About this eBook

While ePUB is an open standard widely employed for the publication of electronic books, support and features may vary from one device to another. Every effort has been made to ensure that this book will display faithfully on all devices, but it may be necessary to adjust the settings of your particular device for optimum readability.

There are many examples of code employed throughout this book. Due to the flowing nature of text in the ePUB format, this code, which is written line by line, may not always display correctly. For example, comments that begin with `//` may overflow to the next line. If you are trying to recreate the code that you see in the eBook and discover that it is not working as expected, ensure that you are following the correct formatting procedures as outlined in this eBook and in the PEP 8 Style Guide for Python Code.

Python Essentials 2

by

The OpenEDG Python Institute


OPENEDG

Open Education and Development Group
1013 Centre Road, Suite 405
Wilmington, DE
19805, United States

First published in the USA in 2023 by the Open Education and Development Group

ISBN: 979-8-9877622-2-6

Image credits

Portrait of Guido Van Rossum at the Dropbox headquarters in 2014
CC BY-SA 4.0, Photograph by Daniel Stroud.

Cover Design

Konrad Papka

Trademarks & Disclaimer

All the code examples in the book have been tested on Python 3.4, 3.6, 3.7, 3.8, and 3.9, and should work with any subsequent versions of Python 3.x.

Bulk purchase and custom book design

This book may be purchased in bulk in either ePUB or PDF format. Additionally, it may be possible to customize the layout of the book to suit your needs. To discuss these options, email services@openedg.org

TABLE OF CONTENTS

WELCOME TO PYTHON ESSENTIALS 2

LEARN PYTHON – THE LANGUAGE OF TODAY AND TOMORROW

The course picks up where Python Essentials 1 leaves off. Its main goal is to teach you the skills related to the more advanced aspects of Python programming, as well as with general coding techniques and object-oriented programming (OOP).

The course is recommended for aspiring developers who are interested in pursuing careers connected with Software Development, Security, Networking, and the Internet of Things (IoT).

INTRODUCTION

Python is one of the fastest growing programming languages in the world, and is used in almost every sector and industry, from gaming, to medicine, to nuclear physics. It is essential for any would-be programmer to have at least a foundational knowledge of Python.

Luckily, Python is also one of the easiest programming languages to learn. With its focus on real-world words and syntax, a beginner learner of Python can start writing simple programs within minutes

Goals of this book

This book is designed to teach you the basics of Python programming, even if you have zero programming experience

Additionally, it prepares you to take the *PCAP – Certified Associate in Python Programming* exam, which can be taken through the OpenEDG testing platform TestNow™.

At the end of this book, you will find the complete syllabus for the PCAP Python Certified Associate-Level Python Programmer exam

Learning Tools

Edube

The material found in this book may also be accessed online at www.edube.org. Here it is possible to take other courses such as JavaScript Essentials, or C/C++ Essentials, and progress to the intermediate and advances Python courses. Furthermore, through the Edube platform, you can purchase exam vouchers and schedule an exam.

Sandbox

The Edube educational platform offers an interactive programming sandbox, where you can try out the code examples shown in this book. The Sandbox becomes available as soon as you create an account on Edube.

Answers

Throughout this book you will find quizzes and exercises. You can find the answers, hints, and sample solutions and the back of the book in the Appendices.

About the course

This course has been designed and developed by the OpenEDG Python Institute in partnership with the **Cisco Networking Academy**.

The course has been created **for anyone and everyone** who wants to learn Python and modern programming techniques. It will particularly appeal to:

- **aspiring programmers** and learners interested in learning programming for fun and for job-related tasks;
- learners looking to gain fundamental skills and knowledge for role as a software developer, data analyst, or tester;
- industry professionals wishing to explore technologies that are connected with Python, or that utilize it as a foundation;
- team leaders, product managers, and project managers who want to understand the terminology and processes in the software development cycle to more effectively manage and communicate with production and development teams.
- During the course you will have access to **hands-on practice materials, labs, quizzes, assessments, and tests** to learn how to utilize the skills and knowledge gained from studying the resources and performing coding tasks, and interact with some real-life programming challenges and situations.

Syllabus

In this course you will learn:

- how to adopt general coding techniques and best practices in your projects;
- how to process strings;
- how to use object-oriented programming in Python;
- how to import and use Python modules, including the math, random, platform, os, time, datetime, and calendar modules;
- how to create and use your own Python modules and packages;
- how to use the exception mechanism in Python;
- how to use generators, iterators, and closures in Python;
- how to process files.

The course is divided into four modules:

Module 1
Modules, Packages and PIP

Module 2
Strings, string and list methods, and exceptions

Module 3
Object-Oriented Programming

Module 4
Miscellaneous (generators, iterators, closures, file streams, processing text and binary files, the os, time, datetime, and calendar module)

PART 1: MODULES, PACKAGES, AND PIP

ONE – INTRODUCTION TO MODULES IN PYTHON

Computer code has a tendency to grow. We can say that code that doesn't grow is probably completely unusable or abandoned. A real, wanted, and widely used code develops continuously, as both users' demands and users' expectations develop in their own rhythms. A code which is not able to respond to users' needs will be forgotten quickly, and instantly replaced with a new, better, and more flexible code. Be prepared for this, and never think that any of your programs is eventually completed. The completion is a transition state and usually passes quickly, after the first bug report. Python itself is a good example how the rule acts.

Growing code is in fact a growing problem. A larger code always means tougher maintenance. Searching for bugs is always easier where the code is smaller, just as finding a mechanical breakage is simpler when the machinery is simpler and smaller. Moreover, when the code being created is expected to be really big — you can use a total number of source lines as a useful, but not very accurate, measure of a code's size — you may want to divide it into many parts, implemented in parallel by a few, a dozen, several dozen, or even several hundred individual developers.

Of course, this cannot be done using one large source file, which is edited by all programmers at the same time. This will surely lead to a spectacular disaster. If you want such a software project to be completed successfully, you have to have the means allowing you to divide all the tasks among the developers and join all the created parts into one working whole.

For example, a certain project can be divided into two main parts: the user interface, which is the part that communicates with the user using widgets and a graphical screen; and the logic, which is the part processing data and producing results.

Each of these parts can be most likely divided into smaller ones, and so on. Such a process is often called decomposition. For example, if you were asked to arrange a wedding, you wouldn't do everything yourself – you would find a number of professionals and split the task between them all. How do you divide a piece of software into separate but cooperating parts? This is the question. Modules are the answer.

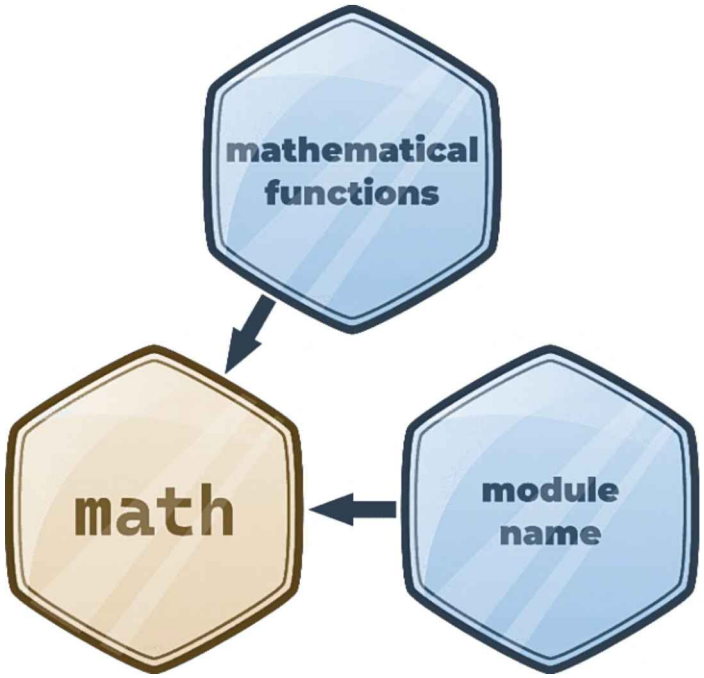
How to make use of a module

So what is a module? the Python Tutorial defines it as a file containing Python definitions and statements, which can be later imported and used when necessary. The handling of modules consists of two different issues. The first, and probably the most common, happens when you want to use an already existing module, written by someone else, or created by yourself during your work on some complex project – in this case you are the module's user. The second occurs when you want to create a brand new module, either for your own use, or to make other programmers' lives easier – you are the module's supplier.

Let's discuss them separately. First of all, a module is identified by its name. If you want to use any module, you need to know the name. A rather large number of modules is delivered together with Python itself. You can think of them as a kind of "Python extra equipment".

All these modules, along with the built-in functions, form the Python standard library – a special sort of library where modules play the roles of books. We can even say that folders play the roles of shelves. If you want to take a look at the full list of all "volumes" collected in that library, you can find it here: https://docs.python.org/3/library/index.html.

Each module consists of entities, like a book consists of chapters. These entities can be functions, variables, constants, classes, and objects. If you know how to access a particular module, you can make use of any of the entities it stores.



Let's start the discussion with one of the most frequently used modules, named math. Its name speaks for itself – the module contains a rich collection of entities (not only functions) which enable a programmer to effectively implement calculations demanding the use of mathematical functions, like *sin()* or *log()*.

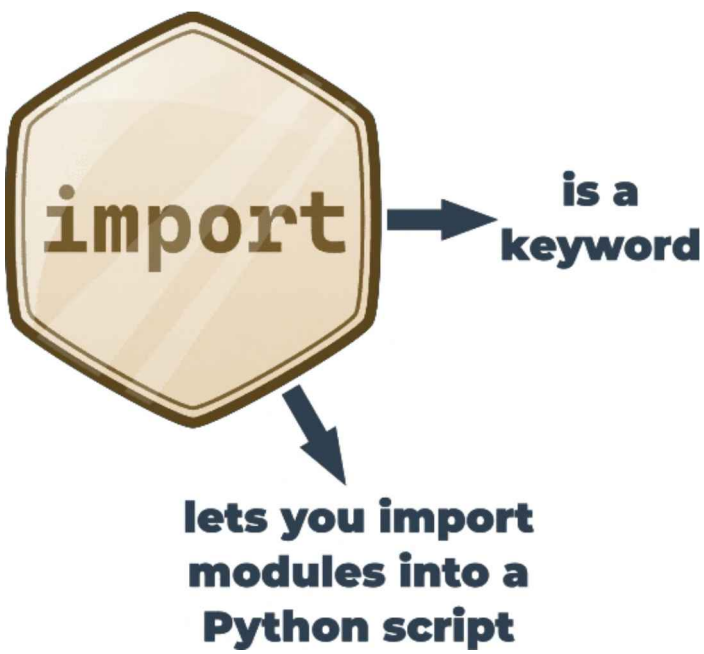
Importing a module

To make a module usable, you must import it. Think of it like of taking a book off the shelf. Importing a module is done by an instruction named import.

NOTE: import is also a keyword, with all the consequences of this fact.

Let's assume that you want to use two entities provided by the math module:

- a symbol, or constant, representing a precise, or as precise as possible using double floating-point arithmetic, value of π (although using a Greek letter to name a variable is fully possible in Python, the symbol is named pi – it's a more convenient solution, especially for that part of the world which neither has nor is going to use a Greek keyboard)
- a function named `sin()`, the computer equivalent of the mathematical *sine* function.



Both these entities are available through the `math` module, but the way in which you can use them strongly depends on how the import has been done. The simplest way to import a particular module is to use the import instruction as follows:

```
1  import math
2
```

The clause contains the import keyword and the name of the module which is subject to import. The instruction may be located anywhere in your code, but it must be placed before the first use of any of the module's entities. If you want to, or have to, import more than one module, you can do it by repeating the import clause, which is the preferred way:

```
1  import math
2  import sys
3
```

or by listing the modules after The `import` keyword, like here:

```
1  import math, sys
2
```

The instruction imports two modules, first the one named `math` and then the second named sys. The modules' list may be arbitrarily long.

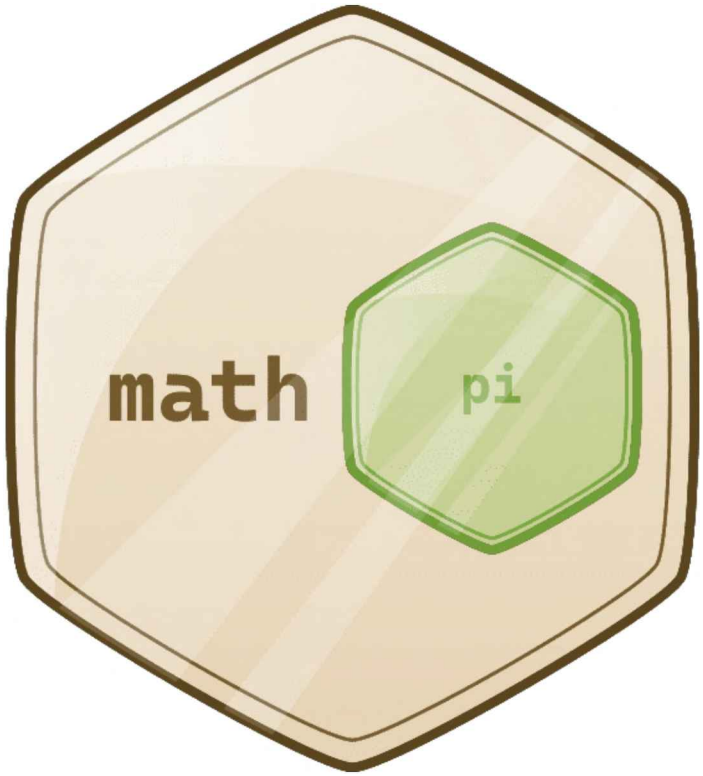
Namespace

To continue, you need to become familiar with an important term: namespace. Don't worry, we won't go into great detail – this explanation is going to be as short as possible. A namespace is a space, understood in a non-physical context, in which some names exist and the names don't conflict with each other (i.e. there are not two different objects of the same name). We can say that each social group is a namespace – the group tends to name each of its members in a unique way (e.g. parents won't give their children the same first names).

This uniqueness may be achieved in many ways, such as by using nicknames along with the first names, which works inside a small group like a class in a school, or by assigning special identifiers to all members of the group — the US Social Security Number is a good example of such a practice.

Inside a certain namespace, each name must remain unique. This may mean that some names may disappear when any other entity of an already known name enters the namespace. We'll show you how it works and how to control it, but first, let's return to imports.

If the module of a specified name exists and is accessible (a module is in fact a Python source file), Python imports its contents. In other words, all the names defined in the module become known, but they don't enter your code's namespace. This means that you can have your own entities named sin or pi and they won't be affected by the import in any way. At this point, you may be wondering how to access the pi coming from the math module. To do this, you have to qualify the pi with the name of its original module.



Importing a module: continued

This first example won't be very advanced – we just want to print the value of **sin(½π)**.

This is how we test it.

```
1  import math
2  print(math.sin(math.pi/2))
3
```

The code outputs the expected value: `1.0`. Look at following the snippet. This is the way in which you qualify the names of `pi` and `sin` with the name of its originating module:

```
1  math.pi
2  math.sin
3
```

It's simple, you put the name of the module (e.g. math), a dot (i.e. .), and the name of the entity (e.g. pi). Such a form clearly indicates the namespace in which the name exists.

NOTE: Using this qualification is compulsory if a module has been imported by the import module instruction. It doesn't matter if any of the names from your code and from the module's namespace are in conflict or not. Removing any of the two qualifications will make the code erroneous. There is no other way to enter math's namespace if you did the following:

```
1  import math
2
```

Now we're going to show you how the two namespaces (yours and the module's one) can coexist. Take a look at the following example. We've defined our own `pi` and `sin` here.

```
import math

def sin(x):
2 * x == pi:
0.99999999
else:
None


pi = 3.14
print(sin(pi/2))
print(math.sin(math.pi/2))

```

The code should produce the following output:

```
0.99999999
1.0
```

As you can see, the entities don't affect each other. In the second method, The `import`'s syntax precisely points out which module's entity (or entities) is acceptable in the code:

```
from math import pi

```

The instruction consists of the following elements: the from keyword; the name of the module to be selectively imported; the import keyword; and the name or list of names of the entity/entities which are being imported into the namespace.

The instruction has this effect: the listed entities, and only those ones, are imported from the indicated module; and the names of the imported entities are accessible without qualification.

NOTE: No other entities are imported. Moreover, you cannot import additional entities using a qualification – a line like this one will cause an error (e is Euler's number: 2.71828...):

```
print(math.e)

```

Let's rewrite the previous script to incorporate the new technique. Here it is:

```
from math import sin, pi

print(sin(pi/2))

```

The output should be the same as previously, as in fact we've used the same entities as before: `1.0`. Copy the code and run the program. Does the code look simpler? Maybe, but the look is not the only effect of this kind of import. Let's show you that. Analyze the following code:

- line 1: carry out the selective import;
- line 3: make use of the imported entities and get the expected result (`1.0`)
- lines 5 through 12: redefine the meaning of `pi` and `sin` – in effect, they supersede the original (imported) definitions within the code's namespace;
- line 15: get `0.99999999`, which confirms our conclusions.

```
from math import sin, pi

print(sin(pi / 2))

pi = 3.14


def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None


print(sin(pi / 2))

```

Let's do another test. Look at the following code:

```
1  pi = 3.14
2
3
4  def sin(x):
5      if 2 * x == pi:
6          return 0.99999999
7      else:
8          return None
9
10
11 print(sin(pi / 2))
12
13 from math import sin, pi
14
15 print(sin(pi / 2))
16
```

Here, we've reversed the sequence of the code's operations:

- lines 1 through 8: define our own `pi` and `sin`;
- line 11: make use of them (`0.99999999` appears on the screen)
- line 13: carry out the import – the imported symbols supersede their previous definitions within the namespace;
- line 15: get `1.0` as a result.

Importing a module: *

In the third method, the `import`'s syntax is a more aggressive form of the previous one:

```
1  from module import *
2
```

As you can see, the name of an entity, or the list of entity names, is replaced with a single asterisk (`*` .). Such an instruction imports all entities from the indicated module. Is it convenient? Yes, it is, as it relieves you of the duty of enumerating all the names you need. Is it unsafe? Yes, it is – unless you know all the names provided by the module, you may not be able to avoid name conflicts. Treat this as a temporary solution, and try not to use it in regular code.

The as keyword

If you use the import module variant and you don't like a particular module's name (e.g. it's the same as one of your already defined entities, so qualification becomes troublesome) you can give it any name you like – this is called **aliasing**.

Aliasing causes the module to be identified under a different name than the original. This may shorten the qualified names, too. Creating an alias is done together with importing the module, and demands the following form of the import instruction:

```
1  import module as alias
2
```

The "module" identifies the original module's name while the "alias" is the name you wish to use instead of the original.

NOTE: `as` is a keyword.

Aliasing

If you need to change the word `math`, you can introduce your own name, just like in the example:

```
1  import math as m
2
3  print(m.sin(m.pi/2))
4
```

NOTE: after successful execution of an aliased import, the **original module name becomes inaccessible** and must not be used.

In turn, when you use The `from module import name` variant and you need to change the entity's name, you make an alias for the entity. This will cause the name to be replaced by the alias you choose. This is how it can be done:

```
1  from module import name as alias
2
```

As previously, the original (unaliased) name becomes inaccessible. The phrase `name as alias` can be repeated – use commas to separate the multiplied phrases, like this:

```
from module import n as a, m as b, o as c

```

The example may look a bit weird, but it works:

```
from math import pi as PI, sin as sine

print(sine(PI/2))

```

Summary

1. If you want to import a module as a whole, you can do it using The `import module_name` statement. You are allowed to import more than one module at once using a comma-separated list. For example:

```
import mod1
import mod2, mod3, mod4

```

although the latter form is not recommended due to stylistic reasons, and it's better and prettier to express the same intention in more a verbose and explicit form, such as:

```
import mod2
import mod3
import mod4

```

2. If a module is imported in this manner and you want to access any of its entities, you need to prefix the entity's name using dot notation. For example:

```
import my_module

result = my_module.my_function(my_module.my_data)

```

The snippet makes use of two entities coming from The `my_module` module: a function named `my_function()` and a variable named `my_data`. Both names **must be prefixed by** `my_module`. None of the imported entity names conflicts with the identical names existing in your code's namespace.

3. You are allowed not only to import a module as a whole, but to import only individual entities from it. In this case, the imported entities **must not be** prefixed when used. For example:

```
from module import my_function, my_data

result = my_function(my_data)

```

This way – despite its attractiveness – is not recommended because of the danger of causing conflicts with names derived from importing the code's namespace.

4. The most general form of this statement allows you to import **all entities** offered by a module:

```
from my_module import *

result = my_function(my_data)

```

Note: this `import`'s variant is not recommended due to the same reasons as previously (the threat of a naming conflict is even more dangerous here).

5. You can change the name of the imported entity "on the fly" by using The `as` phrase of The `import`. For example:

```
from module import my_function as fun, my_data as dat

result = fun(dat)

```

Quiz

Question 1: You want to invoke the function `make_money()` contained in the module named `mint`. Your code begins with the following line:

```
import mint
```

What is the proper form of the function's invocation?

Question 2: You want to invoke the function `make_money()` contained in the module named `mint`. Your code begins with the following line:

```
from mint import make_money
```

What is the proper form of the function's invocation?

Question 3: You've written a function named `make_money` on your own. You need to import a function of the same name from The `mint` module and don't want to rename any of your previously defined names. Which variant of The `import` statement may help you with the issue?

Question 4: What form of The `make_money` function invocation is valid if your code starts with the following line?

```
from mint import*
```

Check Answers

TWO – SELECTED PYTHON MODULES (MATH, RANDOM, PLATFORM)

Before we start going through some standard Python modules, we want to introduce the `dir()` function to you. It has nothing to do with the `dir` command you know from Windows and Unix consoles, as `dir()` doesn't show the contents of a disk directory/folder, but there is no denying that it does something really similar: it is able to reveal all the names provided through a particular module. There is one condition: the module has to have been previously imported as a whole using the `import module` instruction – `from module` is not enough. The function returns an alphabetically sorted list containing all entities' names available in the module identified by a name passed to the function as an argument:

```
1  dir(module)
2
```

NOTE: If the module's name has been aliased, you must use the alias, not the original name. Using the function inside a regular script doesn't make much sense, but it is still possible. For example, you can run the following code to print the names of all entities within the math module:

```
1  import math
2
3  for name in dir(math):
4  print(name, end="\t")
5
```

The example code should produce the following output:

```
__doc__ __loader__ __name__ __package__ __spec__ acos acosh asin asinh atan atan2
atan2 atanh ceilcopysign cos cosh degrees e erf erfc exp expm1 fabs factorial floor
fmod frexp fsum gamma hypot isfinite isinf isnan ldexp lgamma log log10 log1p
log2 modf pi pow radians sin sinh sqrt tan tanh trunc
```

Have you noticed these strange names beginning with __ at the top of the list? We'll tell you more about them when we talk about the issues related to writing your own modules. Some of the names might bring back memories from math lessons, and you probably won't have any problems guessing their meanings.

Using the `dir()` function inside a code may not seem very useful – usually you want to know a particular module's contents before you write and run the code. Fortunately, you can execute the function directly in the Python console (IDLE), without needing to write and run a separate script. This is how it can be done:

```
1  import math
2  dir(math)
3
```

You should see something similar to this:

Selected functions from the math module

Let's start with a quick preview of some of the functions provided by The `math` module. We've chosen them arbitrarily, but that doesn't mean that the functions we haven't mentioned here are any less significant. Dive into the module's depths yourself – we don't have the space or the time to talk about everything in detail here. The first group of The `math`'s functions is connected with trigonometry:

- `sin(x)` → the sine of x;
- `cos(x)` → the cosine of x;
- `tan(x)` → the tangent of x.

All these functions take one argument, an angle measurement expressed in radians, and return the appropriate result. Be careful with `tan()` – not all arguments are accepted. Of course, there are also their inverted versions:

- `asin(x)` → the arcsine of x;
- `acos(x)` → the arccosine of x;
- `atan(x)` → the arctangent of x.

These functions take one argument and return a measure of an angle in radians – mind the domains! To effectively operate on angle measurements, the `math` module provides you with the following entities:

- `Pi` → a constant with a value that is an approximation of π;
- `radians(x)` → a function that converts x from degrees to radians;
- `degrees(x)` → acting in the other direction (from radians to degrees)

Now look at the code. The example program isn't very sophisticated, but can you predict its results?

```
from math import pi, radians, degrees, sin, cos, tan, asin

ad = 90
ar = radians(ad)
ad = degrees(ar)

print(ad == 90.)
print(ar == pi / 2.)
print(sin(ar) / cos(ar) == tan(ar))
print(asin(sin(ar)) == ar)

```

Apart from the circular functions listed previously, The `math` module also contains a set of their hyperbolic analogues:

- `sinh(x)` → the hyperbolic sine;
- `cosh(x)` → the hyperbolic cosine;
- `tanh(x)` → the hyperbolic tangent;
- `asinh(x)` → the hyperbolic arcsine;
- `acosh(x)` → the hyperbolic arccosine;
- `atanh(x)` → the hyperbolic arctangent.

Another group of The `math`'s functions is formed by functions which are connected with exponentiation:

- `e` → a constant with a value that is an approximation of Euler's number (e)
- `exp(x)` → finding the value of e^x;
- `log(x)` → the natural logarithm of x
- `log(x, b)` → the logarithm of x to base b
- `log10(x)` → the decimal logarithm of x (more precise than `log(x, 10)`)
- `log2(x)` → the binary logarithm of x (more precise than `log(x, 2)`)

NOTE: The `pow()` function:

`pow(x, y)` → finding the value of x^y (mind the domains)

This is a built-in function, and doesn't have to be imported. Take a look at this code. Can you predict its output?

```
from math import e, exp, log

print(pow(e, 1) == exp(log(e)))
print(pow(2, 2) == exp(2 * log(2)))
print(log(e, e) == exp(0))

```

The last group consists of some general-purpose functions like:

- `ceil(x)` → the ceiling of x — the smallest integer greater than or equal to x;
- `floor(x)` → the floor of x — the largest integer less than or equal to x;
- `trunc(x)` → the value of x truncated to an integer (be careful – it's not an equivalent either of ceil or floor)
- `factorial(x)` → returns x! — x has to be an integral and not a negative;
- `hypot(x, y)` → returns the length of the hypotenuse of a right-angle triangle with the leg lengths equal to x and y (the same as `sqrt(pow(x, 2) + pow(y, 2))` but more precise)

Analyze the program carefully. It demonstrates the fundamental differences between `ceil()`, `floor()` and `trunc()`.

```
from math import ceil, floor, trunc

x = 1.4
y = 2.6

print(floor(x), floor(y))
print(floor(-x), floor(-y))
print(ceil(x), ceil(y))
print(ceil(-x), ceil(-y))
print(trunc(x), trunc(y))
print(trunc(-x), trunc(-y))

```

Run the program and check its output.

Is there real randomness in computers?

Another module worth mentioning is the one named `random`. It delivers some mechanisms allowing you to operate with pseudo-random numbers. Note the prefix pseudo – the 16 numbers generated by the modules may look random in the sense that you cannot predict their subsequent values, but don't forget that they all are calculated using very refined algorithms.

The algorithms aren't random – they are deterministic and predictable. Only those physical processes which run completely out of our control, like the intensity of cosmic radiation, may be used as a source of actual random data. Data produced by deterministic computers cannot be random in any way.

A random number generator takes a value called a seed, treats it as an input value, calculates a "random" number based on it (the method depends on a chosen algorithm) and produces a new seed value. The length of a cycle in which all seed values are unique may be very long, but it isn't infinite – sooner or later the seed values will start repeating, and the generating values will repeat too. This is normal. It's a feature, not a mistake, or a bug.

The initial seed value, set during the program start, determines the order in which the generated values will appear. The random factor of the process may be augmented by setting the seed with a number taken from the current time – this may ensure that each program launch will start from a different seed value; ergo, it will use different random numbers. Fortunately, such an initialization is done by Python during module import.

Selected functions from The `random` module

The `random` function

The most general function named `random()` — not to be confused with the module's name — produces a float number x coming from the range `(0.0, 1.0)` – in other words: (0.0 <= x < 1.0). The following example program will produce five pseudo-random values – as their values are determined by the current, rather unpredictable, seed value, you can't guess them:

```
1  from random import random
2
3  for i in range(5):
4  print(random())
5
```

Run the program. This is what we've got:

```
0.9535768927411208
0.5312710096244534
0.8737691983477731
0.5896799172452125
0.02116716297022092
```

The seed function

The `seed()` function is able to directly set the generator's seed. We'll show you two of its variants: `seed()`, which sets the seed with the current time; and `seed(int_value)`, which sets the seed with the integer value `int_value`. We've modified the previous program – in effect, we've removed any trace of randomness from the code:

```
1  from random import random, seed
2
3  seed(0)
4
5  for i in range(5):
6  print(random())
7
```

Due to the fact that the seed is always set with the same value, the sequence of generated values always looks the same. Run the program. This is what we've got:

```
0.844421851525
0.75795440294
0.420571580831
0.258916750293
0.511274721369
```

And you?

NOTE: your values may be slightly different than ours if your system uses more precise or less precise floating-point arithmetic, but the difference will be seen quite far from the decimal point.

The `randrange` and `randint` functions

If you want integer random values, one of the following functions would fit better: `randrange(end)`, `randrange(beg, end)`, `randrange(beg, end, step)`, or `randint(left, right)`. The first three invocations will generate an integer taken pseudo randomly from the respective range: `range(end)`, `range(beg, end)`, or `range(beg, end, step)`. Note the implicit right-sided exclusion.

The last function is an equivalent of `randrange(left, right+1)` – it generates the integer value `1`, which falls in the range [left, right]. There is no exclusion on the right side. Look at this code. This sample program will consequently output a line consisting of three zeros and either a zero or one at the fourth place.

```
1  from random import randrange, randint
2
3  print(randrange(1), end=   )
4  print(randrange(0, 1), end=' ')
5  print(randrange(0, 1, 1), end=' ')
6  print(randint(0, 1))
7
```

The previous functions have one important disadvantage – they may produce repeating values even if the number of subsequent invocations is not greater than the width of the specified range. Look at the following code – the program very likely outputs a set of numbers in which some elements are not unique:

```
1  from random import randint
2
3  for i in range(10):
4      print(randint(1, 10), end=',')
5
```

This is what we got in one of the launches:

```
9,4,5,4,5,8,9,4,8,4,
```

The `choice` and `sample` functions

As you can see, this is not a good tool for generating numbers in a lottery. Fortunately, there is a better solution than writing your own code to check the uniqueness of the "drawn" numbers. It's a function named in a very suggestive way: `choice(sequence)`, and `sample(sequence, elements_to_choose)`.

The first variant chooses a "random" element from the input sequence and returns it. The second one builds a list, or a sample, consisting of the `elements_to_choose` element "drawn" from the input sequence. In other words, the function chooses some of the input elements, returning a list with the choice. The elements in the sample are placed in random order.

NOTE: The `elements_to_choose` must not be greater than the length of the input sequence.

Look at the following code:

```
1  from random import choice, sample
2
3  my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4
5  print(choice(my_list))
6  print(sample(my_list, 5))
7  print(sample(my_list, 10))
8
```

Again, the output of the program is not predictable. Our results looked like this:

```
4
[3, 1, 8, 9, 10]
[10, 8, 5, 1, 6, 4, 3, 9, 7, 2]
```

How to know where you are

Sometimes, it may be necessary to find out information unrelated to Python. For example, you may need to know the location of your program within the greater environment of the computer. Imagine your program's environment as a pyramid consisting of a number of layers or platforms. The layers are your running code, located at the top of it; Python, or more precisely, its runtime environment, which lies directly below it; the OS (operating system) – Python's environment provides some of its functionalities using the operating system's services, because Python, although very powerful, isn't omnipotent – it's forced to use many helpers if it's going to process files or communicate with physical devices; and finally the bottom-most layer, the hardware – the processor (or processors), network interfaces, human interface devices (mice, keyboards, etc.) and all other machinery needed to make the computer run. The OS knows how to drive it, and uses lots of tricks to conduct all parts in a consistent rhythm.

This means than some of your, or rather your program's, actions have to travel a long way to be successfully performed. Imagine that your code wants to create a file, so it invokes one of Python's functions. Python accepts the order, rearranges it to meet local OS requirements — it's like putting the stamp "approved" on your request — and sends it down. This may remind you of a chain of command. The OS checks if the request is reasonable and valid, for example, whether the file name conforms to some syntax rules, and tries to create the file; such an operation, seemingly very simple, isn't atomic – it consists of many minor steps taken by the hardware, which is responsible for activating storage devices (hard disk, solid state devices, etc.) to satisfy the OS's needs.

Usually, you're not aware of all that fuss – you want the file to be created and that's that. But sometimes you want to know more – for example, the name of the OS which hosts Python, and some characteristics describing the hardware that hosts the OS. There is a module providing some means to allow you to know where you are and what components work for you. The module is named platform. We'll show you some of the functions it provides to you.

Selected functions from The `platform` module

The `platform` function

The `platform` module lets you access the underlying platform's data, that is, the hardware, operating system, and interpreter version information. There is a function that can show you all the underlying layers in one glance, named `platform`, too. It just returns a string describing the environment; thus, its output is rather addressed to humans than to automated processing. You'll see this soon. This is how you can invoke it:

```
1  platform(aliased = False, terse = False)
2
```

And now:

- `aliased` → when set to `True` (or any non-zero value) it may cause the function to present the alternative underlying layer names instead of the common ones;
- `terse` → when set to `True` (or any non-zero value) it may convince the function to present a briefer form of the result (if possible)

We ran our sample program:

```
1  from platform import platform
2
3  print(platform())
4  print(platform(1))
5  print(platform(0, 1))
6
```

Using three different platforms, this is what we get:

Intel x86 + Windows ® Vista (32 bit):

```
Windows-Vista-6.0.6002-SP2
Windows-Vista-6.0.6002-SP2
Windows-Vista
```

Intel x86 + Gentoo Linux (64 bit):

```
Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_CPU_@_2.20GHz-with-gentoo-2.3
Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_CPU_@_2.20GHz-with-gentoo-2.3
Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_CPU_@_2.20GHz-with-glibc2.3.4
```

Raspberry PI2 + Raspbian Linux (32 bit)

```
Linux-4.4.0-1-rpi2-armv7l-<>with-debian-9.0
Linux-4.4.0-1-rpi2-armv7l-<>with-debian-9.0
Linux-4.4.0-1-rpi2-armv7l-<>with-glibc2.9
```

You can also run the sample program in IDLE on your local machine to check what output you will have.

The `machine` function

Sometimes, you may just want to know the generic name of the processor which runs your OS together with Python and your code – a function named `machine()` will tell you that. As previously, the function returns a string. Again, we ran the sample program:

```
1  from platform import machine
2
3  print(machine())
4
```

On three different platforms:

Intel x86 + Windows ® Vista (32 bit):

```
x86
```

Intel x86 + Gentoo Linux (64 bit):

```
x86_64
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
armv7l
```

The `processor` function

The `processor()` function returns a string filled with the real processor name, if possible. Once again, we ran the sample program:

```
1  from platform import processor
2
3  print(processor())
4
```

On three different platforms:

Intel x86 + Windows ® Vista (32 bit):

```
x86
```

Intel x86 + Gentoo Linux (64 bit):

```
Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
armv7l
```

Test this on your local machine.

The `system` function

A function named `system()` returns the generic OS name as a string.

```
1  from platform import system
2
3  print(system())
4
```

Our example platforms presented themselves like this:

Intel x86 + Windows ® Vista (32 bit):

```
Windows
```

Intel x86 + Gentoo Linux (64 bit):

```
Linux
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
Linux
```

The `version` function

The OS version is provided as a string by the `version()` function.

```
1  from platform import version
2
3  print(version())
4
```

Run the code and check its output. This is what we got:

Intel x86 + Windows ® Vista (32 bit):

```
6.0.6002
```

Intel x86 + Gentoo Linux (64 bit):

```
#1 SMP PREEMPT Fri Jul 21 22:44:37 CEST 2017
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
#1 SMP Debian 4.4.6-1+rpi14 (2016-05-05)
```

The `python_implementation` and The `python_version_tuple` functions

If you need to know what version of Python is running your code, you can check it using a number of dedicated functions. Two of them are `python_implementation()`, which returns a string denoting the Python implementation — expect CPython here, unless you decide to use any non-canonical Python branch — and `python_version_tuple()`, which returns a three-element tuple filled with the major part of Python's version, the minor part, and the patch level number.

```
1  from platform import python_implementation, python_version_tuple
2
3  print(python_implementation())
4
5  for atr in python_version_tuple():
6  print(atr)
7
```

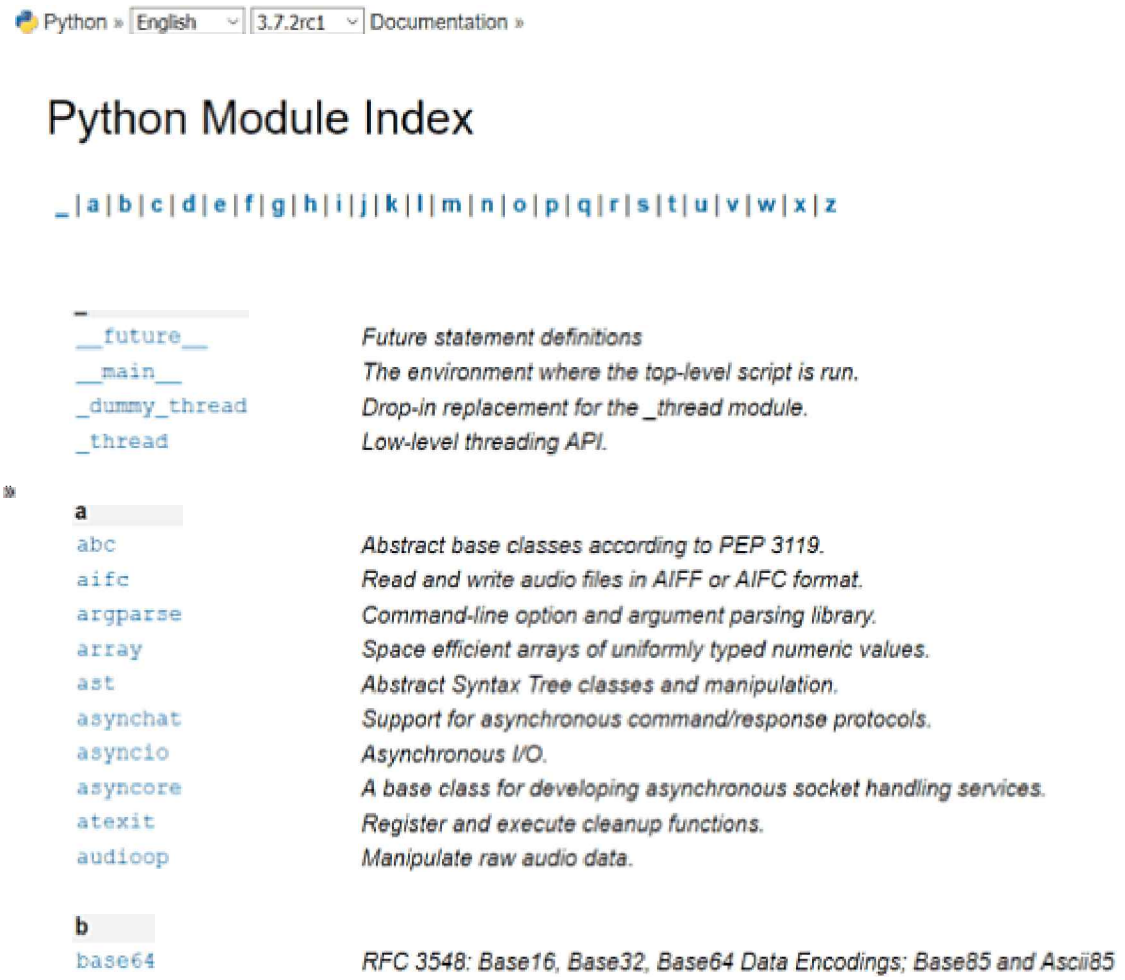
Our example program produced the following output:

```
CPython
3
7
7
```

It's very likely that your version of Python will be different.

Python Module Index

We have only covered the basics of Python modules here. Python's modules make up their own universe, in which Python itself is only a galaxy, and we would venture to say that exploring the depths of these modules can take significantly more time than getting acquainted with "pure" Python. Moreover, the Python community all over the world creates and maintains hundreds of additional modules used in very niche applications like genetics, psychology, or even astrology. These modules aren't distributed along with Python, or through official channels, which makes the Python universe broader – almost infinite.



Don't worry – you won't need all these modules. Many of them are very specific. All you need to do is find the modules you want, and teach yourself how to use them. It's easy.

Summary

1. A function named `dir()` can show you a list of the entities contained inside an imported module. For example:

```
1  import os
2  dir(os)
3
```

It prints out the list of all The `os` module's facilities you can use in your code.

2. The `math` module couples more than 50 symbols (functions and constants) that perform mathematical operations (like `sine()`, `pow()`, `factorial()`) or provide important values (like π and the Euler symbol e).

3. The `random module` groups more than 60 entities designed to help you use pseudo-random numbers. Don't forget the prefix "pseudo", as there is no such thing as a real random number when it comes to generating them using the computer's algorithms.

4. The `platform` module contains about 70 functions which let you dive into the underlying layers of the OS and hardware. Using them allows you to get to know more about the environment in which your code is executed.

5. Python Module Index (https://docs.python.org/3/py-modindex.html is a community-driven directory of modules available in the Python universe. If you want to find a module fitting your needs, start your search there.

Quiz

Question 1: What is the expected value of The `result` variable after the following code is executed?

```
import math
result = math.e == math.exp(1)
```

Question 2: (Complete the sentence) Setting the generator's seed with the same value each time your program is run guarantees that...

Question 3: Which of The `platform` module's functions will you use to determine the name of the CPU running inside your computer?

Question 4: What is the expected output of the following snippet?

```
import platform
print(len(platform.python_version_tuple()))
```

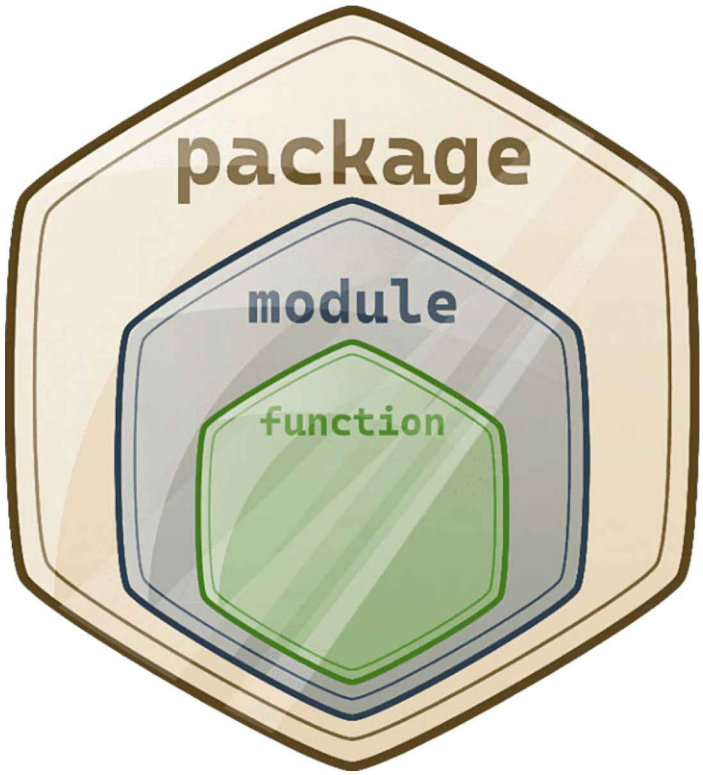
Check Answers

THREE – MODULES AND PACKAGES

Writing your own modules doesn't differ much from writing ordinary scripts. There are some specific aspects you must be aware of, but it definitely isn't rocket science. You'll see this soon enough. Let's summarize some important issues.

A module is a kind of container filled with functions – you can pack as many functions as you want into one module and distribute it across the world.

Of course, just like in a library, where nobody expects scientific works to be put among comic books, it's generally a good idea not to mix functions with different application areas within one module. Therefore, group your functions carefully and name the module containing them in a clear and intuitive way. Don't, for example, give the name `arcade_games` to a module containing functions intended to partition and format hard disks.



Making many modules may cause a little mess – sooner or later you'll want to group your modules exactly in the same way as you've previously grouped functions. Is there a more general container than a module? Yes, there is. It's a package; in the world of modules, a package plays a similar role to a folder/directory in the world of files.

Your first module

Step 1

In this section you're going to be working locally on your machine. Let's start from scratch. Create an empty file, just like this:

```
1
2
```

module.py

You will need two files to repeat these experiments. The first of them will be the module itself. It's empty now. Don't worry, you're going to fill it with actual code soon. We've named the file `module.py`. Not very creative, but simple and clear.

Step 2

The second file contains the code using the new module. Its name is main.py. Its content is very brief so far:

```
1  import module
2
```

main.py

NOTE: Both files have to be located in the same folder. We strongly encourage you to create an empty, new folder for both files. Some things will be easier then. Launch IDLE (or any other IDE you prefer) and run the `main.py file`. What do you

see? You should see nothing. This means that Python has successfully imported the contents of the `module.py` file. It doesn't matter that the module is empty for now. The very first step has been done, but before you take the next step, we want you to take a look into the folder in which both files exist.

Do you notice something interesting? A new subfolder has appeared – can you see it? Its name is `__pycache__`. Take a look inside. What do you see? There is a file named `module.cpython-xy.pyc`, more or less, where `x` and `y` are digits derived from your version of Python (e.g. they will be `3` and `8` if you use Python 3.8). The name of the file is the same as your module's name — `module` here. The part after the first dot says which Python implementation has created the file — `CPython` here — and its version number. The last part (`pyc`) comes from the words *Python* and *compiled*. You can look inside the file – the contents are completely unreadable to humans. It has to be like that, as the file is intended for Python's use only.

When Python imports a module for the first time, it translates its contents into a somewhat compiled shape. The file doesn't contain machine code – it's internal Python semi-compiled code, ready to be executed by Python's interpreter. As such, the file doesn't require lots of the checks needed for a pure source file, the execution starts faster, and it runs faster, too. Thanks to that, every subsequent import will go quicker than interpreting the source text from scratch.

Python is able to check if the module's source file has been modified or not. In this case, the `pyc` file will be rebuilt, in which case the `pyc` file may be run at once. As this process is fully automatic and transparent, you don't have to keep it in mind.

Step 3

Now we've put a little something into the module file:

```
1  print("I like to be a module.")
2
```

module.py

Can you notice any differences between a module and an ordinary script? There are none so far. It's possible to run this file like any other script. Try it for yourself. What happens? You should see the following line inside your console:

```
I like to be a module.
```

Step 4

Let's go back to The `main.py` file:

```
1  import module
2
```

main.py

Run it. What do you see? Hopefully, you see something like this:

```
I like to be a module.
```

What does it actually mean? When a module is imported, its contents are implicitly executed by Python. It gives the module the chance to initialize some of its internal aspects (e.g. it may assign some variables with useful values).

NOTE: The initialization takes place only once, when the first import occurs, so the assignments done by the module aren't repeated unnecessarily.

Imagine that there is a module named `mod1`, there is a module named `mod2` which contains the `import mod1` instruction, and there is a main file containing the `import mod1` and `import mod2` instructions. At first glance, you may think that `mod1` will be imported twice – fortunately, only the first import occurs. Python remembers the imported modules and silently omits all subsequent imports.

Step 5

But Python does much more than just imports the module. It also creates a variable called __name__. Moreover, each source file uses its own, separate version of the variable – it isn't shared between modules. We'll show you how to use it. Modify the module a bit:

```
1  print("I like to be a module.")
2  print(__name__)
3
```

module.py

Now run The `module.py` file. You should see the following lines:

```
I like to be a module

__main__
```

Now run the main.py file. And? Do you see the same as us?

```
I like to be a module

module
```

We can say that when you run a file directly, its `__name__` variable is set to `__main__`, and when a file is imported as a module, its `__name__` variable is set to the file's name, excluding .py.

Step 6

This is how you can make use of The `__main__` variable in order to detect the context in which your code has been activated:

```
1  if __name__ == "__main__":
2      print("I prefer to be a module.")
3  else:
4      print("I like to be a module.")
5
```

module.py

There's a cleverer way to utilize the variable, however. If you write a module filled with a number of complex functions, you can use it to place a series of tests to check if the functions work properly. Each time you modify any of these functions, you can simply run the module to make sure that your amendments didn't spoil the code. These tests will be omitted when the code is imported as a module.

Step 7

This module will contain two simple functions, and if you want to know how many times the functions have been invoked, you need a counter initialized to zero when the module is being imported. You can do it this way:

```
1  counter = 0
2
3  if __name__ == "__main__":
4      print("I prefer to be a module.")
5  else:
6      print("I like to be a module.")
7
```

module.py

Step 8

Introducing such a variable is absolutely correct, but may cause important side effects that you must be aware of. Take a look at the modified `main.py` file:

```
1  import module
2  print(module.counter)
3
```

main.py

As you can see, the main file tries to access the module's counter variable. Is this legal? Yes, it is. Is it usable? It may be very usable. Is it safe? That depends – if you trust your module's users, there's no problem; however, you may not want the rest of the world to see your personal/private variable. Unlike many other programming languages, Python has no means of allowing you to hide such variables from the eyes of the module's users.

You can only inform your users that this is your variable, that they may read it, but that they should not modify it under any circumstances. This is done by preceding the variable's name with _ (one underscore) or __ (two underscores), but remember, it's only a convention. Your module's users may obey it or they may not.

Of course, we'll follow the convention. Now let's put two functions into the module – they'll evaluate the sum and product of the numbers collected in a list. In addition, let's add some ornaments there and remove any superfluous remnants.

Step 9

Okay. Let's write some brand new code in our `module.py` file. The updated module is ready here:

```
#!/usr/bin/env python3

"" module.py - an example of a Python module ""

__counter = 0


def suml(the_list):
global __counter
  __counter += 1
 the_sum = 0
 for element in the_list:
   the_sum += element
return the_sum


def prodl(the_list):
global __counter
  __counter += 1
 prod = 1
 for element in the_list:
   prod *= element
return prod


if __name__ == "__main__":
print("I prefer to be a module, but I can do some tests for you.")
my_list = [i+1for i in  range(5)]
print(suml(my_list) == 15)
print(prodl(my_list) == 120)

```

module.py

A few elements need some explanation, we think. The line starting with `#!` has many names – it may be called *shabang, shebang, hashbang, poundbang* or even *hashpling* (don't ask us why). The name itself means nothing here – its role is more important.

From Python's point of view, it's just a comment as it starts with `#`. For Unix and Unix-like OSs, including MacOS, such a line instructs the OS how to execute the contents of the file — in other words, what program needs to be launched to interpret the text. In some environments, especially those connected with web servers, the absence of that line will cause trouble.

A string, maybe a multiline, placed before any module instructions, including imports, is called the doc-string, and should briefly explain the purpose and contents of the module. The functions defined inside the module — `suml()` and `prodl()` — are available for import. We've used the `__name__` variable to detect when the file is run stand-alone, and seized this opportunity to perform some simple tests.

