
for german_word, english_translation in german_to_english\
_dictionary:
    print("The German word " + german_word + " means " + \
english_translation + " in English")
```

This iteration is going to print the same thing in the console as the code block before the last one.

We can also have nested for loops, for example, let us say that we want to iterate through a list of numbers and finding a sum of each element with each other element of a list. We can do that using nested `for` loops:

```
numbers = [1, 2, 3]
sum_of_numbers = []  # Empty list

for first_number in numbers:
    for second_number in numbers:  # Loop through the lis\
t and add the numbers
        current_sum = first_number + second_number
        # add current first_number from the first_list to\
 the second_number from the second_list
        sum_of_numbers.append(current_sum)


print(sum_of_numbers)
# [2, 3, 4, 3, 4, 5, 4, 5, 6]
```

Stopping a for-loop

Sometimes we may need to exit a `for` loop before it reaches the end. This may be the case when a condition has been met or we have found what we were looking for and there is no need to continue any further.

In those situations, we can use `break` to stop any other iteration of the `for` loop.

Let us assume that we want to check whether there is a negative number in a list. In case we find that positive number, we stop searching for it.

Let us implement this using `break`:

```
my_list = [1, 2, -3, 4, 0]

for element in my_list:
    print("Current number: ", element)
    if element < 0:
        print("We just found a negative number")
        break

# Current number:  1
# Current number:  2
# Current number:  -3
# We just found a negative number
```

As we can see, the moment we reach -3, we break from the `for` loop and stop.

Skipping an iteration

There can also be cases when we want to skip certain iterations since we are not interested in them and they do not matter that much. We can do that using `continue` which prevents code execution below it in that code block and moves the execution procedure towards the next iteration:

```
my_sum = 0
my_list = [1, 2, -3, 4, 0]

for element in my_list:
    if element < 0:  # Do not include negative numbers in\
 the sum
        continue
    my_sum += element

print(my_sum)  # 7
```

`pass` is a statement that can be used to help us when we are about to implement a method, or something but haven't done yet and do not want to get errors.

It helps us execute the program even if some parts of the code are missing:

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

for element in my_list:
    pass  # Do nothing
```

Wrap Up

In conclusion, Python offers conditional statements to help you control the flow of your program. The `if` statement lets you run a block of code only if a certain condition is met. The `elif` statement lets you run a block of code only if another condition is met. And the `else` statement lets you run a block of code only if no other condition is met. These statements are very useful for controlling the flow of your program.

Functions

There can be plenty of cases when we need to use the same code block again and again. Our first guess would be to write it as many times as we want.

Objectively, it does work, but the truth is, this is a really bad practice. We are doing repetitive work that can be quite boring and also prone to many mistakes that can be overlooked.

This is the reason why we need to start using code blocks that we can define once and use that same code of instructions anywhere else.

Just think about this in real life: You see a YouTube video that has been recorded and uploaded to YouTube once. It is then going to be watched by many other people, but the video still remains the same one that was uploaded initially.

In other words, we use methods as a representative of a set of coding instructions that are then supposed to be called anywhere else in the code and that we do not have to write it repeatedly. In cases when we want to modify this method, we simply change it at the place where it was first declared and other places where it is called do not have to do anything.

To define a method in Python, we start by using the `def` keyword, then the name of the function and then a list of arguments that we expect to be used. After that, we need to start writing the body of the method in a new line after an indentation.

```
1  def add(first_number, second_number):
2      our_sum = first_number + second_number
3      return our_sum
```

As you can see from the coloring, both `def` and `return` are keywords in Python that you cannot use to name your variables.

Now, everywhere we want this `add()` to be called, we can just call it there and not have to worry about implementing it entirely.

Since we have defined this method, we can call it in the following way:

```
result = add(1, 5)

print(result)  # 6
```

You can get the impression that this is such a simple method and start asking, why are we even bothering to write a method for it?

You are right. This was a very simple method just to get you introduced to the way we can implement functions.

Let us write a function that finds the sum of numbers that are between two specified numbers:

```
def sum_in_range(starting_number, ending_number):
    result = 0

    while starting_number < ending_number:
        result = result + starting_number
        starting_number = starting_number + 1

    return result
```

This is now a set of instructions that you can call in other places and do not have to write all of it again.

```
result = sum_in_range(1, 5)

print(result)  # 10
```

Note that functions define a scope such that variables that are defined in there are not accessible outside. For example, we cannot access the variable named `product` outside the scope of the function:

```
def multiply_in_range(starting_number, ending_number):
    product = 1
    while starting_number < ending_number:
        product = product * starting_number
        starting_number = starting_number + 1
    return product
```

`product` is accessible only inside the body of this method.

Default arguments

When we call functions, we may make some of the arguments optional by writing an initial value for them at the header of the function.

Let us take an example of getting a user's first name as a required argument and let the second argument be an optional one.

```
def get_user(first_name, last_name=""):
    return f"Hi {first_name} {last_name}"
```

Let us now call this function with both arguments:

```
user = get_user("Durim", "Gashi")

print(user)  # Hi Durim Gashi
```

We can now call that same function even though the the second argument is not specified:

```
user = get_user("Durim")

print(user)  # Hi Durim
```

Keyword argument list

We can define arguments of functions as keywords:

```
# The first argument is required. The other two are optio\
nal
def get_user(number, first_name='', last_name=''):
    return f"Hi {first_name} {last_name}"
```

Now, we can call this function by writing arguments as keywords:

```
user = get_user(1, last_name="Gashi")

print(user)  # Hi  Gashi
```

As you can see, we can omit `first_name` since it is not required. We can also change the ordering of the arguments when calling the function and it will still work the same:

```
user = get_user(1, last_name="Gashi", first_name='Durim')

print(user)  # Hi Durim Gashi
```

Data lifecycle

Variables that are declared inside a function cannot be accessed outside it. They are isolated.

Let us see an example to illustrate this:

```
def counting():
    count = 0  # This is not accessible outside of the fu\
nction.


counting()

print(count)  # This is going to throw an error when exec\
uting, since count is only declared inside the function a\
nd is not acessible outside that
```

Similarly, we cannot change variables inside functions that have been declared outside functions and that are not passed as arguments:

```
count = 3331


def counting():
    count = 0  # This is a new variable


counting()

print(count)  # 3331
# This is declared outside the function and has not been \
changed
```

Changing data inside functions

We can change mutable data that is passed through a function as arguments. Mutable data represents data that can be modified even after it has been declared, for example lists are mutable data.

```
names = ["betim", "durim", "gezim"]


def capitalize_names(current_list):

    for i in range(len(current_list)):
        current_list[i] = current_list[i].capitalize()

    print("Inside the function:", current_list)

    return current_list


capitalize_names(names)  # Inside the function: ['Betim',\
 'Durim', 'Gezim']
```

```
16
17 print("Outside the function:", names)  # Outside the func\
18 tion: ['Betim', 'Durim', 'Gezim']
```