Step 10

Now it's possible to use the updated module – this is one way:

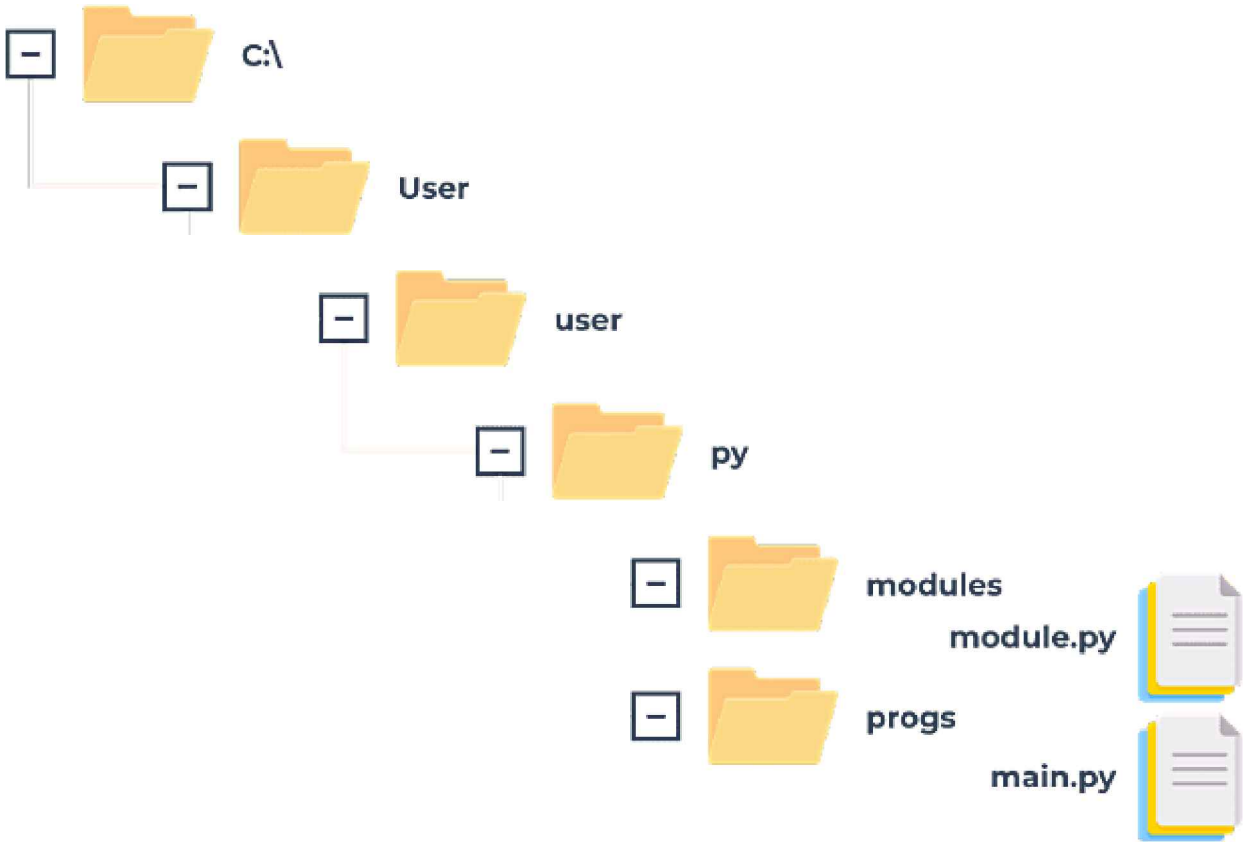
```
from module import suml, prodl

zeroes = [0 for i in range(5)]
ones = [1 for i in range(5)]
print(suml(zeroes))
print(prodl(ones))

```

main.py

Step 11

It's time to make our example more complicated – so far we've assumed that the main Python file is located in the same folder/directory as the module to be imported. Let's give up this assumption and conduct the following thought experiment: we are using Windows ® OS. This assumption is important, as the file name's shape depends on it; the main Python script lies in `C:\Users\user\py\progs` and is named main.py; and the module to import is located in `C:\Users\user\py\modules` How do we deal with it? To answer this question, we have to talk about how Python searches for modules. There's a special variable (actually a list) storing all locations (folders/ directories) that are searched in order to find a module which has been requested by the import instruction.

Python browses these folders in the order in which they are listed in the list – if the module cannot be found in any of these directories, the import fails. Otherwise, the first folder containing a module with the desired name will be taken into consideration. If any of the remaining folders contains a module of that name, it will be ignored. The variable is named `path`, and it's accessible through the module named `sys`. This is how you can check its regular value:




```
1  import sys
2
3  for p in sys.path:
4    print(p)
5
```

We've launched the code inside The `C:\User\user` folder, and this is what we've got:

```
C:\Users\user
C:\Users\user\AppData\Local\Programs\Python\Python36-32\python36.zip
C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib
C:\Users\user\AppData\Local\Programs\Python\Python36-32
C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib\site-packages
```

NOTE: The folder in which the execution starts is listed in the first path's element. Note once again that there is a zip file listed as one of the path's elements – it's not an error. Python is able to treat zip files as ordinary folders – this can save lots of storage. Can you figure out how we can solve our problem now? We can add a folder containing the module to the path variable, as it's fully modifiable.

Step 12

One of several possible solutions looks like this:

```
1  from sys import path
2
3  path.append('..\\modules')
4
5  importmodule
6
7  zeroes = [0 for i in range(5)]
8  ones = [1 for i in range(5)]
9  print(module.suml(zeroes))
10 print(module.prodl(ones))
11
```

main.py

NOTE: We've doubled the \ inside folder name – do you know why? Because a backslash is used to escape other characters – if you want to get just a backslash, you have to escape it. We've used the relative name of the folder – this will work if you start the `main.py` file directly from its home folder, and won't work if the current directory doesn't fit the relative path; you can always use an absolute path, like this:

```
path.append('C:\\Users\\user\\py\\modules')
```

We've used the `append()` method – in effect, the new path will occupy the last element in the path list; if you don't like the idea, you can use `insert()` instead.

Your first package

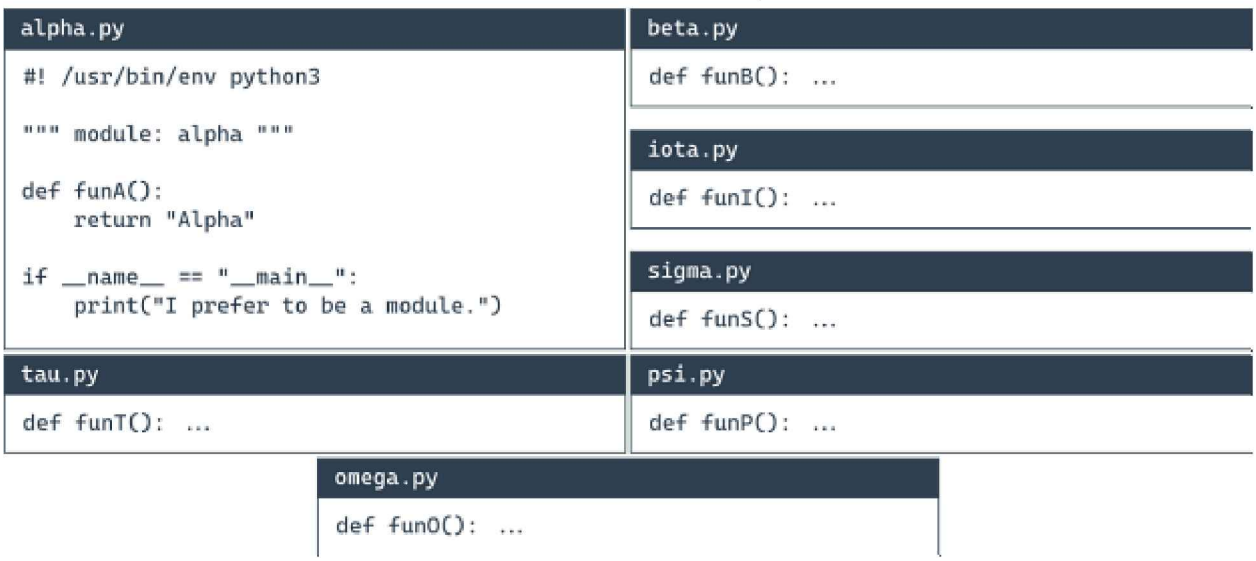
Step 1

Imagine that in the not-so-distant future you and your associates write a large number of Python functions. Your team decides to group the functions in separate modules, and this is the final result of the ordering:

```
1  #! /usr/bin/env python3
2
3  """ module: alpha """"
4
5  def funA():
6    return "Alpha"
7
8  if __name__ == "__main__":
9    print("I prefer to be a module.")
10
```

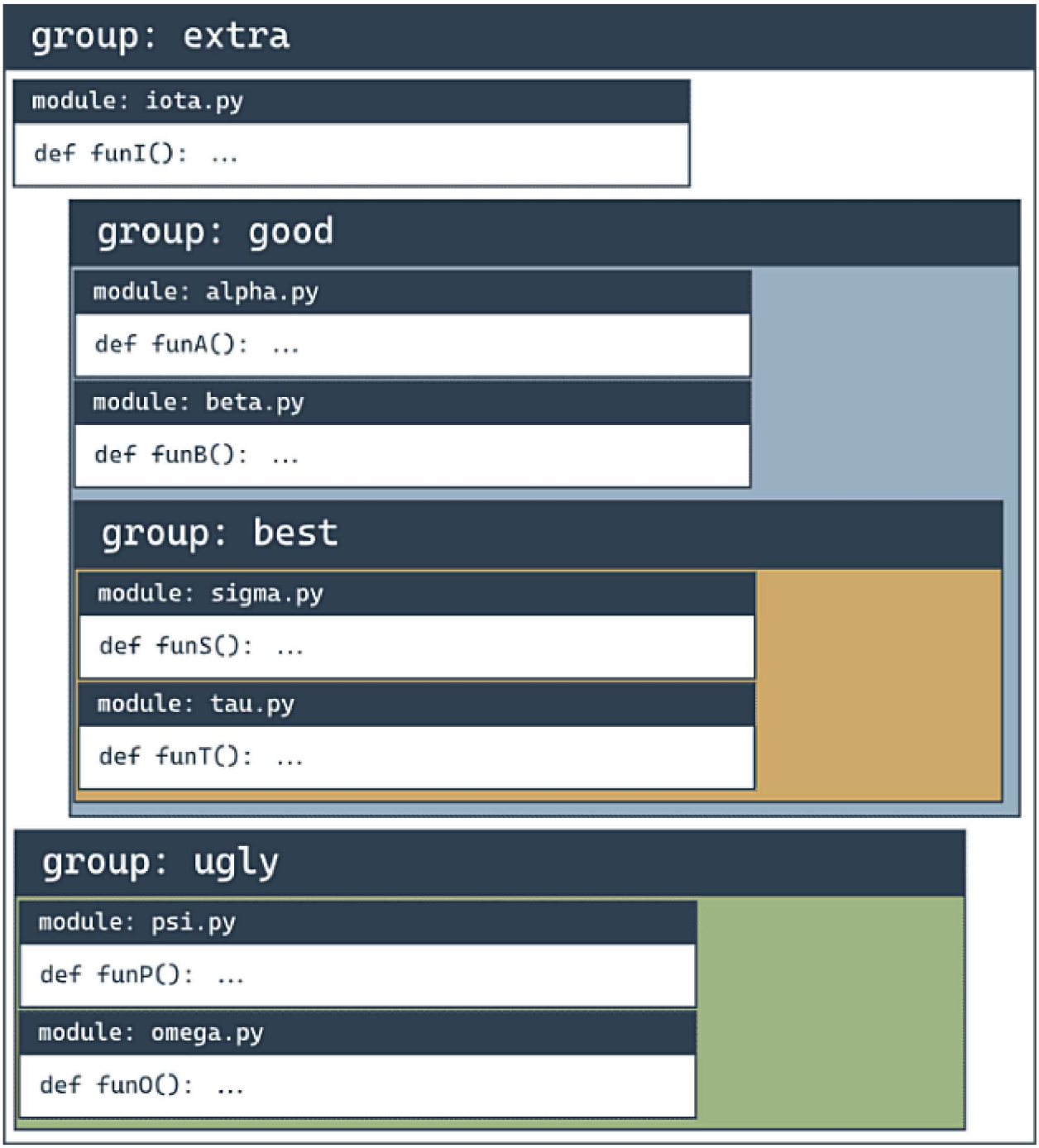
alpha.py

NOTE: we've presented the whole content for The `alpha.py` module only – assume that all the modules look similar (they contain one function named `funX`, where `X` is the first letter of the module's name).



Step 2

Suddenly, somebody notices that these modules form their own hierarchy, so putting them all in a flat structure won't be a good idea. After some discussion, the team comes to the conclusion that the modules have to be grouped. All participants agree that the following tree structure perfectly reflects the mutual relationships between the modules: Let's review this from the bottom up: the `ugly` group contains two modules: `psi` and `omega`; the `best` group contains two modules: `sigma` and `tau`; the `good` group contains two modules (`alpha` and `beta`) and one subgroup (`best`)l the `extra` group contains two subgroups (`good` and `ugly`) and one module (`iota`).



Does it look bad? Not at all – analyze the structure carefully. It resembles something, doesn't it? It looks like a directory structure. Let's build a tree reflecting the projected dependencies between the modules.

Step 3

This is how the tree currently looks:


extra
iota.py
good
alpha.py
beta.py
best
sigma.py
tau.py
ugly

Such a structure is almost a package (in the Python sense). It lacks the fine detail to be both functional and operative.

If you assume that *extra* is the name of a newly created package (think of it as the package's root), it will impose a naming rule which allows you to clearly name every entity from the tree.

For example:

- the location of a function named funT() from the tau module may be described as:

```
extra.good.best.tau.funT()
```

- a function marked as:

```
extra.ugly.psi.funP()
```

comes from The `psi` module being stored in The `ugly` subpackage of the extra package.

Step 4

Now there are two questions to answer. Firstly, how do you transform such a tree, or rather a subtree, into a real Python package? In other words, how do you convince Python that this tree is not just a bunch of junk files, but a set of modules? And secondly, where do you put the subtree to make it accessible to Python?

The first question has a surprising answer: packages, like modules, may require initialization. The initialization of a module is done by an unbound code, which is not a part of any function, located inside the module's file. As a package is not a file, this technique is useless for initializing packages. You need to use a different trick instead – Python expects there to be a file with a very unique name inside the package's folder:
`__init__.py`.

The contents of the file are executed when any of the package's modules is imported. If you don't want any special initializations, you can leave the file empty, but you can't omit it.

Step 5

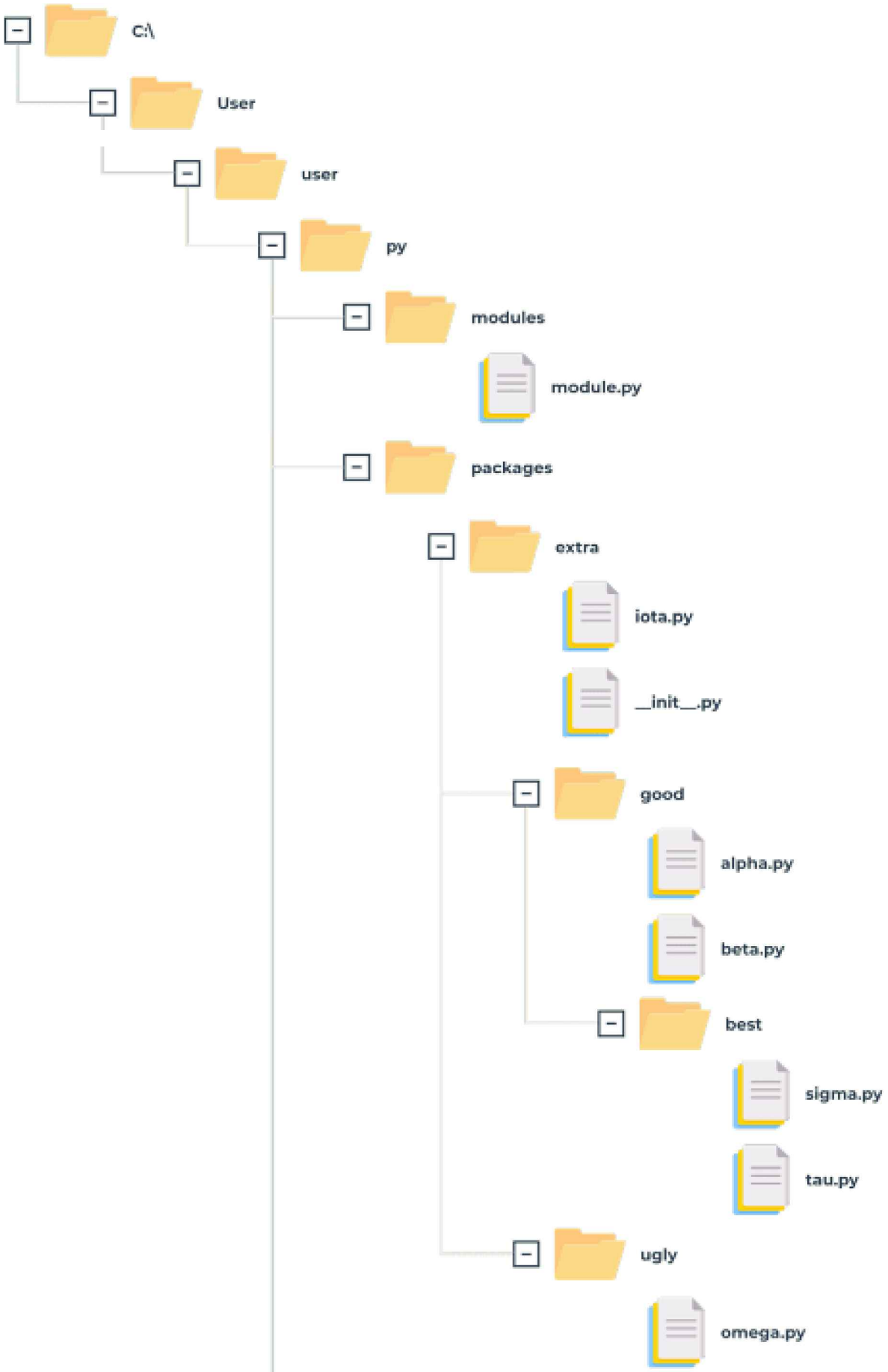
Remember: the presence of the `__init__.py` file finally makes up the package.


extra
iota.py
__init__.py
good
alpha.py
beta.py
best
sigma.py
tau.py

NOTE: It's not only the *root* folder that can contain the `__init.py__` file – you can put it inside any of its subfolders (sub packages) too. It may be useful to do this if some of the sub packages require individual treatment and special kinds of initialization. Now it's time to answer the second question – where do you put the subtree to make it accessible to Python? The answer is simple: anywhere. You only have to ensure that Python is aware of the package's location; you already know how to do that. You're ready to make use of your first package.

Step 6

Let's assume that the working environment looks as follows:


C:\
User
user
py
modules
module.py
packages
extra
iota.py
__init__.py
good
alpha.py
beta.py
best
sigma.py
tau.py
ugly
omega.py

We've prepared a zip file containing all the files from the packages branch. You can download it and use it for your own experiments, but remember to unpack it in the folder presented in the scheme; otherwise, it won't be accessible to the code from the main file. You'll be continuing your experiments using the `main2.py` file.

Step 7

We are going to access the `funI()` function from the `iota` module from the top of the `extra` package. It forces us to use qualified package names.

Associate this with naming folders and subfolders – the conventions are very similar:

```
1  from sys import path
2  path.append('..\\packages')
3
4  import extra.iota
5  print(extra.iota.funI())
6
```

main2.pyl

NOTE: We've modified the `path` variable to make it accessible to Python. The `import` doesn't point directly to the module, but specifies the fully qualified path from the top of the package. Replacing `import extra.iota` with `import iota` will cause an error. This variant is valid too:

```
1  from sys import path
2  path.append('..\\packages')
3
4  from extra.iota import funI
5  print(funI())
6
```

main.2.py

Note the qualified name of The `iota` module.

Step 8

Now let's reach all the way to the bottom of the tree – this is how to get access to the `sigma` and `tau` modules:

```
1  from sys import path
2
3  path.append('..\\packages')
4
5  import extra.good.best.sigma
6  from extra.good.best.tau import funT
7
8  print(extra.good.best.sigma.funS())
9  print(funT())
10
```

main2.py

You can make your life easier by using aliasing:

```
1  from sys import path
2
3  path.append('..\\packages')
4
5
6  import extra.good.best.sigma as sig
7  import extra.good.alpha as alp
8
9  print(sig.funS())
10 print(alp.funA())
11
```

main2.py

Step 9

Let's assume that we've zipped the whole subdirectory, starting from and including the `extra` folder, and let's get a file named `extrapack.zip`. Next, we put the file inside the `packages` folder.

Now we are able to use the zip file in a role of packages:

```
1  from sys import path
2
3  path.append('..\\packages\\extrapack.zip')
4
5  import extra.good.best.sigma as sig
6  import extra.good.alpha as alp
7  from extra.iota import funI
8  from extra.good.beta import funB
9
10 print(sig.funS())
11 print(alp.funA())
12 print(funI())
13 print(funB())
14
```

main2.py

Summary

1. While a module is designed to couple together some related entities such as functions, variables, or constants, a package is a container which enables the coupling of several related modules under one common name. Such a container can be distributed as-is, in other words, as a batch of files deployed in a directory sub-tree, or it can be packed inside a zip file.

2. During the very first import of the actual module, Python translates its source code into a semi-compiled format stored inside the pyc files, and deploys these files into the `__pycache__` directory located in the module's home directory.

3. If you want to tell your module's user that a particular entity should be treated as private (i.e. not to be explicitly used outside the module) you can mark its name with either the _ or __ prefix. Don't forget that this is only a recommendation, not an order.

4. The names *shabang, shebang, hasbang, poundbang*, and *hashpling* describe the digraph written as `#!`, used to instruct Unix-like OSs how the Python source file should be launched. This convention has no effect under MS Windows.

5. If you want convince Python that it should take into account a non-standard package's directory, its name needs to be inserted/appended into/to the import directory list stored in The `path` variable contained in The `sys` module.

6. A Python file named `__init__.py` is implicitly run when a package containing it is subject to import, and is used to initialize a package and/or its sub-packages (if any). The file may be empty, but must not be absent.

Quiz

Question 1: You want to prevent your module's user from running your code as an ordinary script. How will you achieve such an effect?

Question 2: Some additional and necessary packages are stored inside The `D:\Python\Project\Modules` directory. Write a code ensuring that the directory is traversed by Python in order to find all requested modules.

Question 3: The directory mentioned in the previous exercise contains a sub-tree of the following structure:

```
abc
 |__ def
     |__ mymodule.py
```

Assuming that `D:\Python\Project\Modules` has been successfully appended to The `sys.path` list, write an import directive letting you use all The `mymodule` entities.

Check Answers

FOUR – PYTHON PACKAGE INSTALLER (PIP)

Python is a very powerful instrument – we hope you've experienced this yourself already. Many people from around the world feel this way, and they use Python on a regular basis to develop what they can do in many completely different fields of activity. This means that Python has become an interdisciplinary tool employed in countless applications. We can't go through all the spheres in which Python brilliantly shows off its abilities, so let us just tell you about the most impressive ones.

First of all, Python has turned into a leader of research on artificial intelligence. Data mining, one of the most promising modern scientific disciplines, utilizes Python as well. Mathematicians, psychologists, geneticists, meteorologists, linguists – all these people already use Python, or if they don't already, we're sure that they will very soon. There is no escaping this trend.

Of course, it doesn't make any sense to get all Python users to write their code from scratch, keeping them perfectly isolated from the outside world and from other programmers' achievements. This would be both unnatural and counterproductive.

The most preferable and efficient thing is to enable all Python community members to freely exchange their codes and experiences. In this model, nobody is forced to start work from scratch, as there's a high probability that someone else has been working on the same problem, or one very similar.

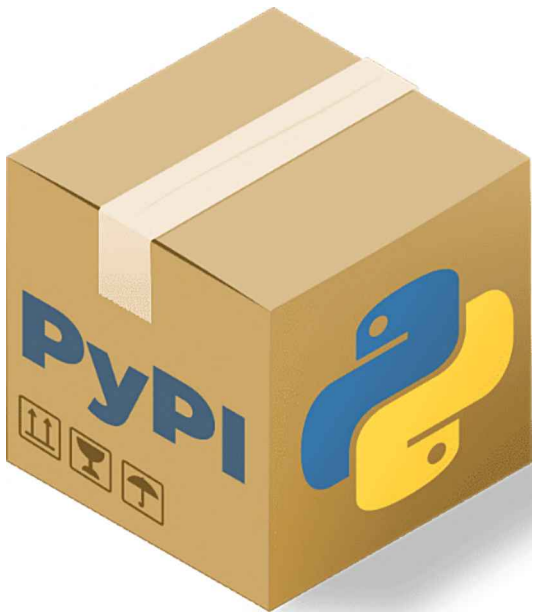
As you know, Python was created as open-source software, and this also works as an invitation for all coders to maintain the whole Python ecosystem as an open, friendly, and free environment. To make the model work and evolve, some additional tools should be provided, tools that help the creators to publish, maintain, and take care of their code.

These same tools should help users to make use of the code, both the already existing code, and also the new code appearing every day. Thanks to that, writing new code for new challenges is not like building a new house, starting at the foundations. Moreover, the programmer is free to modify someone else's code in order to adapt it to their own needs, and in effect build a completely new product that can be used by another developer. The process seems to have no end. Fortunately.

To make this world go round, two basic entities have to be established and kept in motion: a centralized repository of all available software packages; and a tool allowing users to access the repository. Both these entities already exist and can be used at any time. The repository (or *repo* for short) we mentioned before is named PyPI (it's short for Python Package Index) and it's maintained by a workgroup named the Packaging Working Group, a part of the Python Software Foundation, whose main task is to support Python developers in efficient code dissemination.

You can find their website here: https://wiki.python.org/psf/PackagingWG.

The PyPI website (administered by PWG) is located at the address: https://pypi.org/.



When we popped in there for a while in October 2023, we found that PyPI was home to 488,000 projects, consisting of over 9,300,000 files managed by 750,000 users. These three numbers alone clearly show the potency of the Python community and the importance of developer cooperation.

We must point out that PyPI is not the only existing Python repository. On the contrary, there are lots of them, created for projects and led by many larger and smaller Python communities. It's likely that someday you and your colleagues may want to create your own repos. Anyway, PyPI is the most important Python repo in the world. If we modify the classic saying a little, we can state that "all Python roads lead to PyPI", and that's no exaggeration at all.

The PyPI repo: the Cheese Shop

The PyPI repo is sometimes referred to as the Cheese Shop. Really. Does that sound a little strange to you? Don't worry, it's all perfectly innocent. We refer to the repo as a shop, because you go there for the same reasons you go to other shops: to fulfill your needs. If you want some cheese, you go to the cheese shop. If you want a piece of software, you go to the software shop. Fortunately, the analogy ends here – you don't need any money to take some software out of the repo shop.

PyPI is completely free, and you can just pick a code and use it – you'll encounter neither cashier nor security guard. Of course, it doesn't absolve you from being polite and honest. You have to obey all the licensing terms, so don't forget to read them.

"Okay", you say, "the shop is clear now, but what does cheese have to do with Python?" The Cheese Shop is one of the most famous Monty Python sketches. It depicts the surrealist adventure of an Englishman trying to buy some cheese. Unfortunately, the shop he visits, immodestly named Ye National Cheese Emporium, has no cheese in stock at all. Of course, it's meant to be ironic. As you already know, PyPI has lots of software in stock and it's available 24/7. It's fully entitled to identify itself as *Ye International Python Software Emporium*.

PyPI is a very specific shop, not just because it offers all its products for free. It also requires a special tool to make use of it. Fortunately, this tool is also free, so if you want to make your own digital cheeseburger by using the goods offered by the PyPI Shop, you'll need a free tool named `pip`.

No, you haven't misheard. Just `pip`. It's another acronym, of course, but its nature is more complex than the previously mentioned PyPI, as it's an example of a recursive acronym, which means that the acronym refers to itself, which means that explaining it is an infinite process. Why? Because `pip` means *"pip installs packages"*, and the `pip` inside *"pip installs packages"* means *"pip installs packages"* and…

Let's stop there. Thank you for your cooperation. By the way, there are a few other very famous recursive acronyms. One of them is *Linux*, which can be interpreted as *"Linux is Not Unix"*.

How to install `pip`

The question that should be put now is: how to get a proper cheese knife? In other words, how to ensure that pip is installed and ready to work? The most precise answer is "it depends". Really. Some Python installations come with pip, some don't. What's more, it doesn't only depend on the OS you use, although this is a very important factor. Let's start with MS Windows.

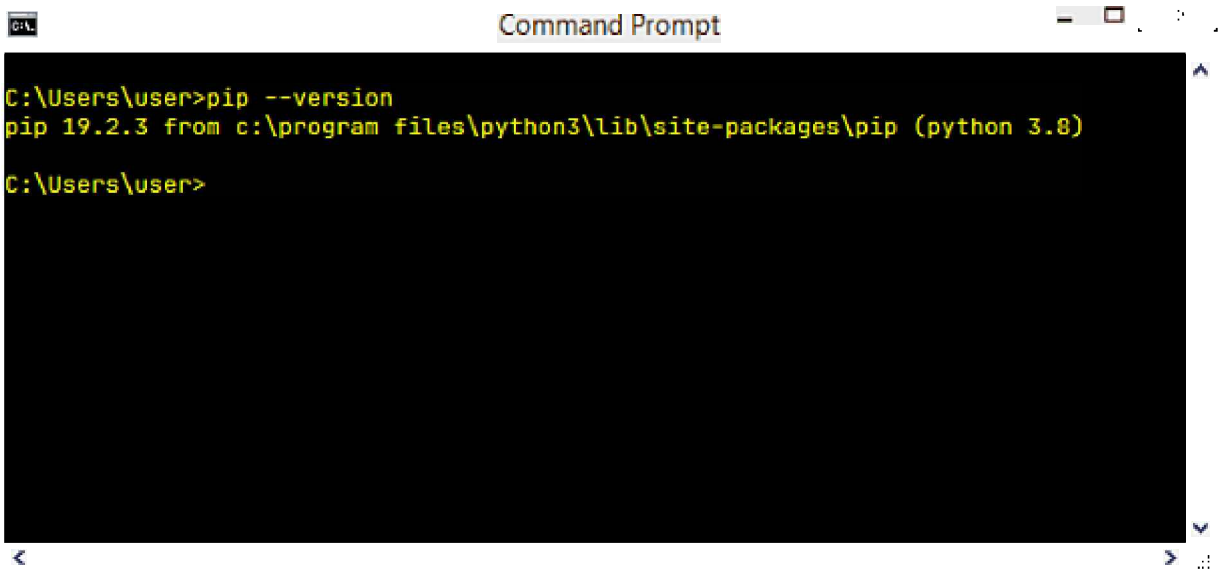
`pip` on MS Windows

The MS Windows Python installer already contains `pip`, and so no other steps need to be taken in order to install it. Unfortunately, if the PATH variable is misconfigured, `pip` may become unavailable. To verify that we haven't misled you, try to do this:

- open the Windows console (*CMD* or *PowerShell*, whatever you prefer)
- execute the following command:

```
pip --version
```

- in the most optimistic scenario (and we really want that to happen) you'll see something like this:



The absence of this message may mean that the PATH variable either incorrectly points to the location of the Python binaries, or doesn't point to it at all; for example, our PATH variable contains the following substring:

```
C:\Program Files\Python3\Scripts\;C:\Program Files\Python3\;
```

The easiest way to reconfigure the PATH variable is to reinstall Python, instructing the installer to set it for you.

`pip` on Linux

Different Linux distributions may behave differently when it comes to using `pip`. Some of them (like *Gentoo*), which are closely bound to Python and which use it internally, may have `pip` preinstalled and are instantly ready to work.

Don't forget that some Linuces may utilize more than one Python version concurrently, e.g. one Python 2 and one Python 3 coexisting side by side. Such systems may launch Python 2 as the default version, and it may be necessary to explicitly specify the program name as python3. In this case there may be two different pips identified as `pip` (or `pip2`) and `pip3`. Check it carefully.

Open the terminal window and issue the following command:

```
pip --version
```

```
user@host ~ $ pip --version
pip 20.0.2 from /usr/lib64/python2.7/site-packages/pip (python 2.7)
user@host ~ $
```

An answer similar to the one shown in the previous picture determines that you've launched `pip` from Python 2, so the next try should look as follows:

```
pip3 --version
```

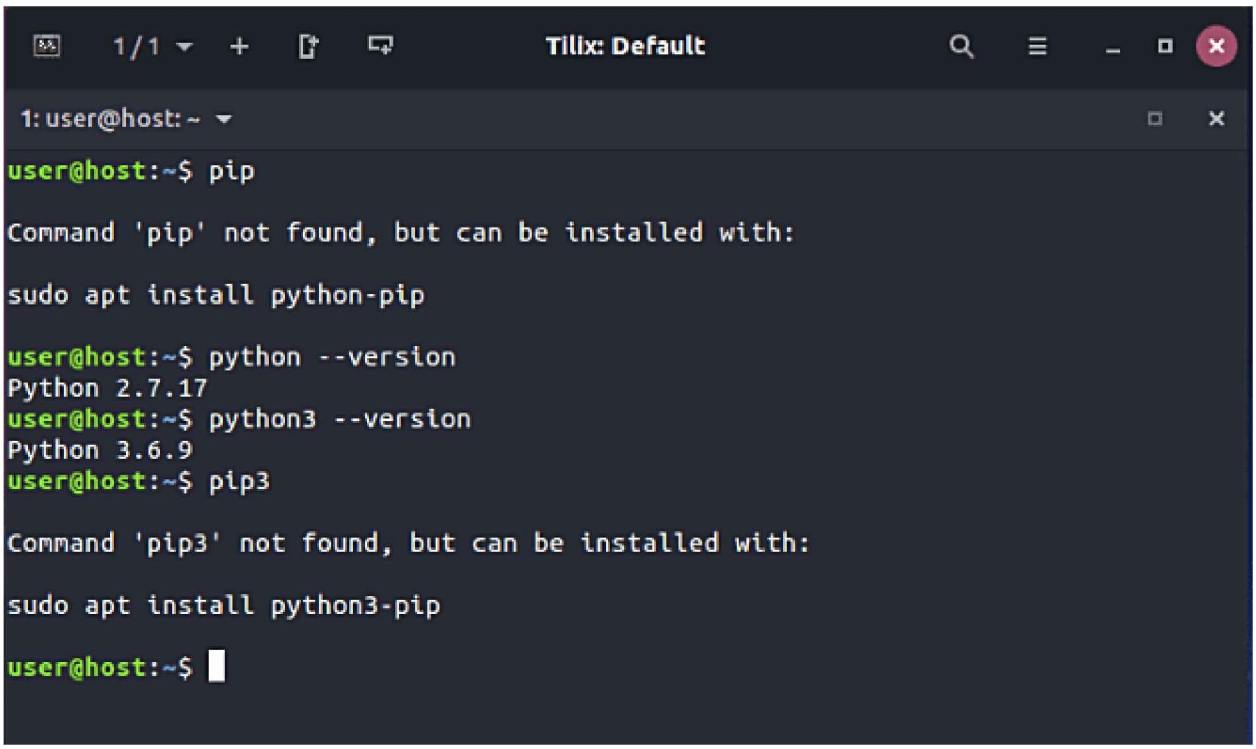
```
user@host ~ $ pip3 --version
pip 20.0.2 from /usr/lib64/python3.6/site-packages/pip (python 3.6)
user@host ~ $
```

As you can see, we're now sure that we're using the appropriate version of `pip`.

Unfortunately, some Linux distributions don't have `pip` preinstalled, even if Python itself is installed by default. Some versions of Ubuntu may behave this way. In this case, you have two possibilities: either install `pip` as a system package using a dedicated package manager (e.g. `apt` in Debian-like systems) or install pip using internal Python mechanisms.

The former is definitely better. Although there are some smart scripts that are able to download and install `pip` by ignoring the OS, we discourage you from using them. This method can get you into trouble.

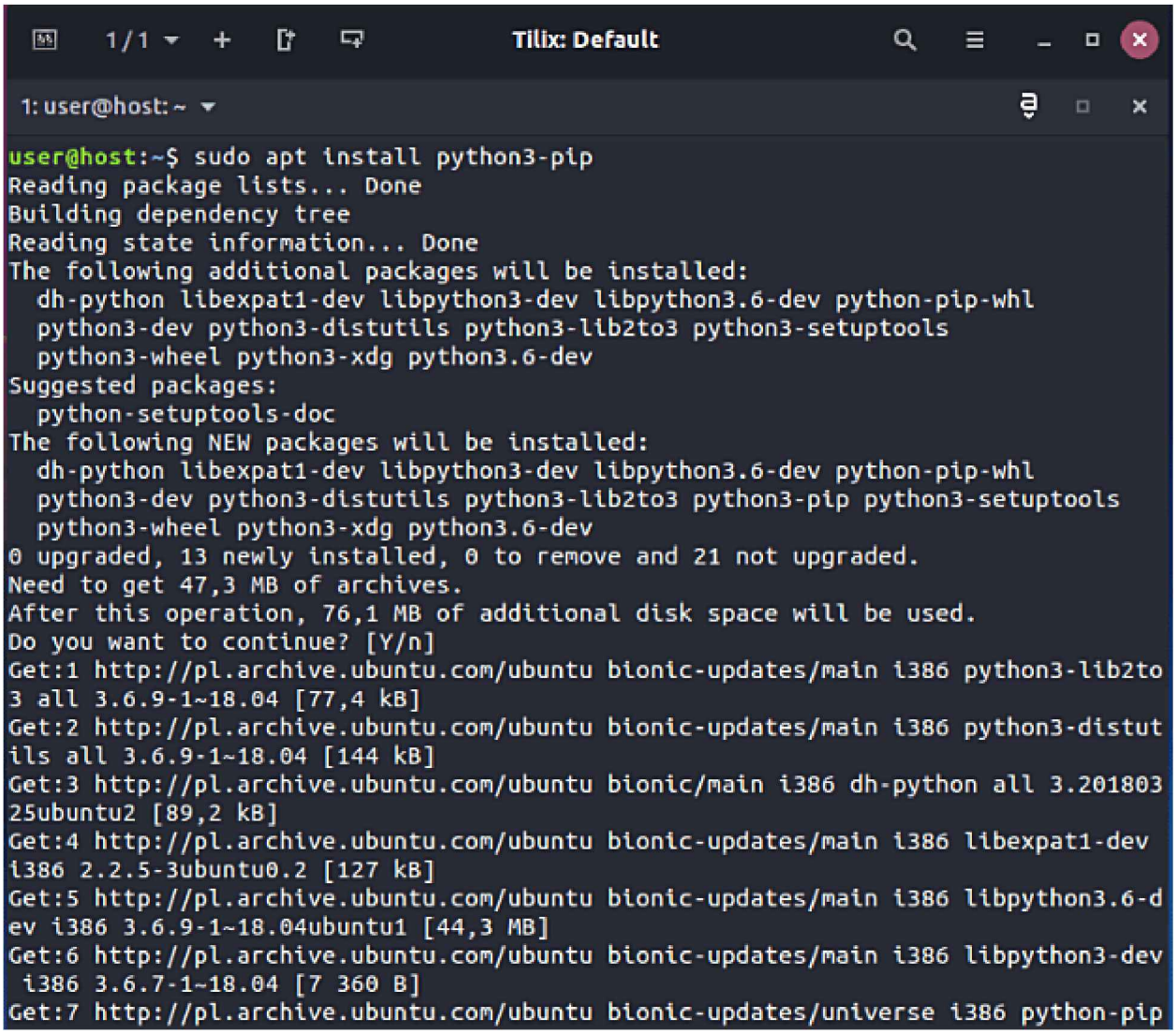
Look – we've tried to launch `pip3` and we've failed. Our OS (we used *Ubuntu Budgie* this time) suggested using `apt` in order to install the package named `python3-pip`:



That's good advice, and we're going to follow it, but it has to be stated that we'll need administrative rights to do it. Don't forget that different Linuces may use different package managers. For example, it could be `pacman` if you use Arch Linux, or `yum` used by distributions derived from Red Hat.

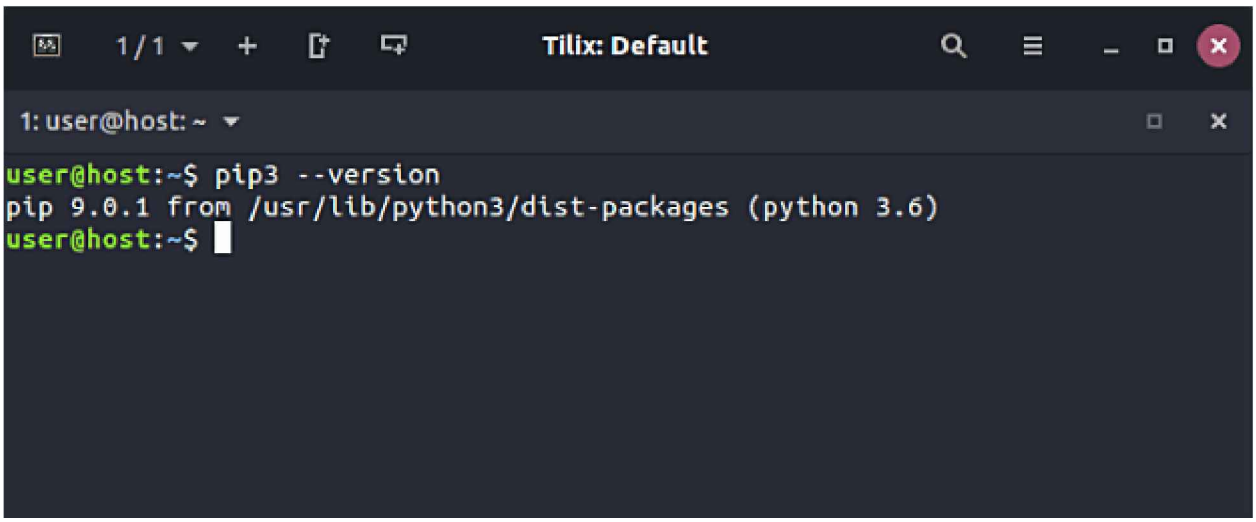
Anyway, all these methods should get `pip` (or `pip3`) installed and working.

Look what happened when we followed the OS suggestion:



As you can see, the OS decided to install not only `pip` itself, but also a couple of additional components needed by `pip`. This is normal – don't be alarmed.

When `apt` finishes its job, we are finally able to utilize `pip3`:



If you're a Mac user and you've installed Python 3 using the *brew* installer, `pip` is already present in your system and ready to work. Check it by issuing the previously mentioned command:

```
1  pip3 --version
2
```

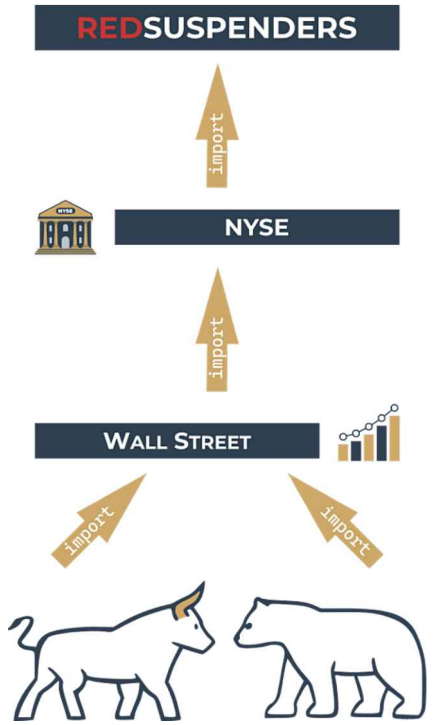
and wait for the response.

This is what we saw:

Dependencies

Now that we're sure that `pip` is ready at our command, we're going to limit our focus to MS Windows only, as its behavior is, or should be, the same in all OSs, but before we start, we need to explain an important issue and tell you about dependencies.

Imagine that you've created a brilliant Python application named *redsuspenders*, able to predict stock exchange rates with 99% accuracy. By the way, if you actually do that, please contact us immediately. Of course, you've used some existing code to achieve this goal — for example, your app imports a package named `nyse` containing some crucial functions and classes. Moreover, the `nyse` package imports another package named `wallstreet`, while the `wallstreet` package imports other two essential packages named `bull` and `bear`.



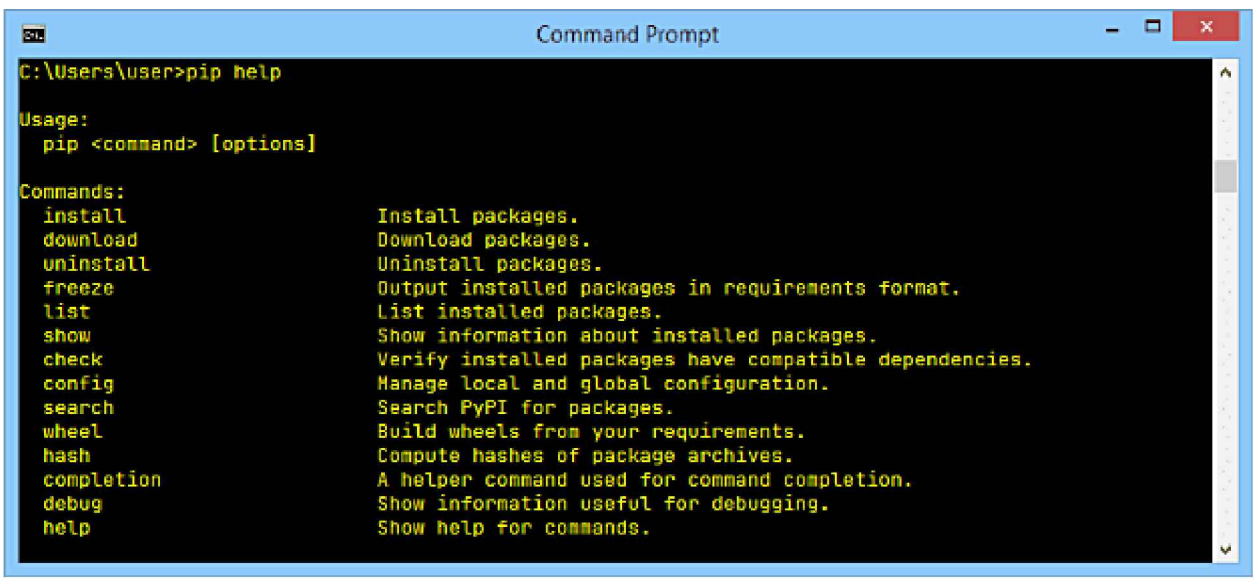
As you've probably already guessed, the connections between these packages are crucial, and if somebody decides to use your code (but remember, we've already called dibs on it) they will also have to ensure that all required packages are in place. To make a long story short, we can say that dependency is a phenomenon that appears every time you're going to use a piece of software that relies on other software. Note that dependency may include, and generally does include more than one level of software development. Does this mean that a potential `nyse` package user is obliged to trace all dependencies and manually install all the needed packages? That would be horrible, wouldn't it? Yes, it's definitely horrible, so you shouldn't be surprised that the process of arduously fulfilling all the subsequent requirements has its own name, and it's called *dependency hell*. How do we deal with that? Is every user doomed to visit hell in order to run the code for the first time? Fortunately not – `pip` can do all of this for you. It can discover, identify, and resolve all dependencies. Moreover, it can do it in the cleverest way, avoiding any unnecessary downloads and reinstalls.

How to use `pip`

Now we're ready to ask pip what it can do for us. Let's do it – issue the following command:

```
pip help
```

and wait for `pip`'s response. This is what it looks like:



Don't forget that you may be obliged to replace `pip` with `pip3` if your environment requires this. The list produced by `pip` summarizes all the available operations, and the last of them is `help`, which we've just used already. If you want to know more about any of the listed operations, you can use the following form of `pip` invocation:

```
1  pip help operation
2
```

For example, the line:

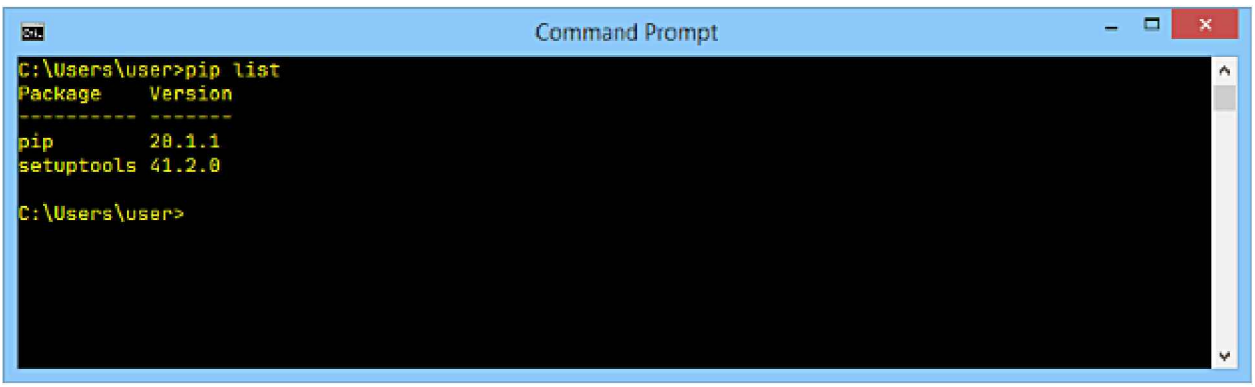
```
1  pip help install
2
```

will show you detailed information about using and parameterizing The `install` command.

If you want to know what Python packages have been installed so far, you can use The `list` operation – just like this:

```
1  pip list
2
```

The output you'll see is rather unpredictable. Don't be surprised if the contents of your screen ends up being completely different. Ours look as follows:



As you can see, there are two columns in the list, one showing the name of the installed package, and the other showing the version of the package. We can't predict the state of your Python installation. The only thing we know for sure is that your list contains the two lines we see on our list: `pip` and `setuptools`. This happens because the OS is convinced that a user wanting pip will very likely need the `setuptools` soon. It's not wrong.

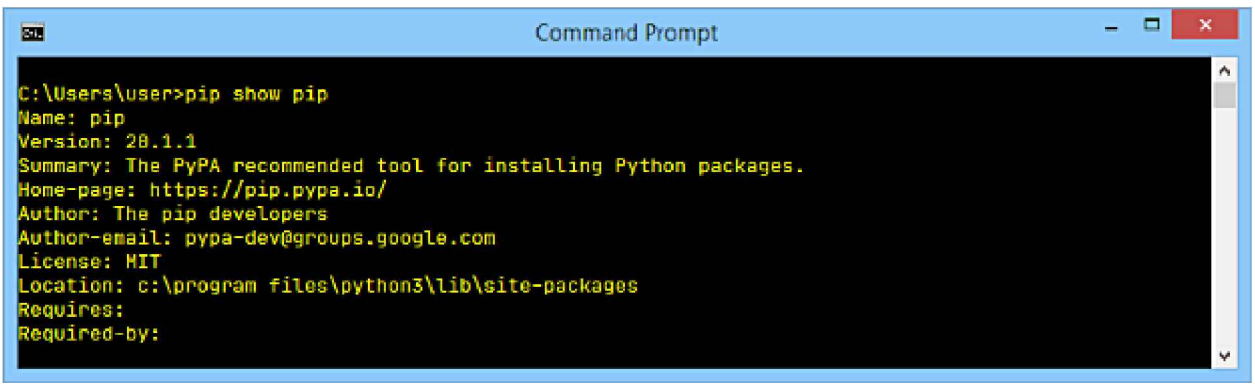
The `pip` list isn't very informative, and it may happen that it won't satisfy your curiosity. Fortunately, there's a command that can tell you more about any of the installed packages — note the word *installed*. The syntax of the command looks as follows:

```
1  pip show package_name
2
```

We're going to use it in a slightly deceptive way – we want to convince `pip` to confess something about itself. This is how we do it:

```
1  pip show pip
2
```

It looks a bit odd, doesn't it? Despite this, it works fine, and `pip`'s self-presentation looks consistent and current:



You may ask where this data comes from? Is `pip` really so perceptive? Not at all – the information appearing on the screen is taken from inside the package being shown. In other words, the package's creator is obliged to equip it with all the needed data, or to express it more precisely, metadata. Look at the two lines at the bottom of the output. They show which packages are needed to successfully utilize the package (`Requires:`) and which packages need the package to be successfully utilized (`Required-by:`)

As you can see, both properties are empty. Feel free to try to use the `show` command in relation to any other installed package. The power of `pip` comes from the fact that it's actually a gateway to the Python software universe. Thanks to that, you can browse and install any of the hundreds of ready-to-use packages gathered in the PyPI repositories. Don't forget that `pip` is not able to store all PyPI content locally. In fact, it's unnecessary and it would be uneconomical.

In effect, `pip` uses the Internet to query PyPI and to download the required data. This means that you have to have a network connection working whenever you're going to ask `pip` for anything that may involve direct interactions with the PyPI infrastructure. One of these cases occurs when you want to search through PyPI in order to find a desired package. This kind of search is initiated by the following command:

```
1  pip search anystring
2
```

The `anystring` you provide will be searched in the names of all the packages and the summary strings of all the packages. Be aware of the fact that some searches may generate a real avalanche of data, so try to be as specific as possible. For example, an innocent-looking query like this one:

```
1  pip search pip
2
```

produces more than 100 lines of results. Try it yourself – don't take our word for it. By the way, the search is case insensitive. If you're not a fan of console reading, you can use the alternative way of browsing PyPI content offered by a search engine, available at https://pypi.org/search.