In case of immutable data, we can only modify the variable inside a function, but the actual value outside that function is going to remain unchanged. Immutable data are strings and numbers:

```
1  name = "Betim"
2
3
4  def say_hello(current_param):
5      current_param = current_param + " Gashi"
6      name = current_param  # name is a local variable
7      print("Value inside the function:", name)
8      return current_param
9
10
11 say_hello(name)  # Value inside the function: Betim Gashi
12
13 print("Value outside the function:", name)  # Value outsi\
14 de the function: Betim
```

If we really want to update immutable variables through a function, we can assign a returning value of a function to the immutable variable:

```
name = "Betim"


def say_hello(current_param):
    current_param = current_param + " Gashi"
    name = current_param  # name is a local variable
    print("Value inside the function", name)
    return current_param


# Here we are assigning the value of name to the current_\
param that is returned from the function
name = say_hello(name)  # Value inside the function Betim\
 Gashi

# Value outside the function: Betim Gashi
print("Value outside the function:", name)
```

Lambda functions

Lambda functions are anonymous functions that can be used to return an output. We can write lambda functions using the following syntax pattern:

```
lambda parameters: expression
```

The expression can only be written in a single line.

Let us start illustrating these anonymous functions using a few examples.

Let us start with a function that multiples each input with 10:

```
tenfold = lambda number : number * 10

print(tenfold(10))  # 100
```

Let us write another example in which we check whether the given argument is positive or not:

```
is_positive =  lambda a : f'{a} is positive' if a > 0 els\
e f'{a} is not positive'


print(is_positive(3))  # 3 is positive

print(is_positive(-1))  # -1 is not positive
```

Note that we cannot use the `if` clause without the `else` clause inside a lambda function.

At this point, you may wonder, why do we need to use lambda functions, since they seem to be almost the same as other functions.

We can see that illustrated in the following section.

Functions as arguments of functions

So far, we have seen ways of calling functions using numbers and strings. We can actually call functions with any type of Python object.

We can even provide as argument of a function an entire function, which can provide a level of abstraction that can be quite useful.

Let us see an example where we want to do a few conversions from one unit into another:

```
def convert_to_meters(feet):
    return feet * 0.3048


def convert_to_feet(meters):
    return meters / 0.3048


def convert_to_miles(kilometers):
    return kilometers / 1.609344


def convert_to_kilometers(miles):
    return miles * 1.609344
```

Now, we can make a general function and pass another function as an argument:

```
def conversion(operation, argument):
    return operation(argument)
```

We can now call `conversion()` like the following:

```
result = conversion(convert_to_miles, 10)

print(result)  # 6.2137119223733395
```

As you can see, we have written `convert_to_miles` as a parameter of the function `conversion()`. We can use other already defined functions like that:

```
result = conversion(convert_to_feet, 310)

print(result)  # 1017.0603674540682
```

We can now make use of lambdas and make this type of abstraction much simpler.

Instead of writing all those four functions, we can simply write a concise lambda function and use it as a parameter when calling the `conversion()` function:

```
def conversion(operation, argument):
    return operation(argument)


result = conversion(lambda kilometers: kilometers / 1.609\
344, 10)

print(result)  # 6.2137119223733395
```

This is of course simpler.

Let us use a few other examples with built-in functions.

map()

map() is a built-in function that creates a new object by getting results by calling a function on each element of an existing list:

```
map(function_name, my_list)
```

Let us see an example of writing a lambda function as the function of a map.

Let us triple each number in a list using list comprehension:

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

triple_list = [x * 3 for x in my_list]

print(triple_list)  # [3, 6, 9, 12]
```

We can implement that using a `map()` function and a lambda function:

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

triple_list = map(lambda x: x * 3, my_list)

print(triple_list)  # [3, 6, 9, 12]
```

This creates a new list. The old list is not changed.

filter()

This is another built-in function that can be used to filter elements of a list that satisfy a condition.

Let us first filter out negative elements from a list using list comprehension:

```
my_list = [3, -1, 2, 0, 14]

non_negative_list = [x for x in my_list if x >= 0]

print(non_negative_list)  # [3, 2, 0, 14]
```

Now, let us filter elements using `filter()` and a lambda function. This function returns an object which we can convert into a list using `list()`:

```
my_list = [3, -1, 2, 0, 14]

non_negative_filter_object = filter(lambda x: x >= 0, my_\
list)

non_negative_list = list(non_negative_filter_object)

print(non_negative_list)  # [3, 2, 0, 14]
```

This should now give you the intuition needed on how you can call functions with other functions as arguments and why lambdas are useful and important.

Decorators

A decorator represents a function that accepts another function as an argument.

We can think of it as a dynamic way of changing the way a function, method, or class behaves without having to use subclasses.

Once a function is being passed as an argument to a decorator, it will be modified and then returned as a new function.

Let us start with a basic function that we want to decorate:

```
def reverse_list(input_list):
    return input_list[::-1]
```

In this example, we are simply returning a reversed list.

We can also write a function that accepts another function as an argument:

```
def reverse_list(input_list):
    return input_list[::-1]


def reverse_input_list(another_function, input_list):
    # we are delegating the execution to another_function\
()
    return another_function(input_list)


result = reverse_input_list(reverse_list, [1, 2, 3])

print(result)  # [3, 2, 1]
```

We can also nest a function into another function:

```
def reverse_input_list(input_list):
    # reverse_list() is now a local function that is not \
accessible from the outside
    def reverse_list(another_list):
        return another_list[::-1]

    result = reverse_list(input_list)
    return result               # Return the result of th\
e local function


result = reverse_input_list([1, 2, 3])
print(result)  # [3, 2, 1]
```

In this example, `reverse_list()` now is a local function and cannot be called outside the scope of the `reverse_input_list()` function.

Now we can write our first decorator:

```
def reverse_list_decorator(input_function):
    def function_wrapper():
        returned_result = input_function()
        reversed_list = returned_result[::-1]
        return reversed_list

    return function_wrapper
```

`reverse_list_decorator()` is a decorator function that takes as input another function. To call it, we need to write another function:

```
# Function that we want to decorate
def get_list():
    return [1, 2, 3, 4, 5]
```

Now we can call the decorator with our new function as an argument:

```
decorator = reverse_list_decorator(get_list)  # This retu\
rns a reference to the function

result_from_decorator = decorator()  # Here we call the a\
ctual function using parenthesis

# We can now print the result in the console
print(result_from_decorator)  # [5, 4, 3, 2, 1]
```

Here is the complete example:

```
def reverse_list_decorator(input_function):
    def function_wrapper():
        returned_result = input_function()
        reversed_list = returned_result[::-1]
        return reversed_list

    return function_wrapper

# Function that we want to decorate
def get_list():
    return [1, 2, 3, 4, 5]


# This returns a reference to the function
decorator = reverse_list_decorator(get_list)

# Here we call the actual function using the parenthesis
result_from_decorator = decorator()

# We can now print the result in the console
print(result_from_decorator)  # [5, 4, 3, 2, 1]
```

We can also call a decorator using annotations. To do that, we use the `@` sign before the name of the decorator that we want to call and put it right above the name of the function:

```
# Function that we want to decorate
@reverse_list_decorator  # The annotation of the decorato\
r function
def get_list():
    return [1, 2, 3, 4, 5]
```

Now, we can simply call the function`get_list()` and the decorator is going to be applied in it:

```
result_from_decorator = get_list()

print(result_from_decorator)  # [5, 4, 3, 2, 1]
```

Stacking decorators

We can also use more than one decorator for a single function. Their order of execution starts from top to bottom, meaning that the decorator that has been defined first is applied first, then the second one, and so on.

Let us do a simple experiment and apply the same decorator that we defined in the previous section twice.

Let us first understand what does that mean.

So we first call the decorator to reverse a list:

`[1, 2, 3, 4, 5]` to `[5, 4, 3, 2, 1]`

Then we apply it again, but now with the returned result from the previous calling of the decorator:

`[5, 4, 3, 2, 1]` ⇒ `[1, 2, 3, 4, 5]`

In other words, reversing a list and then reversing that reversed list again is going to return the original ordering of the list.

Let us see this with decorators:

```
@reverse_list_decorator
@reverse_list_decorator
def get_list():
    return [1, 2, 3, 4, 5]


result = get_list()

print(result)  # [1, 2, 3, 4, 5]
```

Let us explain this with another example.

Let us implement another decorator that only returns numbers that are larger than 1. We then want to reverse that return list with our existing decorator.

```
def positive_numbers_decorator(input_list):
    def function_wrapper():
        # Get only numbers larger than 0
        numbers = [number for number in input_list() if n\
umber > 0]
        return numbers

    return function_wrapper
```

Now we can call this decorator and the other decorator that we have implemented:

```
@positive_numbers_decorator
@reverse_list_decorator
def get_list():
    return [1, -2, 3, -4, 5, -6, 7, -8, 9]


result = get_list()
print(result)  # [9, 7, 5, 3, 1]
```

Here is the complete example:

```
def reverse_list_decorator(input_function):
    def function_wrapper():
        returned_result = input_function()
        reversed_list = returned_result[::-1]  # Reverse \
the list
        return reversed_list

    return function_wrapper


# First decoorator
def positive_numbers_decorator(input_list):
    def function_wrapper():
        # Get only numbers larger than 0
        numbers = [number for number in input_list() if n\
umber > 0]
        return numbers

    return function_wrapper

# Function that we want to decorate


@positive_numbers_decorator
@reverse_list_decorator
def get_list():
    return [1, -2, 3, -4, 5, -6, 7, -8, 9]


result = get_list()
print(result)  # [9, 7, 5, 3, 1]
```

Passing arguments in decorator functions

We can also pass arguments to decorator functions:

```
def add_numbers_decorator(input_function):
    def function_wrapper(a, b):
        result = 'The sum of {} and {} is {}'.format(
            a, b, input_function(a, b))  # calling the in\
put function with arguments
        return result
    return function_wrapper


@add_numbers_decorator
def add_numbers(a, b):
    return a + b


print(add_numbers(1, 2))  # The sum of 1 and 2 is 3
```

Built-in decorators

Python comes with multiple built-in decorators, such as `@classmethod`, `@staticmethod`, `@property`, etc. These are covered in the next chapter.

Wrap up

In conclusion, Python is an excellent language for writing functions because they are easy to write.

Lambda functions are a great way to make small, concise functions in Python. They're perfect for when you don't need a full-blown function, or when you just want to test out a snippet of code.

Python decorators are a great way to improve code readability and maintainability. They allow you to modularize your code and make it more organized. Additionally, they can be used to perform

various tasks such as logging, exception handling, and testing. So if you're looking for a way to clean up your Python code, consider using decorators.

Object Oriented Programming

If you go to buy a cookie at a local store, you are going to get a piece of the cookie that has been produced in many other copies.

There has been a cookie cutter at a factory that has been used to produce a large number of cookies that are then distributed all throughout different stores where it is then ready to be served to the end customers.

We can think of that cookie cutter as a blueprint that has been designed once and is used afterwards. This blueprint is also used in computer programming.

A blueprint that is used to create countless other copies is called a **class**. We can think of a class like a class called **Cookie**, **Factory**, **Building**, **Book**, **Pencil**, etc. We can use the class of **Pencil** as a blueprint to create as many instances as we want of it that we call objects.

In other words, blueprints are classes that are used as cookie cutters, whereas the cookies that are served at different stores are objects.

Object Oriented Programming represents a way of organizing a program in classes and objects. Classes are used to create objects. Objects interact with each other.

We do not use the exact same blueprint for every object that is out there. There is a blueprint for producing books, another one for producing pencils, and so on. We need to categorize them based on attributes and their functionalities.

An object that is created from the Pencil class can have a color type, a manufacturer, a specific thickness, etc. These are the **attributes**.

A *pencil* object can also write which represents its functionality, or its **method**.

We use classes and objects in different programming languages, including Python.

Let us see how does a very basic Bicycle class look like in Python:

```
1 class Bicycle:
2     pass
```

We have used the keyword class to indicate that we are about to start writing a class and then we type the name of the class.

We have added the part pass because we want to let Python interpreter to not yell at us by throwing errors for not continuing to write the remaining part of the code that belongs to this class.

Now, if we want to create new objects from this class Bicycle, we can simply write the name of the object (which can be any variable name that you want) and initiailize it with the constructor method Bicycle() that is used to create new objects:

```
1 favorite_bike = Bicycle()
```

In this case, favorite_bike is an object that is created from the class Bicycle. It gets all the functionalities and attributes of the class Bicycle.

We can enrich our Bicycle class and include additional attributes so that we can have custom bikes, tailored to our needs.

To do that, we can define a constructor method called *init* as follows:

```
class Bicycle:
    def __init__(self, manufacturer, color, is_mountain_b\
ike):
        self.manufacturer = manufacturer
        self.color = color
        self.is_mountain_bike = is_mountain_bike
```

Note the usage of underscores before and after the name `init` of the method. They represent indicators to the Python interpreter to treat that method as a special method.

This is a method that does not return anything. It is a good practice to define it as the first method of the class, so that other developers can also see it being at a specific line.

Now, if we want to create new objects using this blueprint of bicycles, we can simply write:

```
bike = Bicycle("Connondale", "grey", True)
```

We have provided our custom parameters for this bike and are passing them to the constructor method so that we get a new bike with those specific attributes in return. As you can probably tell, we are creating a grey, mountain bike of the brand `Connondale`.

We can also create objects from classes by using optional arguments as follows:

```
class Bicycle:
    # All the following attributes are optional
    def __init__(self, manufacturer=None, color='grey', i\
s_mountain_bike=False):
        self.manufacturer = manufacturer
        self.color = color
        self.is_mountain_bike = is_mountain_bike
```

Now we have just created this object with these attributes, which are not currently accessible outside the scope of the class. This means that we have created this new object from the Bicycle class, but its corresponding attributes are not accessible. To access them, we can implement methods that help us access them.