Assuming that your search is successful, you can use pip to install the package onto your computer. Two possible scenarios may be put into action now: firstly, you want to install a new package for you only – it won't be available for any other user account existing on your computer; this procedure is the only one available if you can't elevate your permissions and act as a system administrator; or you've decided to install a new package system-wide – you have administrative rights and you're not afraid to use them. To distinguish between these two actions, pip uses a dedicated option named `--user` (note the double dash). The presence of this option instructs pip to act locally on behalf of your (non-administrative) user. If you don't add this, pip assumes that you're a system administrator and it'll do nothing to correct you if you're not.

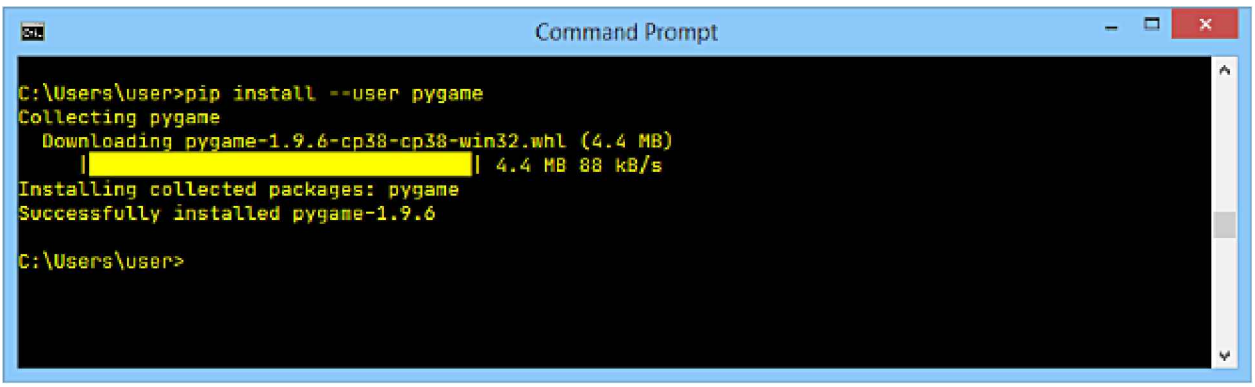
In our case, we're going to install a package named pygame – it's an extended and complex library allowing programmers to develop computer games using Python. The project has been in development since the year 2000, so it's a mature and reliable piece of code. If you want to know more about the project and about the community which leads it, visit https://www.pygame.org. If you're a system administrator, you can install pygame using the following command:

```
1  pip install pygame
2
```

If you're not an admin, or you don't want to fatten up your OS by installing pygame system-wide, you can install it for you only:

```
1  pip install --user pygame
2
```

It's up to you which of these procedures you want to take place. Pip has a habit of displaying fancy textual animation indicating the installation progress, so watch the screen carefully – don't miss the show! If the process is successful, you'll see something like this:



We encourage you to use:

```
1  pip show pygame
2
```

and

```
1  pip list
2
```

to get more information about what actually happened.

A simple test program

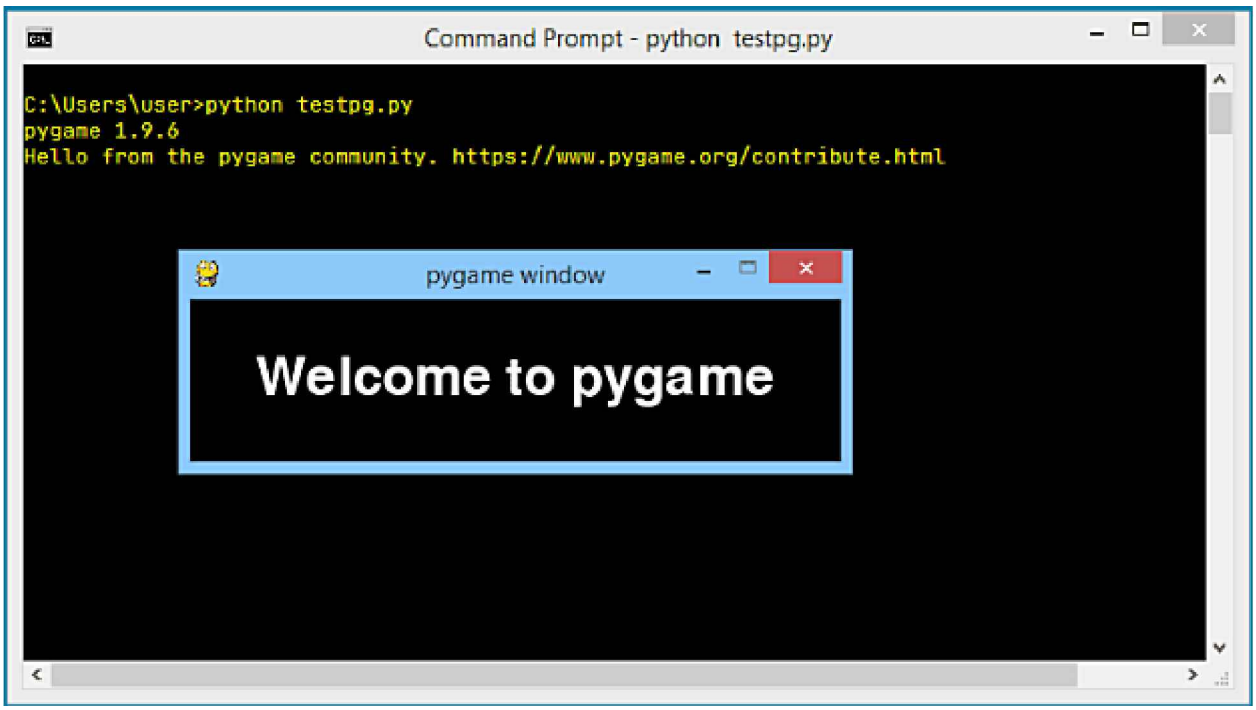
Now that `pygame` is finally accessible, we can try to use it in a very simple test program:

```
 1  import pygame
 2
 3  run = True
 4  width = 400
 5  height = 100
 6  pygame.init()
 7  screen = pygame.display.set_mode((width, height))
 8  font = pygame.font.SysFont(None, 48)
 9  text = font.render("Welcome to pygame", True, (255, 255, 255))
10  screen.blit(text, ((width - text.get_width()) // 2, (height - text.get_height())
    // 2))
11  pygame.display.flip()
12  while run:
13   in pygame.event.get():
14  type == pygame.QUIT\
15  type == pygame.MOUSEBUTTONUP\
16  type == pygame.KEYUP:
17  False
18
```

Let's comment on it briefly.

- line 1: import `pygame` and let it serve us;
- line 3: the program will run as long as The `run` variable is `True`;
- lines 4 and 5: determine the window's size;
- line 6: initialize The `pygame` environment;
- line 7: prepare the application window and set its size;
- line 8: make an object representing the default font of size 48 points;
- line 9: make an object representing a given text – the text will be anti-aliased (`True`) and white (`255,255,255`)
- line 10: insert the text into the (currently invisible) screen buffer;
- line 11: flip the screen buffers to make the text visible;
- line 12: The `pygame` main loop starts here;
- line 13: get a list of all pending `pygame` events;
- lines 14 through 16: check whether the user has closed the window or clicked somewhere inside it or pressed any key;
- line 15: if yes, stop executing the code.

This is what we expect from our impressive code:



The `pip install` has two important additional abilities. It is able to update a locally installed package – for example, if you want to make sure that you're using the latest version of a particular package, you can run the following command:

```
pip install -U package_name
```

where `-U` means update.

NOTE: this form of the command makes use of The `--user` option for the same purpose as presented previously:

Its second ability is to install a user-selected version of a package. `pip` installs the newest available version by default. To achieve this goal you should use the following syntax:

```
pip install package_name==package_version
```

Note the double equals sign:

```
pip install pygame==1.9.2
```

If any of the currently installed packages are **no longer needed** and you want to get rid of them, `pip` will be useful, too. Its `uninstall` command will execute all the needed steps.

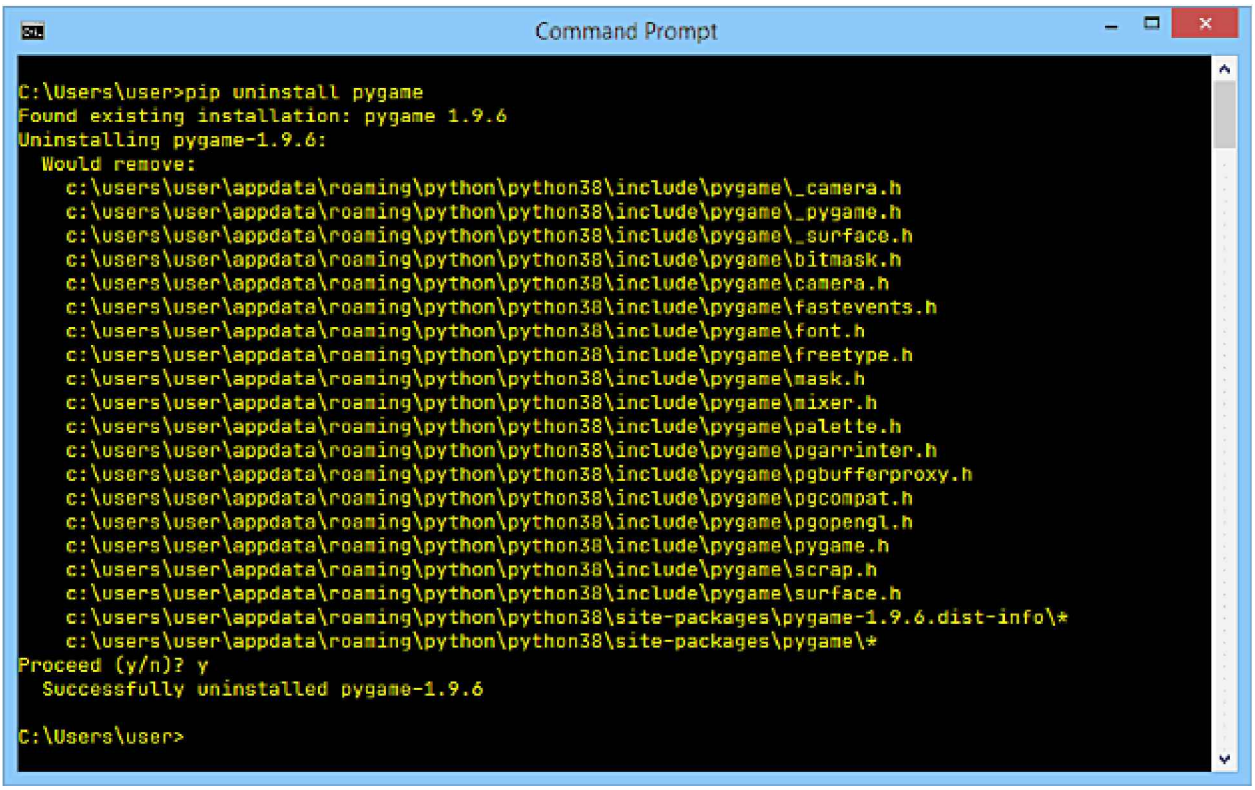
The required syntax is clear and simple:

```
pip uninstall package_name
```

so if you don't want `pygame` anymore, you can execute the following command:

```
pip uninstall pygame
```

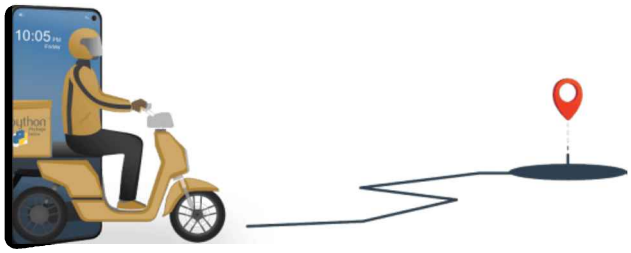
`Pip` will want to know if you're sure about the choice you're making – be prepared to give the right answer. The process looks like this:



Use `pip`!

`Pip`'s capabilities don't end here, but the command set we've presented to you is enough to start successfully managing packages that aren't a part of the regular Python installation.

We hope we've encouraged you to carry out your own experiments with `pip` and the Python package universe. PyPI invites you to dive into its extensive resources. Some say that one of the most important programming virtues is laziness. Don't get us wrong – we don't want you to spend all day napping on the couch and dreaming of Python code. A lazy programmer is a programmer who looks for existing solutions and analyzes the available code before they start to develop their own software from scratch. This is why PyPI and `pip` exist – use them!



Summary

1. A repository (or `repo` for short) designed to collect and share free Python code exists and works under the name Python Package Index (PyPI) although it's also likely that you come across the very niche name the Cheese Shop. The Shop's website is available at https://pypi.org/.

2. To make use of the Cheese Shop, a specialized tool has been created and its name is pip (*pipinstallspackages* while `pip` stands for... ok, never mind). As pip may not be deployed as a part of the standard Python installation, it is possible that you will need to install it manually. Pip is a console tool.

3. To check pip's version one the following commands should be issued:

```
pip --version
```

or

```
pip3 --version
```

Check yourself which of these works for you in your OS's environment.

4. The list of the main pip activities looks as follows:

- `pip help operation` – shows a brief description of `pip`;
- `pip list` – shows a list of the currently installed packages;
- `pip show package_name` – shows `package_name` info including the package's dependencies;
- `pip search anystring` – searches through PyPI directories in order to find packages whose names contain `anystring`;
- `pip install name` – installs `name` system-wide (expect problems when you don't have administrative rights);
- `pip install --user name` – installs `name` for you only; no other platform user will be able to use it;
- `pip install -U name` – updates a previously installed package;
- `pip uninstall name` – uninstalls a previously installed package.

Quiz

Question 1: Where does the name "The Cheese Shop" come from?

Question 2: Why should I ensure which one of `pip` and *pip3* works for me?

Question 3: How can I determine if my `pip` works with either Python 2 or Python 3?

Question 4: Unfortunately, I don't have administrative right. What should I do to install a package system-wide?

Check Answers

PART 2: STRINGS, STRING AND LIST METHODS, EXCEPTIONS

FIVE – CHARACTERS AND STRINGS VS. COMPUTERS

You've written some interesting programs since you've started this course, but all of them have processed only one kind of data – numbers. As you know, and you can see this all around you, lots of computer data are not numbers: first names, last names, addresses, titles, poems, scientific papers, emails, court judgements, declarations of love, and much, much more. All these data must be stored, input, output, searched, and transformed by contemporary computers just like any other data, no matter if they are single characters or multivolume encyclopedias. How is it possible? How can you do it in Python? This is what we'll discuss now. Let's start with how computers understand single characters.

Computers store characters as numbers. Every character used by a computer corresponds to a unique number, and vice versa. This assignment must include more characters than you might expect. Many of them are invisible to humans, but essential to computers. Some of these characters are called whitespaces, while others are named control characters, because their purpose is to control input/output devices. An example of a whitespace that is completely invisible to the naked eye is a special code, or a pair of codes, which are used to mark the ends of the lines inside text files However, different operating systems may treat this issue differently. People do not see this sign, but are able to observe the effect of their application where the lines are broken.

We can create virtually any number of character-number assignments, but life in a world in which every type of computer used a different character encoding would not be very convenient. This system has led to a need to introduce a universal and widely accepted standard implemented by almost all computers and operating systems all over the world.

The one named ASCII (short for American Standard Code for Information Interchange) is the most widely used, and you can assume that nearly all modern devices use that code , including computers, printers, mobile phones, tablets, etc.. The code provides space for 256 different characters, but we are interested only in the first 128. If you want to see how the code is constructed, look at the following table. Look at it carefully – there are some interesting facts. Look at the code of the most common character – the space. This is 32.

CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE
(NUL)	0	(space)	32	@	64	`	96
(SOH)	1	!	33	A	65	a	97
(STX)	2	"	34	B	66	b	98
(ETX)	3	#	35	C	67	c	99
(EOT)	4	$	36	D	68	d	100
(ENQ)	5	%	37	E	69	e	101
(ACK)	6	&	38	F	70	f	102
(BEL)	7	'	39	G	71	g	103
(BS)	8	(	40	H	72	h	104
(HT)	9	)	41	I	73	i	105
(LF)	10	*	42	J	74	j	106
(VT)	11	+	43	K	75	k	107
(FF)	12	,	44	L	76	l	108
(CR)	13	-	45	M	77	m	109
(SO)	14	.	46	N	78	n	110
(SI)	15	/	47	O	79	o	111
(DLE)	16	0	48	P	80	p	112
(DC1)	17	1	49	Q	81	q	113
(DC2)	18	2	50	R	82	r	114
(DC3)	19	3	51	S	83	s	115
(DC4)	20	4	52	T	84	t	116
(NAK)	21	5	23	U	85	u	117
(SYN)	22	6	54	V	86	v	118
(ETB)	23	7	55	W	87	w	119

(CAN)	24	8	56	X	88	x	120
(EM)	25	9	57	Y	89	y	121
(SUB)	26	:	58	Z	90	z	122
(ESC)	27	;	59	[	91	{	123
(FS)	28	<	60	\	92	|	124
(GS)	29	=	61	]	93	}	125
(RS)	30	>	62	^	94	~	126
(US)	31	?	63	_	95		127

Now check the code of the lower-case letter *a*. This is *97*. And now find the upper-case *A*. Its code is *65*. Now work out the difference between the code of *a* and *A*. It is equal to *32*. That's the code of a *space*. Interesting, isn't it? Also note that the letters are arranged in the same order as in the Latin alphabet.

I18N

Of course, the Latin alphabet is not sufficient for the whole of mankind. Users of that alphabet are in the minority. It was necessary to come up with something more flexible and capacious than ASCII, something able to make all the software in the world amenable to internationalization, because different languages use completely different alphabets, and sometimes these alphabets are not as simple as the Latin one.

The word *internationalization* is commonly shortened to I18N. Why? Look carefully – there is an *I* at the front of the word, next there are *18* different letters, and an *N* at the end. Despite the slightly humorous origin, the term is officially used in many documents and standards.

The software I18N is a standard in present times. Each program has to be written in a way that enables it to be used all around the world, among different cultures, languages, and alphabets. A classic form of ASCII code uses eight bits for each sign. Eight bits mean 256 different characters. The first 128 are used for the standard Latin alphabet (both upper-case and lower-case characters). Is it possible to push all the other national characters used around the world into the remaining 128 locations? No. It isn't.

Code points and code pages

We need a new term now: a code point. A code point is a number which makes a character. For example, *32* is a code point which makes a *space* in ASCII encoding. We can say that standard ASCII code consists of 128 code points. As standard ASCII occupies 128 out of 256 possible code points, you can only make use of the remaining 128. It's not enough for all possible languages, but it may be sufficient for one language, or for a small group of similar languages. Can you set the higher half of the code points differently for different languages? Yes, you can. Such a concept is called a code page.

A code page is a standard for using the upper 128 code points to store specific national characters. For example, there are different code pages for Western Europe and Eastern Europe, Cyrillic and Greek alphabets, Arabic and Hebrew languages, and so on. This means that the one and same code point can make different characters when used in different code pages. For example, the code point 200 makes Č (a letter used by some Slavic languages) when utilized by the ISO/IEC 8859-2 code page, and makes Ш (a Cyrillic letter) when used by the ISO/IEC 8859-5 code page.

In consequence, to determine the meaning of a specific code point, you have to know the target code page. In other words, the code points derived from the code page concept are ambiguous.

Unicode

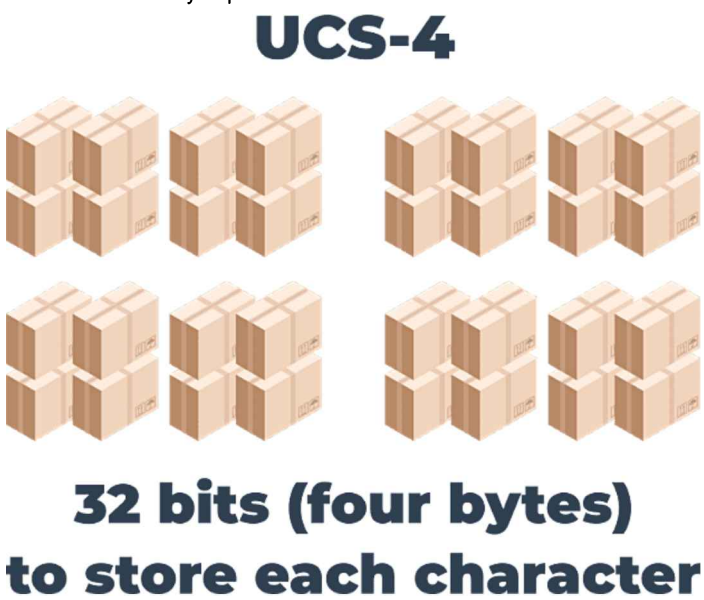
Code pages helped the computer industry to solve I18N issues for some time, but it soon turned out that they would not be a permanent solution. The concept that solved the problem in the long term was Unicode. Unicode assigns unique unambiguous characters (letters, hyphens, ideograms, etc.) to more than a million code points. The first 128 Unicode code points are identical to ASCII, and the first 256 Unicode code points are identical to the ISO/IEC 8859-1 code page, which is a code page designed for western European languages.

UCS-4

The Unicode standard says nothing about how to code and store the characters in the memory and files. It only names all available characters and assigns them to planes — a group of characters of similar origin, application, or nature.

There is more than one standard describing the techniques used to implement Unicode in actual computers and computer storage systems. The most general of them is UCS-4. The name comes from Universal Character Set.

UCS-4 uses 32 bits (four bytes) to store each character, and the code is just the Unicode code points' unique number. A file containing UCS-4 encoded text may start with a BOM (byte order mark), an unprintable combination of bits announcing the nature of the file's contents. Some utilities may require it.



As you can see, UCS-4 is a rather wasteful standard – it increases a text's size by four times compared to standard ASCII. Fortunately, there are smarter forms of encoding Unicode texts.

UTF-8

One of the most commonly used is UTF-8. The name is derived from Unicode Transformation Format. The concept is very smart. UTF-8 uses as many bits for each of the code points as it really needs to represent them. For example, all Latin characters and all standard ASCII characters occupy eight bits, and non-Latin characters occupy 16 bits.

CJK (China-Japan-Korea) ideographs occupy 24 bits. Due to features of the method used by UTF-8 to store the code points, there is no need to use the BOM, but some of the tools look for it when reading the file, and many editors set it up during the save. Python 3 fully supports Unicode and UTF-8, so you can use Unicode/UTF-8 encoded characters to name variables and other entities, and you can use them during all input and output. This means that Python3 is completely I18Ned.

Summary

1. Computers store characters as numbers. There is more than one possible way of encoding characters, but only some of them gained worldwide popularity and are commonly used in IT: these are ASCII (used mainly to encode the Latin alphabet and some of its derivates) and UNICODE (able to encode virtually all alphabets being used by humans).

2. A number corresponding to a particular character is called a codepoint.

3. UNICODE uses different ways of encoding when it comes to storing the characters using files or computer memory: two of them are UCS-4 and UTF-8. The latter is the most common as it wastes less memory space.

Quiz

Question 1: What is BOM?

Question 2: Is Python 3 I18Ned?

Check Answers

SIX – THE NATURE OF STRINGS IN PYTHON

Let's do a brief review of the nature of Python's strings. First of all, Python's strings are immutable sequences. It's very important to note this, because it means that you should expect some familiar behavior from them. Let's analyze the following code to understand what we're talking about:

Take a look at Example 1. The `len()` function used for strings returns a number of characters contained by the arguments. The snippet outputs 2. Any string can be empty. Its length is 0 then – just like in Example 2. Don't forget that a backslash (\) used as an escape character is not included in the string's total length. The code in `Example 3`, therefore, outputs 3. Run the three example codes and check.

```
# Example 1

word = 'by'
print(len(word))


# Example 2

empty = ''
print(len(empty))


# Example 3

i_am = 'I\'m'
print(len(i_am))

```

Multiline strings

Now is a very good moment to show you another way of specifying strings inside the Python source code. Note that the syntax you already know won't let you use a string occupying more than one line of text. For this reason, the code here is erroneous:

```
multiline = 'Line #1
Line #2'


print(len(multiline))
```

Fortunately, for these kinds of strings, Python offers separate, convenient, and simple syntax. Look at the following code. This is what it looks like.

```
multiline = '''Line #1
Line #2'''

print(len(multiline))

```

As you can see, the string starts with three apostrophes, not one. The same tripled apostrophe is used to terminate it. The number of text lines put inside such a string is arbitrary. The snippet outputs 15. Count the characters carefully. Is this result correct or not? It looks okay at first glance, but when you count the characters, it doesn't. `Line #1` contains seven characters. Two such lines comprise 14 characters. Did we lose a character? Where? How? No, we didn't. The missing character is simply invisible – it's a whitespace. It's located between the two text lines. It's denoted as: `\n`.

Do you remember? It's a special control character used to force a line feed (hence its name: LF). You can't see it, but it counts. Multiline strings can be delimited by triple quotes, too, just like here:

```
multiline = """Line #1
Line #2"""

print(len(multiline))

```

Choose the method that is more comfortable for you. Both work the same.

Operations on strings

Like other kinds of data, strings have their own set of permissible operations, although they're rather limited compared to numbers. In general, strings can be concatenated (joined) or replicated.
The first operation is performed by the `+` operator (note: it's not an addition) while the second by the `*` operator (note again: it's not a multiplication). The ability to use the same operator against completely different kinds of data, like numbers vs. strings, is called overloading, as such an operator is overloaded with different duties.

Analyze the example:

```
1  str1 = 'a'
2  str2 = 'b'
3
4  print(str1 + str2)
5  print(str2 + str1)
6  print(5 * 'a')
7  print('b' * 4)
8
```

The + operator used against two or more strings produces a new string containing all the characters from its arguments. Note thatthe order matters – this overloaded +, in contrast to its numerical version, is not commutative. The * operator needs a string and a number as arguments; in this case, the order doesn't matter – you can put the number before the string, or vice versa, the result will be the same – a new string created by the nth replication of the argument's string. The snippet produces the following output:

```
ab
ba
aaaaa
bbbb
```

NOTE: Shortcut variants of these operators are also applicable for strings (+= and *=).

ord()

If you want to know a specific character's ASCII/UNICODE code point value, you can use a function named `ord()` (as in `ordinal`). The function needs a one-character string as its argument – breaching this requirement causes a `TypeError` exception, and returns a number representing the argument's code point. Look at the following code, and run it.

```
1  # Demonstrating the ord() function.
2
3  char_1 = 'a'
4  char_2 = ' '  # space
5
6  print(ord(char_1))
7  print(ord(char_2))
8
```

The snippet outputs:

```
97
32
```

Now assign different values to `char_1` and `char_2`, e.g., α (Greek alpha), and ę (a letter in the Polish alphabet); then run the code and see what result it outputs. Carry out your own experiments.

chr()

If you know the code point (number) and want to get the corresponding character, you can use a function named `chr()`. The function takes a code point and returns its character. Invoking it with an invalid argument (e.g. a negative or invalid code point) causes `ValueError` or `TypeError` exceptions.

Run the following code.

```
1  # Demonstrating the chr() function.
2
3  print(chr(97))
4  print(chr(945))
5
```

The example snippet outputs:

```
a
α
```

NOTE

- `chr(ord(x)) == x`
- `ord(chr(x)) == x`

Again, run your own experiments.

Strings as sequences

Indexing

We told you before that Python strings are sequences. It's time to show you what that actually means. Strings aren't lists, but you can treat them like lists in many particular cases. For example, if you want to access any of a string's characters, you can do it using indexing, just like in the following example. Run the program:

```
1  # Indexing strings.
2
3  the_string = 'silly walks'
4
5  for ix in range(len(the_string)):
6      print(the_string[ix], end=' ')
7
8  print()
9
```

Be careful – don't try to pass a string's boundaries – it will cause an exception. The example output is:

```
s i l l y   w a l k s
```

By the way, negative indices behave as expected, too. Check this yourself.

Iterating

Iterating through the strings works, too. Look at the following example:

```
1  # Iterating through a string.
2
3  the_string = 'silly walks'
4
5  for character in the_string:
6      print(character, end=   )
7
8  print()
9
```

The output is the same as previously.

Slices

Moreover, everything you know about slices is still usable. We've gathered some examples showing how slices work in the string world. Look at the following code, analyze it, and run it.

```
1  # Slices
2
3  alpha = "abdefg"
4
5  print(alpha[1:3])
6  print(alpha[3:])
7  print(alpha[:3])
8  print(alpha[3:-2])
9  print(alpha[-3:4])
10 print(alpha[::2])
11 print(alpha[1::2])
12
```

You won't see anything new in the example, but we want you to be sure that you can explain all the lines of the code. The code's output is:

```
bd
efg
abd
e
e
```

```
adf
beg
```

Now do your own experiments.

The `in` and `not in` operators

The `in` operator

The `in` operator shouldn't surprise you when applied to strings – it simply checks if its left argument (a string) can be found anywhere within the right argument (another string). The result of the check is simply `True` or `False`. Look at the following example program. This is how the `in` operator works:

```
1  alphabet = "abcdefghijklmnopqrstuvwxyz"
2
3  print("f" in alphabet)
4  print("F" in alphabet)
5  print("1" in alphabet)
6  print("ghi" in alphabet)
7  print("Xyz" in alphabet)
8
```

The example output is:

```
True
False
False
True
False
```

The `not in` operator

As you probably suspect, the `not in` operator is also applicable here. This is how it works:

```
1  alphabet = "abcdefghijklmnopqrstuvwxyz"
2
3  print("f" not in alphabet)
4  print("F" not in alphabet)
5  print("1" not in alphabet)
6  print("ghi" not in alphabet)
7  print("Xyz" not in alphabet)
8
```

The example output is:

```
False
True
True
False
True
```

Python strings are immutable

We've also told you that Python's strings are immutable. This is a very important feature. This primarily means that the similarity of strings and lists is limited. Not everything you can do with a list may be done with a string. The first important difference doesn't allow you to use the `del` instruction to remove anything from a string. The example here won't work:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
del alphabet[0]
```

The only thing you can do with `del` and a string is to remove the string as a whole. Python strings don't have the `append()` method – you cannot expand them in any way. The following example is erroneous:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
alphabet.append("A")
```

With the absence of the `append()` method, the `insert()` method is illegal, too:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
```

A string's immutability doesn't limit your ability to operate with strings. The only consequence is that you have to remember about it, and implement your code in a slightly different way – look at the following example code.

```
1  alphabet = "bcdefghijklmnopqrstuvwxy"
2
3  alphabet = "a" + alphabet
4  alphabet = alphabet + "z"
5
6  print(alphabet)
7
```

This form of code is fully acceptable, will work without bending Python's rules, and will bring the full Latin alphabet to your screen:

```
abcdefghijklmnopqrstuvwxyz
```

You may want to ask if creating a new copy of a string each time you modify its contents worsens the effectiveness of the code. Yes, it does. A bit. It's not a problem at all, though.

Operations on strings: continued

`min()`

Now that you understand that strings are sequences, we can show you some less obvious sequence capabilities. We'll present them using strings, but don't forget that lists can adopt the same tricks, too. Let's start with a function named `min()`. The function finds the minimum element of the sequence passed as an argument. There is one condition: the sequence — string or list, it doesn't matter — cannot be empty, or else you'll get a `ValueError` exception.

```
1  # Demonstrating min() - Example 1:
2  print(min("aAbByYzZ"))
3
4
5  # Demonstrating min() - Examples 2 & 3:
6  t = 'The Knights Who Say "Ni!"'
7  print('[' + min(t) + ']')
8
9  t = [0, 1, 2]
10 print(min(t))
11
```

The Example 1 program outputs:

```
A
```

NOTE: It's an upper-case A. Why? Recall the ASCII table? Which letters occupy first locations – upper or lower?
We've prepared two more examples to analyze: Examples 2 & 3. As you can see, they present more than just strings. The expected output looks as follows:

```
[ ]
0
```

NOTE: We've used the square brackets to prevent the space from being overlooked on your screen.

`max()`

Similarly, a function named `max()` finds the maximum element of the sequence. Look at this Example 1:

```
# Demonstrating max() - Example 1:
print(max("aAbByYzZ"))


# Demonstrating max() - Examples 2 & 3:
t = 'The Knights Who Say "Ni!"'
print('[' + max(t) + ']')

t = [0, 1, 2]
print(max(t))

```

NOTE: It's a lower-case z.

Now let's see The `max()` function applied to the same data as previously. Look at Examples 2 & 3. The expected output is:

```
[y]
2
```

The `index()` method

The `index()` method (it's a method, not a function) searches the sequence from the beginning, in order to find the first element of the value specified in its argument.

NOTE: The element searched for must occur in the sequence – its absence will cause a ValueError exception.

The method returns the index of the first occurrence of the argument, which means that the lowest possible result is `0`, while the highest is the length of the argument decremented by `1`.

```
# Demonstrating the index() method:
print("aAbByYzZaA".index("b"))
print("aAbByYzZaA".index("Z"))
print("aAbByYzZaA".index("A"))

```

Therefore, the example outputs:

```
2
7
1
```

The `list()` function

The `list()` function takes its argument (a string) and creates a new list containing all the string's characters, one per list element.

NOTE: It's not strictly a string function – `list()` is able to create a new list from many other entities (e.g. from tuples and dictionaries).

Take a look at the code example.

```
# Demonstrating the list() function:
print(list("abcabc"))

```

The example outputs:

```
['a', 'b', 'c', 'a', 'b', 'c']
```

The `count()` method

The `count()` method counts all occurrences of the element inside the sequence. The absence of such elements doesn't cause any problems. Look at the second example. Can you guess its output?

```
# Demonstrating the count() method:
print("abcabc".count("b"))
print('abcabc'.count("d"))

```

It is:

```
2
```

```
0
```

Python strings have a significant number of methods intended exclusively for processing characters. Don't expect them to work with any other collections. The complete list is presented here: https://docs.python.org/3.4/library/stdtypes.html#string-methods. We're going to show you the ones we consider the most useful.

Summary

1. Python strings are immutable sequences and can be indexed, sliced, and iterated like any other sequence, as well as being subject to the `in` and `not in` operators. There are two kinds of strings in Python:

- one-line strings, which cannot cross line boundaries – we denote them using either apostrophes (`'string'`) or quotes (`"string"`)
- multi-line strings, which occupy more than one line of source code, delimited by trigraphs:

```
'''
string
'''
```

or

```
"""
string
"""
```

2. The length of a string is determined by the `len()` function. The escape character (`\`) is not counted. For example:

```
1  print(len("\n\n"))
2
```

Outputs: `2`

3. Strings can be concatenated using the `+` operator, and replicated using the `*` operator. For example:

```
1  asterisk = '*'
2  plus = "+"
3  decoration = (asterisk + plus) * 4 + asterisk
4  print(decoration)
5
```

Outputs: `*********`

4. The pair of functions `chr()` and `ord()` can be used to create a character using its codepoint, and to determine a codepoint corresponding to a character. Both of the following expressions are always true:

```
1  chr(ord(character)) == character
2  ord(chr(codepoint)) == codepoint
3
```

5. Some other functions that can be applied to strings are:

- `list()` – creates a list consisting of all the string's characters;
- `max()` – finds the character with the maximal codepoint;
- `min()` – finds the character with the minimal codepoint.

6. The method named `index()` finds the index of a given substring inside the string.

Quiz

Question 1: What is the length of the following string assuming there is no whitespaces between the quotes?

```
"""
"""
```

Question 2: What is the expected output of the following code?

```
s = 'yesteryears'
the_list = list(s)
print(the_list[3:6])
```

Question 3: What is the expected output of the following code?

```
for ch in "abc":
    print(chr(ord(ch) + 1), end='')
```

Check Answers

SEVEN – STRING METHODS

Let's go through some standard Python string methods. We're going to go through them in alphabetical order – to be honest, any order has as many disadvantages as advantages, so the choice may as well be random. First, let's look at the `capitalize()` method. It does exactly what it says – it creates a new string filled with characters taken from the source string, but it tries to modify them in the following way: if the first character inside the string is a letter (note: the first character is an element with an index equal to `0`, not just the first visible character), it will be converted to upper-case; all remaining letters from the string will be converted to lower-case.

Don't forget that the original string from which the method is invoked is not changed in any way – a string's immutability must be obeyed without reservation. The modified string, capitalized in this case, is returned as a result – if you don't use it in any way, that is, if you don't assign it to a variable, or pass it to a function/method, then it will disappear without a trace.

NOTE: Methods don't have to be invoked from within variables only. They can be invoked directly from within string literals. We're going to use that convention regularly – it will simplify the examples, as the most important aspects will not disappear among unnecessary assignments. Take a look at the following example. Run it.

```
# Demonstrating the capitalize() method:
print('aBcD'.capitalize())

```

This is what it prints:

```
Abcd
```

Try some more advanced examples and test their output:

```
print("Alpha".capitalize())
print('ALPHA'.capitalize())
print(' Alpha'.capitalize())
print('123'.capitalize())
print("αβγδ".capitalize())

```

The `center()` method

The one-parameter variant of the `center()` method makes a copy of the original string, trying to center it inside a field of a specified width. The centering is actually done by adding some spaces before and after the string.

Don't expect this method to demonstrate any sophisticated skills. It's rather simple. The following example uses brackets to clearly show you where the centered string actually begins and terminates.

```
# Demonstrating the center() method:
print('[' + 'alpha'.center(10) + ']')

```

Its output looks as follows:

```
[  alpha  ]
```

If the target field's length is too small to fit the string, the original string is returned. You can see the `center()` method in more examples here:

```
print('[' + 'Beta'.center(2) + ']')
print('[' + 'Beta'.center(4) + ']')
print('[' + 'Beta'.center(6) + ']')

```

Run the previous snippets and check what output they produce.

The two-parameter variant of `center()` makes use of the character from the second argument, instead of a space. Analyze the following example:

```
print('[' + 'gamma'.center(20, '*') + ']')

```

This is why the output now looks like this:

```
[*******gamma********]
```

Carry out more experiments.

The `endswith()` method

The `endswith()` method checks if the given string ends with the specified argument and returns `True` or `False`, depending on the check result.

NOTE: The substring must adhere to the string's last character – it cannot just be located somewhere near the end of the string.

Look at our example, analyze it, and run it.

```
# Demonstrating the endswith() method:
if "epsilon".endswith("on"):
    print("yes")
else:
    print("no")

```

It outputs:

```
yes
```

You should now be able to predict the output of the following snippet:

```
t = "zeta"
print(t.endswith("a"))
print(t.endswith("A"))
print(t.endswith("et"))
print(t.endswith("eta"))

```

Run the code to check your predictions.

The `find()` method

The `find()` method is similar to `index()`, which you already know – it looks for a substring and returns the index of the first occurrence of this substring, but it's safer – it doesn't generate an error for an argument containing a non-existent substring (it returns `-1` then), and it works with strings only – don't try to apply it to any other sequence.

Look at the following code. This is how you can use it.

```
# Demonstrating the find() method:
print("Eta".find("ta"))
print("Eta".find("mma"))

```

The example prints:

```
1
-1
```

NOTE: Don't use `find()` if you only want to check if a single character occurs within a string – the `in` operator will be significantly faster. Here is another example:

```
t = 'theta'
print(t.find('eta'))
print(t.find('et'))
print(t.find('the'))
print(t.find('ha'))

```

Can you predict the output? Run it and check your predictions. If you want to perform the find, not from the string's beginning, but from any position, you can use a two-parameter variant of the `find()` method. Look at the example:

```
print('kappa'.find('a', 2))

```

The second argument specifies the index at which the search will be started. It doesn't have to fit inside the string. Among the two a letters, only the second will be found. Run the snippet and check. You can use the `find()` method to search for all the substring's occurrences, like here:

```
the_text = """A variation of the ordinary lorem ipsum
text has been used in typesetting since the 1960s
or earlier, when it was popularized by advertisements
for Letraset transfer sheets. It was introduced to
the Information Age in the mid-1980s by the Aldus Corporation,
which employed it in graphics and word-processing templates
for its desktop publishing program PageMaker (from Wikipedia)"""

fnd = the_text.find('the')
while fnd != -1:
    print(fnd)
    fnd = the_text.find('the', fnd + 1)

```

The code prints the indices of all occurrences of the article *the*, and its output looks like this:

```
15
80
198
221
238
```

There is also a three-parameter mutation of the `find()` method – the third argument points to the first index which won't be taken into consideration during the search. It's actually the upper limit of the search. Look at the following example:

```
print('kappa'.find('a', 1, 4))
print('kappa'.find('a', 2, 4))

```

The second argument specifies the index at which the search will be started. It doesn't have to fit inside the string. Therefore, the modified example outputs:

```
1
-1
```

(*a* cannot be found within the given search boundaries in the second `print()`.

The `isalnum()` method

The parameterless method named `isalnum()` checks if the string contains only digits or alphabetical characters (letters), and returns `True` or `False` according to the result. Look at the example and run it.

```
# Demonstrating the isalnum() method:
print('lambda30'.isalnum())
print('lambda'.isalnum())
print('30'.isalnum())
print('@'.isalnum())
print('lambda_30'.isalnum())
print(''.isalnum())

```

NOTE: Any string element that is not a digit or a letter causes the method to return `False`. An empty string does, too. The example output is:

```
True
True
True
False
False
False
```

Three more intriguing examples are here:

```
t = 'Six lambdas'
print(t.isalnum())

t = '&Alpha;&beta;&Gamma;&delta;'
print(t.isalnum())

t = '20E1'
print(t.isalnum())

```

Run them and check their output.

Hint: the cause of the first result is a space – it's neither a digit nor a letter.

The `isalpha()` method

The `isalpha()` method is more specialized – it's interested in letters only. Look at this example:

```
# Example 1: Demonstrating the isapha() method:
print("Moooo".isalpha())
print('Mu40'.isalpha())

```

Its output is:

```
True
False
```

The `isdigit()` method

In turn, the `isdigit()` method looks at digits only – anything else produces `False` as the result. Look at this example:

```
# Example 2: Demonstrating the isdigit() method:
print('2018'.isdigit())
print("Year2019".isdigit())

```

Its output is:

```
True
False
```

Carry out more experiments.

The `islower()` method

The `islower()` method is a fussy variant of `isalpha()` – it accepts lower-case letters only. Look at this example:

```
# Example: Demonstrating the islower() method:
print("Moooo".islower())
print('moooo'.islower())

```

It outputs:

```
False
True
```

The `isspace()` method

The `isspace()` method identifies whitespaces only – it disregards any other character (the result is `False` then). Look at this example:

```
# Example: Demonstrating the isspace() method:
print(' \n '.isspace())
print(" ".isspace())
print("mooo mooo mooo".isspace())

```

The output is:

```
True
True
False
```

The `isupper()` method

The `isupper()` method is the upper-case version of `islower()` – it concentrates on upper-case letters only. Again, look at this code:

```
# Example: Demonstrating the isupper() method:
print("Moooo".isupper())
print('moooo'.isupper())
print('MOOOO'.isupper())

```

The Example produces the following output:

```
False
False
True
```

The `join()` method

The `join()` method is rather complicated, so let us guide you through it step by step. As its name suggests, the method performs a join – it expects one argument as a list. It must be assured that all the list's elements are strings – the method will raise a `TypeError` exception otherwise. All the list's elements will be joined into one string but the string from which the method has been invoked is used as a separator, put among the strings. The newly created string is returned as a result.

Take a look at the following example:

```
# Demonstrating the join() method:
print(",".join(["omicron", "pi", "rho"]))

```

Let's analyze it:

- the `join()` method is invoked from within a string containing a comma — the string can be arbitrarily long, or it can be empty;
- the `join`'s argument is a list containing three strings;
- the method returns a new string.

Here it is:

```
omicron,pi,rho
```

The `lower()` method

The `lower()` method makes a copy of a source string, replaces all upper-case letters with their lower-case counterparts, and returns the string as the result. Again, the source string remains untouched. If the string doesn't contain any upper-case characters, the method returns the original string.

NOTE: The `lower()` method doesn't take any parameters.

Look at the example:

```
# Demonstrating the lower() method:
print("SiGmA=60".lower())

```

It outputs:

```
sigma=60
```

As usual, carry out your own experiments.

The `lstrip()` method

The parameterless version of the `lstrip()` method returns a newly created string formed from the original one by removing all leading whitespaces. Analyze the example code.

```
1  # Demonstrating the lstrip() method:
2  print("[" + " tau ".lstrip() + "]")
3
```

The brackets are not a part of the result – they only show the result's boundaries. The example outputs:

```
[tau ]
```

The one-parameter version of the `lstrip()` method does the same as its parameterless version, but removes all characters enlisted in its argument (a string), not just whitespaces.

```
1  print("www.cisco.com".lstrip("w."))
2
```

Brackets aren't needed here, as the output looks as follows:

```
cisco.com
```

```
1  print("pythoninstitute.org".lstrip(".org"))
2
```

Surprised? Leading characters, leading whitespaces. Again, experiment with your own examples.

The `replace()` method

The two-parameter `replace()` method returns a copy of the original string in which all occurrences of the first argument have been replaced by the second argument. Look at the example. Run it.

```
1  # Demonstrating the replace() method:
2  print("www.netacad.com".replace("netacad.com", "pythoninstitute.org"))
3  print("This is it!".replace("is", "are"))
4  print("Apple juice".replace("juice", ""))
5
```

The example outputs:

```
www.pythoninstitute.org
Thare are it!
Apple
```

If the second argument is an empty string, replacing means actually removing the first argument's string. What kind of magic happens if the first argument is an empty string? The three-parameter `replace()` variant uses the third argument, a number, to limit the number of replacements. Look at the following modified example code:

```
1  print("This is it!".replace("is", "are", 1))
2  print("This is it!".replace("is", "are", 2))
3
```

Can you guess its output? Run the code and check your guesses.

The `rfind()` method

The one-, two-, and three-parameter versions of the `rfind()` method do nearly the same things as their counterparts, the ones devoid of the r prefix, but start their searches from the end of the string, not the beginning (hence the prefix `r`, for *right*). Take a look at the example code and try to predict its output. Run the code to check if you were right.

```
# Demonstrating the rfind() method:
print("tau tau tau".rfind("ta"))
print("tau tau tau".rfind("ta", 9))
print("tau tau tau".rfind("ta", 3, 9))

```

The `rstrip()` method

Two variants of the `rstrip()` method do nearly the same as `lstrips`, but affect the opposite side of the string. Look at the code example. Can you guess its output? Run the code to check your guesses.

```
# Demonstrating the rstrip() method:
print("[" + " upsilon ".rstrip() + "]")
print("cisco.com".rstrip(".com"))

```

As usual, we encourage you to experiment with your own examples.

The `split()` method

The `split()` method does what it says – it splits the string and builds a list of all detected substrings. The method assumes that the substrings are delimited by whitespaces – the spaces don't take part in the operation, and aren't copied into the resulting list.

If the string is empty, the resulting list is empty too.

Look at the following code:

```
# Demonstrating the split() method:
print("phi     chi\npsi".split())

```

The example produces the following output:

```
['phi', 'chi', 'psi']
```

NOTE: The reverse operation can be performed by The `join()` method.

The `startswith()` method

The `startswith()` method is a mirror reflection of `endswith()` – it checks if a given string starts with the specified substring. Look at the example:

```
# Demonstrating the startswith() method:
print("omega".startswith("meg"))
print("omega".startswith("om"))

print()

```

This is the result from it:

```
False
True
```

The `strip()` method

The `strip()` method combines the effects caused by `rstrip()` and `lstrip()` – it makes a new string lacking all the leading and trailing whitespaces. Look at the second example:

```
# Demonstrating the strip() method:
print("[" + " aleph ".strip() + "]")

```

This is the result it returns:

```
[aleph]
```

The `swapcase()` method

The `swapcase()` method makes a new string by swapping the cases of all letters within the source string: lower-case characters become upper-case, and vice versa. All other characters remain untouched. Look at this example:

```
# Demonstrating the swapcase() method:
print("I know that I know nothing.".swapcase())

print()

```

Can you guess the output? It won't look good, but you must see it:

```
i KNOW THAT i KNOW NOTHING.
```

The `title()` method

The `title()` method performs a somewhat similar function – it changes every word's first letter to upper-case, turning all other ones to lower-case. Look at the example:

```
# Demonstrating the title() method:
print("I know that I know nothing. Part 1.".title())

print()

```

Can you guess its output? This is the result:

```
I Know That I Know Nothing. Part 1.
```

The `upper()` method

Last but not least, the `upper()` method makes a copy of the source string, replaces all lower-case letters with their upper-case counterparts, and returns the string as the result. Look at the following example:

```
# Demonstrating the upper() method:
print("I know that I know nothing. Part 2.".upper())

```

It outputs:

```
I KNOW THAT I KNOW NOTHING. PART 2.
```

Hooray! We've made it to the end of this chapter. Are you surprised with any of the string methods we've discussed so far? Take a couple of minutes to review them, and let's move on to the next part, where we'll show you what great things we can do with strings.

Summary

1. Some of the methods offered by strings are:

- `capitalize()` – changes the first letter of a string to capital letter;
- `center()` – centers the string inside the field of a known length;
- `count()` – counts the occurrences of a given character;
- `join()` – joins all items of a tuple/list into one string;
- `lower()` – converts all the string's letters into lower-case letters;
- `lstrip()` – removes the white characters from the beginning of the string;
- `replace()` – replaces a given substring with another;
- `rfind()` – finds a substring starting from the end of the string;
- `rstrip()` – removes the trailing white spaces from the end of the string;
- `split()` – splits the string into a substring using a given delimiter;
- `strip()` – removes the leading and trailing white spaces;
- `swapcase()` – swaps the letters' cases (lower to upper and vice versa)
- `title()` – makes the first letter in each word upper-case;
- `upper()` – converts all the string's letter into upper-case letters.

2. String content can be determined using the following methods — all of them return Boolean values:

- `endswith()` – does the string end with a given substring?
- `isalnum()` – does the string consist only of letters and digits?
- `isalpha()` – does the string consist only of letters?
- `islower()` – does the string consists only of lower-case letters?
- `isspace()` – does the string consists only of white spaces?
- `isupper()` – does the string consists only of upper-case letters?

- `startswith()` – does the string begin with a given substring?

Quiz

Question 1: What is the expected output of the following code?

```
for ch in "abc123XYX":
    if ch.isupper():
        print(ch.lower(), end='')
    elif ch.islower():
        print(ch.upper(), end='')
    else:
        print(ch, end='')
```

Question 2: What is the expected output of the following code?

```
s1 = 'Where are the snows of yesteryear?'
s2 = s1.split()
print(s2[-2])
```

Question 3: What is the expected output of the following code?

```
the_list = ['Where', 'are', 'the', 'snows?']
s = '*'.join(the_list)
print(s)
```

Question 4: What is the expected output of the following code?