To do that, we are going to define `getters` and `setters`, which represent methods that are used to get and set values of attributes of objects. We are going to use an annotation called `@property` to hep us with that.

Let's see it with code:

```
class Bicycle:
    def __init__(self, manufacturer, color, is_mountain_b\
ike):
        self._manufacturer = manufacturer
        self._color = color
        self._is_mountain_bike = is_mountain_bike

    @property
    def manufacturer(self):
        return self._manufacturer

    @manufacturer.setter
    def manufacturer(self, manufacturer):
        self._manufacturer = manufacturer


bike = Bicycle("Connondale", "Grey", True)

print(bike.manufacturer)  # Connondale
```

We can write getters and setters for all the attributes of the class:

```
class Bicycle:
    def __init__(self, manufacturer, color, is_mountain_b\
ike):
        self._manufacturer = manufacturer
        self._color = color
        self._is_mountain_bike = is_mountain_bike

    @property
    def manufacturer(self):
        return self._manufacturer

    @manufacturer.setter
    def manufacturer(self, manufacturer):
        self._manufacturer = manufacturer

    @property
    def color(self):
        return self._color

    @color.setter
    def color(self, color):
        self._color = color

    @property
    def is_mountain_bike(self):
        return self._is_mountain_bike

    @is_mountain_bike.setter
    def is_mountain_bike(self, is_mountain_bike):
        self.is_mountain_bike = is_mountain_bike

bike = Bicycle("Connondale", "Grey", True)
```

Now that we have defined them, we can call these getter methods as attributes:

```
print(bike.manufacturer)  # Connondale
print(bike.color)  # Grey
print(bike.is_mountain_bike)  # True
```

We can also modify the value that we initially used for any attribute by simply typing the name of object and the attribute that we want to change the content:

```
bike.is_mountain_bike = False
bike.color = "Blue"
bike.manufacturer = "Trek"
```

Our classes can also have other methods as well and not just getters and setters.

Let us define a method inside the class Bicycle that we can then call from any object that we have created from that class:

```
class Bicycle:
    def __init__(self, manufacturer, color, is_mountain_b\
ike):
        self._manufacturer = manufacturer
        self._color = color
        self._is_mountain_bike = is_mountain_bike

    def get_description(self):
        desc = "This is a " + self._color + " bike of the\
 brand " + self._manufacturer
        return desc
```

We have created a very simple method in which we are preparing a string as a result from the attributes of the object that we are creating. We can then call this method like any other method.

Let us see this in action:

```
bike = Bicycle("Connondale", "Grey", True)

print(bike.get_description())  # This is a Grey bike of t\
he brand Connondale
```

Methods

Methods are similar to functions that we have already covered before.

In a nutshell, we group a few statements in a code block called method where we perform some operations that we expect to be done more than once and do not want to write them again and again. In the end, we may not return any result at all.

There are three types of methods in Python:

- instance methods
- class methods
- static methods

Let us briefly talk about the overall structure of methods and then dive a little more into details for each method type.

Parameters

Parameters of a method make it possible for us to pass on dynamic values that can then be taken into consideration when executing the statements that are inside the method.

The `return` statement represents the statement that is going to be the last one to be executed in that method since it is an indicator for the Python interpreter to stop the execution of any other line and return a value.

self argument

The first argument of a method in Python is `self` which is also one of the differences between a method and a function. It represents a reference to the object to which it belongs to. If it is not specified as the first argument of the method when being declared, the first argument is then treated as a reference to the object.

We only write it when we declare the method, but we do not need to include it when we invoke that particular method using an object as a caller. It is not required to be named as `self`, but it is a convention that is widely practiced from developers writing Python code all around the world.

Let us define an instance method inside the class `Bicycle` that we can then call from any object that we have created from that class:

```
class Bicycle:
    def __init__(self, manufacturer, color, is_mountain_b\
ike):
        self._manufacturer = manufacturer
        self._color = color
        self._is_mountain_bike = is_mountain_bike

    def get_description(self):
        desc = "This is a " + self._color + " bike of the\
 brand " + self._manufacturer
        return desc
```

We have created a very simple method in which we are preparing a string as a result from the attributes of the object that we are creating. We can then call this method like any other method:

```
bike = Bicycle("Connondale", "Grey", True)

print(bike.get_description())  # This is a Grey bike of t\
he brand Connondale
# We are not passing any argument when calling the method\
 get_description() since we do not need to include self a\
t all
```

Class methods

We have covered instance methods so far that are methods that we can call with objects.

Class methods are methods that can be called using class names and can be accessed without needing to create any new object at all.

Since it is a specific type of method, we need to tell Python interpreter that it is actually different. We do that by making a change in the syntax.

We use the annotation `@classmethod` above a class method and `cls` similar to the usage of `self` for instance methods. `cls` is just a conventional way of referring to the class that is calling the method, and not that you have to use this name.

Let us declare our first class method:

```
class Article:
    blog = 'https://www.python.org/'

    # the init method is called when an instance of the c\
lass is created
    def __init__(self, title, content):
        self.title = title
        self.content = content

```

```
10      @classmethod
11      def get_blog(cls):
12          return cls.blog
```

Now let us call this class method that we have just declared:

```
1  print(Article.get_blog())  # https://www.python.org/
```

Note that we did not have to write any argument when calling `get_blog()` method. On the other hand, when we declare methods and instance methods, we should always include at least one argument.

Static methods

These are methods that do not have direct relations to class variables or instance variables. They can be seen as utility function that are supposed to help us do something with arguments that are passed when calling them.

We can call them by using both the class name and an object that is created by that class where this method is declared. This means that they do not need to have their first argument related to the object or class calling them which was the case with using parameters `self` for instance methods and `cls` for class methods.

There is no limit to the number of arguments that we can use to call them.

To create it, we need to use the `@staticmethod` annotation.

Let us create a static method:

```
class Article:
    blog = 'https://www.python.org/'

    # the init method is called when an instance of the c\
lass is created
    def __init__(self, title, content):
        self.title = title
        self.content = content

    @classmethod
    def get_blog(cls):
        return cls.blog

    @staticmethod
    def print_creation_date(date):
        print(f'The blog was created on {date}')


article = Article('First Article', 'This is the first art\
icle')

# Calling the static method using the object
article.print_creation_date('2022-07-18')  # The blog was\
 created on 2022-07-18

# Calling the static method using the class name
Article.print_creation_date('2022-07-21')  # The blog was\
 created on 2022-07-21
```

Static methods cannot modify class or instance attributes. They are meant to be like utility functions.

If we try to change a class, we are going to get errors:

```
class Article:
    blog = 'https://www.python.org/'

    # the init method is called when an instance of the c\
lass is created
    def __init__(self, title, content):
        self.title = title
        self.content = content

    @classmethod
    def get_blog(cls):
        return cls.blog

    @staticmethod
    def set_title(self, date):
        self.title = 'A random title'
```

If we try to call this static method now, we are going to get an error:

```
# Calling the static method using the class name
Article.set_title('2022-07-21')
```

```
TypeError: set_title() missing 1 required positional argu\
ment: 'date'
```

This is because static methods do not have any reference to `self` as they are not directly related to objects or classes and so they cannot modify attributes.

Access modifier

When creating classes, we can restrict access to certain attributes and methods so that they are not accessible that easily.

We have `public`and `private` access modifiers.

Let us see both of them.

Public attributes

Public attributes are the ones that are accessible from both inside and outside the class.

By default, all attributes and methods are public in Python. If we want them to be private, we need to specify that.

Let us see an example of public attributes:

```
class Bicycle:
    def __init__(self, manufacturer, color, is_mountain_b\
ike):
        self.manufacturer = manufacturer
        self.color = color
        self.is_mountain_bike = is_mountain_bike

    def get_manufacturer(self):
        return self.manufacturer
```

In the previous code block, both `color` and `get_manufacturer()` are accessible outside the class since they are `public` and can be accessed both inside and outside the class:

```
bike = Bicycle("Connondale", "Grey", True)

print(bike.color)  # Grey
print(bike.get_manufacturer())  # Connondale
```

Private attributes

Private attributes can be accessed directly only from inside the class.

We can make properties attributes by using the double underscore, as can be seen in the following example:

```
class Bicycle:
    def __init__(self, manufacturer, color, is_mountain_b\
ike, old):
        self.manufacturer = manufacturer
        self.color = color
        self.is_mountain_bike = is_mountain_bike
        self.__old = old  # This is a private property
```

Now if we try to access `__old`, we are going to get an error:

```
bike = Bicycle("Connondale", "Grey", True, False)

print(bike.__old)  # AttributeError: 'Bicycle' object has\
 no attribute '__old'
```

Let us now see an example where we are declaring private methods using the double underscore in front of the name of the method that we want to make as private:

```
class Bicycle:
    def __init__(self, manufacturer, color, is_mountain_b\
ike, old):
        self.manufacturer = manufacturer
        self.color = color
        self.is_mountain_bike = is_mountain_bike
        self.__old = old  # This is a private property

    def __get_old(self):  # This is a private method
        return self.__old
```

Now, if we want to call this private method from outside the class, an error is going to be thrown:

```
bike = Bicycle("Connondale", "Grey", True, False)

print(bike.__get_old())  # AttributeError: 'Bicycle' obje\
ct has no attribute '__get_old'
```

It is not a common practice to have private variables in Python. However, developers may see it necessary to restrict access so that specific variables are not carelessly accessed and modified.

Hiding information

When you go out there and use a coffee machine, it is not expected from you to know all the engineering details that are behind that machine. This is also with your car. When you sit in your driving seat, you do not analyze and understand about all the details of every part of the car. You have some basic idea about them, but other than that, you are driving more freely.

This can be thought as a restriction of the access from people outside, so that they do not have to worry about exact details that are going on inside.

We can do that in Python as well.

We have seen so far the foundational blocks of object oriented programming, such as classes and objects.

Classes are blueprints that are used to create instances called objects. We can use objects of different classes to interact with each other and build a robust program.

When we work on our own programs, we may need to not let everyone know about all the details that our classes have. We can hence limit access to them, so that certain attributes are less likely to be accessed unintentionally and be modified wrongfully.

To help us with that, we hide parts of a class and simply provide an interface that has less details about the inner workings of our class.

We can hide data in two ways:

1. Encapsulation
2. Abstraction

Let us begin with Encapsulation.

Encapsulation

Encapsulation is not something special and unique just for Python. Other programming languages use it as well.

In a nutshell, we can define it as binding data and methods in a class. This class is then used to create objects.

We encapsulate classes by using `private` access modifiers that can then restrict direct access to such attributes. This can restrict the control.

We are then supposed to write public methods that can provide access to the outside world.

These methods are called `getters` and `setters`.

A **getter** method is a method that is used to get the value of an attribute.

A **setter** is a method that is used to set the value of an attribute.

Let us define first define a `getter` and a `setter` method that can be used to get values:

```
class Smartphone:
    def __init__(self, type=None):  # defining initialize\
r for case of no argument
        self.__type = type  # setting the type here in th\
e beginning when the object is created

    def set_type(self, value):
        self.__type = value

    def get_type(self):
        return (self.__type)
```

Now, let us use this class to set the type and also get the type:

```
smartphone = Smartphone('iPhone')  # we are setting the t\
ype using the constructor method

# getting the value of the type
print(smartphone.get_type())   # iPhone

# Changing the value of the type
smartphone.set_type('Samsung')

# getting the new value of the type
print(smartphone.get_type())    # Samsung
```

What we have done so far is that we have set and also read the value of a private attribute of an object created from the `Smartphone` class.

We can also define `getters` and `setters`, using the `@property` annotation to hep us with that.

Let's see it with code:

```
class Bicycle:
    def __init__(self, manufacturer, color):
        self._manufacturer = manufacturer
        self._color = color

    @property
    def manufacturer(self):
        return self._manufacturer

    @manufacturer.setter
    def manufacturer(self, manufacturer):
        self._manufacturer = manufacturer

    @property
    def color(self):
        return self._color

    @color.setter
    def color(self, color):
        self._color = color


bike = Bicycle("Connondale", "Grey")
```

Now that we have defined them, we can call these getter methods as attributes:

```
print(bike.manufacturer)  # Connondale
print(bike.color)  # Grey
```

We can also modify the value that we initially used for any attribute by simply typing the name of the object and the attribute that we want to modify:

```
bike.is_mountain_bike = False
bike.color = "Blue"
```

Our classes can also have other methods as well, and not just getters and setters.

Let us define a method inside the class Bicycle that we can then call from any object that we have created from that class:

```
class Bicycle:
    def __init__(self, manufacturer, color, is_mountain_b\
ike):
        self._manufacturer = manufacturer
        self._color = color
        self._is_mountain_bike = is_mountain_bike

    def get_description(self):
        desc = "This is a " + self._color + " bike of the\
 brand " + self._manufacturer
        return desc
```

We have created a very simple method in which we are preparing a string as a result from the attributes of the object that we are creating. We can then call this method like any other method.

Let us see this in action:

```
bike = Bicycle("Connondale", "Grey", True)

print(bike.get_description())  # This is a Grey bike of t\
he brand Connondale
```

But why do we need encapsulation?

This looks quite promising and fancy, but you may not get it quite yet. You may get the impression that you need additional reasons on why you need this type of hiding.

To drive this home, let us take another class, where we have a private attribute called `salary`. Let us say that we don't care about encapsulation and we are only trying to build a class fast and use it in our project for our accountant client.