```
s = 'It is either easy or impossible'
s = s.replace('easy', 'hard').replace('im', '')
print(s)
```

Check Answers

LAB: Your own split

You already know how `split()` works. Now we want you to prove it. Your task is to write your own function, which behaves almost exactly like the original `split()` method, i.e.:

- it should accept exactly one argument – a string;
- it should return a list of words created from the string, divided in the places where the string contains whitespaces;
- if the string is empty, the function should return an empty list;
- its name should be `mysplit()`

Use the template given. Test your code carefully.

Expected output

```
['To', 'be', 'or', 'not', 'to', 'be,', 'that', 'is', 'the', 'question']
['To', 'be', 'or', 'not', 'to', 'be,that', 'is', 'the', 'question']
[]
['abc']
[]
```

Code

```
def mysplit(strng):
    #
    # put your code here
    #


print(mysplit("To be or not to be, that is the question"))
print(mysplit("To be or not to be,that is the question"))
print(mysplit(" "))
print(mysplit(" abc "))
print(mysplit(""))

```

Check Hint

Check Sample Solution

EIGHT – STRING IN ACTION

Python strings can be compared using the same set of operators which are in use in relation to numbers. Take a look at these operators – they can all compare strings, too:

```
==
!=
>
>=
<
<=
```

There is one "but" – the results of such comparisons may sometimes be a bit surprising. Don't forget that Python doesn't understand the subtle linguistic issues – it just compares code point values, character by character. The results you get from such an operation are sometimes astonishing. Let's start with the simplest cases. Two strings are equal when they consist of the same characters in the same order. By the same fashion, two strings are not equal when they don't consist of the same characters in the same order. Both comparisons give `True` as a result:

```
1  'alpha' == 'alpha'
2  'alpha' != 'Alpha'
3
```

The final relation between strings is determined by comparing the first different character in both strings — you should keep ASCII/UNICODE code points in mind at all times. When you compare two strings of different lengths and the shorter one is identical to the beginning of the longer one, the longer string is considered greater.

Just like here:

```
1  'alpha' < 'alphabet'
2
```

The relation is `True`.

String comparison is always case-sensitive. Upper-case letters are taken as lesser than lower-case ones. The expression is `True`:

```
1  'beta' > 'Beta'
2
```

Even if a string contains digits only, it's still not a number. It's interpreted as-is, like any other regular string, and its potential numerical aspect is not taken into consideration in any way. Look at the examples:

```
1  '10' == '010'
2  '10' > '010'
3  '10' > '8'
4  '20' < '8'
5  '20' < '80'
6
```

They produce the following results:

```
False
True
False
True
True
```

Comparing strings against numbers is generally a bad idea. The only comparisons you can perform with impunity are these symbolized by the `==` and `!=` operators. The former always gives `False`, while the latter always produces `True`. Using any of the remaining comparison operators will raise a `TypeError` exception. Let's check it:

```
1  '10' == 10
2  '10' != 10
3  '10' == 1
4  '10' != 1
5  '10' > 10
6
```

The results in this case are:

```
False
True
False
True
TypeError exception
```

Run all the examples, and carry out more experiments.

Sorting

Comparing is closely related to sorting; or rather, sorting is in fact a very sophisticated case of comparing. This is a good opportunity to show you two possible ways to sort lists containing strings. Such an operation is very common in the real world – any time you see a list of names, goods, titles, or cities, you expect them to be sorted. Let's assume that you want to sort the following list:

```
1  greek = ['omega', 'alpha', 'pi', 'gamma']
2
```

In general, Python offers two different ways to sort lists. The first is implemented as a function named `sorted()`. The function takes one argument (a list) and returns a new list, filled with the sorted argument's elements. Note that this description is a bit simplified compared to the actual implementation – we'll discuss it later. The original list remains untouched. Look at the following code, and run it.

```
1  # Demonstrating the sorted() function:
2  first_greek = ['omega', 'alpha', 'pi', 'gamma']
3  first_greek_2 = sorted(first_greek)
4
5  print(first_greek)
6  print(first_greek_2)
7
8  print()
9
```

The snippet produces the following output:

```
['omega', 'alpha', 'pi', 'gamma']
['alpha', 'gamma', 'omega', 'pi']
```

The second method affects the list itself – no new list is created. Ordering is performed in situ by the method named `sort()`.

```
1  # Demonstrating the sort() method:
2  second_greek = ['omega', 'alpha', 'pi', 'gamma']
3  print(second_greek)
4
5  second_greek.sort()
6  print(second_greek)
7
```

The output hasn't changed:

```
['omega', 'alpha', 'pi', 'gamma']
['alpha', 'gamma', 'omega', 'pi']
```

If you need an order other than non-descending, you have to convince the function/method to change its default behaviors. We'll discuss it soon.

Strings vs. numbers

There are two additional issues that should be discussed here: how to convert a number (an integer or a float) into a string, and vice versa. Moreover, it's a routine way to process input/output data. The numberstring conversion is simple, as it is always possible. It's done by a function named `str()`:

```
itg = 13
flt = 1.3
si = str(itg)
sf = str(flt)

print(si +     + sf)

```

The code outputs:

```
13 1.3
```

The reverse transformation (string-number) is possible when and only when the string represents a valid number. If the condition is not met, expect a `ValueError` exception. Use the `int()` function if you want to get an integer, and `float()` if you need a floating-point value. Just like here:

```
si = '13'
sf = '1.3'
itg = int(si)
flt = float(sf)

print(itg + flt)

```

This is what you'll see:

```
14.3
```

In the next section, we're going to show you some simple programs that process strings.

Summary

1. Strings can be compared to other strings using general comparison operators, but comparing them to numbers gives no reasonable result, because no string can be equal to any number. For example:

- `string == number` is always `False`;
- `string != number` is always `True`;
- `string >= number` always raises an exception.

2. Sorting lists of strings can be done by either a function named `sorted()`, creating a new, sorted list; or by a method named `sort()`, which sorts the list in *situ*.

3. A number can be converted to a string using the `str()` function.

4. Most strings can be converted to numbers using either the `int()` or `float()` function. The conversion fails if a string doesn't contain a valid number image, and an exception is raised then.

Quiz

Question 1: Which of the following lines describe a true condition?

```
'smith' > 'Smith'
'Smiths' < 'Smith'
'Smith' > '1000'
'11' < '8'

```

Question 2: What is the expected output of the following code?

```
s1 = 'Where are the snows of yesteryear?'
s2 = s1.split()
s3 = sorted(s2)
print(s3[1])

```

Question 3: What is the expected result of the following code?

```
s1 = '12.8'
i = int(s1)
s2 = str(i)
f = float(s2)
print(s1 == s2)

```

Check Answers

LAB: An LED Display

You've surely seen a *seven-segment display*.

It's a device (sometimes electronic, sometimes mechanical) designed to present one decimal digit using a subset of seven segments. If you still don't know what it is, refer to the following Wikipedia article. Your task is to write a program which is able to simulate the work of a seven-display device, although you're going to use single LEDs instead of segments.

Each digit is constructed from 13 LEDs (some lit, some dark, of course) – that's how we imagine it:

```
# ### ### # # ### ### ### ### ### ###
#   #   # # # #   #     # # # # # # #
# ### ### ### ### ###   # ### ### # #
# #     #   #   # # #   # # #   # # #
# ### ###   # ### ###   # ### ### ###
```

NOTE: the number 8 shows all the LED lights on.

Your code has to *display* any non-negative integer number entered by the user.

TIP: using a list containing patterns of all ten digits may be very helpful.

Test Data

Sample input

```
123
```

Sample output

```
  # ### ###
  #   #   #
  # ### ###
  # #     #
  # ### ###
```

Sample input

```
9081726354
```

Sample output

```
### ### ###   # ### ### ### ### ### # #
# # # # # #   #   #   # #     # #   # #
### # # ###   #   # ### ### ### ### ###
  # # # # #   #   # #   # #   #   #   #
### ### ###   #   # ### ### ### ###   #
```

Check Hint

Check Sample Solution

NINE – FOUR SIMPLE PROGRAMS

We're going to show you four simple programs in order to present some aspects of string processing in Python. They are purposefully simple, but the lab problems will be significantly more complicated. The first problem we want to show you is called the Caesar cipher – more details here: https://en.wikipedia.org/wiki/Caesar_cipher.

This cipher was (probably) invented and used by Gaius Julius Caesar and his troops during the Gallic Wars. The idea is rather simple – every letter of the message is replaced by its nearest consequent (*A* becomes *B*, *B* becomes *C*, and so on). The only exception is *Z*, which becomes *A*. The following program is a very simple (but working) implementation of the algorithm.

```
1  # Caesar cipher.
2  text = input("Enter your message: ")
3  cipher = ''
4  for char in text:
5      if not char.isalpha():
6          continue
7      char = char.upper()
8      code = ord(char) + 1
9      if code > ord('Z'):
10         code = ord('A')
11     cipher += chr(code)
12
13 print(cipher)
14
```

We've written it using the following assumptions:

- it accepts Latin letters only (note: the Romans didn't use whitespaces or digits)
- all letters of the message are in upper case (note: the Romans knew only capitals)

Let's trace the code:

- line 02: ask the user to enter the open (unencrypted), one-line message;
- line 03: prepare a string for an encrypted message (empty for now)
- line 04: start the iteration through the message;
- line 05: if the current character is not alphabetic...
- line 06: ...ignore it;
- line 07: convert the letter to upper-case (it's preferable to do it blindly, rather than check whether it's needed or not)
- line 08: get the code of the letter and increment it by one;
- line 09: if the resulting code has "left" the Latin alphabet (if it's greater than the *Z* code)...
- line 10: ...change it to the A code;
- line 11: append the received character to the end of the encrypted message;
- line 13: print the cipher.

The code, fed with this message:

```
AVE CAESAR
```

Outputs:

```
BWFDBFTBS
```

Do your own tests.

The Caesar Cipher: decrypting a message

The reverse transformation should now be clear to you – let's just present you with the code as-is, without any explanations. Look at this code. Check carefully if it works. Use the cryptogram from the previous program.

```
1  # Caesar cipher - decrypting a message.
2  cipher = input('Enter your cryptogram: ')
3  text = ''
4  for char in cipher:
5      if not char.isalpha():
6          continue
7      char = char.upper()
8      code = ord(char) - 1
9      if code < ord('A'):
10         code = ord('Z')
11     text += chr(code)
12
13 print(text)
14
```

The Numbers Processor

The third program shows a simple method allowing you to input a line filled with numbers, and to process them easily.

NOTE: the routine `input()` function, combined together with The `int()` or `float()` functions, is unsuitable for this purpose.

The processing will be extremely easy – we want the numbers to be summed. Look at the code. Let's analyze it.

```
1  # Numbers Processor.
2
3  line = input("Enter a line of numbers - separate them with spaces: ")
4  strings = line.split()
5  total = 0
6  try:
7      for substr in strings:
8          total += float(substr)
9      print("The total is:", total)
10 except:
11     print(substr, "is not a number.")
12
```

Using list comprehension may make the code slimmer. You can do that if you want. Let's present our version:

- line 03: ask the user to enter a line filled with any number of numbers (the numbers can be floats)
- line 04: split the line receiving a list of substrings;
- line 05: initiate the total sum to zero;
- line 06: as the string-float conversion may raise an exception, it's best to continue by using the try-except block;
- line 07: iterate through the list...
- line 08: ...and try to convert all its elements into float numbers; if it works, increase the sum;
- line 09: everything is good so far, so print the sum;
- line 10: the program ends here in the case of an error;
- line 11: print a diagnostic message showing the user the reason for the failure.

The code has one important weakness – it displays a bogus result when the user enters an empty line. Can you fix it?

The IBAN Validator

The fourth program implements (in a slightly simplified form) an algorithm used by European banks to specify account numbers. The standard named IBAN (International Bank Account Number) provides a simple and fairly reliable method for validating account numbers against simple typos that can occur during rewriting of the number, for example, from paper documents, like invoices or bills, into computers. You can find more details here: https://en.wikipedia.org/wiki/International_Bank_Account_Number.

An IBAN-compliant account number consists of several things. The first is a two-letter country code taken from the ISO 3166-1 standard (e.g. FR for France, GB for Great Britain, DE for Germany, and so on). Next comes two check digits used to perform the validity checks – fast and simple, but not fully reliable, tests, showing whether a number is invalid (distorted by a typo) or seems to be good. Finally, there is the actual account number (up to 30 alphanumeric characters – the length of that part depends on the country).

The standard says that validation requires the following steps (according to Wikipedia):

- Step 1: Check that the total IBAN length is correct as per the country (this program won't do that, but you can modify the code to meet this requirement if you wish; note: you have to teach the code all the lengths used in Europe)
- Step 2: Move the four initial characters to the end of the string (i.e., the country code and the check digits)
- Step 3: Replace each letter in the string with two digits, thereby expanding the string, where *A = 10*, *B = 11* ... *Z = 35*;
- Step 4: Interpret the string as a decimal integer and compute the remainder of that number by modulo-dividing it by *97*; If the remainder is *1*, the check digit test is passed and the IBAN might be valid.

Look at the following code. Let's analyze it:

```
1  # IBAN Validator.
2
3  iban = input("Enter IBAN, please: ")
4  iban = iban.replace(' ','')
5
6  if not iban.isalnum():
7      print("You have entered invalid characters.")
8  elif len(iban) < 15:
9      print("IBAN entered is too short.")
10 elif len(iban) > 31:
11     print("IBAN entered is too long.")
12 else:
13     iban = (iban[4:] + iban[0:4]).upper()
14     iban2 = ''
15     for ch in iban:
16         if ch.isdigit():
17             iban2 += ch
18         else:
19             iban2 += str(10 + ord(ch) - ord('A'))
20     iban = int(iban2)
21     if iban % 97 == 1:
22         print("IBAN entered is valid.")
23     else:
24         print("IBAN entered is invalid.")
25
```

- line 03: ask the user to enter the IBAN (the number can contain spaces, as they significantly improve number readability...
- line 04: ...but remove them immediately)
- line 05: the entered IBAN must consist of digits and letters only – if it doesn't...
- line 06: ...output the message;
- line 07: the IBAN mustn't be shorter than 15 characters (this is the shortest variant, used in Norway)
- line 08: if it is shorter, the user is informed;
- line 09: moreover, the IBAN cannot be longer than 31 characters (this is the longest variant, used in Malta)
- line 10: if it is longer, make an announcement;
- line 11: start the actual processing;
- line 12: move the four initial characters to the number's end, and convert all letters to upper case (step 02 of the algorithm)
- line 13: this is the variable used to complete the number, created by replacing the letters with digits (according to the algorithm's step 03)
- line 14: iterate through the IBAN;
- line 15: if the character is a digit...
- line 16: just copy it;
- line 17: otherwise...
- line 18: ...convert it into two digits (note the way it's done here)
- line 19: the converted form of the IBAN is ready – make an integer out of it;
- line 20: is the remainder of the division of `iban2` by `97` equal to `1`?
- line 21: If yes, then success;
- line 22: Otherwise...
- line 23: ...the number is invalid.

Let's add some test data (all these numbers are valid – you can invalidate them by changing any character).

- British: `GB72 HBZU 7006 7212 1253 00`
- French: `FR76 30003 03620 00020216907 50`
- German: `DE02100100100152517108`

If you are an EU resident, you can use you own account number for tests.

Summary

1. Strings are key tools in modern data processing, as most useful data are actually strings. For example, using a web search engine (which seems quite trivial these days) utilizes extremely complex string processing, involving unimaginable amounts of data.

2. Comparing strings in a strict way (as Python does) can be very unsatisfactory when it comes to advanced searches (e.g. during extensive database queries). Responding to this demand, a number of fuzzy string comparison algorithms has been created and implemented. These algorithms are able to find strings which aren't equal in the Python sense, but are similar. One such concept is the Hamming distance, which is used to determine the similarity of two strings. If this problem interests you, you can find out more about it here: https://en.wikipedia.org/wiki/Hamming_distance. Another solution of the same kind, but based on a different assumption, is the Levenshtein distance described here: https://en.wikipedia.org/wiki/Levenshtein_distance.

3. Another way of comparing strings is finding their *acoustic* similarity, which means a process leading to determine if two strings sound similar (like "raise" and "race"). Such a similarity has to be established for every language (or even dialect) separately. An algorithm used to perform such a comparison for the English language is called Soundex and was invented – you won't believe – in 1918. You can find out more about it here https://en.wikipedia.org/wiki/Soundex.

4. Due to limited native float and integer data precision, it's sometimes reasonable to store and process huge numeric values as strings. This is the technique Python uses when you force it to operate on an integer number consisting of a very large number of digits.

LAB: Improving the Caesar cipher

You are already familiar with the Caesar cipher, and this is why we want you to improve the code we showed you recently. The original Caesar cipher shifts each character by one: *a* becomes *b*, *z* becomes *a*, and so on. Let's make it a bit harder, and allow the shifted value to come from the range 1..25 inclusive. Moreover, let the code preserve the letters' case (lower-case letters will remain lower-case) and all nonalphabetical characters should remain untouched. Your task is to write a program which:

- asks the user for one line of text to encrypt;
- asks the user for a shift value (an integer number from the range 1..25 – note: you should force the user to enter a valid shift value (don't give up and don't let bad data fool you!)
- prints out the encoded text.

Test your code using the data we've provided.

Test Data

Sample input

```
abcxyzABCxyz 123
2
```

Sample output

```
cdezabCDEzab 123
```

Sample input

```
The die is cast
25
```

Sample output

```
Sgd chd hr bzrs
```

Check Sample Solution

LAB: Palindromes

Do you know what a palindrome is? It's a word which look the same when read forward and backward. For example, "kayak" is a palindrome, while "loyal" is not. Your task is to write a program which asks the user for some text and checks whether the entered text is a palindrome, and prints the result.

NOTE: Assume that an empty string isn't a palindrome; treat upper- and lowercase letters as equal; spaces are not taken into account during the check – treat them as non-existent; there are more than a few correct solutions – try to find more than one. Test your code using the data we've provided.

Test your code using the data we've provided.

Test Data

Sample input

```
Ten animals I slam in a net
```

Sample output

```
It's a palindrome
```

Sample input

```
Eleven animals I slam in a net
```

Sample output

```
It's not a palindrome
```

Check Sample Solution

LAB: Anagrams

An anagram is a new word formed by rearranging the letters of a word, using all the original letters exactly once. For example, the phrases "rail safety" and "fairy tales" are anagrams, while "I am" and "You are" are not. Your task is to write a program which asks the user for two separate texts; and checks whether, the entered texts are anagrams and prints the result.

NOTE: Assume that two empty strings are not anagrams; treat upper- and lower-case letters as equal; spaces are not taken into account during the check – treat them as non-existent.

Test your code using the data we've provided.

Test Data

Sample input

```
Listen
Silent
```

Sample output

```
Anagrams
```

Sample input

```
modern
norman
```

Sample output

```
Not anagrams
```

Check Hint

Check Sample Solution

LAB: The Digit of Life

Some say that the *Digit of Life* is a digit evaluated using somebody's birthday. It's simple – you just need to sum all the digits of the date. If the result contains more than one digit, you have to repeat the addition until you get exactly one digit. For example:

- 1 January 2017 = 2017 01 01
- 2 + 0 + 1 + 7 + 0 + 1 + 0 + 1 = 12
- 1 + 2 = 3

3 is the digit we searched for and found. Your task is to write a program which asks the user her/his birthday (in the format YYYYMMDD, or YYYYDDMM, or MMDDYYYY – actually, the order of the digits doesn't matter)and outputs the *Digit of Life* for the date. Test your code using the data we've provided.

Test Data

Sample input

```
19991229
```

Sample output

```
6
```

Sample input

```
20000101
```

Sample output

```
4
```

Check Hint

Check Sample Solution

LAB: Find a word!

Let's play a game. We will give you two strings: one being a word (e.g. "dog") and the second being a combination of any characters. Your task is to write a program which answers the following question: are the characters comprising the first string hidden inside the second string? For example, if the second string is given as `"vcxzxduybfdsobywuefgas"`, the answer is yes, and if the second string is `"vcxzxdcybfdstbywuefsas"`, the answer is `no` (as the letters "d", "o", or "g" don't appear in this order).

HINT: You should use the two-argument variants of the `find()` functions inside your code; don't worry about case sensitivity. Test your code using the data we've provided.

Test Data

Sample input

```
donor
Nabucodonosor
```

Sample output

```
Yes
```

Sample input

```
donut
Nabucodonosor
```

Sample output

```
No
```

Check Hint

Check Sample Solution

LAB: Sudoku

As you probably know, Sudoku is a number-placing puzzle played on a 9x9 board. The player has to fill the board in a very specific way:

- each row of the board must contain all digits from 1 to 9 (the order doesn't matter)
- each column of the board must contain all digits from 1 to 9 (again, the order doesn't matter)
- each of the nine 3x3 "tiles" (we will name them "sub-squares") of the table must contain all digits from 1 to 9.

If you need more details, you can find them here.

Your task is to write a program which:

- reads 9 rows of the Sudoku, each containing 9 digits (check carefully if the data entered are valid)
- outputs `Yes` if the Sudoku is valid, and `No` otherwise.

Test your code using the data we've provided.

Test Data

Sample input

```
295743861
431865927
876192543
387459216
612387495
549216738
763524189
928671354
154938672
```

Sample output

```
Yes
```

Sample input

```
195743862
431865927
876192543
387459216
612387495
549216738
763524189
928671354
254938671
```

Sample output

```
No
```

Check Hint
Check Sample Solution

TEN – ERRORS, THE PROGRAMMER'S DAILY BREAD

Anything that can go wrong, will go wrong. This is Murphy's law, and it works everywhere and always. Your code's execution can go wrong, too. If it can, it will. Look at the following code. There are at least two possible ways it can "go wrong". Can you see them?

```
1  import math
2
3  x = float(input("Enter x: "))
4  y = math.sqrt(x)
5
6  print("The square root of", x, "equals to", y)
7
```

As a user is able to enter a completely arbitrary string of characters, there is no guarantee that the string can be converted into a float value – this is the first vulnerability of the code. The second is that the `sqrt()` function fails if it gets a negative argument.

You may get one of the following error messages. Something like this:

```
Enter x: Abracadabra


Traceback (most recent call last):


  File "sqrt.py", line 3, in <module>


    x = float(input("Enter x: "))


ValueError: could not convert string to float: 'Abracadabra'
```

Or something like this:

```
Enter x: -1


Traceback (most recent call last):


  File "sqrt.py", line 4, in <module>


    y = math.sqrt(x)


ValueError: math domain error
```

Can you protect yourself from such surprises? Of course you can, and you should do it in order to be considered a good programmer.

Exceptions

Each time your code tries to do something wrong/foolish/irresponsible/crazy/unenforceable, Python does two things: it stops your program; and it creates a special kind of data, called an exception. Both of these activities are called raising an exception. We can say that Python always raises an exception (or that an exception has been raised) when it has no idea what to do with your code. What happens next? The raised exception expects somebody or something to notice it and take care of it. If nothing happens to take care of the raised exception, the program will be forcibly terminated, and you will see an error message sent to the console by Python; otherwise, if the exception is taken care of and handled properly, the suspended program can be resumed and its execution can continue.

Python provides effective tools that allow you to observe exceptions, identify them and handle them efficiently. This is possible due to the fact that all potential exceptions have their unambiguous names, so you can categorize them and react appropriately. You know some exception names already. Take a look at the following diagnostic message:

You know some exception names already. Take a look at the following diagnostic message:

```
ValueError: math domain error
```

`ValueError` is just the exception name. Let's get familiar with some other exceptions. Look at the code. Run the (obviously incorrect) program.

```
1  value = 1
2  value /= 0
3
```

You will see the following message in reply:

```
Traceback (most recent call last):
File "div.py", line 2, in <module>
value /= 0
ZeroDivisionError: division by zero
```

This exception error is called `ZeroDivisionError`. Look at this code. What will happen when you run it? Check.

```
1  my_list = []
2  x = my_list[0]
3
```

You will see the following message in reply:

```
Traceback (most recent call last):
File "lst.py", line 2, in <module>
x = list[0]
IndexError: list index out of range
```

This is The `IndexError`.

How do you handle exceptions? The word *try* is the key to the solution. What's more, it's a keyword, too. The recipe for success is as follows: first, you have to try to do something; next, you have to check whether everything went well. But wouldn't it be better to check all circumstances first and then do something only if it's safe? Just like this example:

```
1  first_number = int(input("Enter the first number: "))
2  second_number = int(input("Enter the second number: "))
3
4  if second_number != 0:
5      print(first_number / second_number)
6  else:
7      print("This operation cannot be done.")
8
9  print("THE END.")
10
```

Admittedly, this way may seem to be the most natural and understandable, but in reality, this method doesn't make programming any easier. All these checks can make your code bloated and illegible. Python prefers a completely different approach. Look at the code. This is the favorite Python approach.

```
1  first_number = int(input("Enter the first number: "))
2  second_number = int(input("Enter the second number: "))
3
4  try:
5      print(first_number / second_number)
6  except:
7      print("This operation cannot be done.")
8
9  print("THE END.")
10
```

NOTE: The `try` keyword begins a block of the code which may or may not be performing correctly; next, Python tries to perform the risky action; if it fails, an exception is raised and Python starts to look for a solution. The `except` keyword starts a piece of code which will be executed if anything inside the `try` block goes wrong – if an exception is raised inside a previous `try` block, it will fail here, so the code located after the `except` keyword should provide an adequate reaction to the raised exception. Returning to the previous nesting level ends the `try-except` section.

Run the code and test its behavior. Let's summarize this:

```
1  try:
2      :
3      :
4  except:
5      :
6      :
7
```

In the first step, Python tries to perform all instructions placed between the `try:` and `except:` statements. If nothing is wrong with the execution and all instructions are performed successfully, the execution jumps to the point after the last line of the `except:` block, and the block's execution is considered complete. If anything goes wrong inside the `try:` and `except:` block, the execution immediately jumps out of the block and into the first instruction located after the `except:` keyword; this means

that some of the instructions from the block may be silently omitted. Look at the following code. It will help you understand this mechanism.

```
try:
    print("1")
    x = 1 / 0
    print("2")
except:
    print("Oh dear, something went wrong...")

print("3")

```

This is the output it produces:

```
1
Oh dear, something went wrong...
3
```

NOTE: The `print("2")` instruction was lost in the process.

This approach has one important disadvantage – if there is a possibility that more than one exception may skip into an `except:` branch, you may have trouble figuring out what actually happened. Just like this code. Run it and see what happens.

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
except:
    print("Oh dear, something went wrong...")

print("THE END.")

```

The message: `Oh dear, something went wrong...` appearing in your console says nothing about the reason, while there are two possible causes of the exception: non-integer data entered by the user; or an integer value equal to 0 assigned to the x variable.

Technically, there are two ways to solve the issue: build two consecutive `try-except` blocks, one for each possible exception reason (easy, but will cause unfavorable code growth), or use a more advanced variant of the instruction.

It looks like this:

```
try:
    :
except exc1:
    :
except exc2:
    :
except:
    :

```

This is how it works: if the `try` branch raises the `exc1` exception, it will be handled by the `except exc1:` block; similarly, if the `try` branch raises the `exc2` exception, it will be handled by the `except exc2:` block; if the `try` branch raises any other exception, it will be handled by the unnamed `except` block.

Let's move on to the next part of the course and see it in action. Look at this code. Our solution is there.

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
    print(y)
except ZeroDivisionError:
    print("You cannot divide by zero, sorry.")
except ValueError:
    print("You must enter an integer value.")
except:
    print("Oh dear, something went wrong...")

print("THE END.")

```

The code, when run, produces one of the following four variants of output: if you enter a valid, non-zero integer value (e.g. `5`) it says:

```
0.2
THE END.
```

If you enter `0`, it says:

```
You cannot divide by zero, sorry.
THE END.
```

If you enter any non-integer string, you see:

```
You must enter an integer value.
THE END.
```

If you press Ctrl-C while the program is waiting for the user's input, which causes an exception named `KeyboardInterrupt`, the program says:

```
Oh dear, something went wrong...
THE END.
```

Don't forget that the `except` branches are searched in the same order in which they appear in the code. You must not use more than one `except` branch with a certain exception name. The number of different except branches is arbitrary – the only condition is that if you use `try`, you must put at least one `except` (named or not) after it. The `except` keyword must not be used without a preceding `try`. If any of the `except` branches is executed, no other branches will be visited. If none of the specified except branches matches the raised exception, the exception remains unhandled. If an unnamed `except` branch exists, that is, one without an exception name, it has to be specified as the last.

```
try:
    :
except exc1:
    :
except exc2:
    :
except:
    :

```

Let's continue the experiments now. Look at the following code. We've modified the previous program – we've removed the `ZeroDivisionError` branch.

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
    print(y)
except ValueError:
    print("You must enter an integer value.")
except:
    print("Oh dear, something went wrong...")

print("THE END.")

```

What happens if the user enters `0` as an input? As there are no dedicated branches for division by zero, the raised exception falls into the general, unnamed, branch; in this case, the program will say:

```
Oh dear, something went wrong...
THE END.
```

Try it yourself. Run the program. Let's spoil the code once again. Look at the program. This time, we've removed the unnamed branch.

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
    print(y)
except ValueError:
    print("You must enter an integer value.")

print("THE END.")

```

The user enters `0` once again, and the exception raised won't be handled by `ValueError` – it has nothing to do with it as there's no other branch, and you should to see this message:

```
Traceback (most recent call last):
File "exc.py", line 3, in <module>
y = 1 / x
ZeroDivisionError: division by zero
```

You've learned a lot about exception handling in Python. Next, we will focus on built-in Python exceptions and their hierarchies.

Summary

1. An exception is an event during program execution caused by an abnormal situation. The exception should be handled to avoid the termination of the program. The part of your code that is suspected of being the source of the exception should be put inside the `try` branch. When the exception happens, the execution of the code is not terminated, but instead jumps into the `except` branch. This is the place where the handling of the exception should take place. The general scheme for such a construction looks as follows:

```
# The code that always runs smoothly.
:
try:
    :
    # Risky code.
    :
except:
    :
    # Crisis management takes place here.
    :
:
# Back to normal.
:

```

2. If you need to handle more than one exception coming from the same `try` branch, you can add more than one `except` branch, but you have to label them with different exception names, like this:

```
# The code that always runs smoothly.
:
try:
    :
    # Risky code.
    :
except Except_1:
    # Crisis management takes place here.
except Except_2:
    # We save the world here.
:
# Back to normal.
:

```

At most, one of The `except` branches is executed – none of the branches is performed when the raised exception doesn't match any of the specified exceptions.

3. You cannot add more than one anonymous (unnamed) `except` branch after the named ones.

```
# The code that always runs smoothly.
:
try:
    :
    # Risky code.
    :
except Except_1:
    # Crisis management takes place here.
except Except_2:
    # We save the world here.
except:
    # All other issues fall here.
:
# Back to normal.
:

```

Quiz

Question 1: What is the expected output of the following code?

```
try:
    print("Let's try to do this")
    print("#"[2])
    print("We succeeded!")
except:
    print("We failed")
print("We're done")
```

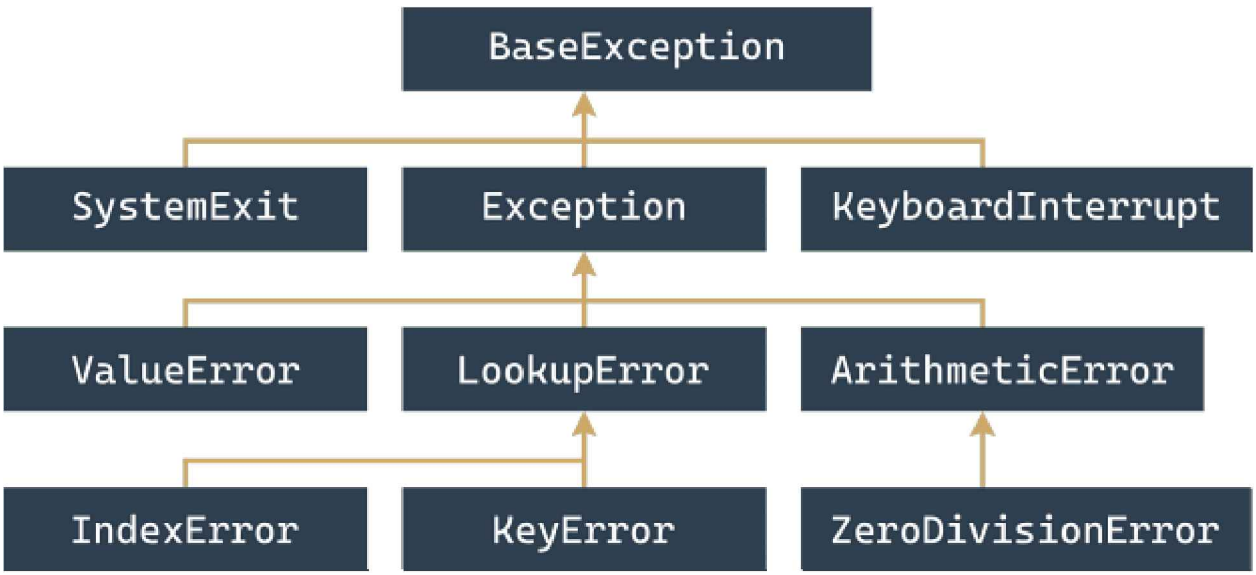
Question 2: What is the expected output of the following code?

```
try:
    print("alpha"[1/0])
except ZeroDivisionError:
    print("zero")
except IndexError:
    print("index")
except:
    print("some")
```

Check Answers

ELEVEN – THE ANATOMY OF EXCEPTIONS

Python 3 defines 63 built-in exceptions, and all of them form a treeshaped hierarchy, although the tree is a bit weird as its root is located on top. Some of the built-in exceptions are more general and include other exceptions, while others are completely concrete, in that they represent themselves only. We can say that the closer to the root an exception is located, the more general or abstract it is. In turn, the exceptions located at the branches' ends — we can call them leaves — are concrete. Take a look at the figure:



It shows a small section of the complete exception tree. Let's begin by examining the tree from The `ZeroDivisionError` leaf.

NOTE: `ZeroDivisionError` is a special case of a more general exception class named `ArithmeticError`. `ArithmeticError` is a special case of a more general exception class named just `Exception`. `Exception` is a special case of a more general class named `BaseException`.

We can describe it in the following way (note the direction of the arrows – they always point to the more general entity): `ZeroDivisionError`. We're going to show you how this generalization works. Let's start with some really simple code. Look at this code. It is a simple example to start with. Run it.

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("Ooopsss...")

print("THE END.")

```

The output we expect to see looks like this:

```
Ooopsss...
THE END.
```

Now look at the following code:

```
try:
    y = 1 / 0
except ArithmeticError:
    print("Ooopsss...")

print("THE END.")

```

Something has changed in it – we've replaced `ZeroDivisionError` with `ArithmeticError`. You already know that `ArithmeticError` is a general class including (among others) The `ZeroDivisionError` exception. Thus, the code's output remains unchanged. Test it.

This also means that replacing the exception's name with either `Exception` or `BaseException` won't change the program's behavior. Let's summarize: each exception raised falls into the first matching branch; the matching branch doesn't have to specify the same exception exactly – it's enough that the exception is more general (more abstract) than the raised one.

Look at this code. What will happen here?

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("Zero Division!")
except ArithmeticError:
    print("Arithmetic problem!")

print("THE END.")

```

The first matching branch is the one containing `ZeroDivisionError`. It means that your console will show:

```
Zero division!
THE END.
```

Will it change anything if we swap the two `except` branches around? Just like this one here:

```
try:
    y = 1 / 0
except ArithmeticError:
    print("Arithmetic problem!")
except ZeroDivisionError:
    print("Zero Division!")

print("THE END.")

```

The change is radical – the code's output is now:

```
Arithmetic problem!
THE END.
```

Why, if the exception raised is the same as previously? The exception is the same, but the more general exception is now listed first – it will catch all zero divisions too. It also means that there's no chance that any exception hits the `ZeroDivisionError` branch. This branch is now completely unreachable. Remember that the order of the branches matter! Don't put more general exceptions before more concrete ones, as this will make the latter one unreachable and useless. Moreover, it will make your code messy and inconsistent, and Python won't generate any error messages regarding this issue. If you want to handle two or more exceptions in the same way, you can use the following syntax:

```
try:
    :
except (exc1, exc2):
    :
```

You simply have to put all the engaged exception names into a commaseparated list and not forget the parentheses. If an exception is raised inside a function, it can be handled either inside the function or outside the function. Let's start with the first variant – look at the following code.

```
def bad_fun(n):
    try:
        return 1 / n
    except ArithmeticError:
        print("Arithmetic Problem!")
    return None

bad_fun(0)

print("THE END.")

```

The `ZeroDivisionError` exception (being a concrete case of the `ArithmeticError` exception class) is raised inside the `bad_fun()` function, and it doesn't leave the function – the function itself takes care of it. The program outputs:

```
Arithmetic problem!
THE END.
```

It's also possible to let the exception propagate outside the function. Let's test it now. Look at the following code:

```
def bad_fun(n):
    return 1 / n

try:
    bad_fun(0)
except ArithmeticError:
    print("What happened? An exception was raised!")

print("THE END.")

```

The problem has to be solved by the invoker (or by the invoker's invoker, and so on). The program outputs:

```
What happened? An exception was raised!
THE END.
```

NOTE: The exception raised can cross function and module boundaries, and travel through the invocation chain looking for a matching `except` clause able to handle it.

If there is no such clause, the exception remains unhandled, and Python solves the problem in its standard way – by terminating your code and emitting a diagnostic message. Now we're going to suspend this discussion, as we want to introduce you to a brand new Python instruction.

The `raise` instruction raises the specified exception named `exc` as if it was raised in a normal (natural) way:

```
raise exc

```

NOTE: `raise` is a keyword.

The instruction enables you to simulate raising actual exceptions (e.g. to test your handling strategy) and to partially handle an exception and make another part of the code responsible for completing the handling (separation of concerns).

Look at this code. This is how you can use it in practice.

```
def bad_fun(n):
    raise ZeroDivisionError


try:
    bad_fun(0)
except ArithmeticError:
    print("What happened? An error?")

print("THE END.")

```

The program's output remains unchanged. In this way, you can test your exception handling routine without forcing the code to do stupid things. The `raise` instruction may also be utilized in the following way (note the absence of the exception's name):

```
raise

```

There is one serious restriction: this kind of `raise` instruction may be used inside the `except` branch only; using it in any other context causes an error. The instruction will immediately re-raise the same exception as currently handled. Thanks to this, you can distribute the exception handling among different parts of the code. Look at the code. Run it – you'll see it in action.

```
def bad_fun(n):
    try:
        return n / 0
    except:
        print("I did it again!")
        raise


try:
    bad_fun(0)
except ArithmeticError:
    print("I see!")

print("THE END.")

```

The `ZeroDivisionError` is raised twice, first, inside the `try` part of the code (this is caused by actual zero division), and then inside the `except` part by the `raise` instruction. In effect, the code outputs:

- first, inside The `try` part of the code (this is caused by actual zero division)
- second, inside The `except` part by The `raise` instruction.

In effect, the code outputs:

```
I did it again!
I see!
THE END.
```

Now is a good moment to show you another Python instruction, named `assert`. This is a keyword.

```
assert expression3

```

How does it work? It evaluates the expression. If the expression evaluates to `True`, or a non-zero numerical value, or a non-empty string, or any other value different than `None`, it won't do anything else, otherwise, it automatically and immediately raises an exception named `AssertionError`. In this case, we say that the assertion has failed.

How it can be used? You may want to put it into your code where you want to be absolutely safe from evidently wrong data, and where you aren't absolutely sure that the data has been carefully examined before (e.g. inside a function used by someone else). Raising an `AssertionError` exception secures your code from producing invalid results, and clearly shows the nature of the failure. Assertions don't supersede exceptions or validate the data – they are their supplements. If exceptions and data validation are like careful driving, assertion can play the role of an airbag. Let's see the `assert` instruction in action. Look at the code and run it.

```
import math

x = float(input("Enter a number: "))
assert x >= 0.0

x = math.sqrt(x)

print(x)

```

The program runs flawlessly if you enter a valid numerical value greater than or equal to zero; otherwise, it stops and emits the following message:

```
Traceback (most recent call last):
File ".main.py", line 4, in <module>
assert x >= 0.0
AssertionError
```

Summary

1. You cannot add more than one anonymous (unnamed) `except` branch after the named ones.

```
# The code that always runs smoothly.
:
try:
    :
    # Risky code.
    :
except Except_1:
    # Crisis management takes place here.
except Except_2:
    # We save the world here.
except:
    # All other issues fall here.
:
# Back to normal.
:

```

2. All the predefined Python exceptions form a hierarchy, i.e. some of them are more general (the one named `BaseException` is the most general one) while others are more or less concrete (e.g. `IndexError` is more concrete than `LookupError`). You shouldn't put more general exceptions before the more concrete ones inside the same `except` branch sequence. For example, you can do this:

```
try:
    # Risky code.
except IndexError:
    # Taking care of mistreated lists
except LookupError:
    # Dealing with other erroneous lookups

```

But don't do this (unless you're absolutely sure that you want some part of your code to be useless).

```
try:
    # Risky code.
except LookupError:
    # Dealing with erroneous lookups
except IndexError:
     You'll never get here

```

3. The Python statement `raise ExceptionName` can raise an exception on demand. The same statement, but lacking *ExceptionName*, can be used inside The `except` branch **only**, and raises the same exception which is currently being handled.

4. The Python statement `assert expression` evaluates the *expression* and raises The `AssertionError` exception when the *expression* is equal to zero, an empty string, or `None`. You can use it to protect some critical parts of your code from devastating data.

Quiz

Question 1: What is the expected output of the following code?

```
try:
    print(1/0)
except ZeroDivisionError:
    print("zero")
except ArithmeticError:
    print("arith")
except:
    print("some")
```

Question 2: What is the expected output of the following code?

```
try:
    print(1/0)
except ArithmeticError:
    print("arith")
except ZeroDivisionError:
    print("zero")
except:
    print("some")
```

Question 3: What is the expected output of the following code?

```
def foo(x):
    assert x
    return 1/x


try:
    print(foo(0))
except ZeroDivisionError:
    print("zero")
except:
    print("some")
```

Check Answers

TWELVE – USEFUL EXCEPTIONS

We're going to show you a short list of the most useful exceptions. While it may sound strange to call "useful" a thing or a phenomenon which is a visible sign of failure or setback, as you know, to err is human and if anything can go wrong, it will go wrong. Exceptions are as routine and normal as any other aspect of a programmer's life. For each exception, we'll show you its name, its location in the exception tree, a short description, and a concise snippet of code showing the circumstances in which the exception may be raised. There are lots of other exceptions to explore – You can explore when you have time.

`ArithmeticError`

Location: `BaseException` ← `Exception` ← `ArithmeticError`

Description: an abstract exception including all exceptions caused by arithmetic operations like zero division or an argument's invalid domain

`AssertionError`

Location: `BaseException` ← `Exception` ← `AssertionError`

Description: a concrete exception raised by the assert instruction when its argument evaluates to `False`, `None`, `0`, or an empty string

Code

```
1  from math import tan, radians
2  angle = int(input('Enter integral angle in degrees: '))
3
4  # We must be sure that angle != 90 + k * 180
5  assert angle % 180 != 90
6  print(tan(radians(angle)))
7
```

`BaseException`

Location: `BaseException`

Description: the most general (abstract) of all Python exceptions – all other exceptions are included in this one; it can be said that the following two `except` branches are equivalent: `except:` and `except BaseException:`.

`IndexError`

Location: `BaseException` ← `Exception` ← `LookupError` ← `IndexError`

Description: a concrete exception raised when you try to access a non-existent sequence's element (e.g. a list's element)

Code

```
1  # The code shows an extravagant way
2  # of leaving the loop.
3
4  the_list = [1, 2, 3, 4, 5]
5  ix = 0
6  do_it = True
7
8  while do_it:
9      try:
10         print(the_list[ix])
11         ix += 1
12     except IndexError:
13         do_it = False
14
15 print('Done')
16
```

`KeyboardInterrupt`

Location: `BaseException` ← `KeyboardInterrupt`

Description: a concrete exception raised when the user uses a keyboard shortcut designed to terminate a program's execution (*Ctrl-C* in most OSs); if handling this exception doesn't lead to program termination, the program continues its execution.

NOTE: this exception is not derived from the Exception class. Run the program in IDLE.

Code

```
# This code cannot be terminated
# by pressing Ctrl-C.

from time import sleep

seconds = 0

while True:
    try:
        print(seconds)
        seconds += 1
        sleep(1)
    except KeyboardInterrupt:
        print("Don't do that!")

```

LookupError

Location: `BaseException` ← `Exception` ← `LookupError`

Description: an abstract exception including all exceptions caused by errors resulting from invalid references to different collections (lists, dictionaries, tuples, etc.)

MemoryError

Location: `BaseException` ← `Exception` ← `MemoryError`

Description: a concrete exception raised when an operation cannot be completed due to a lack of free memory.

Code

```
# This code causes the MemoryError exception.
# Warning: executing this code may affect your OS.
# Don't run it in production environments!

string = 'x'
try:
    while True:
        string = string + string
        print(len(string))
except MemoryError:
    print('This is not funny!')

```

OverflowError

Location: `BaseException` ← `Exception` ← `ArithmeticError` ← `OverflowError`

Description: a concrete exception raised when an operation produces a number too big to be successfully stored.

Code

```
# The code prints subsequent
# values of exp(k), k = 1, 2, 4, 8, 16, ...

from math import exp

ex = 1

try:
    while True:
        print(exp(ex))
        ex *= 2
except OverflowError:
    print('The number is too big.')

```

ImportError

Location: `BaseException` ← `Exception` ← `StandardError` ← `ImportError`

Description: a concrete exception raised when an import operation fails.

Code

```
>
# One of these imports will fail - which one?

try:
    import math
    import time
    import abracadabra:

except:
    print('One of your imports has failed.')

```

KeyError

Location: `BaseException` ← `Exception` ← `LookupError` ← `KeyError`

Description: a concrete exception raised when you try to access a non-existent element in a collection (e.g. a dictionary's element).

Code

```
# How to abuse the dictionary
# and how to deal with it?

dictionary = {'a': 'b', 'b': 'c', 'c': 'd'}
ch = 'a'

try:
    while True:
        ch = dictionary[ch]
        print(ch)
except KeyError:
    print('No such key:', ch)

```

We are done with exceptions for now, but they'll return when we discuss object-oriented programming in Python. You can use them to protect your code from bad accidents, but you also have to learn how to dive into them, exploring the information they carry.

Exceptions are in fact objects – however, we can tell you nothing about this aspect until we present you with classes, objects, and the like. For the time being, if you'd like to learn more about exceptions on your own, look into the Standard Python Library at https://docs.python.org/3.6/library/exceptions.html.

LAB: Reading `int`s safely

Your task is to write a function able to input integer values and to check if they are within a specified range. The function should:

- accept three arguments: a prompt, a low acceptable limit, and a high acceptable limit;
- if the user enters a string that is not an integer value, the function should emit the message `Error: wrong input`, and ask the user to input the value again;
- if the user enters a number which falls outside the specified range, the function should emit the message `Error: the value is not within permitted range (min..max)` and ask the user to input the value again;
- if the input value is valid, return it as a result.

Test data

Test your code carefully. This is how the function should react to the user's input:

```
Enter a number from -10 to 10: 100
Error: the value is not within permitted range (-10..10)
Enter a number from -10 to 10: asd
Error: wrong input
Enter number from -10 to 10: 1
The number is: 1
```

Code

```
def read_int(prompt, min, max):
    #
    # Write your code here.
    #


v = read_int("Enter a number from -10 to 10: ", -10, 10)

print("The number is:", v)

```

Check Sample Solution

Section Summary

1. Some abstract built-in Python exceptions are:

- ArithmeticError
- BaseException
- LookupError

2. Some concrete built-in Python exceptions are:

- AssertionError
- ImportError
- IndexError
- KeyboardInterrupt
- KeyError
- MemoryError
- OverflowError

Section Quiz

Question 1: Which of the exceptions will you use to protect your code from being interrupted through the use of the keyboard?

Question 2: What is the name of the most general of all Python exceptions?

Question 3: Which of the exceptions will be raised through the following unsuccessful evaluation?

```
huge_value = 1E250 ** 2
```

Check Answers

PART 3: OBJECT-ORIENTED PROGRAMMING

THIRTEEN – THE FOUNDATIONS OF OOP

Let's take a step outside of computer programming and computers in general, and discuss object programming issues. Nearly all of the programs and techniques you have used till now fall under the procedural style of programming. Admittedly, you have made use of some built-in objects, but when referring to them, we just mentioned the absolute minimum.

The procedural style of programming was the dominant approach to software development for decades of IT, and it is still in use today. Moreover, it isn't going to disappear in the future, as it works very well for specific types of projects (generally, not very complex ones and not large ones, but there are lots of exceptions to that rule).

The object approach is quite young (much younger than the procedural approach) and is particularly useful when applied to big and complex projects carried out by large teams consisting of many developers. This kind of understanding of a project's structure makes many important tasks easier, for example, dividing the project into small independent parts, and developing different project elements independently.

Python is a universal tool for both object and procedural programming. It may be successfully utilized in both spheres. Furthermore, you can create lots of useful applications, even if you know nothing about classes and objects, but you have to keep in mind that some of the problems (e.g. graphical user interface handling) may require a strict object approach. Fortunately, object programming is relatively simple.

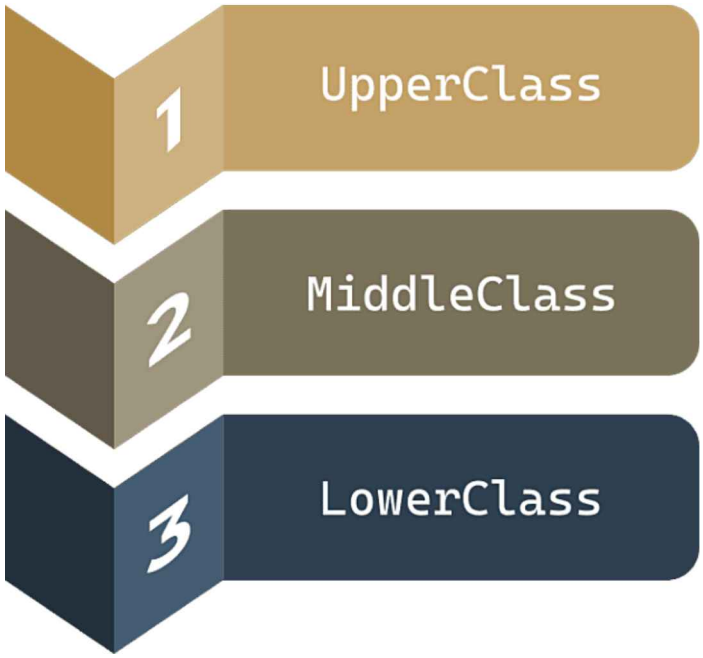
The procedural vs. the object-oriented approach

In the procedural approach, it's possible to distinguish two different and completely separate worlds: the world of data, and the world of code. The world of data is populated with variables of different kinds, while the world of code is inhabited by code grouped into modules and functions. Functions are able to use data, but not vice versa. Furthermore, functions are able to abuse data, in other words, to use the value in an unauthorized manner (e.g. when the sine function gets a bank account balance as a parameter).

We said in the past that data cannot use functions. But is this entirely true? Are there some special kinds of data that can use functions? Yes, there are – the ones named methods. These are functions which are invoked from within the data, not beside them. If you can see this distinction, you've taken the first step into object programming. The object approach suggests a completely different way of thinking. The data and the code are enclosed together in the same world, divided into classes.

Every class is like a recipe which can be used when you want to create a useful object (this is where the name of the approach comes from). You may produce as many objects as you need to solve your problem. Every object has a set of traits (they are called properties or attributes – we'll use both words synonymously) and is able to perform a set of activities (which are called methods).

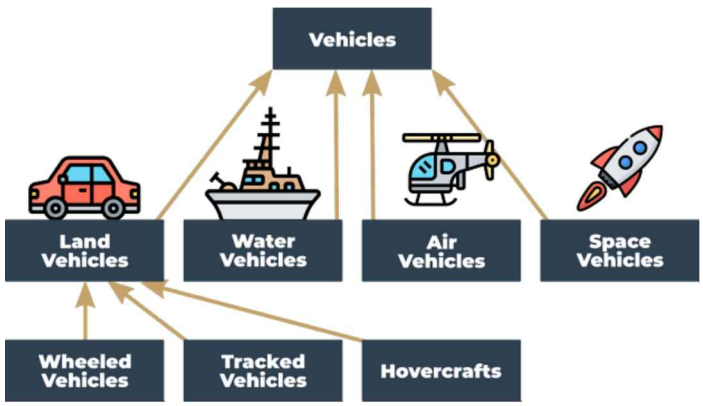
The recipes may be modified if they are inadequate for specific purposes and, in effect, new classes may be created. These new classes inherit properties and methods from the originals, and usually add some new ones, creating new, more specific tools. Objects are incarnations of ideas expressed in classes, like a cheesecake on your plate is an incarnation of the idea expressed in a recipe printed in an old cookbook. The objects interact with each other, exchanging data or activating their methods. A properly constructed class (and thus, its objects) is able to protect the sensible data and hide it from unauthorized modifications. There is no clear border between data and code: they live as one in objects.



All these concepts are not as abstract as you may at first suspect. On the contrary, they all are taken from real-life experiences, and therefore are extremely useful in computer programming: they don't create artificial life – they reflect real facts, relationships, and circumstances.

Class hierarchies

The word *class* has many meanings, but not all of them are compatible with the ideas we want to discuss here. The *class* that we are concerned with is like a *category*, as a result of precisely defined similarities. We'll try to point out a few classes which are good examples of this concept.



Let's look for a moment at vehicles. All existing vehicles (and those that don't exist yet) are related by a single, important feature: the ability to move. You may argue that a dog moves, too; is a dog a vehicle? No, it isn't. We have to improve the definition, enrich it with other criteria, in order to distinguish vehicles from other beings, and create a stronger connection. Let's take the following circumstances into consideration: vehicles are artificially created entities used for transportation, moved by forces of nature, and directed (driven) by humans.

Based on this definition, a dog is not a vehicle. The *vehicles* class is very broad. Too broad. We have to define some more specialized classes, then. The specialized classes are the subclasses. The *vehicles* class will be a superclass for them all.

NOTE: The hierarchy grows from top to bottom, like tree roots, not branches. The most general, and the widest, class is always at the top (the superclass) while its descendants are located below (the subclasses).

By now, you can probably point out some potential subclasses for the *Vehicles* superclass. There are many possible classifications. We've chosen subclasses based on the environment, and say that there are (at least) four subclasses:

- land vehicles;
- water vehicles;
- air vehicles;
- space vehicles.

In this example, we'll discuss the first subclass only – land vehicles. If you wish, you can continue with the remaining classes.

Land vehicles may be further divided, depending on the method with which they impact the ground. So, we can enumerate:

- wheeled vehicles;
- tracked vehicles;

- hovercrafts.

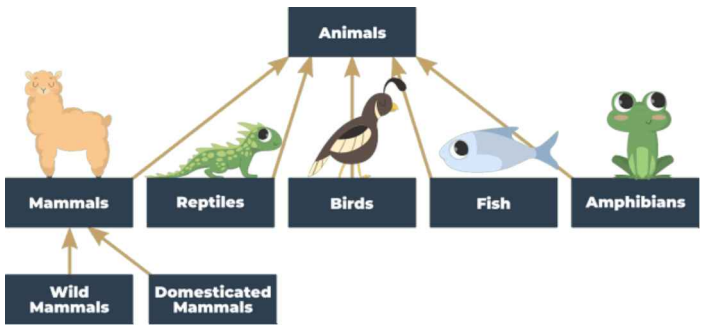
The hierarchy we've created is illustrated by the figure. Note the direction of the arrows – they always point to the superclass. The toplevel class is an exception – it doesn't have its own superclass.

Another example is the hierarchy of the taxonomic kingdom of animals. We can say that all *animals* (our top-level class) can be divided into five subclasses:

- mammals;
- reptiles;
- birds;
- fish;
- amphibians.

We'll take the first one for further analysis. We have identified the following subclasses:

- wild mammals;
- domesticated mammals.



Try to extend the hierarchy any way you want, and find the right place for humans.

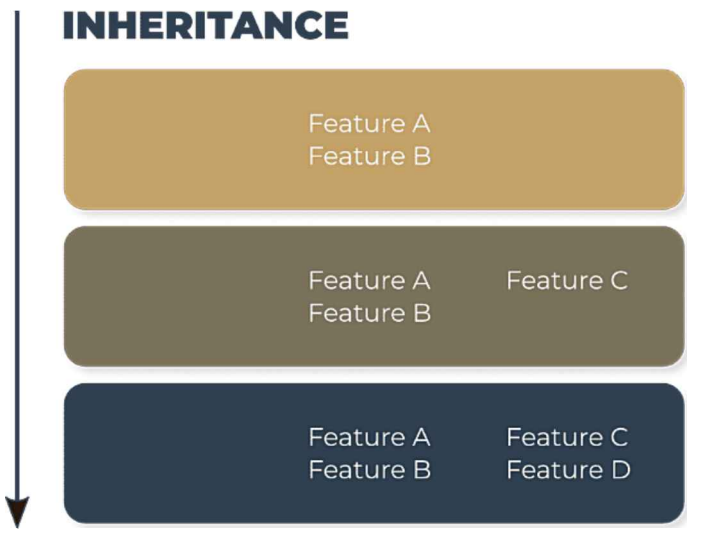
What is an object?

A class (among other definitions) is a set of objects. An object is a being belonging to a class. An object is an incarnation of the requirements, traits, and qualities assigned to a specific class. This may sound simple, but note the following important circumstances. Classes form a hierarchy. This may mean that an object belonging to a specific class belongs to all the superclasses at the same time. It may also mean that any object belonging to a superclass may not belong to any of its subclasses. For example: any personal car is an object belonging to the *wheeled vehicles* class. It also means that the same car belongs to all superclasses of its home class; therefore, it is a member of the *vehicles* class, too.

Your dog (or your cat) is an object included in the *domesticated mammals* class, which explicitly means that it is included in the *animals* class as well. Each subclass is more specialized (or more concrete) than its superclass. Conversely, each superclass is more general (more abstract) than any of its subclasses. Note that we've presumed that a class may only have one superclass – this is not always true, but we'll discuss this issue more a bit later.

Inheritance

Let's define one of the fundamental concepts of object programming, named inheritance. Any object bound to a specific level of a class hierarchy inherits all the traits (as well as the requirements and qualities) defined inside any of the superclasses. The object's home class may define new traits (as well as requirements and qualities) which will be inherited by any of its subclasses. You shouldn't have any problems matching this rule to specific examples, whether it applies to animals, or to vehicles.



What does an object have?

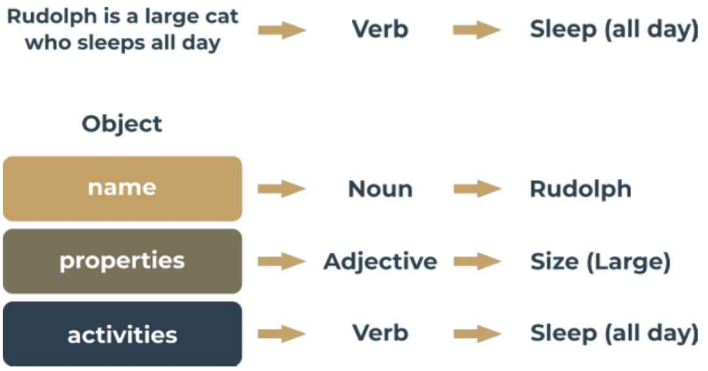
The object programming convention assumes that every existing object may be equipped with three groups of attributes: an object has a name that uniquely identifies it within its home namespace, although there may be some anonymous objects, too; an object has a set of individual properties which make it original, unique, or outstanding, although it's possible that some objects may have no properties at all; an object has a set of abilities to perform specific activities, able to change the object itself, or some of the other objects.

There is a hint (although this doesn't always work) which can help you identify any of these three spheres. Whenever you describe an object and you use:

- a noun – you probably define the object's name;
- an adjective – you probably define the object's property;
- a verb – you probably define the object's activity.

Two sample phrases should serve as a good example:

- Rudolph is a large cat who sleeps all day.



Object name = Rudolph
Home class = Cat
Property = Size (large)
Activity = Sleep (all day)

- A pink Cadillac went quickly.

Object name = Cadillac
Home class = Wheeled vehicles
Property = Color (pink)
Activity = Go (quickly)

Your first class

Object programming is the art of defining and expanding classes. A class is a model of a very specific part of reality, reflecting properties and activities found in the real world. The classes defined at the beginning are too general and imprecise to cover the largest possible number of real cases. There's no obstacle to defining new, more precise subclasses. They'll inherit everything from their superclass, so the work that went into its creation isn't wasted. The new class may add new properties and new activities, and therefore may be more useful in specific applications. Obviously, it may be used as a superclass for any number of newly created subclasses.

The process doesn't need to have an end. You can create as many classes as you need. The class you define has nothing to do with the object: the existence of a class does not mean that any of the compatible objects will automatically be created. The class itself isn't able to create an object – you have to create it yourself, and Python allows you to do this.

It's time to define the simplest class and to create an object. Take a look at the following example:

```
1  class TheSimplestClass:
2      pass
3
```

We've defined a class. The class is rather poor: it has neither properties nor activities. It's empty, actually, but that doesn't matter for now. The simpler the class, the better for our purposes. The definition begins with the keyword `class`. The keyword is followed by an identifier which will name the class (note: don't confuse it with the object's name – these are two different things). Next, you add a colon (`:`), as classes, like functions, form their own nested block. The content inside the block define all the class's properties and activities. The `pass` keyword fills the class with nothing. It doesn't contain any methods or properties.

Your first object

The newly defined class becomes a tool that is able to create new objects. The tool has to be used explicitly, on demand. Imagine that you want to create one (exactly one) object of the `TheSimplestClass` class. To do this, you need to assign a variable to store the newly created object of that class, and create an object at the same time. You do it in the following way:

```
1  my_first_object = TheSimplestClass()
2
```

NOTE: The class name tries to pretend that it's a function – can you see this? We'll discuss it soon. The newly created object is equipped with everything the class brings; as this class is completely empty, the object is empty, too. The act of creating an object of the selected class is also called an instantiation (as the object becomes an instance of the class).

Let's leave classes alone for a short moment, as we're now going to tell you a few words about stacks. We know the concept of classes and objects may not be fully clear yet. Don't worry, we'll explain everything very soon.

Summary

1. A class is an idea (more or less abstract) which can be used to create a number of incarnations – such an incarnation is called an object.

2. When a class is derived from another class, their relation is named inheritance. The class which derives from the other class is named a subclass. The second side of this relation is named superclass. A way to present such a relation is an inheritance diagram, where:

- superclasses are always presented above their subclasses;
- relations between classes are shown as arrows directed from the subclass toward its superclass.

3. Objects are equipped with:

- a name which identifies them and allows us to distinguish between them;
- a set of properties (the set can be empty)
- a set of methods (can be empty, too)

4. To define a Python class, you need to use the class keyword. For example:

```
1  class This_Is_A_Class:
2      pass
3
```

5. To create an object of the previously defined class, you need to use the class as if it were a function. For example:

```
1  this_is_an_object = This_Is_A_Class()
2
```

Quiz

Question 1: If we assume that pythons, vipers, and cobras are subclasses of the same superclass, what would you call it?

Question 2: Try to name a few python class subclasses.

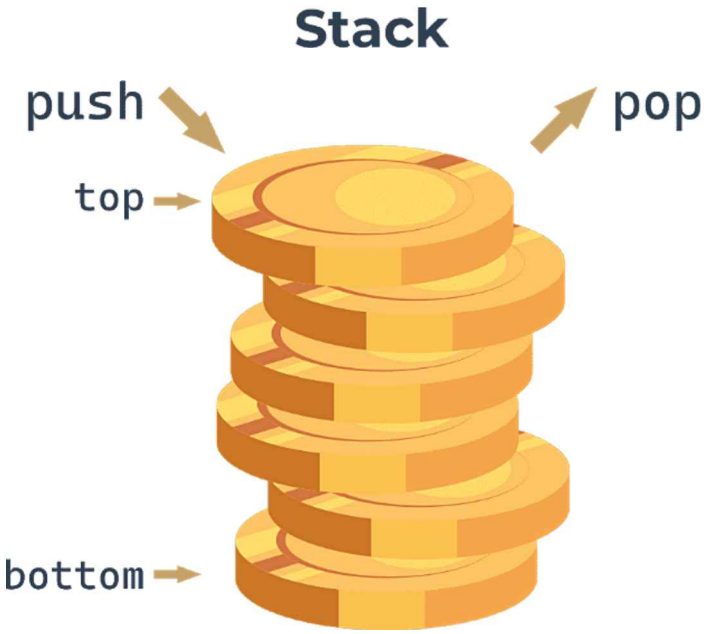
Question 3: Can you name one of your classes just "class"?

Check Answers

FOURTEEN – A SHORT JOURNEY FROM PROCEDURAL TO OBJECT APPROACH

Let's talk about stacks. A stack is a structure developed to store data in a very specific way. Imagine a stack of coins. You aren't able to put a coin anywhere else but on the top of the stack. Similarly, you can't get a coin off the stack from any place other than the top of the stack. If you want to get the coin that lies on the bottom, you have to remove all the coins from the higher levels.

The alternative name for a stack (but only in IT terminology) is LIFO. It's an abbreviation for a very clear description of the stack's behavior: Last In – First Out. The coin that came last onto the stack will leave first.



A stack is an object with two elementary operations, conventionally named push (when a new element is put on the top) and pop (when an existing element is taken away from the top). Stacks are used very often in many classical algorithms, and it's hard to imagine the implementation of many widely used tools without the use of stacks. Let's implement a stack in Python. This will be a very simple stack, and we'll show you how to do it in two independent approaches: procedural and objective. Let's start with the first one.

The stack – the procedural approach

First, you have to decide how to store the values which will arrive onto the stack. We suggest using the simplest of methods, and employing a list for this job. Let's assume that the size of the stack is not limited in any way. Let's also assume that the last element of the list stores the top element. The stack itself is already created:

```
1  stack = []
2
```

We're ready to define a function that puts a value onto the stack. Here are the presuppositions for it: the name for the function is `push`; the function gets one parameter (this is the value to be put onto the stack); the function returns nothing; the function appends the parameter's value to the end of the stack; This is how we've done it – take a look:

```
1  def push(val):
2      stack.append(val)
3
```

Now it's time for a function to take a value off the stack. This is how you can do it: the name of the function is `pop`; the function doesn't get any parameters; the function returns the value taken from the stack the function reads the value from the top of the stack and removes it.

The function is here:

```
1  def pop():
2      val = stack[-1]
3      del stack[-1]
4      returnval
5
```

NOTE: The function doesn't check if there is any element in the stack.

Let's assemble all the pieces together to set the stack in motion. The complete program pushes three numbers onto the stack, pulls them off, and prints their values on the screen. You can see it in the following code.

```
stack = []


def push(val):
    stack.append(val)


def pop():
    val = stack[-1]
    del stack[-1]
    return val


push(3)
push(2)
push(1)

print(pop())
print(pop())
print(pop())

```

The program outputs the following text to the screen:

```
1
2
3
```

Test it.

The stack – the procedural approach vs. the object-oriented approach

The procedural stack is ready. Of course, there are some weaknesses, and the implementation could be improved in many ways (harnessing exceptions to work is a good idea), but in general the stack is fully implemented, and you can use it if you need to. But the more often you use it, the more disadvantages you'll encounter.

The essential variable (the stack list) is highly vulnerable; anyone can modify it in an uncontrollable way, destroying the stack, in effect; this doesn't mean that it's been done maliciously – on the contrary, it may happen as a result of carelessness, e.g. when somebody confuses variable names; imagine that you have accidentally written something like this:

```
stack[0] = 0

```

Furthermore, the functioning of the stack will be completely disorganized. It may also happen that one day you need more than one stack; you'll have to create another list for the stack's storage, and probably other `push` and `pop` functions too. You might also need not only `push` and `pop` functions, but also some other conveniences; you could certainly implement them, but try to imagine what would happen if you had dozens of separately implemented stacks.

The object-oriented approach delivers solutions for each of these problems. Firstly, it provides the ability to hide (protect) selected values against unauthorized access is called encapsulation; the encapsulated values can be neither accessed nor modified if you want to use them exclusively. In addition, when you have a class implementing all the needed stack behaviors, you can produce as many stacks as you want; you needn't copy or replicate any part of the code. Finally, you have the ability to enrich the stack with new functions comes from inheritance; you can create a new class (a subclass) which inherits all the existing traits from the superclass, and adds some new ones.

Procedural Approach vs. Objective Approach

Let's now write a brand new stack implementation from scratch. This time, we'll use the object approach, guiding you step by step into the world of object-oriented programming.

The stack – the object approach

Of course, the main idea remains the same. We'll use a list as the stack's storage. We only have to know how to put the list into the class. Let's start from the absolute beginning – this is how the object stack begins

```
1  class Stack:
2
```

Now, we expect two things from it: first, we want the class to have one property as the stack's storage – we have to "install" a list inside each object of the class (note: each object has to have its own list – the list mustn't be shared among different stacks); ad second, we want the list to be hidden from the class users' sight.

How is this done? In contrast to other programming languages, Python has no means of allowing you to declare such a property just like that. Instead, you need to add a specific statement or instruction. The properties have to be added to the class manually. How do you guarantee that such an activity takes place every time the new stack is created? There is a simple way to do it – you have to equip the class with a specific function – its specificity is dual: it has to be named in a strict way; and it is invoked implicitly, when the new object is created.

Such a function is called a constructor, as its general purpose is to construct a new object. The constructor should know everything about the object's structure, and must perform all the needed initializations. Let's add a very simple constructor to the new class. Take a look at the snippet:

```
class Stack:

    def __init__(self):

        print("Hi!")



stack_object = Stack()
```

And now:

- the constructor's name is always `__init__`;
- it has to have at least one parameter (we'll discuss this later); the parameter is used to represent the newly created object – you can use the parameter to manipulate the object, and to enrich it with the needed properties; you'll make use of this soon;
- note: the obligatory parameter is usually named `self` – it's only a convention, but you should follow it – it simplifies the process of reading and understanding your code.

Run the following code.

```
1  class Stack: # Defining the Stack class.
2      def __init__(self):  # Defining the constructor function.
3          print("Hi!")
4
5
6  stack_object = Stack()  # Instantiating the object.
7
```

Here is its output:

```
Hi!
```

NOTE: There is no trace of invoking the constructor inside the code. It has been invoked implicitly and automatically. Let's make use of that now.

Any change you make inside the constructor that modifies the state of the `self parameter` will be reflected in the newly created object. This means you can add any property to the object and the property will remain there until the object finishes its life or the property is explicitly removed. Now let's add just one property to the new object – a list for a stack. We'll name it `stack_list`. Just like the code here:

```
1  class Stack:
2      def __init__(self):
3          self.stack_list = []
4
5
6  stack_object = Stack()
7  print(len(stack_object.stack_list))
8
```

NOTE: We've used dot notation, just like when invoking methods; this is the general convention for accessing an object's properties. You need to name the object, put a dot (`.`) after it, and specify the desired property's name; don't use parentheses! You don't want to invoke a method – you want to access a property. If you set a property's value for the very first time, like in the constructor, you are creating it; from that moment on, the object has got the property and is ready to use its value.

We've done something more in the code – we've tried to access the `stack_list` property from outside the class immediately after the object has been created; we want to check the current length of the stack – have we succeeded? Yes, we have – the code produces the following output:

```
0
```

This is not what we want from the stack. We prefer `stack_list` to be hidden from the outside world. Is that possible? Yes, and it's simple, but not very intuitive. Take a look – we've added two underscores before the `stack_list` name – nothing more:

```
class Stack:
    def __init__(self):
        self.__stack_list = []


stack_object = Stack()
print(len(stack_object.__stack_list))

```

The change invalidates the program. Why? When any class component has a name starting with two underscores (`__`), it becomes private – this means that it can be accessed only from within the class. You cannot see it from the outside world. This is how Python implements the encapsulation concept. Run the program to test our assumptions – an `AttributeError` exception should be raised.

The object approach: a stack from scratch

Now it's time for the two functions (methods) implementing the *push* and *pop* operations. Python assumes that a function of this kind (a class activity) should be immersed inside the class body – just like a constructor. We want to invoke these functions to `push` and `pop` values. This means that they should both be accessible to every class's user (in contrast to the previously constructed list, which is hidden from the ordinary class's users).

Such a component is called public, so you can't begin its name with two (or more) underscores. There is one more requirement – the name must have no more than one trailing underscore. As no trailing underscores at all fully meets the requirement, you can assume that the name is acceptable. The functions themselves are simple. Take a look:

```
class Stack:
    def __init__(self):
        self.__stack_list = []


def push(self, val):
        self.__stack_list.append(val)


def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


stack_object = Stack()

stack_object.push(3)
stack_object.push(2)
stack_object.push(1)

print(stack_object.pop())
print(stack_object.pop())
print(stack_object.pop())

```

However, there's something really strange in the code. The functions look familiar, but they have more parameters than their procedural counterparts. Here, both functions have a parameter named `self` at the first position of the parameters list. Is it needed? Yes, it is.

All methods have to have this parameter. It plays the same role as the first constructor parameter. It allows the method to access entities (properties and activities/methods) carried out by the actual object. You cannot omit it. Every time Python invokes a method, it implicitly sends the current object as the first argument. This means that a method is obligated to have at least one parameter, which is used by Python itself – you don't have any influence on it.

If your method needs no parameters at all, this one must be specified anyway. If it's designed to process just one parameter, you have to specify two, and the first one's role is still the same. There is one more thing that requires explanation – the way in which methods are invoked from within the `__stack_list` variable. Fortunately, it's much simpler than it looks: the first stage delivers the object as a whole → `self`; next, you need to get to the `__stack_list` list → `self.__stack_list`; and with `__stack_list` ready to be used, you can perform the third and last step → `self.__stack_list.append(val)`. The class declaration is complete, and all its components have been listed. The class is ready for use.

Having such a class opens up some new possibilities. For example, you can now have more than one stack behaving in the same way. Each stack will have its own copy of private data, but will utilize the same set of methods. This is exactly what we want for this example. Analyze the code:

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


stack_object_1 = Stack()
stack_object_2 = Stack()

stack_object_1.push(3)
stack_object_2.push(stack_object_1.pop())

print(stack_object_2.pop())

```

There are two stacks created from the same base class. They work independently. You can make more of them if you want to. Run the code and see what happens. Carry out your own experiments. Analyze the following snippet – we've created three objects of the class Stack. Next, we've juggled them up. Try to predict the value to be outputted.

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


little_stack = Stack()
another_stack = Stack()
funny_stack = Stack()

little_stack.push(1)
another_stack.push(little_stack.pop() + 1)
funny_stack.push(another_stack.pop() - 2)

print(funny_stack.pop())

```

So, what's the result? Run the program and check if you're right. Now let's go a little further. Let's add a new class for handling stacks. The new class should be able to evaluate the sum of all the elements currently stored in the stack. We don't want to modify the previously defined stack. It's already good enough in its applications, and we don't want it changed in any way. We want a new stack with new capabilities. In other words, we want to construct a subclass of the already existing `Stack` class.

The first step is easy: just define a new subclass pointing to the class which will be used as the superclass. This is what it looks like:

```
class AddingStack(Stack):
    pass

```

The class doesn't define any new component yet, but that doesn't mean that it's empty. It gets all the components defined by its superclass – the name of the superclass is written before the colon directly after the new class name.

From the new stack, we want the push method not only to `push` the value onto the stack but also to add the value to the `sum` variable;, and we want the pop function not only to `pop` the value off the stack but also to subtract the value from the `sum` variable.

Firstly, let's add a new variable to the class. It'll be a private variable, like the stack list. We don't want anybody to manipulate the `sum` value. As you already know, adding a new property to the class is done by the constructor. You already know how to do that, but there is something really intriguing inside the constructor. Take a look:

```
class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0

```

The second line of the constructor's body creates a property named `__sum` – it will store the total of all the stack's values. But the line before it looks different. What does it do? Is it really necessary? Yes, it is. Contrary to many other languages, Python forces you to explicitly invoke a superclass's constructor. Omitting this point will have harmful effects – the object will be deprived of the `__stack_list` list. Such a stack will not function properly. This is the only time you can invoke any of the available constructors explicitly – it can be done inside the subclass's constructor.

Note the syntax:

- you specify the superclass's name (this is the class whose constructor you want to run)
- you put a dot (`.`)after it;
- you specify the name of the constructor;
- you have to point to the object (the class's instance) which has to be initialized by the constructor – this is why you have to specify the argument and use the `self` variable here; note: invoking any method (including constructors) from outside the class never requires you to put the `self` argument at the argument's list – invoking a method from within the class demands explicit usage of the `self` argument, and it has to be put first on the list.

NOTE: It's generally a recommended practice to invoke the superclass's constructor before any other initializations you want to perform inside the subclass. This is the rule we have followed in the snippet.

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0

```

Secondly, let's add two methods. But let us ask you: is it really adding? We have these methods in the superclass already. Can we do something like that? Yes, we can. It means that we're going to change the functionality of the methods, not their names. We can say more precisely that the interface (the way in which the objects are handled) of the class remains the same when changing the implementation at the same time. Let's start with the implementation of the `push` function. We expect it to add the value to the `__sum` variable and to push the value onto the stack.

NOTE: The second activity has already been implemented inside the superclass – so we can use that. Furthermore, we have to use it, as there's no other way to access the `__stackList` variable. This is what the `push` method looks like in the subclass:

```
def push(self, val):
    self.__sum+= val
    Stack.push(self, val)

```

Note the way we've invoked the previous implementation of the `push` method (the one available in the superclass). We have to specify the superclass's name; this is necessary in order to clearly indicate the class containing the method, to avoid confusing it with any other function of the same name; and we have to specify the target object and to pass it as the first argument (it's not implicitly added to the invocation in this context.)

We say that the `push` method has been overridden – the same name as in the superclass now represents a different functionality.

```
class Stack:
    def __init__(self):
        self.__stackList = []

    def push(self, val):
        self.__stackList.append(val)

    def pop(self):
        val = self.__stackList[-1]
        del self.__stackList[-1]
        return val


class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0


# Enter code here.

```

This is the new `pop` function:

```
def pop(self):
    val = Stack.pop(self)
    self.__sum-= val
    returnval

```

So far, we've defined the `__sum` variable, but we haven't provided a method to get its value. It seems to be hidden. How can we reveal it and do it in a way that still protects it from modifications? We have to define a new method. We'll name it `get_sum`. It will return the `__sum` value. Here it is:

```
def get_sum(self):
    return self.__sum

```

So, let's look at the following program. The complete code of the class is there. We can check if it's functioning now, and we do it with the help of a very few additional lines of code.

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0

    def get_sum(self):
        return self.__sum

    def push(self, val):
        self.__sum += val
        Stack.push(self, val)

    def pop(self):
        val = Stack.pop(self)
        self.__sum-= val
        return val


stack_object = AddingStack()

for i in range(5):
    stack_object.push(i)
print(stack_object.get_sum())

for i in range(5):
    print(stack_object.pop())

```

As you can see, we add five subsequent values onto the stack, print their sum, and take them all off the stack.

Summary

1. A stack is an object designed to store data using the LIFO model. The stack usually performs at least two operations, named `push()` and `pop()`.

2. Implementing the stack in a procedural model raises several problems which can be solved by the techniques offered by OOP (Object Oriented Programming).

3. A class method is actually a function declared inside the class and able to access all the class's components.

4. The part of the Python class responsible for creating new objects is called the constructor, and it's implemented as a method of the name `__init__`.

5. Each class method declaration must contain at least one parameter (always the first one) usually referred to as `self`, and is used by the objects to identify themselves.

6. If we want to hide any of a class's components from the outside world, we should start its name with `__`. Such components are called private.

Quiz

Question 1: Assuming that there is a class named `Snakes`, write the very first line of The `Python` class declaration, expressing the fact that the new class is actually a subclass of Snake.

Question 2: Something is missing from the following declaration – what?

```
class Snakes:
    def __init__():
        self.sound = 'Sssssss'
```

Question 3: Modify the code to guarantee that the venomous property is private.

```
class Snakes:
    def __init__(self):
```

```
        self.venomous = True
```

Check Answers

LAB: Counting stack

We showed you recently how to extend *Stack* possibilities by defining a new class (i.e. a subclass) which retains all inherited traits and adds some new ones. Your task is to extend the `Stack` class behavior in such a way so that the class is able to count all the elements that are pushed and popped — we assume that counting pops is enough. Use the `Stack` class we've provided. Follow the hints:

- introduce a property designed to count pop operations and name it in a way which guarantees it is hidden;
- initialize it to zero inside the constructor;
- provide a method which returns the value currently assigned to the counter (name it `get_counter()`).

Complete the following code. Run it to check whether your code outputs 100.

```
class Stack:
    def __init__(self):
        self.__stk = []

    def push(self, val):
        self.__stk.append(val)

    def pop(self):
        val = self.__stk[-1]
        del self.__stk[-1]
        return val


class CountingStack(Stack):
    def __init__(self):
    #
    # Fill the constructor with appropriate actions.
    #

    def get_counter(self):
    #
    # Present the counter's current value to the world.
    #

    def pop(self):
    #
    # Do pop and update the counter.
    #


stk = CountingStack()
for i in range(100):
    stk.push(i)
    stk.pop()
print(stk.get_counter())

```

Check Sample Solution

LAB: Queue aka FIFO

As you already know, a *stack* is a data structure realizing the LIFO (Last In – First Out) model. It's easy and you've already grown perfectly accustomed to it. Let's try something new now. A *queue* is a data model characterized by the term FIFO: First In – First Out.

NOTE: A regular queue (line) you know from shops or post offices works exactly in the same way – a customer who came first is served first too. Your task is to implement the `Queue` class with two basic operations:

- `put(element)`, which puts an element at end of the queue;
- `get()`, which takes an element from the front of the queue and returns it as the result (the queue cannot be empty to successfully perform it.)

HINT Use a list as your storage (just like we did with the stack); `put()` should append elements to the beginning of the list, while `get()` should remove the elements from the end of the list; define a new exception named `QueueError` (choose an exception to derive it from) and raise it when `get()` tries to operate on an empty list.

Complete the following code. Run it to check whether its output is similar to ours.

Expected output

```
1
dog
False
Queue error
```

Code

```
class QueueError(???):  # Choose base class for the new exception.
    #
    # Write code here
    #


class Queue:
    def __init__(self):
        #
        # Write code here
        #

    def put(self, elem):
        #
        # Write code here
        #

    def get(self):
        #
        # Write code here
        #


que = Queue()
que.put(1)
que.put("dog")
que.put(False)
try:
    for i in range(4):
        print(que.get())
except:
    print("Queue error")

```

Check Sample Solution

LAB: Queue aka FIFO: part 2

Your task is to slightly extend the `Queue` class's capabilities. We want it to have a parameter less method that returns `True` if the queue is empty and `False` otherwise. Complete the code we've provided. Run it to check whether it outputs a similar result to ours. You can copy the code we used in the previous lab:

Expected output

```
1
dog
False
Queue empty
```

Code

```
class QueueError(???):
    pass


class Queue:
    #
    # Code from the previous lab.
    #


class SuperQueue(Queue):
    #
    # Write new code here.
    #


que = SuperQueue()
que.put(1)
que.put("dog")
que.put(False)
for i in range(4):
    if not que.isempty():
        print(que.get())
    else:
        print("Queue empty")

```

Check Sample Solution

FIFTEEN – OOP: PROPERTIES

In general, a class can be equipped with two different kinds of data to form a class's properties. You already saw one of them when we were looking at stacks. This kind of class property exists when and only when it is explicitly created and added to an object. As you already know, this can be done during the object's initialization, performed by the constructor. Moreover, it can be done at any moment of the object's life. Furthermore, any existing property can be removed at any time.

Such an approach has some important consequences. Different objects of the same class may possess different sets of properties. There must also be a way to safely check if a specific object owns the property you want to utilize, unless you want to provoke an exception, which is always worth considering. Finally, each object carries its own set of properties – they don't interfere with one another in any way.

Such variables (properties) are called instance variables. The word instance suggests that they are closely connected to the objects (which are class instances), not to the classes themselves. Let's take a closer look at them. Here is an example:

```
class ExampleClass:
    def __init__(self, val = 1):
        self.first = val

    def set_second(self, val):
        self.second = val


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)

example_object_2.set_second(3)

example_object_3 = ExampleClass(4)
example_object_3.third= 5

print(example_object_1.__dict__)
print(example_object_2.__dict__)
print(example_object_3.__dict__)

```

It needs one additional explanation before we go into any more detail. Take a look at the last three lines of the code. Python objects, when created, are gifted with a small set of predefined properties and methods. Each object has got them, whether you want them or not. One of them is a variable named `__dict__` (it's a dictionary). The variable contains the names and values of all the properties (variables) the object is currently carrying. Let's make use of it to safely present an object's contents.

Let's dive into the code now:

– the class named `ExampleClass` has a constructor, which unconditionally creates an instance variable named `first`, and sets it with the value passed through the first argument (from the class user's perspective) or the second argument (from the constructor's perspective); note the default value of the parameter – any trick you can do with a regular function parameter can be applied to methods, too;
– the class also has a method which creates another instance variable, named `second`;
– we've created three objects of the class `ExampleClass`, but all these instances differ:

- `example_object_1` only has the property named `first`;
- `example_object_2` has two properties: `first` and `second`;
- `example_object_3` has been enriched with a property named `third` just on the fly, outside the class's code – this is possible and fully permissible.

The program's output clearly shows that our assumptions are correct – here it is:

```
{'first': 1}
{'first': 2, 'second': 3}
{'first': 4, 'third': 5}
```

There is one additional conclusion that should be stated here: modifying an instance variable of any object has no impact on all the remaining objects. Instance variables are perfectly isolated from each other. Take a look at the following modified example.

```
class ExampleClass:
    def __init__(self, val = 1):
        self.__first = val

    def set_second(self, val = 2):
        self.__second = val


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)

example_object_2.set_second(3)

example_object_3 = ExampleClass(4)
example_object_3.__third = 5


print(example_object_1.__dict__)
print(example_object_2.__dict__)
print(example_object_3.__dict__)

```

It's nearly the same as the previous one. The only difference is in the property names. We've added two underscores (__) in front of them. As you know, such an addition makes the instance variable private – it becomes inaccessible from the outer world. The actual behavior of these names is a bit more complicated, so let's run the program. This is the output:

```
{'_ExampleClass__first': 1}
{'_ExampleClass__first': 2, '_ExampleClass__second': 3}
{'_ExampleClass__first': 4, '__third': 5}
```

Can you see these strange names full of underscores? Where did they come from? When Python sees that you want to add an instance variable to an object and you're going to do it inside any of the object's methods, it mangles the operation in the following way: it puts a class name before your name; and it puts an additional underscore at the beginning. This is why the `__first` becomes `_ExampleClass__first`. The name is now fully accessible from outside the class. You can run a code like this:

```
print(example_object_1._ExampleClass__first)

```

And you'll get a valid result with no errors or exceptions. As you can see, making a property private is limited. The mangling won't work if you add a private instance variable outside the class code. In this case, it'll behave like any other ordinary property.

Class variables

A class variable is a property which exists in just one copy and is stored outside any object.

NOTE: no instance variable exists if there is no object in the class; a class variable exists in one copy even if there are no objects in the class.

Class variables are created differently to their instance siblings. The example will tell you more:

```
class ExampleClass:
    counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.counter += 1


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)
example_object_3 = ExampleClass(4)

print(example_object_1.__dict__, example_object_1.counter)
print(example_object_2.__dict__, example_object_2.counter)
print(example_object_3.__dict__, example_object_3.counter)

```

There is an assignment in the first list of the class definition – it sets the variable named `counter` to `0`; initializing the variable inside the class but outside any of its methods makes the variable a class variable. Accessing such a variable looks the same as accessing any instance attribute – you can see it in the constructor body; as you can see, the constructor increments the variable by one; in effect, the variable counts all the created objects. Running the code will cause the following output:

```
{'_ExampleClass__first': 1} 3
```

```
{'_ExampleClass__first': 2} 3
{'_ExampleClass__first': 4} 3
```

Two important conclusions come from the example: the first is that class variables aren't shown in an object's `__dict__`. This is natural as class variables aren't parts of an object, but you can always try to look into the variable of the same name, albeit at the class level – we'll show you this very soon. The second is that a class variable always presents the same value in all class instances (objects).

Mangling a class variable's name has the same effects as those you're already familiar with. Look at the following example. Can you guess its output?

```
1  class ExampleClass:
2      __counter = 0
3      def __init__(self, val = 1):
4          self.__first = val
5          ExampleClass.__counter += 1
6
7
8  example_object_1 = ExampleClass()
9  example_object_2 = ExampleClass(2)
10 example_object_3 = ExampleClass(4)
11
12 print(example_object_1.__dict__, example_object_1._ExampleClass__counter)
13 print(example_object_2.__dict__, example_object_2._ExampleClass__counter)
14 print(example_object_3.__dict__, example_object_3._ExampleClass__counter)
15
```

Run the program and check if your predictions are correct. Everything works as expected, doesn't it? We told you before that class variables exist even when no class instance (object) had been created. Now we're going to take the opportunity to show you the difference between these two `__dict__` variables, the one from the class and the one from the object. Look at the code. The proof is there.

```
1  class ExampleClass:
2      varia = 1
3      def __init__(self, val):
4          ExampleClass.varia = val
5
6
7  print(ExampleClass.__dict__)
8  example_object = ExampleClass(2)
9
10 print(ExampleClass.__dict__)
11 print(example_object.__dict__)
12
```

Let's take a closer look at it:

1. We define one class named `ExampleClass`;
2. The class defines one class variable named `varia`;
3. The class constructor sets the variable with the parameter's value;
4. Naming the variable is the most important aspect of the example because:

- Changing the assignment to `self.varia = val` would create an instance variable of the same name as the class's one;
- Changing the assignment to `varia = val` would operate on a method's local variable; (we strongly encourage you to test both cases – this will make it easier for you to remember the difference)

5. The first line of the off-class code prints the value of The `ExampleClass.varia` attribute; note – we use the value before the very first object of the class is instantiated.

Run the code and check its output. As you see, the class's `__dict__` contains much more data than its object's counterpart. Most are useless now – the one we want you to check carefully shows the current `varia` value. Note that the object's `__dict__` is empty – the object has no instance variables.

Checking an attribute's existence

Python's attitude to object instantiation raises one important issue – in contrast to other programming languages, you may not expect that all objects of the same class have the same sets of properties. Just like in this example. Look at it carefully.

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


example_object = ExampleClass(1)

print(example_object.a)
print(example_object.b)

```

The object created by the constructor can have only one of two possible attributes: `a` or `b`. Executing the code will produce the following output:

```
1
Traceback (most recent call last):
  File ".main.py", line 11, in
    print(example_object.b)
AttributeError: 'ExampleClass' object has no attribute 'b'
```

As you can see, accessing a non-existing object (class) attribute causes an `AttributeError` exception. The `try-except` instruction gives you the chance to avoid issues with non-existent properties. It's easy – look at the following code.

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


example_object = ExampleClass(1)
print(example_object.a)

try:
    print(example_object.b)
except AttributeError:
    pass

```

As you can see, this action isn't very sophisticated. Essentially, we've just swept the issue under the carpet. Fortunately, there is one more way to cope with the issue. Python provides a function which is able to safely check if any object/class contains a specified property. The function is named hasattr, and expects two arguments to be passed to it, the class or the object being checked, and the name of the property whose existence has to be reported (note: it has to be a string containing the attribute name, not the name alone). The function returns `True` or `False`, and this is how you can utilize it:

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


example_object = ExampleClass(1)
print(example_object.a)

if hasattr(example_object, 'b'):
    print(example_object.b)

```

Don't forget that the `hasattr()` function can operate on classes, too. You can use it to find out if a class variable is available, just like here in the following example. The function returns `True` if the specified class contains a given attribute, and `False` otherwise. Can you guess the code's output? Run it to check your guesses.

```
class ExampleClass:
    attr = 1


print(hasattr(ExampleClass, 'attr'))
print(hasattr(ExampleClass, 'prop'))

```

And one more example – look at this code and try to predict its output:

```
class ExampleClass:
    a = 1
    def __init__(self):
        self.b = 2


example_object = ExampleClass()

print(hasattr(example_object, 'b'))
print(hasattr(example_object, 'a'))
print(hasattr(ExampleClass, 'b'))
print(hasattr(ExampleClass, 'a'))

```

Were you successful? Run the code to check your predictions.

Summary

1. An instance variable is a property whose existence depends on the creation of an object. Every object can have a different set of instance variables. Moreover, they can be freely added to and removed from objects during their lifetime. All object instance variables are stored inside a dedicated dictionary named `__dict__`, contained in every object separately.

2. An instance variable can be private when its name starts with `__`, but don't forget that such a property is still accessible from outside the class using a mangled name constructed as `_ClassName__PrivatePropertyName`..

3. A class variable is a property which exists in exactly one copy, and doesn't need any created object to be accessible. Such variables are not shown as `__dict__` content. All a class's class variables are stored inside a dedicated dictionary named `__dict__`, contained in every class separately.

4. A function named `hasattr()` can be used to determine if any object/class contains a specified property.

For example:

```
class Sample:
    gamma = 0 # Class variable.
    def __init__(self):

        self.alpha = 1 # Instance variable.
        self.__delta = 3 # Private instance variable.


obj = Sample()
obj.beta = 2 # Another instance variable (existing only inside the "obj"
instance.)
print(obj.__dict__)

```

The code outputs:

```
{'alpha': 1, '_Sample__delta': 3, 'beta': 2}
```

Quiz

Question 1: Which of The `Python` class properties are instance variables and which are class variables? Which of them are private?

```
class Python:
    population = 1
    victims = 0
    def __init__(self):
        self.length_ft = 3
        self.__venomous = False
```

Question 2: You're going to negate The `__venomous` property of The `version_2` object, ignoring the fact that the property is private. How will you do this?

```
version_2 = Python()
```

Question 3: Write an expression which checks if The `version_2` object contains an instance property named `constrictor` (yes, constrictor!).

Check Answers

SIXTEEN – OOP: METHODS

Let's summarize all the facts regarding the use of methods in Python classes. As you already know, a method is a function embedded inside a class. There is one fundamental requirement – a method is obliged to have at least one parameter. There are no such things as parameterless methods – a method may be invoked without an argument, but not declared without parameters.

The first, or only, parameter is usually named `self`. We suggest you follow the convention – it's commonly used, and you'll cause a few surprises by using other names for it. The name `self` suggests the parameter's purpose. It identifies the object for which the method is invoked. If you're going to invoke a method, don't pass the argument for the `self` parameter. Python will set it for you. This example shows the difference.

```
1  class Classy:
2      def method(self):
3          print("method")
4
5
6  obj = Classy()
7  obj.method()
8
```

The code outputs:

```
method
```

Note the way we've created the object – we've treated the class name like a function, returning a newly instantiated object of the class. If you want the method to accept parameters other than self, you should place them after `self` in the method's definition and deliver them during invocation without specifying `self`, as previously.

Take a look at the following code:

```
1  class Classy:
2      def method(self, par):
3          print("method:", par)
4
5
6  obj = Classy()
7  obj.method(1)
8  obj.method(2)
9  obj.method(3)
10
```

The code outputs:

```
method: 1
method: 2
method: 3
```

The `self` parameter is used to obtain access to the object's instance and class variables.

```
1  class Classy:
2      varia = 2
3      def method(self):
4          print(self.varia, self.var)
5
6
7  obj = Classy()
8  obj.var = 3
9  obj.method()
10
```

The code outputs:

```
2 3
```

The `self` parameter is also used to invoke other object/class methods from inside the class. Just like here:

```
class Classy:
    def other(self):
        print("other")

    def method(self):
        print("method")
        self.other()


obj = Classy()
obj.method()

```

The code outputs:

```
method
other
```

If you name a method like this: `__init__`, it won't be a regular method – it will be a constructor. If a class has a constructor, it is invoked automatically and implicitly when the object of the class is instantiated. The constructor is obliged to have the `self` parameter (it's set automatically, as usual), and it may, but doesn't need to, have more parameters than just `self`; if this happens, how the class name is used to create the object must reflect the `__init__` definition. The constructor can be used to set up the object, that is, properly initialize its internal state, create instance variables, instantiate any other objects if their existence is needed, etc. Look at this code. This example shows a very simple constructor at work.

```
class Classy:
    def __init__(self, value):
        self.var = value


obj_1 = Classy("object")

print(obj_1.var)

```

Run it. The code outputs:

```
object
```

Note that the constructor cannot return a value, as it is designed to return a newly created object and nothing else, nor can it be invoked directly either from the object or from inside the class. You can invoke a constructor from any of the object's subclasses, but we'll discuss this issue later. As `__init__` is a method, and a method is a function, you can do the same tricks with constructors/methods as you do with ordinary functions. The example shows how to define a constructor with a default argument value.

```
class Classy:
    def __init__(self, value = None):
        self.var = value


obj_1 = Classy("object")
obj_2 = Classy()

print(obj_1.var)
print(obj_2.var)

```

The code outputs:

```
object
None
```

Everything we've said about property name mangling applies to method names – a method whose name starts with is (partially) hidden. The following example shows this effect:

```
class Classy:
    def visible(self):
        print("visible")

    def __hidden(self):
        print("hidden")


obj = Classy()
obj.visible()

try:
    obj.__hidden()
except:
    print("failed")

obj._Classy__hidden()

```

The code outputs:

```
visible
failed
hidden
```

Run the program, and test it.

The inner life of classes and objects

Each Python class and each Python object is pre-equipped with a set of useful attributes which can be used to examine its capabilities. You already know one of these – it's the `__dict__` property. Let's observe how it deals with methods – look at the code.

```
class Classy:
    varia = 1
    def __init__(self):
        self.var = 2

    def method(self):
        pass

    def __hidden(self):
        pass


obj = Classy()

print(obj.__dict__)
print(Classy.__dict__)

```

Run it to see what it outputs. Check the output carefully. Find all the defined methods and attributes. Locate the context in which they exist: inside the object or inside the class. `__dict__` is a dictionary. Another built-in property worth mentioning is `__name__`, which is a string. The property contains the name of the class. It's nothing exciting, just a string.

NOTE: The `__name__` attribute is absent from the object – **it exists only inside classes**.

If you want to find the class of a particular object, you can use a function named `type()`, which is able (among other things) to find a class which has been used to instantiate any object. Look at the code, run it, and see for yourself.

```
class Classy:
    pass


print(Classy.__name__)
obj = Classy()
print(type(obj).__name__)

```

The code outputs:

```
Classy
```

```
Classy
```

Note that a statement like this one will cause an error.

```
1  print(obj.__name__)
2
```

`__module__` is a string – it stores the name of the module which contains the definition of the class. Let's check it – run the following code.

```
1  class Classy:
2      pass
3
4
5  print(Classy.__module__)
6  obj = Classy()
7  print(obj.__module__)
8
```

The code outputs:

```
__main__
__main__
```

As you know, any module named `__main__` is actually not a module, but the file currently being run. `__bases__` is a tuple. The tuple contains classes (not class names) which are direct superclasses for the class. The order is the same as that used inside the class definition. We'll show you only a very basic example, as we want to highlight how inheritance works. Moreover, we're going to show you how to use this attribute when we discuss the object approach aspects of exceptions.

NOTE: Only classes have this attribute – objects don't.

We've defined a function named `printBases()`, designed to present the tuple's contents clearly. Look at the code. Analyze it and run it.

```
1  class SuperOne:
2      pass
3
4
5  class SuperTwo:
6      pass
7
8
9  class Sub(SuperOne, SuperTwo):
10     pass
11
12
13 def printBases(cls):
14     print('( ', end='')
15
16     for x in cls.__bases__:
17         print(x.__name__, end=   )
18     print(')')
19
20
21 printBases(SuperOne)
22 printBases(SuperTwo)
23 printBases(Sub)
24
```

It will output:

```
( object )
( object )
( SuperOne SuperTwo )
```

NOTE: A class without explicit superclasses points to an object (a predefined Python class) as its direct ancestor.

Reflection and introspection

All these means allow the Python programmer to perform two important activities specific to many objective languages. The first is introspection, which is the ability of a program to examine the type or properties of an object at runtime. The second is reflection, which goes a step further, and is the ability of a program to manipulate the values, properties and/or functions of an object at runtime.

In other words, you don't have to know a complete class/object definition to manipulate the object, as the object and/or its class contain the metadata allowing you to recognize its features during program execution.

Investigating classes

What can you find out about classes in Python? the answer is simple – everything. Both reflection and introspection enable a programmer to do anything with any object, no matter where it comes from.

Analyze the code.

```
1  class MyClass:
2      pass
3
4
5  obj = MyClass()
6  obj.a = 1
7  obj.b = 2
8  obj.i = 3
9  obj.ireal = 3.5
10 obj.integer = 4
11 obj.z = 5
12
13
14 def incIntsI(obj):
15     for name in obj.__dict__.keys():
16         if name.startswith('i'):
17             val = getattr(obj, name)
18             if isinstance(val, int):
19                 setattr(obj, name, val + 1)
20
21
22 print(obj.__dict__)
23 incIntsI(obj)
24 print(obj.__dict__)
25
```

The function named `incIntsI()` gets an object of any class, scans its contents in order to find all integer attributes with names starting with `i`, and increments them by one. Impossible? Not at all! This is how it works:

- line 1: define a very simple class...
- lines 3 through 10: ... and fill it with some attributes;
- line 12: this is our function!
- line 13: scan The `__dict__` attribute, looking for all attribute names;
- line 14: if a name starts with `i`...
- line 15: ... use The `getattr()` function to get its current value; note: `getattr()` takes two arguments: an object, and its property name (as a string), and returns the current attribute's value;
- line 16: check if the value is of type integer, and use the function `isinstance()` for this purpose (we'll discuss this later);
- line 17: if the check goes well, increment the property's value by making use of The `setattr()` function; the function takes three arguments: an object, the property name (as a string), and the property's new value.

The code outputs:

```
{'a': 1, 'integer': 4, 'b': 2, 'i': 3, 'z': 5, 'ireal': 3.5}
{'a': 1, 'integer': 5, 'b': 2, 'i': 4, 'z': 5, 'ireal': 3.5}
```

That's all!

Summary

1. A method is a function embedded inside a class. The first (or only) parameter of each method is usually named `self`, which is designed to identify the object for which the method is invoked in order to access the object's properties or invoke its methods.

2. If a class contains a constructor (a method named `__init__`) it cannot return any value and cannot be invoked directly.

3. All classes (but not objects) contain a property named `__name__`, which stores the name of the class. Additionally, a property named `__module__` stores the name of the module in which the class has been declared, while the property named `__bases__` is a tuple containing a class's superclasses.

For example:

```
1  class Sample:
2      def __init__(self):
3          self.name = Sample.__name__
4      def myself(self):
5          print("My name is "+ self.name + " living in a " + Sample.__module__)
6
7
8  obj = Sample()
9  obj.myself()
10
```

The code outputs:

```
My name is Sample living in a __main__
```

Quiz

Question 1: The declaration of The `Snake` class is given below. Enrich the class with a method named `increment()`, adding `1` to The `__victims` property.

```
class Snake:
    def __init__(self):
        self.victims = 0
```

Question 2: Redefine The `Snake` class constructor so that is has a parameter to initialize The `victims` field with a value passed to the object during construction.

Question 3: Can you predict the output of the following code?

```
class Snake:
    pass


class Python(Snake):
    pass


print(Python.__name__, 'is a', Snake.__name__)
print(Python.__bases__[0].__name__, 'can be', Python.__name__)
```

Check Answers

LAB: The Timer class

We need a class able to count seconds. Easy? Not as easy as you may think, as we're going to have some specific requirements. Read them carefully as the class you're about write will be used to launch rockets carrying international missions to Mars. It's a great responsibility. We're counting on you! Your class will be called `Timer`. Its constructor accepts three arguments representing hours (a value from the range [0..23] – we will be using military time), minutes (from the range [0..59]) and seconds (from the range [0..59]). Zero is the default value for all of these parameters. There is no need to perform any validation checks. The class itself should provide the following facilities:

- objects of the class should be "printable", i.e. they should be able to implicitly convert themselves into strings of the following form: "hh:mm:ss", with leading zeros added when any of the values is less than 10;
- the class should be equipped with parameterless methods called `next_second()` and `previous_second()`, incrementing the time stored inside the objects by +1/-1 second respectively.

Use the following hints: all object properties should be private, and you should consider writing a separate function (not method!) to format the time string. Complete the template we've provided. Run your code and check whether the output looks the same as ours.

Expected output

```
23:59:59
00:00:00
23:59:59
```

Code

```
class Timer:
    def __init__( ??? ):
        #
        # Write code here
        #

    def __str__(self):
        #
        # Write code here
        #

    def next_second(self):
        #
        # Write code here
        #

    def prev_second(self):
        #
        # Write code here
        #


timer = Timer(23, 59, 59)
print(timer)
timer.next_second()
print(timer)
timer.prev_second()
print(timer)

```

Check Sample Solution

LAB: Days of the week

Your task is to implement a class called `Weeker`. Yes, your eyes don't deceive you – this name comes from the fact that objects of that class will be able to store and to manipulate the days of the week. The class constructor accepts one argument – a string. The string represents the name of the day of the week and the only acceptable values must come from the following set:

```
Mon Tue Wed Thu Fri Sat Sun
```

Invoking the constructor with an argument from outside this set should raise the `WeekDayError` exception (define it yourself; don't worry, we'll talk about the objective nature of exceptions soon). The class should provide the following facilities:

- objects of the class should be "printable", i.e. they should be able to implicitly convert themselves into strings of the same form as the constructor arguments;
- the class should be equipped with one-parameter methods called `add_days(n)` and `subtract_days(n)`, with n being an integer number and updating the day of week stored inside the object in the way reflecting the change of date by the indicated number of days, forward or backward.