Let us say that we have the following class:

```
class Employee:
    def __init__(self, name=None, email=None, salary=None\
):
        self.name = name
        self.email = email
        self.salary = salary
```

Now, let us create a new `employee` object and initialize its attributes accordingly:

```
# We are creating an object
betim = Employee('Betim', 'betim@company.com', 5000)

print(betim.salary)  # 5000
```

Since `salary` is not being protected in any way, we can set a new salary for this new object without any problem:

```
betim.salary = 25000

print(betim.salary)  # 25000
```

As we can see, this person got five times the salary of what he was getting previously without going through any type of evaluation or interviewing at all. In fact, it happened in a matter of seconds. That's probably going to hit the budget of the company heavily.

We obviously do not want to do that. We want to restrict access to the `salary` attribute so that it is not called from other places. We can do that by using the double underscore before the attribute name as can be shown below:

```
class Employee:
    def __init__(self, name=None, email=None, salary=None\
):
        self.__name = name
        self.__email = email
        self.__salary = salary
```

Let us create a new object:

```
# We are creating an object
betim = Employee('Betim', 'betim@company.com', 1000)
```

Now, if we try to access its attributes, we cannot do so, since they are private attributes:

```
print(betim.salary)  # 1000
```

Trying to access any of the attributes is going to be followed with an error:

```
AttributeError: 'Employee' object has no attribute 'salar\
y'
```

We can simply implement a method that returns the attributes but we are not providing any way for someone to increase their salary sneakily:

```
class Employee:
    def __init__(self, name=None, email=None, salary=None\
):
        self.__name = name
        self.__email = email
        self.__salary = salary

    def get_info(self):
        return self.__name, self.__email, self.__salary
```

Now, we can access the information of objects created by this class:

```
# We are creating an object
betim = Employee('Betim', 'betim@company.com', '5000')

print(betim.get_info())  # ('Betim', 'betim@company.com',\
 '5000')
```

In summary, encapsulation helps us protect properties of objects and provides access them in a controlled fashion.

Inheritance

In real life, we can share many characteristics with other human beings.

We all need to eat food, drink water, work, sleep, move, and so on. These and many other behaviors and characteristics are shared among billions of people all around the world. They are not something unique that only our generation has. These traits have been for many generations that we have not had the chance to even see at all.

This is also something that is going to last for future generations to come.

We can also have certain shared characteristics between objects and classes that we implement ourselves in computer programming using **inheritance**. This includes both attributes and methods.

Let us imagine that we have a class called `Book`. It should contain a title, an author, a number of pages, a category, an ISBN, etc. We are going to keep our class simple and use only two attributes:

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def get_short_book_paragraph(self):
        short_paragraph = "This is a short paragraph of t\
he book."
        return short_paragraph
```

Now, we can create an object from this class and access it:

```
first_book = Book("Atomic Habits", "James Clear")

print(first_book.title)  # Atomic Habits
print(first_book.author)  # James Clear
print(first_book.get_short_book_paragraph())  # This is a\
 short paragraph of the book.
```

Let us now create a subclass of the class `Book` that inherits attributes and methods from the class `Book`, but also has another additional method called `get_book_description()`:

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def get_short_book_paragraph(self):
        short_paragraph = "This is a short paragraph of t\
he book."
        return short_paragraph


class BookDetails(Book):
    def __init__(self, title, author):
        Book.__init__(self, title, author)
        # Here we are call the constructor of the parent \
class Book

    def get_book_details(self):
        description = "Title: " + self.title + ". "
        description += "Author: " + self.author
        return description
```

Note the syntax in which we tell Python that `BookDetails` is a subclass of the class `Book`:

```
class BookDetails(Book):
```

If we try to access this new method from objects of the class `Book`, we are going to get an error:

```
first_book = Book("Atomic Habits", "James Clear")

print(first_book.get_book_details())
# AttributeError: 'Book' object has no attribute 'get_boo\
k_details'
```

This happens because this method `get_book_details()` can be accessed only from objects of `BookDetails`:

```
first_book_details = BookDetails("Atomic Habits", "James \
Clear")

print(first_book_details.get_book_details())
# Title: Atomic Habits. Author: James Clear
```

We can however access any method that is defined in the parent class, which in our case is the `Book` class:

```
first_book_details = BookDetails("Atomic Habits", "James \
Clear")

print(first_book_details.get_short_book_paragraph())
# This is a short paragraph of the book.
```

In the previous classes, `Book` is considered as a parent class or as a superclass, whereas `BookDetails` is considered as a child class, or a subclass.

super() function

There is a special function called `super()` that can be used from a child class to refer to its parent class without writing the exact name of the parent class.

We use it with initializers, or when calling properties or methods of parent classes.

Let us see all three of them illustrated with examples.

Using super() with initializers

We can use `super()` inside the constructor method of the subclass and even call the constructor of the super class:

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


class Cat(Animal):
    def __init__(self, name, age):
        super().__init__(name, age)  # calling the parent\
 class constructor
        self.health = 100  # initializing a new attribute\
 that is not in the parent class
```

We can also replace `super()` with the name of the parent class, which is going to work in the same way again:

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


class Cat(Animal):
    def __init__(self, name, age):
        Animal.__init__(name, age)  # calling the parent \
class constructor
```

```
        self.health = 100  # initializing a new attribute\
 that is not in the parent class
```

Even changing the order of the lines inside the child's constructor will not cause any error at all.

Using super() with class properties of the parent class

We can use `super()` to access class properties of the parent class, which can be useful especially when both the parent and the child class use the same name for an attribute.

To see that in action, let us assume that we have a class attribute called call `name` which is present both in the parent and the child class. We want to access this variable from both the parent class and the child class.

To do that, we simply need to write `super()` and then the name of the variable:

```
class Producer:  # parent class
    name = 'Samsung'


class Seller(Producer):  # child class
    name = 'Amazon'

    def get_product_details(self):
        # Calling the variable from the parent class
        print("Producer:", super().name)

        # Calling the variable from the child class
        print("Seller:", self.name)
```

Now, if we call method `get_product_details()`, we are going to get the following printed in the console:

```
seller = Seller()

seller.get_product_details()

# Producer: Samsung
# Seller: Amazon
```

Using super() with methods of the parent class

We can similarly call methods in the parent class using `super()`.

```
class Producer:  # parent class
    name = 'Samsung'

    def get_details(self):
        return f'Producer name: {self.name}'


class Seller(Producer):  # child class
    name = 'Amazon'

    def get_details(self):
        # Calling the method from the parent class
        print(super().get_details())

        # Calling the variable from the child class
        print(f'Seller name: {self.name}')


seller = Seller()
seller.get_details()

# Producer name: Amazon
# Seller name: Amazon
```

This is all you need to know about `super()`.

Types of inheritance

We can have different types of inheritance based on the relationship of parent classes and child classes:

1. Single
2. Multi-level
3. Hierarchical
4. Multiple
5. Hybrid

1. Single inheritance

We can have a class that inherits only from another class:

```
class Animal:
    def __init__(self):
        self.health = 100

    def get_health(self):
        return self.health


class Cat(Animal):
    def __init__(self, name):
        super().__init__()
        self.health = 150
        self.name = name

    def move(self):
        print("Cat is moving")

cat = Cat("Cat")

# Calling the method from the parent class
```

```
print(cat.get_health())  # 150

# Calling the method from the child class
cat.move()  # Cat is moving
```

2. Multi-level inheritance

This is another type of inheritance where a class inherits from another class which inherits from another class: Class A inherits from Class B which inherits from Class C.