- all the object's properties should be private;

Complete the template we've provided and run your code and check whether your output looks the same as ours.

Expected output

```
Mon
Tue
Sun
Sorry, I can't serve your request.
```

Code

```
class WeekDayError(Exception):
    pass


class Weeker:
    #
    # Write code here.
    #

    def __init__(self, day):
        #
        # Write code here.
        #

    def __str__(self):
        #
        # Write code here.
        #

    def add_days(self, n):
        #
        # Write code here.
        #

    def subtract_days(self, n):
        #
        # Write code here.
        #


try:
    weekday = Weeker('Mon')
    print(weekday)
    weekday.add_days(15)
    print(weekday)
    weekday.subtract_days(23)
    print(weekday)
    weekday = Weeker('Monday')
except WeekDayError:
    print("Sorry, I can't serve your request.")

```

Check Sample Solution

LAB: Points on a plane

Let's visit a very special place – a plane with the Cartesian coordinate system (you can learn more about this concept here: https://en.wikipedia.org/wiki/Cartesian_coordinate_system). Each point located on the plane can be described as a pair of coordinates customarily called x and y. We want you to write a Python class which stores both coordinates as float numbers. Moreover, we want the objects of this class to evaluate the distances between any of the two points situated on the plane.

The task is rather easy if you employ the function named `hypot()` (available through the *math* module) which evaluates the length of the hypotenuse of a right triangle (more details at https://en.wikipedia.org/wiki/Hypotenuse) and https://docs.python.org/3.7/library/math.html#trigonometric-functions.

This is how we imagine the class:

- it's called `Point`;
- its constructor accepts two arguments (*x* and *y* respectively), both of which default to zero;
- all the properties should be private;
- the class contains two parameterless methods called `getx()` and `gety()`, which return each of the two coordinates (the coordinates are stored privately, so they cannot be accessed directly from within the object);

- the class provides a method called `distance_from_xy(x,y)`, which calculates and returns the distance between the point stored inside the object and the other point given as a pair of floats;
- the class provides a method called `distance_from_point(point)`, which calculates the distance (like the previous method), but the other point's location is given as another `Point` class object;

Complete the template we've provided and run your code and check whether your output looks the same as ours.

Expected output

```
1.4142135623730951
1.4142135623730951
```

Code

```
import math


class Point:
    def __init__(self, x=0.0, y=0.0):
        #
        # Write code here
        #

    def getx(self):
        #
        # Write code here
        #

    def gety(self):
        #
        # Write code here
        #

    def distance_from_xy(self, x, y):
        #
        # Write code here
        #

    def distance_from_point(self, point):
        #
        # Write code here
        #


point1 = Point(0, 0)
point2 = Point(1, 1)
print(point1.distance_from_point(point2))
print(point2.distance_from_xy(2, 0))

```

Check Sample Solution

LAB: Triangle

Now we're going to embed the `Point` class (see Lab 3.4.1.14) inside another class. Also, we're going to put three points into one class, which will let us define a triangle. How can we do it? The new class will be called `Triangle` and this is what we want:

- the constructor accepts three arguments – all of them are objects of the `Point` class;
- the points are stored inside the object as a private list;
- the class provides a parameter less method called `perimeter()`, which calculates the perimeter of the triangle described by the three points; the perimeter is the sum of all the lengths of the legs (we mention this for the record, although we are sure that you know it perfectly yourself.)

Complete the template we've provided. Run your code and check whether the evaluated perimeter is the same as ours. Copy the `Point` class code we used in the previous lab.

Expected output

```
3.414213562373095
```

Code

```
import math


class Point:
    #
    # The code copied from the previous lab.
    #


class Triangle:
    def __init__(self, vertice1, vertice2, vertice3):
        #
        # Write code here
        #

    def perimeter(self):
        #
        # Write code here
        #


triangle = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))
print(triangle.perimeter())

```

Check Sample Solution

SEVENTEEN – OOP FUNDAMENTALS: INHERITANCE

Before we start talking about inheritance, we want to present a new, handy mechanism utilized by Python's classes and objects – it's the way in which the object is able to introduce itself. Let's start with an example.

```
1  class Star:
2      def __init__(self, name, galaxy):
3          self.name = name
4          self.galaxy = galaxy
5
6
7  sun = Star("Sun", "Milky Way")
8  print(sun)
9
```

The program prints out just one line of text, which in our case is this:

```
<__main__.Star object at 0x7f1074cc7c50>
```

If you run the same code on your computer, you'll see something very similar, although the hexadecimal number (the substring starting with `0x`) will be different, as it's just an internal object identifier used by Python, and it's unlikely that it would appear the same when the same code is run in a different environment. As you can see, the printout here isn't really useful, and something more specific, or just prettier, may be more preferable. Fortunately, Python offers just such a function.

When Python needs any class/object to be presented as a string (putting an object as an argument in the `print()` function invocation fits this condition) it tries to invoke a method named `__str__()` from the object and to use the string it returns.

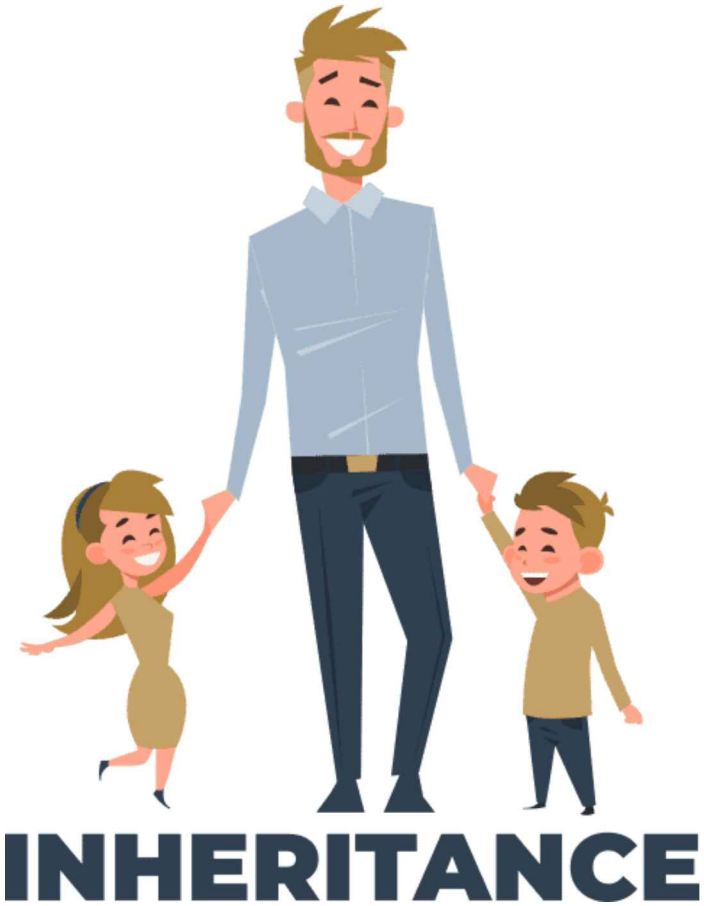
The default `__str__()` method returns the previous string – ugly and not very informative. You can change it just by defining your own method of the name.

We've just done it – look at the following code.

```
1
2  class Star:
3      def __init__(self, name, galaxy):
4          self.name = name
5          self.galaxy = galaxy
6
7      def __str__(self):
8          return self.name + ' in ' + self.galaxy
9
10
11 sun = Star("Sun", "Milky Way")
12 print(sun)
13
```

This new `__str__()` method makes a string consisting of the star's and galaxy's names – nothing special, but the print results look better now, doesn't it? Can you guess the output? Run the code to check if you were right.

The term inheritance is older than computer programming, and it describes the common practice of passing different goods from one person to another upon that person's death. The term, when related to computer programming, has an entirely different meaning. Let's define the term for our purposes. Inheritance is a common practice (in object programming) of passing attributes and methods from the superclass (defined and existing) to a newly created class, called the subclass. In other words, inheritance is a way of building a new class, not from scratch, but by using an already defined repertoire of traits. The new class inherits (and this is the key) all the already existing equipment, but is able to add some new ones if needed. Thanks to that, it's possible to build more specialized (more concrete) classes using some sets of predefined general rules and behaviors.



The most important factor of the process is the relation between the superclass and all of its subclasses (note: if *B* is a subclass of *A* and *C* is a subclass of *B*, this also means than *C* is a subclass of *A*, as the relationship is fully transitive). A very simple example of two-level inheritance is presented here:

```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass

```

All the presented classes are empty for now, as we're going to show you how the mutual relations between the super and subclasses work. We'll fill them with contents soon.

We can say that:

- The `Vehicle` class is the superclass for both The `LandVehicle` and `TrackedVehicle` classes;
- The `LandVehicle` class is a subclass of `Vehicle` and a superclass of `TrackedVehicle` at the same time;
- The `TrackedVehicle` class is a subclass of both The `Vehicle` and `LandVehicle` classes.

We understand this just by reading the code (in other words, we know it because we can see it). Does Python know the same? Is it possible to ask Python about it? Yes, it is.

`issubclass()`

Python offers a function which is able to identify a relationship between two classes, and although its diagnosis isn't complex, it can check if a particular class is a subclass of any other class. This is what it looks like:

```
issubclass(ClassOne, ClassTwo)

```

The function returns `True` if `ClassOne` is a subclass of `ClassTwo`, and `False` otherwise. Let's see it in action – it may surprise you. Look at the following code. Read it carefully.

```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass


for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:
    for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:
        print(issubclass(cls1, cls2), end="\t")
    print()

```

There are two nested loops. Their purpose is to check all possible ordered pairs of classes, and to print the results of the check to determine whether the pair matches the subclass-superclass relationship. Run the code. The program produces the following output:

```
True    False   False
True    True    False
True    True    True
```

Let's make the result more readable:

↓ is a subclass of →	Vehicle	LandVehicle	TrackedVehicle
Vehicle	True	False	False
LandVehicle	True	True	False
TrackedVehicle	True	True	True

Each class is considered to be a subclass of itself.

isinstance()

As you already know, an object is an incarnation of a class. This means that the object is like a cake baked using a recipe which is included inside the class. This can bring up some important issues. Let's assume that you've got a cake (e.g. as an argument passed to your function). You want to know what recipe has been used to make it. Why? Because you want to know what to expect from it, e.g. whether it contains nuts or not, which is crucial information to some people. Similarly, it can be crucial if the object does have (or doesn't have) certain characteristics. In other words, whether it is an object of a certain class or not. Such a fact could be detected by the function named `isinstance()`:

```
isinstance(objectName, ClassName)

```

The function returns `True` if the object is an instance of the class, or `False` otherwise. Being an instance of a class means that the object (the cake) has been prepared using a recipe contained in either the class or one of its superclasses. Don't forget: if a subclass contains at least the same equipment as any of its superclasses, it means that objects of the subclass can do the same as objects derived from the superclass, ergo, it's an instance of its home class and any of its superclasses. Let's test it. Analyze the following code.

```
class Vehicle:
    pass


class LandVehicle(Vehicle):
    pass


class TrackedVehicle(LandVehicle):
    pass


my_vehicle = Vehicle()
my_land_vehicle = LandVehicle()
my_tracked_vehicle = TrackedVehicle()

for obj in [my_vehicle, my_land_vehicle, my_tracked_vehicle]:
    for cls in [Vehicle, LandVehicle, TrackedVehicle]:
        print(isinstance(obj, cls), end="\t")
    print()

```

We've created three objects, one for each of the classes. Next, using two nested loops, we check all possible object-class pairs to find out if the objects are instances of the classes. Run the code. This is what we get:

```
True    False   False
True    True    False
True    True    True
```

Let's make the result more readable once again. Does the table confirm our expectations?

↓ is an instance of →	Vehicle	LandVehicle	TrackedVehicle
my_vehicle	True	False	False
my_land_vehicle	True	True	False
my_tracked_vehicle	True	True	True

The `is` operator

There is also a Python operator worth mentioning, as it refers directly to objects – here it is:

```
object_one is object_two

```

The `is` operator checks whether two variables (`object_one` and `object_two` here) refer to the same object. Don't forget that variables don't store the objects themselves, but only the handles pointing to the internal Python memory. Assigning a value of an object variable to another variable doesn't copy the object, but only its handle. This is why an operator like `is` may be very useful in particular circumstances. Take a look at the code.

```
class SampleClass:
    def __init__(self, val):
        self.val = val


object_1 = SampleClass(0)
object_2 = SampleClass(2)
object_3 = object_1
object_3.val += 1

print(object_1 is object_2)
print(object_2 is object_3)
print(object_3 is object_1)
print(object_1.val, object_2.val, object_3.val)

string_1 = "Mary had a little "
string_2 = "Mary had a little lamb"
string_1 += "lamb"

print(string_1 == string_2, string_1 is string_2)

```

Let's analyze it. There is a very simple class equipped with a simple constructor, creating just one property. The class is used to instantiate two objects. The former is then assigned to another variable, and its `val` property is incremented by one. Afterward, the `is` operator is applied three times to check all possible pairs of objects, and all `val` property values are also printed. The last part of the code carries out another experiment. After three assignments, both strings contain the same texts, but these texts are stored in different objects. The code prints:

```
False
False
True
1 2 1
True False
```

The results prove that `object_1` and `object_3` are actually the same objects, while `string_1` and `string_2` aren't, despite their contents being the same.

How Python finds properties and methods

Now we're going to look at how Python deals with inheriting methods. Take a look at the example.

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

    def __str__(self):
        return "My name is " + self.name + "."


class Sub(Super):
    def __init__(self, name):
        Super.__init__(self, name)


obj = Sub("Andy")

print(obj)

```

Let's analyze it. There is a class named `Super`, which defines its own constructor used to assign the object's property, named `name`. The class defines the `__str__()` method, too, which makes the class able to present its identity in clear text form. The class is next used as a base to create a subclass named `Sub`. The `Sub` class defines its own constructor, which invokes the one from the superclass. Note how we've done it: `Super.__init__(self, name)`. We've explicitly named the superclass, and pointed to the method to invoke `__init__()`, providing all needed arguments. We've instantiated one object of class `Sub` and printed it. The code outputs:

```
My name is Andy.
```

NOTE: As there is no `__str__()` method within the `Sub` class, the printed string is to be produced within the `Super` class. This means that the `__str__()` method has been inherited by the `Sub` class.

Look at the code. We've modified it to show you another way to access any entity defined inside the superclass. In the last example, we explicitly named the superclass. In this example, we make use of the `super()` function, which accesses the superclass without needing to know its name:

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

    def __str__(self):
        return "My name is " + self.name + "."


class Sub(Super):
    def __init__(self, name):
        super().__init__(name)


obj = Sub("Andy")

print(obj)

```

In the last example, we explicitly named the superclass. In this example, we make use of the `super()` function, which accesses the superclass without needing to know its name:

```
super().__init__(name)

```

The `super()` function creates a context in which you don't have to (moreover, you mustn't) pass the self-argument to the method being invoked – this is why it's possible to activate the superclass constructor using only one argument.

NOTE: You can use this mechanism not only to invoke the superclass constructor, but also to get access to any of the resources available inside the superclass.

Let's try to do something similar, but with properties (more precisely: with class variables). Take a look at this example.

```
# Testing properties: class variables.
class Super:
    supVar = 1


class Sub(Super):
    subVar = 2


obj = Sub()

print(obj.subVar)
print(obj.supVar)

```

As you can see, the `Super` class defines one class variable named `supVar`, and the `Sub` class defines a variable named `subVar`. Both these variables are visible inside the object of class `Sub` – this is why the code outputs:

```
2

1
```

The same effect can be observed with instance variables – take a look at the second example.

```
# Testing properties: instance variables.
class Super:
    def __init__(self):
        self.supVar = 11


class Sub(Super):
    def __init__(self):
        super().__init__()
        self.subVar = 12


obj = Sub()

print(obj.subVar)
print(obj.supVar)

```

The `Sub` class constructor creates an instance variable named `subVar`, while the `Super` constructor does the same with a variable named supVar. As previously, both variables are accessible from within the object of class `Sub`. The program's output is:

```
12
11
```

NOTE: The existence of the `supVar` variable is obviously conditioned by the `Super` class constructor invocation. Omitting it would result in the absence of the variable in the created object. Try it yourself.

It's now possible to formulate a general statement describing Python's behavior. When you try to access any object's entity, Python will try to find it inside the object itself and find it in all classes involved in the object's inheritance line from bottom to top, and it will try to do it in that order. If both fail, an exception (`AttributeError`) is raised.

The first condition may need some additional attention. As you know, all objects deriving from a particular class may have different sets of attributes, and some of the attributes may be added to the object a long time after the object's creation. The following example summarizes this in a three-level inheritance line. Analyze it carefully.

```
1  class Level1:
2      variable_1 = 100
3      def __init__(self):
4          self.var_1 = 101
5
6      def fun_1(self):
7          return 102
8
9
10 class Level2(Level1):
11     variable_2 = 200
12     def __init__(self):
13         super().__init__()
14         self.var_2 = 201
15
16     def fun_2(self):
17         return 202
18
19
20 class Level3(Level2):
21     variable_3 = 300
22     def __init__(self):
23         super().__init__()
24         self.var_3 = 301
25
26     def fun_3(self):
27         return 302
28
29
30 obj = Level3()
31
32 print(obj.variable_1, obj.var_1, obj.fun_1())
33 print(obj.variable_2, obj.var_2, obj.fun_2())
34 print(obj.variable_3, obj.var_3, obj.fun_3())
35
```

All the comments we've made so far are related to single inheritance, when a subclass has exactly one superclass. This is the most common situation, and the recommended one, too. Python, however, offers much more here. In the next lessons we're going to show you some examples of multiple inheritance. Multiple inheritance occurs when a class has more than one superclass. Syntactically, such inheritance is presented as a comma-separated list of superclasses put inside parentheses after the new class name – just like here:

```
class SuperA:
    var_a = 10
    def fun_a(self):
        return 11


class SuperB:
    var_b = 20
    def fun_b(self):
        return 21


class Sub(SuperA, SuperB):
    pass

obj = Sub()

print(obj.var_a, obj.fun_a())
print(obj.var_b, obj.fun_b())

```

The `Sub` class has two superclasses: `SuperA` and `SuperB`. This means that the `Sub` class inherits all the goods offered by both `SuperA` and `SuperB`. The code prints:

```
10 11
20 21
```

Now it's time to introduce a brand new term – overriding. What do you think will happen if more than one of the superclasses defines an entity of a particular name? Let's analyze the example.

```
class Level1:
    var = 100
    def fun(self):
        return 101


class Level2(Level1):
    var = 200
    def fun(self):
        return 201


class Level3(Level2):
    pass


obj = Level3()

print(obj.var, obj.fun())

```

Both, `Level1` and `Level2` classes define a method named `fun()` and a property named var. Does this mean that the `Level3` class object will be able to access two copies of each entity? Not at all. The entity defined later (in the inheritance sense) overrides the same entity defined earlier. This is why the code produces the following output:

```
200 201
```

As you can see, the var class variable and `fun()` method from the `Level2` class override the entities of the same names derived from the `Level1` class. This feature can be intentionally used to modify default (or previously defined) class behaviors when any of its classes needs to act in a different way to its ancestor. We can also say that Python looks for an entity from bottom to top, and is fully satisfied with the first entity of the desired name.

How does it work when a class has two ancestors offering the same entity, and they lie on the same level? In other words, what should you expect when a class emerges using multiple inheritance? Let's look at this. Let's take a look at the following example:

```
class Left:
    var = "L"
    var_left = "LL"
    def fun(self):
        return "Left"


class Right:
    var = "R"
    var_right = "RR"
    def fun(self):
        return "Right"


class Sub(Left, Right):
    pass


obj = Sub()

print(obj.var, obj.var_left, obj.var_right, obj.fun())

```

The `Sub` class inherits goods from two superclasses, `Left` and `Right` (these names are intended to be meaningful). There is no doubt that the class variable `var_right` comes from the `Right` class, and `var_left` comes from `Left` respectively. This is clear. But where does `var` come from? Is it possible to guess it? the same problem is encountered with the `fun()` method – will it be invoked from `Left` or from `Right`? Let's run the program – its output is:

```
L LL RR Left
```

This proves that both unclear cases have a solution inside the `Left` class. Is this a sufficient premise to formulate a general rule? Yes, it is. We can say that Python looks for object components in the following order: inside the object itself; in its superclasses, from bottom to top; if there is more than one class on a particular inheritance path, Python scans them from left to right.

Do you need anything more? Just make a small amendment in the code – replace: `class Sub(Left, Right):` with: `class Sub(Right, Left):`, then run the program again, and see what happens. What do you see now? We see:

```
R LL RR Right
```

Do you see the same, or something different?

How to build a hierarchy of classes

Building a hierarchy of classes isn't just art for art's sake. If you divide a problem among classes and decide which of them should be located at the top and which should be placed at the bottom of the hierarchy, you have to carefully analyze the issue, but before we show you how to do it (and how not to do it), we want to highlight an interesting effect. It's nothing extraordinary (it's just a consequence of the general rules presented earlier), but remembering it may be key to understanding how some codes work, and how the effect may be used to build a flexible set of classes. Take a look at the code.

```
class One:
    def do_it(self):

        print("do_it from One")
    def doanything(self):
        self.do_it()


class Two(One):
    def do_it(self):
        print("do_it from Two")


one = One()
two = Two()

one.doanything()
two.doanything()

```

Let's analyze it. There are two classes, named `One` and `Two`, while `Two` is derived from `One`. Nothing special. However, one thing looks remarkable – the `do_it()` method. The `do_it()` method is defined twice: originally inside `One` and subsequently inside `Two`. The essence of the example lies in the fact that it is invoked just once – inside `One`.

The question is – which of the two methods will be invoked by the last two lines of the code? The first invocation seems to be simple, and it is simple actually – invoking `doanything()` from the object named `one` will obviously activate the first of the methods. The second invocation needs some attention. It's simple too if you keep in mind how Python finds class components.

The second invocation will launch `do_it()` in the form existing inside the `Two` class, regardless of the fact that the invocation takes place within the `One` class. In effect, the code produces the following output:

```
do_it from One
do_it from Two
```

NOTE: The situation in which the subclass is able to modify its superclass behavior, just like in the example, is called polymorphism. The word comes from Greek (polys: "many, much" and morphe, "form, shape"), which means that one and the same class can take various forms depending on the redefinitions done by any of its subclasses. The method, redefined in any of the subclasses, thus changing the behavior of the superclass, is called virtual. In other words, no class is given once and for all. Each class's behavior may be modified at any time by any of its subclasses.

We're going to show you how to use polymorphism to extend class flexibility. Look at the following example.

```
1  import time
2
3  class TrackedVehicle:
4      def control_track(left, stop):
5          pass
6
7      def turn(left):
8          control_track(left, True)
9          time.sleep(0.25)
10         control_track(left, False)
11
12
13 class WheeledVehicle:
14     def turn_front_wheels(left, on):
15         pass
16
17     def turn(left):
18         turn_front_wheels(left, True)
19         time.sleep(0.25)
20         turn_front_wheels(left, False)
21
```

Does it resemble anything? Yes, of course it does. It refers to the example shown at the beginning of the module when we talked about the general concepts of object programming. It may look weird, but we didn't use inheritance in any way – just to show you that it doesn't limit us – and we managed to get ours. We've defined two separate classes able to produce two different kinds of land vehicles. The main difference between them is in how they turn. A wheeled vehicle just turns the front wheels (generally).

A tracked vehicle has to stop one of the tracks. Can you follow the code? A tracked vehicle performs a turn by stopping and moving on one of its tracks (this is done by the `control_track()` method, which will be implemented later). A wheeled vehicle turns when its front wheels turn (this is done by the `turn_front_wheels()` method). The `turn()` method uses the method suitable for each particular vehicle.

Can you see what's wrong with the code? The `turn()` methods look too similar to leave them in this form. Let's rebuild the code – we're going to introduce a superclass to gather all the similar aspects of the driving vehicles, moving all the specifics to the subclasses. Look at the code again.

```
1  import time
2
3  class Vehicle:
4      def change_direction(left, on):
5          pass
6
7      def turn(left):
8          change_direction(left, True)
9          time.sleep(0.25)
10         change_direction(left, False)
11
12
13 class TrackedVehicle(Vehicle):
14     def control_track(left, stop):
15         pass
16
17     def change_direction(left, on):
18         control_track(left, on)
19
20
21 class WheeledVehicle(Vehicle):
22     def turn_front_wheels(left, on):
23         pass
24
25     def change_direction(left, on):
26         turn_front_wheels(left, on)
27
```

This is what we've done. We defined a superclass named `Vehicle`, which uses the `turn()` method to implement a general scheme of turning, while the turning itself is done by a method named `change_direction()`; note: the former method is empty, as we are going to put all the details into the subclass (such a method is often called an abstract method, as it only demonstrates some possibility which will be instantiated later). We defined a subclass named `TrackedVehicle` (note: it's derived from the `Vehicle` class) which instantiated the `change_direction()` method by using the specific (concrete) method named `control_track()`. Respectively, the subclass named `WheeledVehicle` does the same trick, but uses the `turn_front_wheels()` method to force the vehicle to turn.

The most important advantage (omitting readability issues) is that this form of code enables you to implement a brand new turning algorithm just by modifying the `turn()` method, which can be done in just one place, as all the vehicles will obey it. This is how polymorphism helps the developer to keep the code clean and consistent.

Inheritance is not the only way to construct adaptable classes. You can achieve the same goals (not always, but very often) by using a technique named composition. Composition is the process of composing an object using other different objects. The objects used in the composition deliver a set of desired traits (properties and/or methods) so we can say that they act like blocks used to build a more complicated structure.

It can be said that inheritance extends a class's capabilities by adding new components and modifying existing ones; in other words, the complete recipe is contained inside the class itself and all its ancestors; the object takes all the class's belongings and makes use of them. In addition, composition projects a class as a container able to store and use other objects (derived from other classes) where each of the objects implements a part of a desired class's behavior.

Let us illustrate the difference by using the previously defined vehicles. The previous approach led us to a hierarchy of classes in which the topmost class was aware of the general rules used in turning the vehicle, but didn't know how to control the appropriate components (wheels or tracks).

The subclasses implemented this ability by introducing specialized mechanisms. Let's do (almost) the same thing, but using composition. The class – like in the previous example – is aware of how to turn the vehicle, but the actual turn is done by a specialized object stored in a property named `controller`. The `controller` is able to control the vehicle by manipulating the relevant vehicle's parts.

Take a look at the code – this is what it could look like.

```
import time

class Tracks:
    def change_direction(self, left, on):
        print("tracks: ", left, on)


class Wheels:
    def change_direction(self, left, on):
        print("wheels: ", left, on)


class Vehicle:
    def __init__(self, controller):
        self.controller = controller

    def turn(self, left):
        self.controller.change_direction(left, True)
        time.sleep(0.25)
        self.controller.change_direction(left, False)


wheeled = Vehicle(Wheels())
tracked = Vehicle(Tracks())

wheeled.turn(True)
tracked.turn(False)

```

There are two classes named `Tracks` and `Wheels` – they know how to control the vehicle's direction. There is also a class named `Vehicle` which can use any of the available controllers (the two already defined, or any others defined in the future) – the `controller` itself is passed to the class during initialization.

In this way, the vehicle's ability to turn is composed using an external object, not implemented inside the `Vehicle` class.

In other words, we have a universal vehicle and can install either tracks or wheels onto it.

The code produces the following output:

```
wheels: True True
wheels:  True False
tracks:  False True
tracks:  False False
```

Single inheritance vs. multiple inheritance

As you already know, there are no obstacles to using multiple inheritance in Python. You can derive any new class from more than one previously defined classes. There is only one "but". The fact that you can do it does not mean you have to.

Don't forget that a single inheritance class is always simpler, safer, and easier to understand and maintain; multiple inheritance is always risky, as you have many more opportunities to make a mistake in identifying these parts of the superclasses which will

effectively influence the new class. Multiple inheritance may make overriding extremely tricky; moreover, using the super() function becomes ambiguous. Multiple inheritance also violates the single responsibility principle (more details here: https://en.wikipedia.org/wiki/Single_responsibility_principle) as it makes a new class of two (or more) classes that know nothing about each other;

We strongly suggest multiple inheritance as the last of all possible solutions – if you really need the many different functionalities offered by different classes, composition may be a better alternative.

What is Method Resolution Order (MRO) and why is it that not all inheritances make sense?

MRO, in general, is a way (you can call it a strategy) in which a particular programming language scans through the upper part of a class's hierarchy in order to find the method it currently needs. It's worth emphasizing that different languages use slightly (or even completely) different MROs. Python is a unique creature in this respect, however, and its customs are a bit specific. We're going to show you how Python's MRO works in two peculiar cases that are clear-cut examples of problems which may occur when you try to use multiple inheritance too recklessly. Let's start with a snippet that initially may look simple. Look at the code we've prepared for you.

```
1  class Top:
2      def m_top(self):
3          print("top")
4
5
6  class Middle(Top):
7      def m_middle(self):
8          print("middle")
9
10
11 class Bottom(Middle):
12     def m_bottom(self):
13         print("bottom")
14
15
16 object = Bottom()
17 object.m_bottom()
18 object.m_middle()
19 object.m_top()
20
```

We're sure that if you analyze the snippet yourself, you won't see any anomalies in it. Yes, you're perfectly right – it looks clear and simple, and raises no concerns. If you run the code, it will produce the following, predictable output:

```
bottom
middle
top
```

No surprises so far. Let's make a tiny change to this code. Have a look:

```
1  class Top:
2      def m_top(self):
3          print("top")
4
5
6  class Middle(Top):
7      def m_middle(self):
8          print("middle")
9
10
11 class Bottom(Middle, Top):
12     def m_bottom(self):
13         print("bottom")
14
15
16 object = Bottom()
17 object.m_bottom()
18 object.m_middle()
19 object.m_top()
20
```

Can you see the difference? It's hidden in this line:

```
1  class Bottom(Middle, Top):
2
```

In this exotic way, we've turned a very simple code with a clear singleinheritance path into a mysterious multiple-inheritance riddle. “Is it valid?” you may ask. Yes, it is. “How is that possible?” you should ask now, and we hope that you really feel the need to ask this question.

As you can see, the order in which the two superclasses have been listed between parenthesis is compliant with the code's structure: the `Middle` class precedes the `Top` class, just like in the real inheritance path. Despite its oddity, the sample is correct and works as expected, but it has to be stated that this notation doesn't bring any new functionality or additional meaning.

Let's modify the code once again – now we'll swap both superclass names in the `Bottom` class definition. This is what the snippet looks like now:

```
class Top:
    def m_top(self):
        print("top")


class Middle(Top):
    def m_middle(self):
        print("middle")


class Bottom(Top, Middle):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()

```

To anticipate your question, we'll say that this amendment has spoiled the code, and it won't run anymore. What a pity. The order we tried to force (Top, Middle) is incompatible with the inheritance path which is derived from the code's structure. Python won't like it. This is what we'll see:

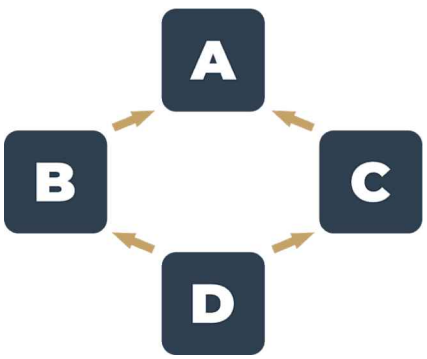
```
TypeError: Cannot create a consistent method resolution order (MRO) for bases Top,
Middle
```

We think that the message speaks for itself. Python's MRO cannot be bent or violated, not just because that's the way Python works, but also because it's a rule you have to obey.

The diamond problem

The second example of the spectrum of issues that can possibly arise from multiple inheritance is illustrated by a classic problem named the diamond problem. The name reflects the shape of the inheritance diagram – take a look at the picture.

There is the top-most superclass named A. There are two subclasses derived from A, B and C; and there is also the bottom-most subclass named D, derived from B and C (or C and B, as these two variants mean different things in Python) Can you see the diamond there?



Have a look at the code. The same structure, but expressed in Python.

```
class A:
    pass


class B(A):
    pass


class C(A):
    pass


class D(B, C):
    pass


d = D()

```

Some programming languages don't allow multiple inheritance at all, and as a consequence, they won't let you build a diamond – this is the route that Java and C# have chosen to follow since their origins. Python, however, has chosen a different route – it allows multiple inheritance, and it doesn't mind if you write and run code like the one in the example. But don't forget about MRO – it's always in charge. Let's rebuild our example from previously to make it more diamond-like, just like the following:

```
class Top:
    def m_top(self):
        print("top")


class Middle_Left(Top):
    def m_middle(self):
        print("middle_left")


class Middle_Right(Top):
    def m_middle(self):
        print("middle_right")


class Bottom(Middle_Left, Middle_Right):
    def m_bottom(self):
        print("bottom")


object = Bottom()
object.m_bottom()
object.m_middle()
object.m_top()

```

NOTE: Both `Middle` classes define a method of the same name: `m_middle()`.

It introduces a small uncertainty to our sample, although we're absolutely sure that you can answer the following key question: which of the two `m_middle()` methods will actually be invoked when the following line is executed?

```
Object.m_middle()
```

In other words, what will you see on the screen: `middle_left` or `middle_right`? You don't need to hurry – think twice and keep Python's MRO in mind! Are you ready? Yes, you're right. The invocation will activate the `m_middle()` method, which comes from the `Middle_Left` class. The explanation is simple: the class is listed before `Middle_Right` on the `Bottom` class's inheritance list. If you want to make sure that there's no doubt about it, try to swap these two classes on the list and check the results. If you want to experience some more profound impressions about multiple inheritance and precious gemstones, try to modify our snippet and equip the `Top` class with another specimen of the `m_middle()` method, and investigate its behavior carefully. As you can see, diamonds may bring some problems into your life – both the real ones and those offered by Python.

Summary

1. A method named `__str__()` is responsible for converting an object's contents into a (more or less) readable string. You can redefine it if you want your object to be able to present itself in a more elegant form. For example:

```
class Mouse:
    def __init__(self, name):
        self.my_name = name


    def __str__(self):
        return self.my_name


the_mouse = Mouse('mickey')
print(the_mouse) # Prints "mickey".

```

2. A function named `issubclass(Class_1, Class_2)` is able to determine if `Class_1` is a subclass of `Class_2`. For example:

```
class Mouse:
    pass


class LabMouse(Mouse):
    pass


print(issubclass(Mouse, LabMouse), issubclass(LabMouse, Mouse)) # Prints "False
True

```

3. A function named `isinstance(Object, Class)` checks if an object comes from an indicated class. For example:

```
class Mouse:
    pass


class LabMouse(Mouse):
    pass


mickey = Mouse()
print(isinstance(mickey, Mouse), isinstance(mickey, LabMouse)) # Prints "True
False".

```

4. An operator called `is` checks if two variables refer to the same object. For example:

```
class Mouse:
    pass


mickey = Mouse()
minnie = Mouse()
cloned_mickey = mickey
print(mickey is minnie, mickey is cloned_mickey) # Prints "False True".

```

5. A parameterless function named `super()` returns a reference to the nearest superclass of the class. For example:

```
class Mouse:
    def __str__(self):
        return "Mouse"


class LabMouse(Mouse):
    def __str__(self):
        return "Laboratory " + super().__str__()


doctor_mouse = LabMouse();
print(doctor_mouse) # Prints "Laboratory Mouse".

```

6. Methods as well as instance and class variables defined in a superclass are automatically inherited by their subclasses. For example:

```
class Mouse:
    Population = 0
    def __init__(self, name):
        Mouse.Population += 1
        self.name = name

    def __str__(self):
        return "Hi, my name is " + self.name

class LabMouse(Mouse):
    tpass

professor_mouse = LabMouse("Professor Mouser")
print(professor_mouse, Mouse.Population) # Prints "Hi, my name is Professor
Mouser 1"

```

7. In order to find any object/class property, Python looks for it inside the object itself and all classes involved in the object's inheritance line from bottom to top. If there is more than one class on a particular inheritance path, Python scans them from left to right, and if both fail, the `AttributeError` exception is raised.

8. If any of the subclasses defines a method/class variable/instance variable of the same name as existing in the superclass, the new name overrides any of the previous instances of the name. For example:

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

    def __str__(self):
    return "My name is " + self.name

class AncientMouse(Mouse):
    def __str__(self):
    return "Meum nomen est " + self.name

mus = AncientMouse("Caesar") # Prints "Meum nomen est Caesar"
print(mus)

```

Quiz

Exercises

Scenario

Assume that the following piece of code has been successfully executed:

```
class Dog:
    kennel = 0
    def __init__(self, breed):
        self.breed = breed
        Dog.kennel += 1
    def __str__(self):
        return self.breed + " says: Woof!"


class SheepDog(Dog):
    def __str__(self):
        return super().__str__() + " Don't run away, Little Lamb!"


class GuardDog(Dog):
    def __str__(self):
        return super().__str__() + " Stay where you are, Mister Intruder!"


rocky = SheepDog("Collie")
luna = GuardDog("Dobermann")

```

Question 1: The declaration of The `Snake` class is given here. Enrich the class with a method named `increment()`, adding `1` to The `__victims` property.

```
print(rocky)
print(luna)
```

Question 2: What is the expected output of the following piece of code?

```
print(issubclass(SheepDog, Dog), issubclass(SheepDog, GuardDog))
print(isinstance(rocky, GuardDog), isinstance(luna, GuardDog))
```

Question 3: What is the expected output of the following piece of code?

```
print(luna is luna, rocky is luna)
print(rocky.kennel)
```

Question 4: Define a `SheepDog`'s subclass named `LowlandDog`, and equip it with an `__str__()` method overriding an inherited method of the same name. The new dog's `__str__()` method should return the string "{Dog} says: Woof! I don't like mountains!".

Check Answers

EIGHTEEN – EXCEPTIONS ONCE AGAIN

Discussing object programming offers a very good opportunity to return to exceptions. The object-oriented nature of Python's exceptions makes them a very flexible tool, able to fit to specific needs, even those you don't yet know about.

Before we dive into the objective face of exceptions, we want to show you some syntactical and semantic aspects of how Python treats the `try-except` block, as it offers a little more than what we have presented so far. The first feature we want to discuss here is an additional, possible branch that can be placed inside (or rather, directly behind) the *try-except* block – it's the part of the code starting with else – just like in this example.

```
1  def reciprocal(n):
2      try:
3          n = 1 / n
4      except ZeroDivisionError:
5          print("Division failed")
6          return None
7      else:
8          print("Everything went fine")
9          return n
10
11
12 print(reciprocal(2))
13 print(reciprocal(0))
14
```

A code labelled in this way is executed when (and only when) no exception has been raised inside the `try:` part. We can say that exactly one branch can be executed after `try:` – either the one beginning with `except` (don't forget that there can be more than one branch of this kind) or the one starting with else.

NOTE: The `else:` branch has to be located after the last `except` branch.

The example code produces the following output:

```
Everything went fine
0.5
Division failed
None
```

The `try-except` block can be extended in one more way – by adding a part headed by the `finally` keyword (it must be the last branch of the code designed to handle exceptions).

NOTE: These two variants (`else` and `finally`) aren't dependent in any way, and they can coexist or occur independently.

The `finally` block is always executed (it finalizes the `try-except` block execution, hence its name), no matter what happened earlier, even when raising an exception, no matter whether this has been handled or not. Look at the following code.

```
1  def reciprocal(n):
2      try:
3          n = 1 / n
4      except ZeroDivisionError:
5          print("Division failed")
6          n = None
7      else:
8          print("Everything went fine")
9      finally:
10         print("It's time to say goodbye")
11         return n
12
13
14 print(reciprocal(2))
15 print(reciprocal(0))
16
```

It outputs:

```
Everything went fine
It's time to say goodbye
0.5
Division failed
It's time to say goodbye
None
```

Exceptions are classes

All the previous examples were content with detecting a specific kind of exception and responding to it in an appropriate way. Now we're going to delve deeper, and look inside the exception itself. You probably won't be surprised to learn that exceptions are classes. Furthermore, when an exception is raised, an object of the class is instantiated, and goes through all levels of program execution, looking for the except branch that is prepared to deal with it. Such an object carries some useful information which can help you to precisely identify all aspects of the pending situation. To achieve that goal, Python offers a special variant of the exception clause – you can find it in the following code.

```
1  try:
2      i = int("Hello!")
3  except Exception as e:
4      print(e)
5      print(e.__str__())
6
```

As you can see, the `except` statement is extended, and contains an additional phrase starting with the `as` keyword, followed by an identifier. The identifier is designed to catch the exception object so you can analyze its nature and draw some useful conclusions.

NOTE: The identifier's scope covers its `except` branch, and doesn't go any further.

The example presents a very simple way of utilizing the received object – just print it out (as you can see, the output is produced by the object's `__str__()` method) and it contains a brief message describing the reason. The same message will be printed if there is no fitting `except` block in the code, and Python is forced to handle it alone. All the built-in Python exceptions form a hierarchy of classes. There is no obstacle to extending it if you find it reasonable. Look at this code.

```
1  def print_exception_tree(thisclass, nest = 0):
2      if nest > 1:
3          print("   |" * (nest - 1), end=  )
4      if nest > 0:
5          print("   +---", end=  )
6
7      print(thisclass.__name__)
8
9      for subclass in thisclass.__subclasses__():
10         print_exception_tree(subclass, nest + 1)
11
12
13 print_exception_tree(BaseException)
14
```

This program dumps all predefined exception classes in the form of a tree-like printout. As a tree is a perfect example of a recursive data structure, a recursion seems to be the best tool to traverse through it. The `print_exception_tree()` function takes two arguments: a point inside the tree from which we start traversing the tree; and a nesting level, which we'll use to build a simplified drawing of the tree's branches.

Let's start from the tree's root – the root of Python's exception classes is the `BaseException` class (it's a superclass of all other exceptions). For each of the encountered classes, perform the same set of operations. First, print its name, taken from the `__name__` property, then iterate through the list of subclasses delivered by the `__subclasses__()` method, and recursively invoke the `print_exception_tree()` function, incrementing the nesting level respectively.

Note how we've drawn the branches and forks. The printout isn't sorted in any way – you can try to sort it yourself, if you want a challenge. Moreover, there are some subtle inaccuracies in the way in which some branches are presented. That can be fixed, too, if you wish.

This is what it looks like:

```
BaseException
    +---Exception
    |    +---TypeError
    |    +---StopAsyncIteration
    |    +
    |    +---ImportError
    |    |    +---ModuleNotFoundError
    |    |    +---ZipImportError
    |    +---OSError
    |    |    +---ConnectionError
    |    |    |    +---BrokenPipeError
    |    |    |    +---ConnectionAbortedError
    |    |    |    +---ConnectionRefusedError
    |    |    |    +---ConnectionResetError
    |    |    +---BlockingIOError
```

```
|    |    +---ChildProcessError
|    |    +---FileExistsError
|    |    +---FileNotFoundError
|    |    +---IsADirectoryError
|    |    +---NotADirectoryError
|    |    +---InterruptedError
|    |    +---PermissionError
|    |    +---ProcessLookupError
|    |    +---TimeoutError
|    |    +---UnsupportedOperation
|    |    +---herror
|    |    +---gaierror
|    |    +---timeout
|    |    +---Error
|    |    |    +---SameFileError
|    |    +---SpecialFileError
|    |    +---ExecError
|    |    +---ReadError
|    +---EOFError
|    +---RuntimeError
|    |    +---RecursionError
|    |    +---NotImplementedError
|    |    +---_DeadlockError
|    |    +---BrokenBarrierError
|    +---NameError
|    |    +---UnboundLocalError
|    +---AttributeError
|    +---
|    |    +---IndentationError
|    |    |    +---TabError
|    +---LookupError
|    |    +---IndexError
|    |    +---KeyError
|    |    +---CodecRegistryError
|    +---ValueError
|    |    +---UnicodeError
|    |    |    +---UnicodeEncodeError
|    |    |    +---UnicodeDecodeError
|    |    |    +---UnicodeTranslateError
|    |    +---UnsupportedOperation
|    +---AssertionError
|    +---ArithmeticError
|    |    +---FloatingPointError
|    |    +---OverflowError
|    |    +---ZeroDivisionError
|    +---SystemError
|    |    +---CodecRegistryError
|    +---ReferenceError
|    +---BufferError
|    +---MemoryError
|    +---Warning
```

```
|    |    +---UserWarning
|    |    +---DeprecationWarning
|    |    +---PendingDeprecationWarning
|    |    +---SyntaxWarning
|    |    +---RuntimeWarning
|    |    +---FutureWarning
|    |    +---ImportWarning
|    |    +---UnicodeWarning
|    |    +---BytesWarning
|    |    +---ResourceWarning
|    +---error
|    +---Verbose
|    +---Error
|    +---TokenError
|    +---StopTokenizing
|    +---Empty
|    +---Full
|    +---_OptionError
|    +---TclError
|    +---SubprocessError
|    |    +---CalledProcessError
|    |    +---TimeoutExpired
|    +---Error
|    |    +---NoSectionError
|    |    +---DuplicateSectionError
|    |    +---DuplicateOptionError
|    |    +---NoOptionError
|    |    +---InterpolationError
|    |    |    +---InterpolationMissingOptionError
|    |    |    +---InterpolationSyntaxError
|    |    |    +---InterpolationDepthError
|    |    +---ParsingError
|    |    |    +---MissingSectionHeaderError
|    +---InvalidConfigType
|    +---InvalidConfigSet
|    +---InvalidFgBg
|    +---InvalidTheme
|    +---EndOfBlock
|    +---BdbQuit
|    +---error
|    +---_Stop
|    +---PickleError
|    |    +---PicklingError
|    |    +---UnpicklingError
|    +---_GiveupOnSendfile
|    +---error
|    +---LZMAError
|    +---RegistryError
|    +---ErrorDuringImport
+---GeneratorExit
+---SystemExit
```

```
        +---KeyboardInterrupt
```

Detailed anatomy of exceptions

Let's take a closer look at the exception's object, as there are some really interesting elements here (we'll return to the issue soon when we consider Python's input/output base techniques, as their exception subsystem extends these objects a bit).

The `BaseException` class introduces a property named `args`. It's a tuple designed to gather all arguments passed to the class constructor. It is empty if the construct has been invoked without any arguments, or contains just one element when the constructor gets one argument (we don't count the `self` argument here), and so on. We've prepared a simple function to print the `args` property in an elegant way. You can see the function in the following code.

```
1  def print_args(args):
2      lng = len(args)
3      if lng == 0:
4          print("")
5      elif lng == 1:
6          print(args[0])
7      else:
8          print(str(args))
9
10
11 try:
12     raise Exception
13 except Exception as e:
14     print(e, e.__str__(), sep=' : ' ,end=' : ')
15     print_args(e.args)
16
17 try:
18     raise Exception("my exception")
19 except Exception as e:
20     print(e, e.__str__(), sep=' : ', end=' : ')
21     print_args(e.args)
22
23 try:
24     raise Exception("my", "exception")
25 except Exception as e:
26     print(e, e.__str__(), sep=' : ', end=' : ')
27     print_args(e.args)
28
```

We've used the function to print the contents of the `args` property in three different cases, where the exception of the `Exception` class is raised in three different ways. To make it more spectacular, we've also printed the object itself, along with the result of the `__str__()` invocation.

The first case looks routine – there is just the name `Exception` after the `raise` keyword. This means that the object of this class has been created in a most routine way. The second and third cases may look a bit weird at first glance, but there's nothing odd here – these are just the constructor invocations. In the second `raise` statement, the constructor is invoked with one argument, and in the third, with two. As you can see, the program output reflects this, showing the appropriate contents of the `args` property:

```
:  :
my exception : my exception : my exception
('my', 'exception') : ('my', 'exception') : ('my', 'exception')
```

How to create your own exception

The exceptions hierarchy is neither closed nor finished, and you can always extend it if you want or need to create your own world populated with your own exceptions. It may be useful when you create a complex module which detects errors and raises exceptions, and you want the exceptions to be easily distinguishable from any others brought by Python. This is done by defining your own, new exceptions as subclasses derived from predefined ones.

NOTE: If you want to create an exception which will be utilized as a specialized case of any built-in exception, derive it from just this one. If you want to build your own hierarchy, and don't want it to be closely connected to Python's exception tree, derive it from any of the top exception classes, like Exception.

Imagine that you've created a brand new arithmetic, ruled by your own laws and theorems. It's clear that division has been redefined, too, and has to behave in a different way than routine dividing. It's also clear that this new division should raise its own exception, different from the built-in `ZeroDivisionError`, but it's reasonable to assume that in some circumstances, you (or your arithmetic's user) may want to treat all zero divisions in the same way. Demands like these may be fulfilled in the manner shown.

```
class MyZeroDivisionError(ZeroDivisionError):
    pass


def do_the_division(mine):
    if mine:
        raise MyZeroDivisionError("some worse news")
    else:
        raise ZeroDivisionError("some bad news")


for mode in [False, True]:
    try:
        do_the_division(mode)
    except ZeroDivisionError:
        print('Division by zero')

for mode in [False, True]:
    try:
        do_the_division(mode)
    except MyZeroDivisionError:
        print('My division by zero')
    except ZeroDivisionError:
        print('Original division by zero')

```

Look at the code, and let's analyze it. We've defined our own exception, named `MyZeroDivisionError`, derived from the built-in `ZeroDivisionError`. As you can see, we've decided not to add any new components to the class. In effect, an exception of this class can be – depending on the desired point of view – treated like a plain `ZeroDivisionError`, or considered separately. The `do_the_division()` function raises either a `MyZeroDivisionError` or `ZeroDivisionError` exception, depending on the argument's value. The function is invoked four times in total, while the first two invocations are handled using only one `except` branch (the more general one) and the last two ones with two different branches, able to distinguish the exceptions (don't forget: the order of the branches makes a fundamental difference!)

When you're going to build a completely new universe filled with completely new creatures that have nothing in common with all the familiar things, you may want to build your own exception structure. For example, if you work on a large simulation system which is intended to model the activities of a pizza restaurant, it can be desirable to form a separate hierarchy of exceptions. You can start building it by defining a general exception as a new base class for any other specialized exception. We've done in in the following way:

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza= pizza

```

NOTE: We're going to collect more specific information here than a regular `Exception` does, so our constructor will take two arguments, one specifying a pizza as a subject of the process, and one containing a more or less precise description of the problem. As you can see, we pass the second parameter to the superclass constructor, and save the first inside our own property.

A more specific problem (like an excess of cheese) can require a more specific exception. It's possible to derive the new class from the already defined `PizzaError` class, like we've done here:

```
class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese= cheese

```

The `TooMuchCheeseError` exception needs more information than the regular `PizzaError` exception, so we add it to the constructor – the name `cheese` is then stored for further processing. Look at the code. We've coupled together the two previously defined exceptions and harnessed them to work in a small example snippet.

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese


def make_pizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError(pizza, "no such pizza on the menu")
    if cheese > 100:
        raise TooMuchCheeseError(pizza, cheese, "too much cheese")
    print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        make_pizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)

```

One of these is raised inside the `make_pizza()` function when any of these two erroneous situations is discovered: a wrong pizza request, or a request for too much cheese.

NOTE: Removing the branch starting with except `TooMuchCheeseError` will cause all appearing exceptions to be classified as `PizzaError`. Removing the branch starting with except `PizzaError` will cause the `TooMuchCheeseError` exceptions to remain unhandled, and will cause the program to terminate.

The previous solution, although elegant and efficient, has one important weakness. Due to the somewhat easygoing way of declaring the constructors, the new exceptions cannot be used as they are without a full list of required arguments. We'll remove this weakness by setting the default values for all constructor parameters. Take a look:

```
class PizzaError(Exception):
    def __init__(self, pizza='uknown', message=''):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza='uknown', cheese='>100', message=''):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese


def make_pizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError
    if cheese > 100:
        raise TooMuchCheeseError
    print("Pizza ready!")


for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        make_pizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)

```

Now, if the circumstances permit, it is possible to use the class names alone.

Summary

1. The `else:` branch of the `try` statement is executed when there has been no exception during the execution of the `try:` block.

2. The `finally:` branch of the `try` statement is always executed.

3. The syntax `except Exception_Name as an exception_object:` lets you intercept an object carrying information about a pending exception. The object's property named `args` (a tuple) stores all arguments passed to the object's constructor.

4. The exception classes can be extended to enrich them with new capabilities, or to adopt their traits to newly defined exceptions.