Let us implement this in Python:

```
class Creature:
    def __init__(self, alive):
        self.alive = alive

    def is_it_alive(self):
        return self.alive


class Animal(Creature):
    def __init__(self):
        super().__init__(True)
        self.health = 100

    def get_health(self):
        return self.health


class Cat(Animal):
    def __init__(self, name):
        super().__init__()
        self.name = name

    def move(self):
```

```
        print("Cat is moving")


cat = Cat("Cat")

# Calling the method from the parent of the parent class
print(cat.is_it_alive())

# Calling the method from the parent class
print(cat.get_health())  # 150

# Calling the method from the child class
cat.move()  # Cat is moving
```

3. Hierarchical inheritance

When we derive multiple child classes from the same parent class, then we have hierarchical inheritance. These child classes inherit from the parent class:

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

    def get_location(self):
        return self.x, self.y


class Continent(Location):
    pass


class Country(Location):
    pass

```

```
17
18 continent = Continent(0, 0)
19 print(continent.get_location())  # (0, 0)
20
21 country = Country(10, 30)
22 print(country.get_location())  # (10, 30)
```

4. Multiple inheritance

We can have another type of inheritance, namely multiple inheritance which can help us inherit from more than one class at the same time.

Let us assume that we have a class called `Date` and another one called `Time`.

We can then implement another class then inherits from both classes:

```
1  class Date:
2      date = '2022-07-23'  # Hardcoded date
3
4      def get_date(self):
5          return self.date
6
7
8  class Time:
9      time = '20:20:20'  # Hardcoded time
10
11     def get_time(self):
12         return self.time
13
14
15 class DateTime(Date, Time):  # Inheriting from both
16     def get_date_time(self):
17         return self.get_date() + ' ' + self.get_time()  #\
18  getting methods from its parent classes
```

```
19
20
21 date_time = DateTime()
22 print(date_time.get_date_time())  # 2022-07-23 20:20:20
```

5. Hybrid inheritance

Hybrid inheritance is a combination of multiple and multi-level inheritance:

```
1  class Vehicle:
2      def print_vehicle(self):
3          print('Vehicle')
4
5
6  class Car(Vehicle):
7      def print_car(self):
8          print('Car')
9
10
11 class Ferrari(Car):
12     def print_ferrari(self):
13         print('Ferrari')
14
15
16 class Driver(Ferrari, Car):
17     def print_driver(self):
18         print('Driver')
```

Now, if we create an object from the class `Driver`, we can call all methods in all classes:

```
driver = Driver()

# Calling all methods from the subclass
driver.print_vehicle()  # Vehicle
driver.print_car()  # Car
driver.print_ferrari()  # Ferrari
driver.print_driver()  # Driver
```

Polymorphism

This is another important concept from Object Oriented Programming that refers to the possibility of an object behaving like different forms and calling different behaviors.

An example of a built-in function that uses polymorphism is the method `len()` which can be used for both strings and lists:

```
print(len('Python'))  # 6

print(len([2, 3, -43]))  # 3
```

We can take another example with a class called `House`. We can have different subclasses that inherit methods and attributes from that superclass, namely classes such as `Condo`, `Apartment`, `SingleFamilyHouse`, `MultiFamilyHouse`, etc.

Let us assume that we want to implement a method in the `House` class that is supposed to get the area.

Each type of living residence has a different size, so each one of the subclasses should have different implementations.

Now we can define methods into subclasses such as:

- `getAreaOfCondo()`
- `getAreaOfApartment()`

- `getAreaOfSingleFamilyHouse()`
- `getAreaOfMultiFamilyHouse()`

This would force us to remember the names of each subclass, which can be tedious and also prone to errors when we call them.

However, there is a simpler method that we can use that comes from polymorphism.

We can have polymorphism using both methods and inheritance.

Let us first see how we can implement polymorphism using methods.

Polymorphism using methods

Let us say that we have two classes, namely `Condo` and `Apartment`. Both of them have the method `get_area()` that returns a value.

Each of them is going to have a custom implementation.

Now the method that is going to be called depends on the class type of the object:

```
class Condo:
    def __init__(self, area):
        self.area = area

    def get_area(self):
        return self.area


class Apartment:
    def __init__(self, area):
        self.area = area

    def get_area(self):
        return self.area
```

Let us create two objects from these classes:

```
condo = Condo(100)

apartment = Apartment(200)
```

Now, we can put both of them in a list and call the same method for both objects:

```
places_to_live = [condo, apartment]

for place in places_to_live:
    print(place.get_area())  # same method for both objec\
ts
```

After we execute that, we are going to see the following in the console:

```
# 100
# 200
```

This is the polymorphism with methods.

Polymorphism with inheritance

We can not just call a method from a superclass. We can also use the same name but have a different implementation for it for each subclass.

Let us first define a superclass:

```
class House:
    def __init__(self, area):
        self.area = area

    def get_price(self):
        pass
```

Let us then implement subclasses `Condo` and `Apartment` of the superclass `House`:

```
class House:
    def __init__(self, area):
        self.area = area

    def get_price(self):
        pass


class Condo(House):
    def __init__(self, area):
        self.area = area

    def get_price(self):
        return self.area * 100


class Apartment(House):
    def __init__(self, area):
        self.area = area

    def get_price(self):
        return self.area * 300
```

As we can see, both subclasses have the method `get_price()` but different implementation.

We can now create new objects from subclasses and call this method which is going to *polymorph* based on the object that calls it:

```
condo = Condo(100)

apartment = Apartment(200)

places_to_live = [condo, apartment]

for place in places_to_live:
    print(place.get_price())
```

After we execute that, we are going to see the following in the console:

```
# 10000
# 60000
```

This is another example of polymorphism where we have specific implementation of a method that has the same name.

Importing

One of the main benefits of using a popular language such as Python is its large number of libraries that you can use and benefit from.

Many developers around the world are generous with their time and knowledge and publish a lot of really useful libraries that can save us plenty of time both in our professional work, but also on our side projects that we may do for fun.

Here are some of the modules with very useful methods that you can immediately start using in your projects:

- time: Time access and conversions
- csv: CSV File Reading and Writing
- math: Math functions
- email: Create, send, and process email
- urllib: Work with URLs

To import one or more modules, we only need to write `import` and then the name of the modules that we want to import.

Let us import our first module:

```
1  import os
```

Now, let us import multiple modules at once:

```
1  import os, numbers, math
```

Once we have imported a module, we can start using methods that are inside it.