```
try:
    assert __name__ == "__main__"
except:
    print("fail", end=' ')
else:
    print("success", end=' ')
finally:
    print("done")

```

The code outputs: `success done.`

Quiz

Question 1: What is the expected output of the following code?

```
import math


    try:
        print(math.sqrt(9))
    except ValueError:
        print("inf")
    else:
        print("fine")
```

Question 2: What is the expected output of the following code?

```
import math


    try:
        print(math.sqrt(-9))
    except ValueError:
        print("inf")
    else:
        print("fine")
    finally:
        print("the end")
```

Question 3: What is the expected output of the following code?

```
import math


class NewValueError(ValueError):
    def __init__(self, name, color, state):
        self.data = (name, color, state)


try:
    raise NewValueError("Enemy warning", "Red alert", "High readiness")
except NewValueError as nve:
    for arg in nve.args:
        print(arg, end='! ')
```

Check Answers

PART 4: MISCELLANEOUS

NINETEEN – GENERATORS, ITERATORS, AND CLOSURES

Generator – what do you associate this word with? Perhaps it refers to some electronic device. Or perhaps it refers to a heavy machine designed to produce power, electrical or other. A Python generator is a piece of specialized code able to produce a series of values, and to control the iteration process. This is why generators are very often called iterators, and although some may find a very subtle distinction between these two, we'll treat them as one. You may not realize it, but you've encountered generators many, many times before. Take a look at the very simple snippet:

```
1  for i in range(5):
2  print(i)
3
```

The `range()` function is, in fact, a generator, which is (in fact, again) an iterator. What is the difference? A function returns one, well-defined value – it may be the result of a more or less complex evaluation of, something like a polynomial, and is invoked once – only once. A generator returns a series of values, and in general, is (implicitly) invoked more than once.

In the example, the `range()` generator is invoked six times, providing five subsequent values from zero to four, and finally signaling that the series is complete. This process is completely transparent. Let's shed some light on it. Let's show you the iterator protocol. The iterator protocol is a way in which an object should behave to conform to the rules imposed by the context of the `for` and `in` statements. An object conforming to the iterator protocol is called an iterator.

An iterator must provide two methods: `__iter__()` which should return the object itself and which is invoked once (it's needed for Python to successfully start the iteration); and `__next__()` which is intended to return the next value (first, second, and so on) of the desired series – it will be invoked by the `for/in` statements in order to pass through the next iteration; if there are no more values to provide, the method should raise the `StopIteration` exception. Does it sound strange? Not at all. Look at the following example.

```
1  class Fib:
2      def __init__(self, nn):
3          print("__init__")
4          self.__n = nn
5          self.__i = 0
6          self.__p1 = self.__p2 = 1
7
8      def __iter__(self):
9          print("__iter__")
10         return self
11
12     def __next__(self):
13         print("__next__")
14         self.__i += 1
15         if self.__i > self.__n:
16             raise StopIteration
17         if self.__i in [1, 2]:
18             return 1
19         ret = self.__p1 + self.__p2
20         self.__p1, self.__p2 = self.__p2, ret
21         return ret
22
23
24 for i in Fib(10):
25     print(i)
26
```

We've built a class able to iterate through the first `n` values (where `n` is a constructor parameter) of the Fibonacci numbers. Let us remind you – the Fibonacci numbers (Fib_i) are defined as follows:

$Fib_1 = 1$
$Fib_2 = 1$
$Fib_i = Fib_{i-1} + Fib_{i-2}$

In other words, the first two Fibonacci numbers are equal to 1, and any other Fibonacci number is the sum of the two previous ones (e.g., $Fib_3 = 2$, $Fib_4 = 3$, $Fib_5 = 5$, and so on)

Let's dive into the code:

- lines 2 through 6: the class constructor prints a message (we'll use this to trace the class's behavior), prepares some variables (`__n` to store the series limit, `__i` to track the current Fibonacci number to provide, and `__p1` along with `__p2` to save the two previous numbers);
- lines 8 through 10: the `__iter__` method is obliged to return the iterator object itself; its purpose may be a bit ambiguous here, but there's no mystery; try to imagine an object which is not an iterator (e.g., it's a collection of some entities), but one of its components is an iterator able to scan the collection; the `__iter__` method should extract the iterator and entrust it with the execution of the iteration protocol; as you can see, the method starts its action by printing a message;
- lines 12 through 21: the `__next__` method is responsible for creating the sequence; it's somewhat wordy, but this should make it more readable; first, it prints a message, then it updates the number of desired values, and if it reaches the end of the sequence, the method breaks the iteration by raising the StopIteration exception; the rest of the code is simple, and it precisely reflects the definition we showed you earlier;
- lines 24 and 25 make use of the iterator.

The code produces the following output:

```
__init__
__iter__
__next__
1
__next__
1
__next__
2
__next__
3
__next__
5
__next__
8
__next__
13
__next__
21
__next__
34
__next__
55
__next__
```

The iterator object is instantiated first. Next, Python invokes the `__iter__` method to get access to the actual iterator. Then, the `__next__` method is invoked eleven times – the first ten times produce useful values, while the eleventh terminates the iteration. The previous example shows you a solution where the iterator object is a part of a more complex class. The code isn't really sophisticated, but it presents the concept in a clear way. Take a look at the code:

```
class Fib:
    def __init__(self, nn):
        self.__n = nn
        self.__i = 0
        self.__p1 = self.__p2 = 1

    def __iter__(self):
        print("Fib iter")
        return self

    def __next__(self):
        self.__i += 1
        if self.__i > self.__n:
            raise StopIteration
        if self.__i in [1, 2]:
            return 1
        ret = self.__p1 + self.__p2
        self.__p1, self.__p2 = self.__p2, ret
        return ret

class Class:
    def __init__(self, n):
        self.__iter = Fib(n)

    def __iter__(self):
        print("Class iter")
        return self.__iter


object = Class(8)

for i in object:
    print(i)

```

We've built the `Fib` iterator into another class (we can say that we've composed it into the `Class` class). It's instantiated along with `Class`'s object. The object of the class may be used as an iterator when (and only when) it positively answers to the `__iter__` invocation – this class can do it, and if it's invoked in this way, it provides an object able to obey the iteration protocol. This is why the output of the code is the same as previously, although the object of the `Fib` class isn't used explicitly inside the `for` loop's context.

The `yield` statement

The iterator protocol isn't particularly difficult to understand and use, but it is also indisputable that the protocol is rather inconvenient. The main discomfort it brings is the need to save the state of the iteration between subsequent `__iter__` invocations. For example, the `Fib` iterator is forced to precisely store the place in which the last invocation has been stopped (i.e. The evaluated number and the values of the two previous elements). This makes the code larger and less comprehensible. This is why Python offers a much more effective, convenient, and elegant way of writing iterators.

The concept is fundamentally based on a very specific and powerful mechanism provided by the `yield` keyword. You may think of the `yield` keyword as a smarter sibling of the `return` statement, with one essential difference. Take a look at the following function.

```
def fun(n):
    for i in range(n):
        return i

```

It looks strange, doesn't it? It's clear that the `for` loop has no chance to finish its first execution, as the `return` will break it irrevocably. Moreover, invoking the function won't change anything – the `for` loop will start from scratch and will be broken immediately. We can say that such a function is not able to save and restore its state between subsequent invocations. This also means that a function like this cannot be used as a generator. We've replaced exactly one word in the code – can you see it?

```
def fun(n):
    for i in range(n):
        yield i

```

We've added `yield` instead of return. This little amendment turns the function into a generator, and executing the `yield` statement has some very interesting effects. First of all, it provides the value of the expression specified after the `yield` keyword, just like `return`, but doesn't lose the state of the function. All the variables' values are frozen, and wait for the next invocation, when the execution is resumed (not taken from scratch, like after `return`).

There is one important limitation: such a function should not be invoked explicitly as – in fact – it isn't a function anymore; it's a generator object. The invocation will return the object's identifier, not the series we expect from the generator. Due to the same

reasons, the previous function (the one with the `return` statement) may only be invoked explicitly, and must not be used as a generator.

How to build a generator

How to build a generator

Let us show you the new generator in action. This is how we can use it:

```
def fun(n):
    for i in range(n):
        yield i


for v in fun(5):
    print(v)

```

Can you guess the output? What if you need a generator to produce the first *n* powers of *2*? Nothing easier. Just look at the following code:

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2


for v in powers_of_2(8):
    print(v)

```

Can you guess the output? Run the code to check your guesses.

List comprehensions

Generators may also be used within list comprehensions, just like here:

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2


t = [x for x in powers_of_2(5)]
print(t)

```

Run the example and check the output.

The `list()` function

The `list()` function can transform a series of subsequent generator invocations into a real list:

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2


t = list(powers_of_2(3))
print(t)
```

Again, try to predict the output and run the code to check your predictions.

The `in` operator

Moreover, the context created by The `in` operator allows you to use a generator, too. The example shows how to do it:

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2



for i in range(20):
    if i in powers_of_2(4):
        print(i)
```

What's the code's output? Run the program and check.

The Fibonacci number generator

Now let's see a Fibonacci number generator, and ensure that it looks much better than the object-oriented version based on the direct iterator protocol implementation. Here it is:

```
1  def fibonacci(n):
2      p = pp = 1
3      for i in range(n):
4          if i in [0, 1]:
5              yield 1
6          else:
7              n = p + pp
8              pp, p = p, n
9              yield n
10
11 fibs = list(fibonacci(10))
12 print(fibs)
13
```

Guess the output (a list) produced by the generator, and run the code to check if you're right.

More about list comprehensions

You should be able to remember the rules governing the creation and use of a very special Python phenomenon named list comprehension – a simple and very impressive way of creating lists and their contents. In case you need it, we've provided a quick reminder in the following code.

```
1  list_1 = []
2
3  for ex in range(6):
4  list_1.append(10 ** ex)
5
6  list_2 = [10 ** ex for ex in range(6)]
7
8  print(list_1)
9  print(list_2)
10
```

There are two parts inside the code, both creating a list containing a few of the first natural powers of ten. The first employs a routine way to utilize the `for` loop, while the second makes use of the list comprehension and builds the list in situ, without needing a loop, or any other extended code.

It looks like the list is created inside itself – it's not true, of course, as Python has to perform nearly the same operations as in the first snippet, but it is indisputable that the second formalism is simply more elegant, and lets the reader avoid any unnecessary details. The example outputs two identical lines containing the following text:

```
[1, 10, 100, 1000, 10000, 100000]
[1, 10, 100, 1000, 10000, 100000]
```

Run the code to check if we're right. There is a very interesting syntax we want to show you now. Its usability is not limited to list comprehensions, but we have to admit that comprehensions are the ideal environment for it. It's a conditional expression – a way of selecting one of two different values based on the result of a Boolean expression. Look:

expression_one if condition else *expression_two*

It may look a bit surprising at first glance, but you have to keep in mind that it is not a conditional instruction. Moreover, it's not an instruction at all. It's an operator. The value it provides is equal to `expression_one` when the condition is `True`, and `expression_two` otherwise. A good example will tell you more. Look at the code.

```
1  the_list = []
2
3  for x in range(10):
4      the_list.append(1 if x % 2 == 0 else 0)
5
6  print(the_list)
7
```

The code fills a list with `1`s and `0`s – if the index of a particular element is odd, the element is set to `0`, and to `1` otherwise. Simple? Maybe not at first glance. Elegant? Indisputably. Can you use the same trick within a list comprehension? Yes, you can. Look at the example.

```
1  the_list = [1 if x % 2 == 0 else 0 for x in range(10)]
2
3  print(the_list)
4
```

Compactness and elegance – these two words come to mind when looking at the code. So, what do they have in common, generators and list comprehensions? Is there any connection between them? Yes, a rather loose connection, but an unequivocal one. Just one change can turn any list comprehension into a generator.

List comprehensions vs. generators

Now look at the following code and see if you can find the detail that turns a list comprehension into a generator:

```
1  the_list = [1 if x % 2 == 0 else 0 for x in range(10)]
2  the_generator = (1 if x % 2 == 0 else 0 for x in range(10))
3
4  for v in the_list:
5      print(v, end=   )
6  print()
7
8  for v in the_generator:
9      print(v, end=" ")
10 print()
11
```

It's the parentheses. The brackets make a comprehension, the parentheses make a generator. The code, however, when run, produces two identical lines:

```
1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0
```

How can you know that the second assignment creates a generator, not a list? There is some proof we can show you. Apply the `len()` function to both these entities. `len(the_list)` will evaluate to `10`. Clear and predictable. `len(the_generator)` will raise an exception, and you will see the following message:

```
TypeError: object of type 'generator' has no len()
```

Of course, saving either the list or the generator is not necessary – you can create them exactly in the place where you need them – just like here:

```
1  for v in [1 if x % 2 == 0 else 0 for x in range(10)]:
2      print(v, end=" ")
3  print()
4
5  for v in (1 if x % 2 == 0 else 0 for x in range(10)):
6      print(v, end=" ")
7  print()
8
```

NOTE: The same appearance of the output doesn't mean that both loops work in the same way. In the first loop, the list is created and iterated through as a whole – it actually exists when the loop is being executed.

In the second loop, there is no list at all – there are only subsequent values produced by the generator, one by one. Carry out your own experiments.

The lambda function

The lambda function is a concept borrowed from mathematics, more specifically, from a part called *the Lambda calculus*, but these two phenomena are not the same. Mathematicians use *the Lambda calculus* in many formal systems connected with logic, recursion, or theorem probability. Programmers use the lambda function to simplify the code, to make it clearer and easier to understand.

A lambda function is a function without a name (you can also call it an anonymous function). Of course, such a statement immediately raises the question: how do you use anything that cannot be identified? Fortunately, it's not a problem, as you can name such a function if you really need, but, in fact, in many cases the lambda function can exist and work while remaining fully incognito. The declaration of the lambda function doesn't resemble a normal function declaration in any way – see for yourself:

```
lambda parameters: expression
```

Such a clause returns the value of the expression when taking into account the current value of the current lambda argument. As usual, an example will be helpful. Our example uses three lambda functions, but gives them names. Look at it carefully:

```
two = lambda: 2
sqr = lambda x: x * x
pwr = lambda x, y: x ** y

for a in range(-2, 3):
    print(sqr(a), end='   ')
    print(pwr(a, two()))
```

Let's analyze it. The first lambda is an anonymous parameter less function that always returns 2. As we've assigned it to a variable named two, we can say that the function is not anonymous anymore, and we can use the name to invoke it. The second one is a one-parameter anonymous function that returns the value of its squared argument. We've named it as such, too. The third lambda takes two parameters and returns the value of the first one raised to the power of the second one. The name of the variable which carries the lambda speaks for itself. We don't use pow in order to avoid confusion with the built-in function of the same name and the same purpose. The program produces the following output:

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

This example is clear enough to show how lambdas are declared and how they behave, but it says nothing about why they're necessary, and what they're used for, since they can all be replaced with routine Python functions. Where is the benefit?

How to use lambdas and what for?

The most interesting thing about lambdas is that you can use them in their pure form – as anonymous parts of code intended to evaluate a result. Imagine that we need a function (we'll name it print_function) which prints the values of a given (other) function for a set of selected arguments. We want print_function to be universal – it should accept a set of arguments put in a list and a function to be evaluated, both as arguments – we don't want to hardcode anything. Look at the example. This is how we've implemented the idea.

```
def print_function(args, fun):
    for x in args:
        print('f(', x,')=', fun(x), sep='')


def poly(x):
    return 2 * x**2 - 4 * x + 2


print_function([x for x in range(-2, 3)], poly)
```

Let's analyze it. The print_function() function takes two parameters: the first, a list of arguments for which we want to print the results; and the second, a function which should be invoked as many times as the number of values that are collected inside the first parameter.

NOTE: We've also defined a function named poly() – this is the function whose values we're going to print. The calculation the function performs isn't very sophisticated – it's the polynomial (hence its name) of the form:

$$f(x) = 2x^2 - 4x + 2$$

The name of the function is then passed to the print_function() along with a set of five different arguments – the set is built with a list comprehension clause. The code prints the following lines:

```
f(-2)=18
f(-1)=8
f(0)=2
f(1)=0
f(2)=2
```

Can we avoid defining the `poly()` function, as we're not going to use it more than once? Yes, we can – this is the benefit a lambda can bring. Look at the following example. Can you see the difference?

```
1  def print_function(args, fun):
2      for x in args:
3          print('f(', x,')=', fun(x), sep='')
4
5  print_function([x for x in range(-2, 3)], lambda x: 2 * x**2 - 4 * x + 2)
6
```

The `print_function()` has remained exactly the same, but there is no `poly()` function. We don't need it anymore, as the polynomial is now directly inside the `print_function()` invocation in the form of a lambda defined in the following way:

```
1  lambda x: 2 * x**2 - 4 * x + 2
2
```

The code has become shorter, clearer, and more legible. Let us show you another place where lambdas can be useful. We'll start with a description of `map()`, a built-in Python function. Its name isn't too descriptive, its idea is simple, and the function itself is really usable.

Lambdas and the `map()` function

In the simplest of all possible cases, The `map()` function:

```
1  map(function, list)
2
```

It takes two arguments: a function and a list. This description is extremely simplified, as the second `map()` argument may be any entity that can be iterated, such as a tuple, or just a generator), and `map()` can accept more than two arguments. The `map()` function applies the function passed by its first argument to all its second argument's elements, and returns an iterator delivering all subsequent function results. You can use the resulting iterator in a loop, or convert it into a list using the `list()` function. Can you see a role for any lambda here? Look at the code – we've used two lambdas in it.

```
1  list_1 = [x for x in range(5)]
2  list_2 = list(map(lambda x: 2 ** x, list_1))
3  print(list_2)
4
5  for x in map(lambda x: x * x, list_2):
6      print(x, end=' ')
7  print()
8
```

This is the intrigue: build the `list_1` with values from `0` to `4`; next, use `map` along with the first `lambda` to create a new list in which all elements have been evaluated as `2` raised to the power taken from the corresponding element from `list_1`; `list_2` is printed then; in the next step, use the `map()` function again to make use of the generator it returns and to directly print all the values it delivers; as you can see, we've engaged the second `lambda` here – it just squares each element from `list_2`. Try to imagine the same code without lambdas. Would it be any better? It's unlikely.

Lambdas and the `filter()` function

Another Python function which can be significantly beautified by the application of a lambda is `filter()`. It expects the same kind of arguments as `map()`, but does something different – it filters its second argument while being guided by directions flowing from the function specified as the first argument. The function is invoked for each list element, just like in `map()`. The elements which return `True` from the function pass the filter – the others are rejected. The example shows the `filter()` function in action.

```
from random import seed, randint

seed()
data = [randint(-10,10) for x in range(5)]
filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))

print(data)
print(filtered)

```

NOTE: We've made use of the `random` module to initialize the random number generator (not to be confused with the generators we've just talked about) with the `seed()` function, and to produce five random integer values from `-10` to `10` using the `randint()` function. The list is then filtered, and only the numbers which are greater than zero and even areaccepted. Of course, it's not likely that you'll receive the same results, but this is what our results looked like:

```
[6, 3, 3, 2, -7]
[6, 2]
```

A brief look at closures

Let's start with a definition: closure is a technique which allows the storing of values in spite of the fact that the context in which they have been created does not exist anymore. Intricate? A bit. Let's analyze a simple example:

```
def outer(par):
    loc = par


var = 1
outer(var)

print(par)
print(loc)

```

The example is erroneous. The last two lines will cause a `NameError` exception – neither `par` nor `loc` is accessible outside the function. Both the variables exist when and only when the `outer()` function is being executed. Look at this example. We've modified the code significantly.

```
def outer(par):
    loc = par

    def inner():
        return loc
    return inner


var = 1
fun = outer(var)
print(fun())

```

There is a brand new element in it – a function (named `inner`) inside another function (named `outer`). How does it work? Just like any other function except for the fact that `inner()` may be invoked only from within `outer()`. We can say that `inner()` is `outer()`'s private tool – no other part of the code can access it.

The `inner()` function returns the value of the variable accessible inside its scope, as `inner()` can use any of the entities at the disposal of `outer()`. The `outer()` function returns the `inner()` function itself; more precisely, it returns a copy of the `inner()` function, the one which was frozen at the moment of `outer()`'s invocation; the frozen function contains its full environment, including the state of all local variables, which also means that the value of `loc` is successfully retained, although `outer()` ceased to exist a long time ago. In effect, the code is fully valid, and outputs:

```
1
```

The function returned during the `outer()` invocation is a closure. A closure has to be invoked in exactly the same way in which it has been declared. In the following example:

```
def outer(par):
    loc = par

    def inner():
        return loc
    return inner


var = 1
fun = outer(var)
print(fun())

```

The `inner()` function is parameter less, so we have to invoke it without arguments. Now look at the following code. It is fully possible to declare a closure equipped with an arbitrary number of parameters, e.g. one, just like the `power()` function.

```
def make_closure(par):
    loc = par

    def power(p):
        return p ** loc
    return power


fsqr = make_closure(2)
fcub = make_closure(3)

for i in range(5):
    print(i, fsqr(i), fcub(i))

```

This means that the closure not only makes use of the frozen environment, but it can also modify its behavior by using values taken from the outside. This example shows one more interesting circumstance – you can create as many closures as you want using one and the same piece of code. This is done with a function named `make_closure()`. Note: the first closure obtained from `make_closure()` defines a tool squaring its argument; while the second one is designed to cube the argument. This is why the code produces the following output:

```
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
```

Summary

1. An iterator is an object of a class providing at least two methods, not counting the constructor: `__iter__()` is invoked once when the iterator is created and returns the iterator's object itself; `__next__()` is invoked to provide the next iteration's value and raises the `StopIteration` exception when the iteration comes to an end.

2. The `yield` statement can be used only inside functions. The `yield` statement suspends function execution and causes the function to return the yield's argument as a result. Such a function cannot be invoked in a regular way – its only purpose is to be used as a generator (i.e. in a context that requires a series of values, like a `for` loop).

3. A conditional expression is an expression built using the `if-else` operator. For example, this outputs `True`:

```
print(True if 0 >= 0 else False)

```

4. A list comprehension becomes a generator when used inside parentheses (used inside brackets, it produces a regular list). For example:

```
for x in (el * 2 for el in range(5)):
print(x)

```

outputs `02468`.

5. A lambda function is a tool to create anonymous functions. For example:

```
def foo(x, f):
    return f(x)

print(foo(9, lambda x: x ** 0.5))

```

outputs `3.0`.

6. The `map(fun, list)` function creates a copy of a `list` argument, and applies the `fun` function to all of its elements, returning a generator that provides the new list content element by element. For example:

```
short_list = ['mython', 'python', 'fell', 'on', 'the', 'floor']
new_list = list(map(lambda s: s.title(), short_list))
print(new_list)

```

outputs `['Mython', 'Python', 'Fell', 'On', 'The', 'Floor']`.

7. The `filter(fun, list)` function creates a copy of those `list` elements, which cause the `fun` function to return `True`. The function's result is a generator providing the new list content element by element. For example:

```
short_list = [1, "Python", -1, "Monty"]
new_list = list(filter(lambda s: isinstance(s, str), short_list))
print(new_list)

```

outputs `['Python', 'Monty']`.

8. A closure is a technique which can store values even if the context in which they have been created does not exist anymore. For example:

```
def tag(tg):
    tg2 = tg[0] + '/' + tg[1:]

    def inner(str):
        return tg + str + tg2
    return inner


b_tag = tag('<b>')
print(b_tag('Monty Python'))

```

outputs `<b>Monty Python</b>`

Quiz

Question 1: What is the expected output of the following code?

```
class Vowels:
    def __init__(self):
        self.vow = "aeiouy " # Yes, we know that y is not always considered a
vowel.
        self.pos = 0


    def __iter__(self):
        return self


    def __next__(self):
        if self.pos == len(self.vow):
            raise StopIteration
        self.pos += 1
        return self.vow[self.pos - 1]
```

```
vowels = Vowels()
for v in vowels:
    print(v, end=' ')
```

Question 2: Write a lambda function, setting the least significant bit of its integer argument, and apply it to The `map()` function to produce the string `1 3 3 5` on the console.

```
any_list = [1, 2, 3, 4]
even_list = # Complete the line here.
print(even_list)
```

Question 3: What is the expected output of the following code?

```
def replace_spaces(replacement='*'):
    def new_replacement(text):
        return text.replace(' ', replacement)
    return new_replacement



stars = replace_spaces()
print(stars("And Now for Something Completely Different"))
```

Check Answers

NOTE PEP 8, the Style Guide for Python Code, recommends that lambdas should not be assigned to variables, but rather they should be defined as functions.

This means that it is better to use a def statement, and avoid using an assignment statement that binds a lambda expression to an identifier. Analyze the following code:

```
1  # Recommended:
2  def f(x): return 3*x
3
4
5  # Not recommended:
6  f = lambda x: 3*x
7
```

Binding lambdas to identifiers generally duplicates the functionality of the `def` statement. Using `def` statements, on the other hand, generates more lines of code.

It is important to understand that reality often likes to draw its own scenarios, which do not necessarily follow the conventions or formal recommendations. Whether you decide to follow them or not will depend on many things: your preferences, other conventions adopted, company internal guidelines, compatibility with existing code, etc. Be aware of this.

TWENTY – FILES (FILE STREAMS, FILE PROCESSING, DIAGNOSING STREAM PROBLEMS)

One of the most common issues in the developer's job is to process data stored in files while the files are usually physically stored using storage devices – hard, optical, network, or solid-state disks. It's easy to imagine a program that sorts twenty numbers, and it's equally easy to imagine the user of this program entering these twenty numbers directly from the keyboard. It's much harder to imagine the same task when there are 20,000 numbers to be sorted, and there isn't a single user who is able to enter these numbers without making a mistake. It's much easier to imagine that these numbers are stored in the disk file which is read by the program. The program sorts the numbers and doesn't send them to the screen, but instead creates a new file and saves the sorted sequence of numbers there.

If we want to implement a simple database, the only way to store the information between program runs is to save it into a file (or files if your database is more complex). In principle, any non-trivial programming problem relies on the use of files, whether it processes images (stored in files) or multiplies matrices (stored in files), or calculates wages and taxes (reading data stored in files). You may ask why we have waited until now to show you these issues. The answer is very simple – Python's way of accessing and processing files is implemented using a consistent set of objects. There is no better moment to talk about it.

File names

Different operating systems can treat the files in different ways. For example, Windows uses a different naming convention than the one adopted in Unix/Linux systems. If we use the notion of a canonical file name (a name which uniquely defines the location of the file regardless of its level in the directory tree) we can realize that these names look different in Windows and in Unix/Linux.

In addition, Unix/Linux system file names are case-sensitive. Windows systems store the case of letters used in the file name, but don't distinguish between their cases at all. This means that these two strings: `ThisIsTheNameOfTheFile` and `thisisthenameofthefile` describe two different files in Unix/Linux systems, but are the same name for just one file in Windows systems.

The main and most striking difference is that you have to use two different separators for the directory names: `\` in Windows, and `/` in Unix/Linux. This difference is not very important to the normal user, but is very important when writing programs in Python. To understand why, try to recall the very specific role played by the `\` inside Python strings.

Suppose you're interested in a particular file located in the directory `dir`, and named `file`. Suppose also that you want to assign a string containing the name of the file. In Unix/Linux systems, it may look as follows:

```
name = "/dir/file"
```

But if you try to code it for the Windows system:

```
name = "\dir\file"
```

You'll get an unpleasant surprise: either Python will generate an error, or the execution of the program will behave strangely, as if the file name has been distorted in some way. In fact, it's not strange at all, but quite obvious when you think about it for a moment. Python uses the `\` as an escape character (like `\n`).

This means that Windows file names must be written as follows:

```
name = "\\dir\\file"
```

Fortunately, there is also one more solution. Python is smart enough to be able to convert slashes into backslashes each time it discovers that it's required by the OS. This means that the following assignments:

```
name = "/dir/file"
name = "c:/dir/file"
```

will work with Windows, too. Any program written in Python (and not only in Python, because that convention applies to virtually all programming languages) does not communicate with the files directly, but through some abstract entities that are named differently in different languages or environments – the most-used terms are handles or streams, and we'll use them as synonyms here.

The programmer, having a more- or less-rich set of functions/methods, is able to perform certain operations on the stream, which affect the real files using mechanisms contained in the operating system kernel. In this way, you can implement the process of accessing any file, even when the name of the file is unknown at the time of writing the program.

The operations performed with the abstract stream reflect the activities related to the physical file. To connect (bind) the stream with the file, it's necessary to perform an explicit operation. The operation of connecting the stream with a file is called opening the file, while disconnecting this link is named closing the file. Hence, the conclusion is that the very first operation performed on the stream is always `open` and the last one is `close`. The program, in effect, is free to manipulate the stream between these two events and to handle the associated file. This freedom is limited, of course, by the physical characteristics of the file and the way in which the file has been opened.

Let us say again that the opening of the stream can fail, and it may happen due to several reasons: the most common is the lack of a file with a specified name. It can also happen that the physical file exists, but the program is not allowed to open it. There's also the risk that the program has opened too many streams, and the specific operating system may not allow the simultaneous opening of more than `n` files (e.g. 200). A well-written program should detect these failed openings, and react accordingly.

File streams

The opening of the stream is not only associated with the file, but should also declare the manner in which the stream will be processed. This declaration is called an open mode. If the opening is successful, the program will be allowed to perform only the operations which are consistent with the declared open mode.

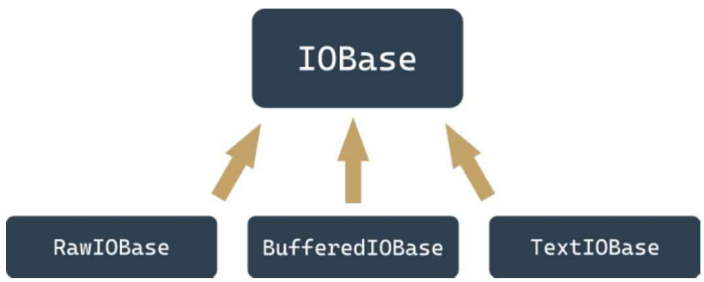
There are two basic operations performed on the stream: read from the stream: the portions of the data are retrieved from the file and placed in a memory area managed by the program (e.g. a variable); and write to the stream: the portions of the data from the memory (e.g. a variable) are transferred to the file.

There are three basic modes used to open the stream. The first is read mode. A stream opened in this mode allows read operations only; trying to write to the stream will cause an exception (the exception is named UnsupportedOperation, which inherits OSError and ValueError, and comes from the `io` module). The second is write mode. A stream opened in this mode allows write operations only; attempting to read the stream will cause the previously mentioned exception. The third is update mode. A stream opened in this mode allows both writes and reads.

Before we discuss how to manipulate the streams, we owe you some explanation. The stream behaves almost like a tape recorder. When you read something from a stream, a virtual head moves over the stream according to the number of bytes transferred from the stream. When you write something to the stream, the same head moves along the stream recording the data from the memory. Whenever we talk about reading from and writing to the stream, try to imagine this analogy. The programming books refer to this mechanism as the current file position, and we'll also use this term. It's necessary now to show you the object responsible for representing streams in programs.

File handles

Python assumes that every file is hidden behind an object of an adequate class. Of course, it's hard not to ask how to interpret the word adequate. Files can be processed in many different ways – some of them depend on the file's contents, some on the programmer's intentions. In any case, different files may require different sets of operations, and behave in different ways. An object of an adequate class is created when you open the file and annihilate it at the time of closing. Between these two events, you can use the object to specify what operations should be performed on a particular stream. The operations you're allowed to use are imposed by the way in which you've opened the file. In general, the object comes from one of the classes shown here:



NOTE: You never use constructors to bring these objects to life. The only way you obtain them is to invoke the function named `open()`.

The function analyses the arguments you've provided, and automatically creates the required object. If you want to get rid of the object, you invoke the method named `close()`. The invocation will sever the connection to the object and the file, and will remove the object.

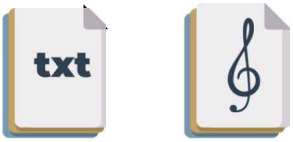
For our purposes, we'll concern ourselves only with streams represented by `BufferIOBase` and `TextIOBase` objects. You'll understand why soon. Due to the type of the stream's contents, all the streams are divided into text and binary streams.

The text streams are structured in lines; that is, they contain typographical characters (letters, digits, punctuation, etc.) arranged in rows (lines), as seen with the naked eye when you look at the contents of a file in an editor. This file is written (or read) mostly character by character, or line by line. The binary streams don't contain text but a sequence of bytes of any value. This sequence can be, for example, an executable program, an image, an audio or a video clip, a database file, etc. Because these files don't contain lines, the reads and writes relate to portions of data of any size. Hence the data is read/written byte by byte, or block by block, where the size of the block usually ranges from one to an arbitrarily chosen value.

Then comes a subtle problem. In Unix/Linux systems, the line ends are marked by a single character named `LF` (ASCII code 10) designated in Python programs as `\n`. Other operating systems, especially these derived from the prehistoric CP/M system (which applies to Windows family systems, too) use a different convention: the end of line is marked by a pair of characters, `CR` and `LF` (ASCII codes 13 and 10) which can be encoded as `\r\n`.

TThis ambiguity can cause various unpleasant consequences. If you create a program responsible for processing a text file, and it is written for Windows, you can recognize the ends of the lines by finding the `\r\n` characters, but the same program running in a Unix/Linux environment can be completely useless, and vice versa: the program written for Unix/Linux systems may be useless in Windows.

Such undesirable features of the program, which prevents or hinders the use of the program in different environments, is called nonportability. Similarly, the trait of the program allowing execution in different environments is called portability. A program endowed with such a trait is called a portable program. Since portability issues were (and still are) very serious, a decision was made to definitely resolve the issue in a way that doesn't engage the developer's attention.



It was done at the level of classes, which are responsible for reading and writing characters to and from the stream. It works in the following way: when the stream is open and it's advised that the data in the associated file will be processed as text (or there is no such advisory at all), it is switched into text mode. During reading/writing of lines from/to the associated file, nothing special occurs in the Unix environment, but when the same operations are performed in the Windows environment, a process called a translation of newline characters occurs: when you read a line from the file, every pair of `\r\n` characters is replaced with a single `\n` character, and vice versa; during write operations, every `\n` character is replaced with a pair of `\r\n` characters. The mechanism is completely transparent to the program, which can be written as if it was intended for processing Unix/ Linux text files only; the source code run in a Windows environment will work properly, too. When the stream is open and it's advised to do so, its contents are taken as-is, without any conversion – no bytes are added or omitted.

Opening the streams

The opening of the stream is performed by a function which can be invoked in the following way:

```
1  stream = open(file, mode = 'r', encoding = None)
2
```

Let's analyze it. The name of the function (`open`) speaks for itself; if the opening is successful, the function returns a stream object; otherwise, an exception is raised (e.g. FileNotFoundError if the file you're going to read doesn't exist). The first parameter of the function (`file`) specifies the name of the file to be associated with the stream. The second parameter (`mode`) specifies the open mode used for the stream; it's a string filled with a sequence of characters, and each of them has its own special meaning (more details soon). The third parameter (`encoding`) specifies the encoding type (e.g. UTF-8 when working with text files). The opening must be the very first operation performed on the stream.

NOTE: The mode and encoding arguments may be omitted – their default values are assumed then. The default opening mode is read in text mode, while the default encoding depends on the platform used.

Let us now present you with the most important and useful open modes. Ready?

Opening the streams: modes

`r` open mode: read

- the stream will be opened in read mode;
- the file associated with the stream must exist and has to be readable, otherwise The `open()` function raises an exception.

`w` open mode: write

- the stream will be opened in write mode;
- the file associated with the stream doesn't need to exist; if it doesn't exist it will be created; if it exists, it will be truncated to the length of zero (erased); if the creation isn't possible (e.g. due to system permissions) The `open()` function raises an exception.

`a` open mode: append

- the stream will be opened in append mode;
- the file associated with the stream doesn't need to exist; if it doesn't exist, it will be created; if it exists the virtual recording head will be set at the end of the file (the previous content of the file remains untouched.)

`r+` open mode: read and update

- the stream will be opened in read and update mode;
- the file associated with the stream must exist and has to be writeable, otherwise The `open()` function raises an exception;
- both read and write operations are allowed for the stream.

`w+` open mode: write and update

- the stream will be opened in write and update mode;
- the file associated with the stream doesn't need to exist; if it doesn't exist, it will be created; the previous content of the file remains untouched;
- both read and write operations are allowed for the stream.

Selecting text and binary modes

If there is a letter `b` at the end of the mode string, it means that the stream is to be opened in binary mode. If the mode string ends with a letter `t`, the stream is opened in text mode. Text mode is the default behavior assumed when no binary/text mode specifier is used. Finally, the successful opening of a file will set the current file position (the virtual reading/writing head) before the first byte of the file if the mode is not `a` and after the last byte of the file if the mode is set to `a`.

TEXT MODE	BINARY MODE	DESCRIPTION
rt	rb	read
wt	wb	write
at	ab	append
r+t	r+b	read and update
w+t	w+b	write and update

EXTRA You can also open a file for its exclusive creation. You can do this using the `x` open mode. If the file already exists, the `open()` function will raise an exception.

Opening the stream for the first time

Imagine that we want to develop a program that reads the contents of the text file named: `C:\Users\User\Desktop\file.txt`. How do we open that file for reading? Here's a relevant snippet of the code:

```
1  try:
2      stream = open(r"C:\Users\User\Desktop\file.txt", "rt")
3  # Processing goes here.
4      stream.close()
5  except Exception as exc:
6      print("Cannot open the file:", exc)
7
```

What's going on here?

We open the try-except block as we want to handle runtime errors softly. We use the `open()` function to try to open the specified file (note the way we've specified the file name). The open mode is defined as text to read, as text is the default setting, we can skip the `t` in the mode string. If we're successful, we get an object from the `open()` function and we assign it to the stream variable. if `open()` fails, we handle the exception by printing the full error information, because it's definitely good to know what exactly happened.

Pre-opened streams

We said earlier that any stream operation must be preceded by the `open()` function invocation. There are three well-defined exceptions to the rule. When our program starts, the three streams are already opened and don't require any extra preparations. What's more, your program can use these streams explicitly if you take care to import the sys module:

```
1  import sys
2
```

That's where the declaration of the three streams is placed. The names of these streams are: `sys.stdin`, `sys.stdout`, and `sys.stderr`.

Let's analyze them:

`sys.stdin`

- `stdin` (standard input)
- The `stdin` stream is normally associated with the keyboard, pre-open for reading and regarded as the primary data source for the running programs;
- the well-known `input()` function reads data from `stdin` by default.

`sys.stdout`

- `stdout` (standard output)
- The `stdout` stream is normally associated with the screen, pre-open for writing, regarded as the primary target for outputting data by the running program;
- the well-known `print()` function outputs the data to The `stdout` stream.

`sys.stderr`

- `stderr` (standard error output)
- The `stderr` stream is normally associated with the screen, pre-open for writing, regarded as the primary place where the running program should send information on the errors encountered during its work;
- we haven't presented any method to send the data to this stream (we will do it soon, we promise)
- the separation of `stdout` (useful results produced by the program) from The `stderr` (error messages, undeniably useful but does not provide results) gives the possibility of redirecting these two types of information to the different targets. More extensive discussion of this issue is beyond the scope of our course. The operation system handbook will provide more information on these issues.

Closing streams

The last operation performed on a stream should be closing. This doesn't include the `stdin`, `stdout`, and `stderr` streams, which don't require it. That action is performed by a method invoked from within the open stream object: `stream.close()`.

The name of the function is definitely self-commenting: `close()`. The function expects exactly no arguments; the stream doesn't need to be opened. The function returns nothing, but raises an IOError exception in case of error.

Most developers believe that the `close()` function always succeeds and thus there is no need to check if it's done its task properly. This belief is only partly justified. If the stream was opened for writing and then a series of write operations were performed, it may happen that the data sent to the stream has not been transferred to the physical device yet (due to a mechanism called caching or buffering). Since the closing of the stream forces the buffers to flush them, it may be that the flushes fail and therefore the `close()` fails too.

We have already mentioned failures caused by functions operating with streams, but we haven't said a word about how exactly we can identify the cause of the failure. The possibility of making a diagnosis exists and is provided by one of the streams' exception components, which we are going to tell you about just now.

Diagnosing stream problems

The `IOError` object is equipped with a property named `errno` (the name comes from the phrase *error number*) and you can access it as follows:

```
try:
    # Some stream operations.
except IOError as exc:
    print(exc.errno)

```

The value of The `errno` attribute can be compared with one of the predefined symbolic constants defined in The `errno` module.

Let's take a look at some selected constants useful for detecting stream errors:

- `errno.EACCES` → Permission denied
  The error occurs when you try, for example, to open a file with the *read only* attribute for writing.
- `errno.EBADF` → Bad file number
  The error occurs when you try, for example, to operate with an unopened stream.
- `errno.EEXIST` → File exists
  The error occurs when you try, for example, to rename a file with its previous name.
- `errno.EFBIG` → File too large
  The error occurs when you try to create a file that is larger than the maximum allowed by the operating system.
- `errno.EISDIR` → Is a directory
  The error occurs when you try to treat a directory name as the name of an ordinary file.
- `errno.EMFILE` → Too many open files
  The error occurs when you try to simultaneously open more streams than acceptable for your operating system.
- `errno.ENOENT` → No such file or directory
  The error occurs when you try to access a non-existent file/directory.
- `errno.ENOSPC` → No space left on device
  The error occurs when there is no free space on the media.

The complete list is much longer (it also includes some error codes not related to the stream processing.) If you are a very careful programmer, you may feel the need to use a sequence of statements similar to those presented in the following code.

```
import errno

try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # Actual processing goes here.
    s.close()
except Exception as exc:
    if exc.errno == errno.ENOENT:
        print("The file doesn't exist.")
    elif exc.errno == errno.EMFILE:
        print("You've opened too many files.")
    else:
        print("The error number is:", exc.errno)

```

Fortunately, there is a function that can dramatically simplify the error handling code. Its name is `strerror()`, and it comes from the `os` module and expects just one argument – an error number. Its role is simple: you give an error number and get a string describing the meaning of the error.

NOTE: If you pass a non-existent error code — a number which is not bound to any actual error — the function will raise a `ValueError` exception.

Now we can simplify our code in the following way:

```
from os import strerror

try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # Actual processing goes here.
    s.close()
except Exception as exc:
    print("The file could not be opened:", strerror(exc.errno))

```

Okay. Now it's time to deal with text files and get familiar with some basic techniques you can use to process them.

Summary

1. A file needs to be open before it can be processed by a program, and it should be closed when the processing is finished. Opening the file associates it with the stream, which is an abstract representation of the physical data stored on the media. The way in which the stream is processed is called open mode. Three open modes exist:

- read mode – only read operations are allowed;
- write mode – only write operations are allowed;
- update mode – both writes and reads are allowed.

2. Depending on the physical file content, different Python classes can be used to process files. In general, the `BufferedIOBase` is able to process any file, while `TextIOBase` is a specialized class dedicated to processing text files (i.e. files containing human-visible texts divided into lines using new-line markers). Thus, the streams can be divided into binary and text ones.

3. The following `open()` function syntax is used to open a file:

open(file_name, mode=open_mode, encoding=text_encoding)

The invocation creates a stream object and associates it with the file named `file_name`, using the specified `open_mode` and setting the specified `text_encoding`, or it raises an exception in the case of an error.

4. Three predefined streams are already open when the program starts:

- `sys.stdin` – standard input;
- `sys.stdout` – standard output;
- `sys.stderr` – standard error output.

5. The `IOError` exception object, created when any file operations fails (including open operations), contains a property named `errno`, which contains the completion code of the failed action. Use this value to diagnose the problem.

Quiz

Question 1: How do you encode an `open()` function's `mode` argument value if you're going to create a new text file to only fill it with an article?

Question 2: What is the meaning of the value represented by `errno.EACESS`?

Question 3: What is the expected output of the following code, assuming that the file named *file* does not exist?

```
import errno


try:
    stream = open("file", "rb")
    print("exists")
    stream.close()
except IOError as error:
    if error.errno == errno.ENOENT:
    print("absent")
    else:
    print("unknown")
```

Check Answers

TWENTY-ONE – WORKING WITH REAL FILES

In this lesson we're going to prepare a simple text file with some short, simple content. We're going to show you some basic techniques you can utilize to read the file contents in order to process them. The processing will be very simple – you're going to copy the file's contents to the console, and count all the characters the program has read in. But remember – our understanding of a text file is very strict. In our sense, it's a plain text file – it may contain only text, without any additional decorations (formatting, different fonts, etc.). That's why you should avoid creating the file using any advanced text processor like MS Word, LibreOffice Writer, or something like this. Use the very basics your OS offers: Notepad, vim, gedit, etc.

If your text files contain some national characters not covered by the standard ASCII charset, you may need an additional step. Your `open()` function invocation may require an argument denoting specific text encoding. For example, if you're using a Unix/Linux OS configured to use UTF-8 as a system-wide setting, the `open()` function may look as follows:

```
stream = open('file.txt', 'rt', encoding='utf-8')

```

Where the encoding argument has to be set to a value which is a string representing proper text encoding (UTF-8, here). Consult your OS documentation to find an encoding name adequate to your environment.

NOTE For the purposes of our experiments with file processing carried out in this chapter, we're going to use a ready set of files (i.e. `tzop.txt`, or `text.txt` files) which you'll be able to work with. If you'd like to work with your own files locally on your machine, we strongly encourage you to do so, and to use IDLE to carry out your own tests. Go to https://edube.org to find the files on the Python Essentials 2 course.

Reading a text file's contents can be performed using several different methods – none of them is any better or worse than any other. It's up to you which of them you prefer. Some of them will sometimes be handier, and sometimes more troublesome. Be flexible. Don't be afraid to change your preferences. The most basic of these methods is the one offered by the `read()` function, which you were able to see in action in the previous lesson. If applied to a text file, the function is able to: read a desired number of characters (including just one) from the file, and return them as a string; read all the file contents, and return them as a string; if there is nothing more to read, in other words, the virtual reading head reaches the end of the file, the function returns an empty string.

We'll start with the simplest variant and use a file named `text.txt`. The file has the following contents:

```
Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.
```

Now look at the code, and let's analyze it.

```
from os import strerror

try:
    counter = 0
    stream = open('text.txt', "rt")
    char = stream.read(1)
    while char != '':
        print(char, end='')
        counter += 1
        char = stream.read(1)
    stream.close()
    print("\n\nCharacters in file:", counter)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))

```

The routine is rather simple. Use the try-except mechanism and open the file of the predetermined name (`text.txt` in our case). Try to read the very first character from the file (`char = stream.read(1)`). If you succeed and you get a positive result of the `while` condition check, output the character. Note the `end=` argument – it's important! You don't want to skip to a new line after every character! Now update the counter (`counter`). Try to read the next character, and the process repeats.

If you're absolutely sure that the file's length is safe and you can read the whole file to the memory at once, you can do it – the `read()` function, invoked without any arguments or with an argument that evaluates to `None`, will do the job for you.

Remember – reading a terabyte-long file using this method may corrupt your OS. Don't expect miracles, computer memory isn't stretchable. Look at the following code. What do you think of it?

```
from os import or

try:
    counter = 0
    stream = open('text.txt', "rt")
    content = stream.read()
    for char in content:
        print(char, end='')
        counter += 1
    stream.close()
    print("\n\nCharacters in file:", counter)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))

```

Let's analyze it. Open the file as previously. Read its contents by one `read()` function invocation. Next, process the text, iterating through it with a regular `for` loop, and updating the counter value at each turn of the loop. The result will be exactly the same as previously.

readline()

If you want to treat the file's contents as a set of lines, not a bunch of characters, the `readline()` method will help you with that. The method tries to read a complete line of text from the file, and returns it as a string in the case of success. Otherwise, it returns an empty string. This opens up new opportunities – now you can also count lines easily, not only characters. Let's make use of it. Look at this code.

```
from os import strerror

try:
    character_counter = line_counter = 0
    stream = open('text.txt', 'rt')
    line = stream.readline()
    while line != '':
        line_counter += 1
        for char in line:
            print(char, end=  )
            character_counter += 1
        line = stream.readline()
    stream.close()
    print("\n\nCharacters in file:", character_counter)
    print("Lines in file:     ", line_counter)
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))

```

As you can see, the general idea is exactly the same as in both previous examples.

readlines()

Another method, which treats text file as a set of lines, not characters, is `readlines()`. The `readlines()` method, when invoked without arguments, tries to read all the file contents, and returns a list of strings, one element per file line. If you're not sure if the file size is small enough and don't want to test the OS, you can convince the `readlines()` method to read not more than a specified number of bytes at once (the returning value remains the same – it's a list of a string). Feel free to experiment with the following example code to understand how the `readlines()` method works:

```
stream = open("text.txt")
print(stream.readlines(20))
print(stream.readlines(20))
print(stream.readlines(20))
print(stream.readlines(20))
stream.close()

```

The maximum accepted input buffer size is passed to the method as its argument. You may expect that `readlines()` can process a file's contents more effectively than `readline()`, as it may need to be invoked fewer times.

NOTE: When there is nothing to read from the file, the method returns an empty list. Use it to detect the end of the file.

To the extent of the buffer's size, you can expect that increasing it may improve input performance, but there is no golden rule for it – try to find the optimal values yourself. Look at the code. We've modified it to show you how to use `readlines()`.

```
from os import strerror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    lines = s.readlines(20)
    while len(lines) != 0:
        for line in lines:
            lcnt += 1
            for ch in line:
                print(ch, end='')
                ccnt += 1
        lines = s.readlines(10)
    s.close()
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:     ", lcnt)
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))

```

We've decided to use a 15-byte-long buffer, but don't take that as a recommendation. We've used such a value to avoid the situation in which the first `readlines()` invocation consumes the whole file. We want the method to be forced to work harder, and to demonstrate its capabilities. There are two nested loops in the code: the outer one uses `readlines()`'s result to iterate through it, while the inner one prints the lines character by character. The last example we want to present shows a very interesting trait of the object returned by the `open()` function in text mode. We think it may surprise you – the object is an instance of the iterable class. Strange? Not at all. Usable? Yes, absolutely.

The iteration protocol defined for the file object is very simple – its `__next__` method just returns the next line read in from the file. Moreover, you can expect that the object automatically invokes `close()` when any of the file reads reaches the end of the file. Look at the code and see how simple and clear it has now become.

```
from os import strerror

try:
    ccnt = lcnt = 0
    for line in open('text.txt', 'rt'):
        lcnt += 1
        for ch in line:
            print(ch, end='')
            ccnt += 1
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file: ", lcnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))

```

Dealing with text files: `write()`

Writing text files seems to be simpler, as in fact there is one method that can be used to perform such a task. The method is named `write()` and it expects just one argument – a string that will be transferred to an open file (don't forget – *open* mode should reflect the way in which the data is transferred – writing a file opened in read mode won't succeed).

No newline character is added to the `write()`'s argument, so you have to add it yourself if you want the file to be filled with a number of lines. The following example shows a very simple code that creates a file named *newtext.txt* (note: the open mode `w` ensures that the file will be created from scratch, even if it exists and contains data) and then puts ten lines into it.

```
from os import strerror

try:
    file = open('newtext.txt', 'wt') # A new file (newtext.txt) is created.
    for i in range(10):
        s = "line #" + str(i+1) + "\n"
        for char in s:
            file.write(char)
    file.close()
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))

```

The string to be recorded consists of the word line, followed by the line number. We've decided to write the string's contents character by character (this is done by the inner `for` loop) but you're not obliged to do it this way. We just wanted to show you that `write()` is able to operate on single characters. The code creates a file filled with the following text:

```
line #1
line #2
```

```
line #3
line #4
line #5
line #6
line #7
line #8
line #9
line #10
```

Can you print the file's contents to the console? We encourage you to test the behavior of the `write()` method locally on your machine. Look at this example. We've modified the previous code to write whole lines to the text file.