```
import math

print(math.sqrt(81))  # 9.0
```

We can also use new names for our imported modules by specifying an alias for them `as alias` where `alias` is any variable name that you want:

```
import math as math_module_that_i_just_imported

result = math_module_that_i_just_imported.sqrt(4)

print(result)  # 2.0
```

Limiting parts that we want to import

There are times when we do not want to import a whole package with all its methods, since we want to avoid the overriding of methods or variables that are in the module with the ones that we want to implement ourselves.

We specify parts that we want to import by using the following form:

```
from module import function
```

Let us take an example of importing only the square root function from `math` module:

```
from math import sqrt

print(sqrt(100))  # 10.0
```

Importing everything

We can also import everything from a module, which can turn out to be a problem. Let us illustrate this with an example.

Let us assume that we want to import everything that is included in the `math` module. We can do that by using the asterisk

```
from math import *  # The asterisk is an indicator to inc\
lude everything when importing
```

Now, let us assume that we want to declare a variable called `sqrt`:

```
sqrt = 25
```

When we try to call the function `sqrt()` from the math module, we are going to get an error, since the interpreter is going to call the latest `sqrt` variable that we have just declared in the previous code block:

```
print(sqrt(100))
```

```
TypeError: 'float' object is not callable
```

Exceptions

When we are implementing Python scripts or doing any type of implementation, we are going to get many errors that are thrown even when the syntax is correct.

These types of errors that happen during execution are called exceptions.

We indeed do not have to surrender and not do anything regarding them. We can write handlers that are there to do something so that the execution of the program does not stop.

Common Exceptions

Here are some of the most common exceptions that happen in Python with definitions taken from the Python documentation[1]:

- **Exception** (This is a class that is as a superclass of most other exception types that happen)
- **NameError** - Raised when a local or global name is not found.
- **AttributeError** - Raised when an attribute reference or assignment fails.
- **SyntaxError** - Raised when the parser encounters a syntax error.
- **TypeError** - Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.
- **ZeroDivisionError** - Raised when the second argument of a division or modulo operation is zero.

[1] https://docs.python.org/3/library/exceptions.html

- **IOError** - Raised when an I/O operation (such as a print statement, the built-in open() function or a method of a file object) fails for an I/O-related reason, e.g., "file not found" or "disk full".
- **ImportError** - Raised when an import statement fails to find the module definition or when a **from** ... **import** fails to find a name that is to be imported.
- **IndexError** - Raised when a sequence subscript is out of range.
- **KeyError** - Raised when a mapping (dictionary) key is not found in the set of existing keys.
- **ValueError** - Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as IndexError.

There are many other error types, but you may not need to see about them now. It is also very unlikely that you are going to see all types of errors all the time.

You can see more types of exception in the Python documentation[2].

Handling exceptions

Let us start with a very simple example and write a program that throws an error on purpose so that we can then fix it.

We are going to do a division by zero, which is something that you have probably seen at school:

```
1 print(5 / 0)
```

If we try to execute that, we are going to be greeted with the following error in the console:

[2] https://docs.python.org/3/library/exceptions.html

```
ZeroDivisionError: division by zero
```

If we were to have such occurrences inside a Python program of any kind, we should catch and wrap this error inside a `try/except` block.

We need to write inside the `try` block the part of the code that we expect is going to throw errors. We then catch those types of errors inside the `except` block by also specifying the type of error that we except to happen.

Let us see the first example.

Let us see how we can deal with that error so that we also get informed that such error happened:

```
try:
    5 / 0
except ZeroDivisionError:
    print('You cannot divide by 0 mate!')
```

As you can see, we are printing a message in the console once we have reached the part where a division by `0` is happening.

We can also omit the part `ZeroDivisionError` completely:

```
try:
    5 / 0
except:
    print('You cannot divide by 0 mate!')
```

However, this is not recommended, since we are catching all types of errors in a single `except` block and we are not sure what type of errors are being caught, which can be quite useful for us.

Let us continue with another type of error.

Let us try to use a variable that is not defined at all:

```
name = 'User'

try:
    person = name + surname  # surname is not declared
except NameError:
    print('A variable is not defined')
```

In the previous example, we have used variable `surname` before declaring it, therefore a `NameError`is going to be thrown.

Let us continue with another type of example that can be common.

When we use lists, it can be a common mistake to use an index that is out of range, meaning that the used index is larger or smaller than the range of indexes of the elements in that list.

Let us illustrate this with an example, where an `IndexError` is going to be thrown:

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

try:
        print(my_list[5])
    # This list only has 4 elements, so its indexes range\
 from 0 to 3
except IndexError:
    print('You have used an index that is out of range')
```

We can also use a single `try` block with multiple `except` errors:

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

try:
    print(my_list[5])
    # This list only has 4 elements, so its indexes range\
 from 0 to 3
except NameError:
    print('You have used an invalid value')
except ZeroDivisionError:
    print('You cannot divide by zero')
except IndexError:
    print('You have used an index that is out of range')
```

In the previous example, we try to initially catch whether there is any variable that is used but not declared. If this error happens, then this `except` block is going to be taking over the execution flow. This execution flow is going to stop there.

Then, we try to check whether we are dividing by zero. If this error is thrown, then this `except` block is going to take over the execution and everything that is inside it is going to be executed. Similarly, we continue with the rest of the errors declared.

We can also put more than one error inside parenthesis to catch multiple exceptions, but this is not going to be helpful for us, since we do not know what specific error has been thrown. In other words, the following method does work, but it is not recommended:

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

try:
    print(my_list[5])
    # This list only has 4 elements, so its indexes range\
 from 0 to 3
except (NameError, ZeroDivisionError, IndexError):
    print('A NameError, ZeroDivisionError, or IndexError \
occurred')
```

finally

After the `try` and `except` are passed, there is another block that we can declare and execute. This block starts with the `finally` keyword and it is executed no matter whether we have an error is being thrown or not:

```
my_list = ['a', 'b']

try:
    print(my_list[0])
except IndexError:
    print('An IndexError occurred')
finally:
    print('The program is ending. This is going to be exe\
cuted.')
```

If we execute the previous block of code, we are going to see the following in the console:

```
a
The program is ending. This is going to be executed.
```

We usually write code that we want to be as a cleanup inside the `finally` block. This includes things like closing a file, or stopping a connection with a database, exiting the program entirely, etc.

try, else, except

We can write statements inside `try` and `except`, but we can also use an `else` block where we can write code that we want to be executed if there are no errors being thrown:

```
my_list = ['a', 'b']

try:
    print(my_list[0])
except IndexError:
    print('An IndexError occurred')
else:
    print('No error occurred. Congratulations!')
```

If we execute the code above, we are going to get the following printed in the console:

```
a
No error occurred. Congratulations!
```

Wrap up

This should be sufficient for you to understand exceptions and ways that you can use to handle them so that there is no sudden interruptions that cause your program to fail unexpectedly.

Afterword

This book represents my attempt to make it quick and easy for you to learn the essentials of Python. There are many other things that Python includes and that are not covered in this book, but we will leave it here.

I hope this serves as a useful reference for you.

Now that you have had the chance to learn how to write Python, go out there and make a positive impact with your lines of code.