```
1  from os import strerror
2
3  try:
4      file = open('newtext.txt', 'wt')
5      for i in range(10):
6          file.write("line #" + str(i+1) + "\n")
7      file.close()
8  except IOError as e:
9      print("I/O error occurred: ", strerror(e.errno))
10
```

The contents of the newly created file are the same.

NOTE: You can use the same method to write to The `stderr` stream, but don't try to open it, as it's always open implicitly.

For example, if you want to send a message string to `stderr` to distinguish it from normal program output, it may look like this:

```
1  import sys
2  sys.stderr.write("Error message")
3
```

What is a bytearray?

Before we start talking about binary files, we have to tell you about one of the specialized classes Python uses to store amorphous data. Amorphous data is data which have no specific shape or form – they are just a series of bytes. This doesn't mean that these bytes cannot have their own meaning, or cannot represent any useful object, such as bitmap graphics. The most important aspect of this is that in the place where we have contact with the data, we are not able to, or simply don't want to, know anything about it.

Amorphous data cannot be stored using any of the previously presented means – they are neither strings nor lists. There should be a special container able to handle such data. Python has more than one such container – one of them is a specialized class name bytearray – as the name suggests, it's an array containing (amorphous) bytes. If you want to have such a container, for example, in order to read in a bitmap image and process it in any way, you need to create it explicitly, using one of the available constructors.

Take a look:

```
1  data = bytearray(10)
2
```

Such an invocation creates a bytearray object able to store ten bytes.

NOTE: such a constructor fills the whole array with zeros.

Bytearrays resemble lists in many respects. For example, they are mutable, they're a subject of the `len()` function, and you can access any of their elements using conventional indexing. There is one important limitation – you mustn't set any byte array elements with a value which is not an integer (violating this rule will cause a `TypeError` exception) and you're not allowed to assign a value that doesn't come from the range 0 to 255 inclusive (unless you want to provoke a `ValueError` exception). You can treat any byte array elements as integer values – just like in the following example.

```
1  data = bytearray(10)
2
3  for i in range(len(data)):
4      data[i] = 10 - i
5
6  for b in data:
7      print(hex(b))
8
```

NOTE: We've used two methods to iterate the byte arrays, and made use of the `hex()` function to see the elements printed as hexadecimal values.

Now we're going to show you how to write a byte array to a binary file – binary, as we don't want to save its readable representation – we want to write a one-to-one copy of the physical memory content, byte by byte. So, how do we write a byte array to a binary file? Look at the following code.

```
from os import strerror

data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 + i

try:
    bf = open('file.bin', 'wb')
    bf.write(data)
    bf.close()
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))

# Your code that reads bytes from the stream should go here.

```

Let's analyze it. First, we initialize the `bytearray` with subsequent values starting from `10`; if you want the file's contents to be clearly readable, replace `10` with something like `ord('a')` – this will produce bytes containing values corresponding to the alphabetical part of the ASCII code, but don't think it will make the file a text file – it's still binary, as it was created with a `wb` flag. Then, we create the file using the `open()` function – the only difference compared to the previous variants is the open mode containing the b flag. The `write()` method takes its argument (`bytearray`) and sends it, as a whole, to the file, and then the stream is then closed in a routine way.

The `write()` method returns a number of successfully written bytes. If the values differ from the length of the method's arguments, it may announce some write errors. In this case, we haven't made use of the result – this may not be appropriate in every case. Try to run the code and analyze the contents of the newly created output file. You're going to use it in the next step.

How to read bytes from a stream

Reading from a binary file requires the use of a specialized method name `readinto()`, as the method doesn't create a new byte array object, but fills a previously created one with the values taken from the binary file.

NOTE: The method returns the number of successfully read bytes. The method tries to fill the whole space available inside its argument; if there are more data in the file than space in the argument, the read operation will stop before the end of the file; otherwise, the method's result may indicate that the byte array has only been filled fragmentarily. The result will show you that, too, and the part of the array not being used by the newly read contents remains untouched. Look at the following complete code:

```
1  from os import strerror
2
3  data = bytearray(10)
4
5  try:
6      binary_file = open('file.bin', 'rb')
7      binary_file.readinto(data)
8      binary_file.close()
9
10     for b in data:
11         print(hex(b), end=' ')
12 except IOError as e:
13     print("I/O error occurred:", strerror(e.errno))
14
```

Let's analyze it: first, we open the file (the one you created using the previous code) with the mode described as `rb`; then, we read its contents into the byte array named `data`, of size ten bytes; finally, we print the byte array contents – are they the same as you expected? Run the code and check if it's working.

An alternative way of reading the contents of a binary file is offered by the method named `read()`. Invoked without arguments, it tries to read all the contents of the file into the memory, making them a part of a newly created object of the bytes class. This class has some similarities to `bytearray`, with the exception of one significant difference – it's immutable. Fortunately, there are no obstacles to creating a byte array by taking its initial value directly from the bytes object, just like here:

```
1  from os import strerror
2
3  try:
4      binary_file = open('file.bin', 'rb')
5      data = bytearray(binary_file.read())
6      binary_file.close()
7
8      for b in data:
9          print(hex(b), end=' ')
10
11 except IOError as e:
12     print("I/O error occurred:", strerror(e.errno))
13
```

Be careful – don't use this kind of read if you're not sure whether the file's contents will fit the available memory. If the `read()` method is invoked with an argument, it specifies the maximum number of bytes to be read. The method tries to read the desired number of bytes from the file, and the length of the returned object can be used to determine the number of bytes actually read. You can use the method just like here:

```
1  try:
2      binary_file = open('file.bin', 'rb')
3      data = bytearray(binary_file.read(5))
4      binary_file.close()
5
6      for b in data:
7          print(hex(b), end=' ')
8
9  except IOError as e:
10     print("I/O error occurred:", strerror(e.errno))
11
```

NOTE: The first five bytes of the file have been read by the code – the next five are still waiting to be processed.

Copying files – a simple and functional tool

Now you're going to amalgamate all this new knowledge, add some fresh elements to it, and use it to write a real code which is able to actually copy a file's contents. Of course, the purpose is not to make a better replacement for commands like *copy* (MS Windows) or *cp* (Unix/Linux) but to see one possible way of creating a working tool, even if nobody wants to use it. Look at the code.

```
1  from os import strerror
2
3  srcname = input("Enter the source file name: ")
4  try:
5      src = open(srcname, 'rb')
6  except IOError as e:
7      print("Cannot open the source file: ", strerror(e.errno))
8      exit(e.errno)
9
10 dstname = input("Enter the destination file name: ")
11 try:
12     dst = open(dstname, 'wb')
13 except Exception as e:
14     print("Cannot create the destination file: ", strerror(e.errno))
15     src.close()
16     exit(e.errno)
17
18 buffer = bytearray(65536)
19 total = 0
20 try:
21     readin = src.readinto(buffer)
22     while readin > 0:
23         written = dst.write(buffer[:readin])
24         total += written
25         readin = src.readinto(buffer)
26 except IOError as e:
27     print("Cannot create the destination file: ", strerror(e.errno))
28     exit(e.errno)
29
30 print(total,'byte(s) succesfully written')
31 src.close()
32 dst.close()
33
```

Let's analyze it:

- lines 3 through 8: ask the user for the name of the file to copy, and try to open it to read; terminate the program execution if the open fails; note: use The `exit()` function to stop program execution and to pass the completion code to the OS; any completion code other than `0` says that the program has encountered some problems; use The `errno` value to specify the nature of the issue;
- lines 10 through 16: repeat nearly the same action, but this time for the output file;
- line 18: prepare a piece of memory for transferring data from the source file to the target one; such a transfer area is often called a buffer, hence the name of the variable; the size of the buffer is arbitrary – in this case, we've decided to use 64 kilobytes; technically, a larger buffer is faster at copying items, as a larger buffer means fewer I/O operations; actually, there is always a limit, the crossing of which renders no further improvements; test it yourself if you want.
- line 19: count the bytes copied – this is the counter and its initial value;
- line 21: try to fill the buffer for the very first time;
- line 22: as long as you get a non-zero number of bytes, repeat the same actions;
- line 23: write the buffer's contents to the output file (note: we've used a slice to limit the number of bytes being written, as `write()` always prefers to write the whole buffer)
- line 24: update the counter;
- line 25: read the next file chunk;
- lines 30 through 32: some final cleaning – the job is done.

LAB: Character frequency histogram

A text file contains some text (nothing unusual) but we need to know how often (or how rare) each letter appears in the text. Such an analysis may be useful in cryptography, so we want to be able to do that in reference to the Latin alphabet.

Your task is to write a program which:

- asks the user for the input file's name;
- reads the file (if possible) and counts all the Latin letters (lower- and upper-case letters are treated as equal)
- prints a simple histogram in alphabetical order (only non-zero counts should be presented)

Create a test file for the code, and check if your histogram contains valid results. Assuming that the test file contains just one line filled with:

```
aBc
```

Expected output

```
a -> 1
b -> 1
c -> 1
```

TIP We think that a dictionary is a perfect data collection medium for storing the counts. The letters may be keys while the counters can be values.

Check Sample Solution

LAB: Sorted character frequency histogram

The previous code needs to be improved. It's okay, but it has to be better.

Your task is to make some amendments, which generate the following results:

- the output histogram will be sorted based on the characters' frequency (the bigger counter should be presented first)
- the histogram should be sent to a file with the same name as the input one, but with the suffix '.hist' (it should be concatenated to the original name)

Assuming that the input file contains just one line filled with:

```
cBabAa
```

Expected output

```
a -> 3
b -> 2
c -> 1
```

TIP: Use a lambda to change the sort order.

Check Sample Solution

LAB: Evaluating students' results

Prof. Jekyll conducts classes with students and regularly makes notes in a text file. Each line of the file contains three elements: the student's first name, the student's last name, and the number of points the student received during certain classes.

The elements are separated with white spaces. Each student may appear more than once inside Prof. Jekyll's file.

The file may look as follows:

```
John      Smith     5
Anna      Boleyn    4.5
John      Smith     2
Anna      Boleyn    11
Andrew    Cox       1.5
```

Your task is to write a program which:

- asks the user for Prof. Jekyll's file name;
- reads the file contents and counts the sum of the received points for each student;
- prints a simple (but sorted) report, just like this one:

```
Andrew    Cox       1.5
Anna      Boleyn    15.5
John      Smith     7.0
```

NOTE Your program must be fully protected against all possible failures: the file's non-existence, the file's emptiness, or any input data failures; encountering any data error should cause immediate program termination, and the error should be presented to the user. Implement and use your own exceptions hierarchy – like the example we've presented; the second exception should be raised when a wrong line is detected, and the third when the source file exists but is empty.

TIP Use a dictionary to store the students' data.

Code

```
class StudentsDataException(Exception):
    pass


class BadLine(StudentsDataException):
    # Write your code here.


class FileEmpty(StudentsDataException):
    # Write your code here.

```

Check Sample Solution

Summary

1. To read a file's contents, the following stream methods can be used:

- `read(number)` – reads The `number` characters/bytes from the file and returns them as a string; is able to read the whole file at once;
- `readline()` – reads a single line from the text file;
- `readlines(number)` – reads The `number` lines from the text file; is able to read all lines at once;
- `readinto(bytearray)` – reads the bytes from the file and fills The `bytearray` with them;

2. To write new content into a file, the following stream methods can be used:

- `write(string)` – writes a `string` to a text file;
- `write(bytearray)` – writes all the bytes of `bytearray` to a file;

3. The `open()` method returns an iterable object which can be used to iterate through all the file's lines inside a `for` loop. For example:

```
for line in open("file", "rt"):
    print(line, end='')
```

The code copies the file's contents to the console, line by line.

NOTE: The stream closes itself automatically when it reaches the end of the file.

Quiz

Question 1: What do we expect from The `readlines()` method when the stream is associated with an empty file?

Question 2: What is the following code intended to do?

```
for line in open("file", "rt"):
    for char in line:
        if char.lower() not in "aeiouy ":
            print(char, end='')
```

Question 3: You're going to process a bitmap stored in a file named image.png, and you want to read its contents as a whole into a bytearray variable named image. Add a line to the following code to achieve this goal.

```
try:
    stream = open("image.png", "rb")
    # Insert a line here.
    stream.close()
except IOError:
    print("failed")
else:
    print("success")
```

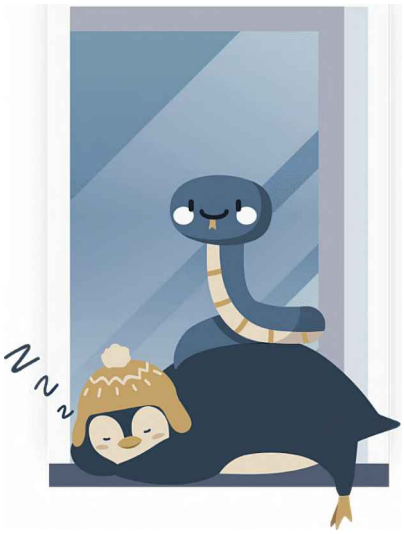
Check Answers

TWENTY-TWO – THE os MODULE – INTERACTING WITH THE OPERATING SYSTEM

In this chapter, you'll learn about a module called os, which lets you interact with the operating system using Python. It provides functions that are available on Unix and/or Windows systems. If you're familiar with the command console, you'll see that some functions give the same results as the commands available on the operating systems. A good example of this is the `mkdir` function, which allows you to create a directory just like the *mkdir* command in Unix and Windows. If you don't know this command, don't worry.

You'll soon have the opportunity to learn the functions of the *os* module, to perform operations on files and directories along with the corresponding commands. In addition to file and directory operations, the `os` module enables you to get information about the operating system, manage processes, and operate on I/O streams using file descriptors.

In a moment, you'll see how to get basic information about your operating system, although process management and working with file descriptors won't be discussed here, because these are more advanced topics that require knowledge of operating system mechanisms. Ready?



Getting information about the operating system

Before you create your first directory structure, you'll see how you can get information about the current operating system. This is really easy because the *os* module provides a function called `uname`, which returns an object containing the following attributes:

- `sysname` – stores the name of the operating system;
- `nodename` – stores the machine name on the network;
- `release` – stores the operating system release;
- `version` – stores the operating system version;
- `machine` – stores the hardware identifier, e.g. x86_64.

Let's look at how it is in practice:

```
1  import os
2  print(os.uname())
3
```

Result:

```
posix.uname_result(sysname='Linux', nodename='192d19f04766', release='4.4.0-164-
generic', version='#192-Ubuntu SMP Fri Sep 13 12:02:50 UTC 2019', machine='x86_64')
```

As you can see, the `uname` function returns an object containing information about the operating system. This code was launched on Ubuntu 16.04.6 LTS, so don't be surprised if you get a different result, because it depends on your operating system. Unfortunately, the `uname` function only works on some Unix systems. If you use Windows, you can use the `uname` function in the `platform` module, which returns a similar result. The `os` module allows you to quickly distinguish the operating system using the `name` attribute, which supports one of the following names:

- `posix` – you'll get this name if you use Unix;
- `nt` – you'll get this name if you use Windows;
- `java` – you'll get this name if your code is written in Jython.

For Ubuntu 16.04.6 LTS, the name attribute returns the name posix:

```
1  import os
2  print(os.name)
3
```

Result:

```
posix
```

NOTE In Unix systems, there's a command called `uname` that, if you run it with the -a option, returns the same information as the `uname` function.

Creating directories in Python

The `os` module provides a function called `mkdir`, which, like the *mkdir* command in Unix and Windows, allows you to create a directory. The `mkdir` function requires a path that can be relative or absolute. Let's recall what both paths look like in practice:

- *my_first_directory* – this is a relative path which will create the my_first_directory directory in the current working directory;
- *./my_first_directory* – this is a relative path that explicitly points to the current working directory. It has the same effect as the previous path;
- *../my_first_directory* – this is a relative path that will create the my_first_directory directory in the parent directory of the current working directory;
- */python/my_first_directory* – this is the absolute path that will create the my_first_directory directory, which in turn is in the python directory in the root directory.

Look at the following code. It shows an example of how to create the *my_first_directory* directory using a relative path. This is the simplest variant of the relative path, which consists of passing only the directory name. If you test the following code, it will output the newly created ['my_first_directory'] directory (and the entire contents of the current working catalog).

```
1  import os
2
3  os.mkdir("my_first_directory")
4  print(os.listdir())
5
```

The `mkdir` function creates a directory in the specified path. Note that running the program twice will raise a `FileExistsError`. This means that we cannot create a directory if it already exists. In addition to the path argument, the `mkdir` function can optionally take the *mode* argument, which specifies directory permissions. However, on some systems, the *mode* argument is ignored.

To change the directory permissions, we recommend the `chmod` function, which works similarly to the *chmod* command on Unix systems. You can find more information about it in the documentation. In the previous example, another function provided by the `os` module named `listdir` is used. The `listdir` function returns a list containing the names of the files and directories that are in the path passed as an argument. If no argument is passed to it, the current working directory will be used (as in the previous example). It's important that the result of the `listdir` function omits the entries '.' and '..', which are displayed, e.g. when using the ls -a command on Unix systems.

NOTE In both Windows and Unix, there's a command called *mkdir*, which requires a directory path. The equivalent of this code that creates the *my_first_directory* directory is the *mkdir my_first_directory* command

Recursive directory creation

The `mkdir` function is very useful, but what if you need to create another directory in the directory you've just created? Of course, you can go to the created directory and create another directory inside it, but fortunately the os module provides a function called `makedirs`, which makes this task easier. The `makedirs` function enables recursive directory creation, which means that all directories in the path will be created. Take a look at the following code and see how it is in practice.

```
1  import os
2
3  os.makedirs("my_first_directory/my_second_directory")
4  os.chdir("my_first_directory")
5  print(os.listdir())
6
```

The code should produce the following result:

```
['my_second_directory']
```

The code creates two directories. The first of them is created in the current working directory, while the second in the *my_first_directory* directory. You don't have to go to the *my_first_directory* directory to create the *my_second_directory* directory, because the *makedirs* function does this for you. In this example, we go to the *my_first_directory* directory to show that the *makedirs* command creates the *my_second_directory* subdirectory. To move between directories, you can use a function called *chdir*, which changes the current working directory to the specified path. As an argument, it takes any relative or absolute path. In our example, we pass the first directory name to it.

NOTE The equivalent of The `makedirs` function on Unix systems is the *mkdir* command with the *-p* flag, while in Windows, simply the *mkdir* command with the path:

- Unix-like systems:

```
mkdir -p my_first_directory/my_second_directory
```

- Windows:

```
mkdir my_first_directory/my_second_directory
```

Where am I now?

You already know how to create directories and how to move between them. Sometimes, when you have a really large directory structure that you navigate, you may not know which directory you're currently working in. As you've probably guessed, the `os` module provides a function that returns information about the current working directory. It's called `getcwd`. Look at the example code provided to see how to use it in practice.

```
1  import os
2
3  os.makedirs("my_first_directory/my_second_directory")
4  os.chdir("my_first_directory")
5  print(os.getcwd())
6  os.chdir("my_second_directory")
7  print(os.getcwd())
8
```

```
.../my_first_directory
.../my_first_directory/my_second_directory
```

In the example, we create the *my_first_directory* directory, and the *my_second_directory* directory inside it. In the next step, we change the current working directory to the *my_first_directory* directory, and then display the current working directory (first line of the result). Next, we go to the *my_second_directory* directory and again display the current working directory (second line of the result). As you can see, The `getcwd` function returns the absolute path to the directories.

NOTE On Unix-like systems, the equivalent of The `getcwd` function is the *pwd* command, which prints the name of the current working directory.

Deleting directories in Python

The `os` module also allows you to delete directories. It gives you the option of deleting a single directory or a directory with its subdirectories. To delete a single directory, you can use a function called `rmdir`, which takes the path as its argument. Look at the following code.

```
1  import os
2
3  os.mkdir("my_first_directory")
4  print(os.listdir())
5  os.rmdir("my_first_directory")
6  print(os.listdir())
7
```

This example is really simple. First, the `my_first_directory` directory is created, and then it's removed using the `rmdir` function. The `listdir` function is used as proof that the directory has been removed successfully. In this case, it returns an empty list. When deleting a directory, make sure it exists and is empty, otherwise an exception will be raised. To remove a directory and its subdirectories, you can use the `removedirs` function, which requires you to specify a path containing all directories that should be removed:

```
1  import os
2
3  os.makedirs("my_first_directory/my_second_directory")
4  os.removedirs("my_first_directory/my_second_directory")
5  print(os.listdir())
6
```

As with The `rmdir` function, if one of the directories doesn't exist or isn't empty, an exception will be raised.

NOTE In both Windows and Unix, there's a command called *rmdir*, which, just like the `rmdir` function, removes directories. What's more, both systems have commands to delete a directory and its contents. In Unix, this is the *rm* command with the *-r* flag.

The `system()` function

All functions presented in this part of the course can be replaced by a function called `system`, which executes a command passed to it as a string. The `system` function is available in both Windows and Unix. Depending on the system, it returns a different result. In Windows, it returns the value returned by the shell after running the command given, while in Unix, it returns the exit status of the process. Try running the following code and see how it is in practice.

```
1  import os
2
3  returned_value = os.system("mkdir my_first_directory")
4  print(returned_value)
5
```

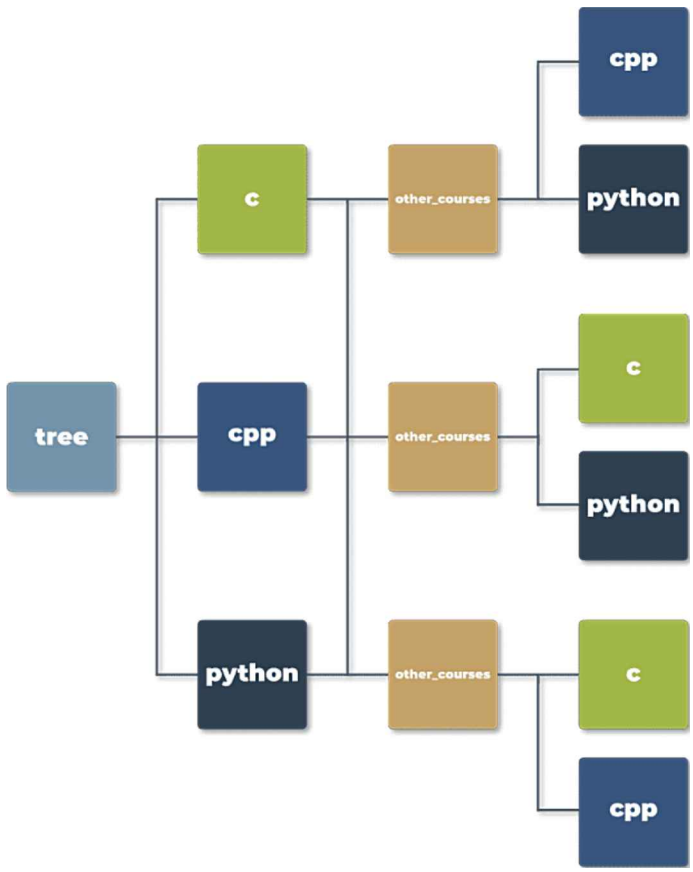
Result:

```
0
```

This example will work in both Windows and Unix. In our case, we receive exit status 0, which indicates success on Unix systems. This means that the *my_first_directory* directory has been created. As part of the exercise, try to list the contents of the directory where you created the *my_first_directory* directory.

LAB: The os module

It goes without saying that operating systems allow you to search for files and directories. While studying this part of the course, you learned about the functions of the `os` module, which have everything you need to write a program that will search for directories in a given location. To make your task easier, we have prepared a test directory structure for you:



Your program should meet the following requirements:

1. Write a function or method called `find` that takes two arguments called `path` and `dir`. The `path` argument should accept a relative or absolute path to a directory where the search should start, while the `dir` argument should be the name of a directory that you want to find in the given path. Your program should display the absolute paths if it finds a directory with the given name.
2. The directory search should be done recursively. This means that the search should also include all subdirectories in the given path.

Sample input

```
path="./tree", dir="python"
```

Sample output

```
.../tree/python
.../tree/cpp/other_courses/python
```

```
.../tree/c/other_courses/python
```

Check Sample Solution

Summary

1. The `uname` function returns an object that contains information about the current operating system. The object has the following attributes:

- `sysname` (stores the name of the operating system)
- `nodename` (stores the machine name on the network)
- `release` (stores the operating system release)
- `version` (stores the operating system version)
- `machine` (stores the hardware identifier, e.g. x86_64.)

2. The `name` attribute available in The `os` module allows you to distinguish the operating system. It returns one of the following three values:

- `posix` (you'll get this name if you use Unix)
- `nt` (you'll get this name if you use Windows)
- `java` (you'll get this name if your code is written in something like Jython)

3. The `mkdir` function creates a directory in the path passed as its argument. The path can be either relative or absolute, e.g:

```
import os


os.mkdir("hello") # the relative path

os.mkdir("/home/python/hello") # the absolute path
```

NOTE: If the directory exists, a `FileExistsError` exception will be thrown. In addition to the `mkdir` function, the `os` module provides the `makedirs` function, which allows you to recursively create all directories in a path.

4. The result of The `listdir()` function is a list containing the names of the files and directories that are in the path passed as its argument. It's important to remember that The `listdir` function omits the entries '.' and '..', which are displayed, for example, when using The `ls -a` command on Unix systems. If the path isn't passed, the result will be returned for the current working directory.

5. To move between directories, you can use a function called `chdir()`, which changes the current working directory to the specified path. As its argument, it takes any relative or absolute path. If you want to find out what the current working directory is, you can use The `getcwd()` function, which returns the path to it.

6. To remove a directory, you can use The `rmdir()` function, but to remove a directory and its subdirectories, use The `removedirs()` function.

7. On both Unix and Windows, you can use the system function, which executes a command passed to it as a string, e.g.:

```
1  import os
2
3  returned_value = os.system("mkdir hello")
4
```

The system function on Windows returns the value returned by the shell after running the command given, while on Unix, it returns the exit status of the process.

Quiz

Question 1: What is the output of the following snippet if you run it on Unix?

```
import os

print(os.name)
```

Question 2: What is the output of the following snippet?

```
import os


os.mkdir("hello")

print(os.listdir())
```

Check Answers

TWENTY-THREE – THE `DATETIME` MODULE – WORKING WITH TIME- AND DATE-RELATED FUNCTIONS

In this chapter, you'll learn about a Python module called datetime. As you can guess, it provides classes for working with date and time. If you think you don't need to delve into this topic, let's talk about examples of using date and time in programming. Date and time have countless uses and it's probably hard to find a production application that doesn't use them. Here are some examples:

- event logging – thanks to the knowledge of date and time, we are able to determine when exactly a critical error occurs in our application. When creating logs, you can specify the date and time format;
- tracking changes in the database – sometimes it's necessary to store information about when a record was created or modified. The `datetime` module will be perfect for this case;
- data validation – you'll soon learn how to read the current date and time in Python. Knowing the current date and time, you'll be able to validate various types of data, e.g. whether a discount coupon entered by a user in our application is still valid;
- storing important information – can you imagine bank transfers without storing the information of when they were made? the date and time of certain actions must be preserved, and we must deal with it.

Date and time are used in almost every area of our lives, so it's important to familiarize yourself with the Python `datetime` module. Are you ready for a new dose of knowledge?

Getting the current local date and creating date objects

One of the classes provided by the `datetime` module is a class called `date`. Objects of this class represent a date consisting of the year, month, and day. Look at the following code to see what it looks like in practice, and get the current local date using the `today` method. Run the code to see what happens.

```
1  from datetime import date
2
3  today = date.today()
4
5  print("Today:", today)
6  print("Year:", today.year)
7  print("Month:", today.month)
8  print("Day:", today.day)
9
```

The `today` method returns a `date` object representing the current local date. Note that the `date` object has three attributes: `year`, `month`, and `day`. Be careful, because these attributes are read-only. To create a date object, you must pass the `year`, `month`, and `day` parameters as follows:

```
1  from datetime import date
2
3  my_date = date(2019, 11, 4)
4  print(my_date)
5
```

Run the example to see what happens.

When creating a `date` object, keep the following restrictions in mind:

PARAMETER	RESTRICTIONS
`year`	The year parameter must be greater than or equal to 1 (MINYEAR constant) and less than or equal to 9999 (MAXYEAR constant).
`month`	The month parameter must be greater than or equal to 1 and less than or equal to 12.
`day`	The day parameter must be greater than or equal to 1 and less than or equal to the last day of the given month and year.

NOTE Later in this course you'll learn how to change the default date format.

Creating a date object from a timestamp

The `date` class gives us the ability to create a `date` object from a *timestamp*. In Unix, the timestamp expresses the number of seconds since January 1, 1970, 00:00:00 (UTC). This date is called the Unix epoch, because this is when the counting of time began on Unix systems. The timestamp is actually the difference between a particular date (including time) and January 1, 1970, 00:00:00 (UTC), expressed in seconds.

To create a date object from a timestamp, we must pass a Unix timestamp to The `fromtimestamp` method. For this purpose, we can use The `time` module, which provides time-related functions. One of them is a function called `time()`, which returns the number of seconds from January 1, 1970 to the current moment in the form of a float number. Take a look at this example.

```
1  from datetime import date
2  import time
3
4  timestamp = time.time()
5  print("Timestamp:", timestamp)
6
7  d = date.fromtimestamp(timestamp)
8  print("Date:", d)
9
```

Run the code to see the output. If you run the sample code several times, you'll be able to see how the timestamp increments itself. It's worth adding that the result of the `time` function depends on the platform, because in Unix and Windows systems, leap seconds aren't counted.

NOTE In this part of the course we'll also talk about the `time` module.

Creating a date object using the ISO format

The `datetime` module provides several methods to create a `date` object. One of them is the `fromisoformat` method, which takes a date in the YYYY-MM-DD format compliant with the ISO 8601 standard. The ISO 8601 standard defines how the date and time are represented. It's often used, so it's worth taking a moment to familiarize yourself with it. Take a look at the picture describing the values required by the format.

Now look at the code and run it. In our example, YYYY is 2019, MM is 11 (November), and DD is 04 (fourth day of November). When substituting the date, be sure to add 0 before a month or a day that is expressed by a number less than 10.

NOTE The `fromisoformat` method has been available in Python since version 3.7.

The `replace()` method

Sometimes you may need to replace the year, month, or day with a different value. You can't do this with the year, month, and day attributes because they're read-only. In this case, you can use the method named `replace`.

Run this code.

```
1  from datetime import date
2
3  d = date(1991, 2, 5)
4  print(d)
5
6  d = d.replace(year=1992, month=1, day=16)
7  print(d)
8
```

Result:

```
1991-02-05
1992-01-16
```

The `year`, `month`, and `day` parameters are optional. You can pass only one parameter to the `replace` method, e.g. `year`, or all three as in the example. The `replace` method returns a changed `date` object, so you must remember to assign it to some variable.

What day of the week is it?

One of the more helpful methods that make working with dates easier is the method called `weekday`. It returns the day of the week as an integer, where 0 is Monday and 6 is Sunday. Run the code.

```
1  from datetime import date
2
3  d = date(2019, 11, 4)
4  print(d.weekday())
5
```

Result:

```
0
```

The `date` class has a similar method called `isoweekday`, which also returns the day of the week as an integer, but 1 is Monday, and 7 is Sunday:

```
1  from datetime import date
2
3  d = date(2019, 11, 4)
4  print(d.isoweekday())
5
```

Result:

```
1
```

As you can see, for the same date we get a different integer, but expressing the same day of the week. The integer returned by the `isoweekday` method follows the ISO 85601 specification.

Creating time objects

You already know how to present a date using the `date` object. The `datetime` module also has a class that allows you to present time. Can you guess its name? Yes, it's called `time`:

```
time(hour, minute, second, microsecond, tzinfo, fold)
```

The `time` class constructor accepts the following optional parameters:

PARAMETER	RESTRICTIONS
`hour`	The hour parameter must be greater than or equal to 0 and less than 23.
`minute`	The minute parameter must be greater than or equal to 0 and less than 59.
`second`	The second parameter must be greater than or equal to 0 and less than 59.
`microsecond`	The microsecond parameter must be greater than or equal to 0 and less than 1000000.
`tzinfo`	The `tzinfo` parameter must be a `tzinfo` subclass object or `None` (default).
`fold`	The fold parameter must be 0 or 1 (default 0).

The `tzinfo` parameter is associated with time zones, while `fold` is associated with wall times. We won't use them during this course, but we encourage you to familiarize yourself with them. Let's look at how to create a time object in practice. Run the following code.

```
1  from datetime import time
2
3  t = time(14, 53, 20, 1)
4
5  print("Time:", t)
6  print("Hour:", t.hour)
7  print("Minute:", t.minute)
8  print("Second:", t.second)
9  print("Microsecond:", t.microsecond)
10
```

Result:

```
Time: 14:53:20.000001
Hour: 14
Minute: 53
Second: 20
Microsecond: 1
```

In the example, we passed four parameters to the class constructor: `hour`, `minute`, `second`, and `microsecond`. Each of them can be accessed using the class attributes.

NOTE Soon we'll tell you how you can change the default time formatting.

The `time` module

In addition to The `time` class, the Python standard library offers a module called `time`, which provides a time-related function. You already had the opportunity to learn the function called `time` when discussing The `date` class. Now we'll look at another useful function available in this module. You must spend many hours in front of a computer while doing this course. Sometimes you may feel the need to take a nap. Why not? Let's write a program that simulates a student's short nap. Have a look at this code.

```
1  import time
2
3  class Student:
4      def take_nap(self, seconds):
5          print("I'm very tired. I have to take a nap. See you later.")
6          time.sleep(seconds)
7          print("I slept well! I feel great!")
8
9  student = Student()
10 student.take_nap(5)
11
```

Result:

```
I'm very tired. I have to take a nap. See you later.
I slept well! I feel great!
```

The most important part of the sample code is the use of the `sleep` function (yes, you may remember it from one of the previous labs earlier in the course), which suspends program execution for the given number of seconds. In our example it's 5 seconds. You're right; it's a very short nap. Extend the student's sleep by changing the number of seconds. Note that the `sleep` function accepts only an integer or a floating point number.

The `ctime()` function

The `time` module provides a function called `ctime`, which converts the time in seconds since January 1, 1970 (Unix epoch) to a string. Do you remember the result of the `time` function? That's what you need to pass to `ctime`. Take a look at this example.

```
1  import time
2
3  timestamp = 1572879180
4  print(time.ctime(timestamp))
5
```

Result:

```
Mon Nov  4 14:53:00 2019
```

The `ctime` function returns a string for the passed timestamp. Here, the timestamp expresses November 4, 2019 at 14:53:00. You can also call the `ctime` function without specifying the time in seconds:

```
1  import time
2  print(time.ctime())
3
```

The `gmtime()` and `localtime()` functions

Some of the functions available in the `time` module require knowledge of the `struct_time` class, but before we get to know them, let's see what the class looks like:

```
1  time.struct_time:
2      tm_year   # Specifies the year.
3      tm_mon    # Specifies the month (value from 1 to 12)
4      tm_mday   # Specifies the day of the month (value from 1 to 31)
5      tm_hour   # Specifies the hour (value from 0 to 23)
6      tm_min    # Specifies the minute (value from 0 to 59)
7      tm_sec    # Specifies the second (value from 0 to 61 )
8      tm_wday    # Specifies the weekday (value from 0 to 6)
9      tm_yday   # Specifies the year day (value from 1 to 366)
10     tm_isdst  # Specifies whether daylight saving time applies (1 - yes, 0 - no,
   -1 - it isn't known)
11     tm_zone   # Specifies the timezone name (value in an abbreviated form)
12     tm_gmtoff # Specifies the offset east of UTC (value in seconds)
13
```

The `struct_time` class also allows access to values using indexes. Index `0` returns the value in `tm_year`, while 8 returns the value in `tm_isdst`. The exceptions are `tm_zone` and `tm_gmoff`, which cannot be accessed using indexes. Let's look at how to use the `struct_time` class in practice. Run the following code.

```
1  import time
2
3  timestamp = 1572879180
4  print(time.gmtime(timestamp))
5  print(time.localtime(timestamp))
6
```

Result:

```
time.struct_time(tm_year=2019, tm_mon=11, tm_mday=4, tm_hour=14, tm_min=53,
tm_sec=0, tm_wday=0, tm_yday=308, tm_isdst=0)
time.struct_time(tm_year=2019, tm_mon=11, tm_mday=4, tm_hour=14, tm_min=53,
tm_sec=0, tm_wday=0, tm_yday=308, tm_isdst=0)
```

The example shows two functions that convert the elapsed time from the Unix epoch to the `struct_time` object. The difference between them is that the `gmtime` function returns the `struct_time` object in UTC, while the `localtime` function returns local time. For the `gmtime` function, the `tm_isdst` attribute is always 0.

The `asctime()` and `mktime()` functions

The time module has functions that expect a `struct_time` object or a tuple that stores values according to the indexes presented when discussing the `struct_time` class. Run this example.

```
1  import time
2
3  timestamp = 1572879180
4  st = time.gmtime(timestamp)
5
6  print(time.asctime(st))
7  print(time.mktime((2019, 11, 4, 14, 53, 0, 0, 308, 0)))
8
```

Result:

```
Mon Nov 4 14:53:00 2019
1572879180.0
```

The first of the functions, called `asctime`, converts a `struct_time` object or a tuple to a string. Note that the familiar `gmtime` function is used to get The `struct_time` object. If you don't provide an argument to The `asctime` function, the time returned by The `localtime` function will be used. The second function called `mktime` converts a `struct_time` object or a tuple that

expresses the local time to the number of seconds since the Unix epoch. In our example, we passed a tuple to it, which consists of the following values:

```
2019 => tm_year
11 => tm_mon
4 => tm_mday
14 => tm_hour
53 => tm_min
0 => tm_sec
0 => tm_wday
308 => tm_yday
0 => tm_isdst
```

Creating `datetime` objects

In The `datetime` module, date and time can be represented either as separate objects or as one object. The class that combines date and time is called `datetime`.

```
datetime(year, month, day, hour, minute, second, microsecond, tzinfo, fold)
```

Its constructor accepts the following parameters:

- `year`: the year parameter must be greater than or equal to 1 (MINYEAR constant) and less than or equal to 9999 (MAXYEAR constant).
- `month`: The month parameter must be greater than or equal to 1 and less than or equal to 12.
- `day`: The day parameter must be greater than or equal to 1 and less than or equal to the last day of the given month and year.
- `hour`: The hour parameter must be greater than or equal to 0 and less than 23.
- `minute`: The minute parameter must be greater than or equal to 0 and less than 59.
- `second`: The second parameter must be greater than or equal to 0 and less than 59.
- `microsecond`: The microsecond parameter must be greater than or equal to 0 and less than 1000000.
- `tzinfo`: The tzinfo parameter must be a tzinfo subclass object or None (default).
- `fold`: The fold parameter must be 0 or 1 (default 0).

Now let's have a look at the code to see how we create a datetime object.

```
Datetime: 2019-11-04 14:53:00
Date: 2019-11-04
Time: 14:53:00
```

The example creates a `datetime` object representing November 4, 2019 at 14:53:00. All parameters passed to the constructor go to read-only class attributes. They're `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, `tzinfo`, and `fold`. The example shows two methods that return two different objects. The method called `date` returns the `date` object with the given year, month, and day, while the method called `time` returns the *time* object with the given hour and minute.

Methods that return the current date and time

The `datetime` class has several methods that return the current date and time. These methods are:

- `today()` – returns the current local date and time with the `tzinfo` attribute set to `None`;
- `now()` – returns the current local date and time the same as the `today` method, unless we pass the optional argument `tz` to it. The argument of this method must be an object of the `tzinfo` subclass;
- `utcnow()` – returns the current UTC date and time with the `tzinfo` attribute set to `None`.

Run the codes to see them all in practice. What can you say about the output?

```
1  from datetime import datetime
2
3  print("today:", datetime.today())
4  print("now:", datetime.now())
5  print("utcnow:", datetime.utcnow())
6
```

As you can see, the result of all the three methods is the same. The small differences are caused by the time elapsed between subsequent calls.

NOTE You can read more about `tzinfo` objects in the documentation.

Getting a timestamp

There are many converters available on the Internet that can calculate a timestamp based on a given date and time, but how can we do it in the `datetime` module? We can do it thanks to the `timestamp` method provided by the `datetime` class. Look at the following code.

```
1  from datetime import datetime
2
3  dt = datetime(2020, 10, 4, 14, 55)
4  print("Timestamp:", dt.timestamp())
5
```

Result:

```
Timestamp: 1601823300.0
```

The `timestamp` method returns a float value expressing the number of seconds elapsed between the date and time indicated by the `datetime` object and January 1, 1970, 00:00:00 (UTC).

Date and time formatting

All `datetime` module classes presented so far have a method called `strftime`. This is a very important method, because it allows us to return the date and time in the format we specify. The `strftime` method takes only one argument in the form of a string specifying a format that can consist of directives. A directive is a string consisting of the character % (percent) and a lowercase or uppercase letter. For example, the directive `%Y` means the year with the century as a decimal number. Let's see it in an example. Run the code.

```
1  from datetime import date
2
3  d = date(2020, 1, 4)
4  print(d.strftime('%Y/%m/%d'))
5
```

Result:

```
2020/01/04
```

In the example, we've passed a format consisting of three directives separated by / (slash) to the `strftime` method. Of course, the separator character can be replaced by another character, or even by a string. You can put any characters in the format, but only recognizable directives will be replaced with the appropriate values. In our format we've used the following directives:

- `%Y` – returns the year with the century as a decimal number. In our example, this is 2020.
- `%m` – returns the month as a zero-padded decimal number. In our example, it's 01.
- `%d` – returns the day as a zero-padded decimal number. In our example, it's 04.

NOTE You can find all available directives here.

Time formatting works in the same way as date formatting, but requires the use of appropriate directives. Let's take a closer look at a few of them.

```
1  from datetime import time
2  from datetime import datetime
3
4  t = time(14, 53)
5  print(t.strftime("%H:%M:%S"))
6
7  dt = datetime(2020, 11, 4, 14, 53)
8  print(dt.strftime("%y/%B/%d %H:%M:%S"))
9
```

Result:

```
14:53:00

20/November/04 14:53:00
```

The first of the formats used concerns only time. As you can guess, `%H` returns the hour as a zero-padded decimal number, `%M` returns the minute as a zero-padded decimal number, while `%S` returns the second as a zero-padded decimal number. In our example, `%H` is replaced by 14, `%M` by 53, and `%S` by 00. The second format used combines date, and time directives. There are two new directives, `%y` and `%B`. The directive `%y` returns the year without a century as a zero-padded decimal number (in our example it's 20). The `%B` directive returns the month as the locale's full name (in our example, it's November).

In general, you've got a lot of freedom in creating formats, but you must remember to use the directives properly. As an exercise, you can check what happens if, for example, you try to use The `%y` directive in the format passed to the time object's *strftime* method. Try to find out why you got this result yourself. Good luck!

The `strftime()` function in The `time` module

You probably won't be surprised to learn that the `strftime` function is available in the time module. It differs slightly from the `strftime` methods in the classes provided by the `datetime` module because, in addition to the format argument, it can also

take (optionally) a tuple or `struct_time` object. If you don't pass a tuple or `struct_time` object, the formatting will be done using the current local time. Take a look at the following example.

```
1  import time
2
3  timestamp = 1572879180
4  st = time.gmtime(timestamp)
5
6  print(time.strftime("%Y/%m/%d %H:%M:%S", st))
7  print(time.strftime("%Y/%m/%d %H:%M:%S"))
8
```

Our result looks as follows:

```
2019/11/04 14:53:00
2020/10/12 12:19:40
```

Creating a format looks the same as for the `strftime` methods in the `datetime` module. In our example, we use the `%Y`, `%m`, `%d`, `%H`, `%M`, and `%S` directives that you already know. In the first function call, we format the `struct_time` object, while in the second call (without the optional argument), we format the local time.

The `strptime()` method

Knowing how to create a format can be helpful when using a method called `strptime` in the `datetime` class. Unlike the `strftime` method, it creates a `datetime` object from a string representing a date and time. The `strptime` method requires you to specify the format in which you saved the date and time. Let's see it in an example.

Take a look at this code.

```
1  from datetime import datetime
2  print(datetime.strptime("2019/11/04 14:53:00", "%Y/%m/%d %H:%M:%S"))
3
```

Result:

```
2019-11-04 14:53:00
```

In the example, we've specified two required arguments. The first is a date and time as a string: `"2019/11/04 14:53:00"`, while the second is a format that facilitates parsing to a `datetime` object. Be careful, because if the format you specify doesn't match the date and time in the string, it'll raise a *ValueError*.

NOTE In the `time` module, you can find a function called `strptime`, which parses a string representing a time to a `struct_time` object. Its use is analogous to the `strptime` method in the `datetime` class:

```
1  import time
2  print(time.strptime("2019/11/04 14:53:00", "%Y/%m/%d %H:%M:%S"))
3
```

Its result will be as follows:

```
time.struct_time(tm_year=2019, tm_mon=11, tm_mday=4, tm_hour=14, tm_min=53,
tm_sec=0, tm_wday=0, tm_yday=308, tm_isdst=-1)
```

Date and time operations

Sooner or later you'll have to perform some calculations on the date and time. Luckily there's a class called `timedelta` in the `datetime` module that was created for just this purpose. To create a `timedelta` object, just perform a subtraction on the `date` or `datetime` objects:

```
1  from datetime import date
2  from datetime import datetime
3
4  d1 = date(2020, 11, 4)
5  d2 = date(2019, 11, 4)
6
7  print(d1 - d2)
8
9  dt1 = datetime(2020, 11, 4, 0, 0, 0)
10 dt2 = datetime(2019, 11, 4, 14, 53, 0)
11
12 print(dt1 - dt2)
13
```

Result:

```
366 days, 0:00:00
365 days, 9:07:00
```

The example shows subtraction for both the `date` and `datetime` objects. In the first case, we receive the difference in days, which is 366 days. Note that the difference in hours, minutes, and seconds is also displayed. In the second case, we receive a different result, because we specified the time that was included in the calculations. As a result, we receive 365 days, 9 hours, and 7 minutes.

Creating `timedelta` objects

You've already learned that a `timedelta` object can be returned as a result of subtracting two `date` or `datetime` objects. You can also create an object yourself. For this purpose, let's get acquainted with the arguments accepted by the class constructor, which are: `days`, `seconds`, `microseconds`, `milliseconds`, `minutes`, `hours`, and `weeks`. Each of them is optional and defaults to 0. The arguments should be integers or floating point numbers, and can be either positive or negative. Let's look at a simple example:

```
1  from datetime import timedelta
2
3  delta = timedelta(weeks=2, days=2, hours=3)
4  print(delta)
5
```

Result:

```
16 days, 3:00:00
```

The result of 16 days is obtained by converting the `weeks` argument to days (2 weeks = 14 days) and adding the `days` argument (2 days). This is normal behavior, because the `timedelta` object only stores days, seconds, and microseconds internally. Similarly, the `hour` argument is converted to seconds. Take a look at the following example:

```
1  from datetime import timedelta
2
3  delta = timedelta(weeks=2, days=2, hours=3)
4  print("Days:", delta.days)
5  print("Seconds:", delta.seconds)
6  print("Microseconds:", delta.microseconds)
7
```

Result:

```
Days: 16
Seconds: 10800
Microseconds: 0
```

The result of 10800 is obtained by converting 3 hours into seconds. In this way the `timedelta` object stores the arguments passed during its creation. Weeks are converted to days, hours and minutes to seconds, and milliseconds to microseconds. You already know how the `timedelta` object stores the passed arguments internally. Let's see how it can be used in practice. Look at some operations supported by the `datetime` module classes. Run the code we've provided.

```
1  from datetime import timedelta
2  from datetime import date
3  from datetime import datetime
4
5  delta = timedelta(weeks=2, days=2, hours=2)
6  print(delta)
7
8  delta2 = delta * 2
9  print(delta2)
10
11 d = date(2019, 10, 4) + delta2
12 print(d)
13
14 dt = datetime(2019, 10, 4, 14, 53) + delta2
15 print(dt)
16
```

Result:

```
16 days, 2:00:00
32 days, 4:00:00
```

```
2019-11-05

2019-11-05 18:53:00
```

The `timedelta` object can be multiplied by an integer. In our example, we multiply the object representing 16 days and 2 hours by 2. As a result, we receive a `timedelta` object representing 32 days and 4 hours. Note that both days and hours have been multiplied by 2. Another interesting operation using the `timedelta` object is adding. In the example, we've added the `timedelta` object to the `date` and `datetime` objects. As a result of these operations, we receive `date` and `datetime` objects increased by days and hours stored in the `timedelta` object. The presented multiplication operation allows you to quickly increase the value of the `timedelta` object, while multiplication can also help you get a date from the future. Of course, the `timedelta`, `date`, and `datetime` classes support many more operations. We encourage you to familiarize yourself with them in the documentation.

LAB: The `datetime` and `time` modules

You've already learned about the `strftime` method, which requires knowledge of directives to create a format. It's now time to put these directives into practice. By the way, you'll have the opportunity to practice working with documentation, because you'll have to find directives that you don't yet know. Here's your task:

Write a program that creates a `datetime` object for November 4, 2020, 14:53:00. The object created should call the `strftime` method with the appropriate format to display the following result:

```
2020/11/04 14:53:00

20/November/04 14:53:00 PM

Wed, 2020 Nov 04

Wednesday, 2020 November 04

Weekday: 3

Day of the year: 309

Week number of the year: 44
```

NOTE Each result line should be created by calling The `strftime` method with at least one directive in the format argument.

Check Sample Solution

Summary

1. To create a `date` object, you must pass the year, month, and day arguments as follows:

```
1  from datetime import date
2
3  my_date = date(2020, 9, 29)
4  print("Year:", my_date.year) # Year: 2020
5  print("Month:", my_date.month) # Month: 9
6  print("Day:", my_date.day) # Day: 29
7
```

The `date` object has three (read-only) attributes: year, month, and day.

2. The `today` method returns a date object representing the current local date:

```
1  from datetime import date
2  print("Today:", date.today()) # Displays: Today: 2020-09-29
3
```

3. In Unix, the timestamp expresses the number of seconds since January 1, 1970, 00:00:00 (UTC). This date is called the "Unix epoch", because it began the counting of time on Unix systems. The timestamp is actually the difference between a particular date (including time) and January 1, 1970, 00:00:00 (UTC), expressed in seconds. To create a date object from a timestamp, we pass a Unix timestamp to the `fromtimestamp` method:

```
1  from datetime import date
2  import time
3
4  timestamp = time.time()
5  d = date.fromtimestamp(timestamp)
6
```

NOTE: The `time` function returns the number of seconds from January 1, 1970 to the current moment in the form of a float number.

4. The constructor of the `time` class accepts six arguments (`hour`, `minute`, `second`, `microsecond`, `tzinfo`, and `fold`). Each of these arguments is optional.

```
from datetime import time

t = time(13, 22, 20)

print("Hour:", t.hour) # Hour: 13
print("Minute:", t.minute) # Minute: 22
print("Second:", t.second) # Second: 20

```

5. The `time` module contains the `sleep` function, which suspends program execution for a given number of seconds, e.g.:

```
import time

time.sleep(10)
print("Hello world!") # This text will be displayed after 10 seconds.

```

6. In the `datetime` module, date and time can be represented either as separate objects, or as one object. The class that combines date and time is called `datetime`. All arguments passed to the constructor go to read-only class attributes. They are `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, `tzinfo`, and `fold`:

```
from datetime import datetime

dt = datetime(2020, 9, 29, 13, 51)
print("Datetime:", dt) # Displays: Datetime: 2020-09-29 13:51:00

```

7. The `strftime` method takes only one argument in the form of a string specifying a format that can consist of directives. A directive is a string consisting of the character `%` (percent) and a lower-case or upper-case letter. Here are some useful directives:

- `%Y` – returns the year with the century as a decimal number;
- `%m` – returns the month as a zero-padded decimal number;
- `%d` – returns the day as a zero-padded decimal number;
- `%H` – returns the hour as a zero-padded decimal number;
- `%M` – returns the minute as a zero-padded decimal number;
- `%S` – returns the second as a zero-padded decimal number.

Example:

```
from datetime import date

d = date(2020, 9, 29)
print(d.strftime('%Y/%m/%d')) # Displays: 2020/09/29

```

8. It's possible to perform calculations on `date` and `datetime` objects, e.g.:

```
from datetime import date

d1 = date(2020, 11, 4)
d2 = date(2019, 11, 4)

d = d1 - d2
print(d) # Displays: 366 days, 0:00:00.
print(d * 2) # Displays: 732 days, 0:00:00.

```

The result of the subtraction is returned as a `timedelta` object that expresses the difference in days between the two dates in the example. Note that the difference in hours, minutes, and seconds is also displayed. The `timedelta` object can be used for further calculations (e.g. you can multiply it by 2).

Quiz

Question 1: What is the output of the following snippet?

```
from datetime import time


t = time(14, 53)
print(t.strftime("%H:%M:%S"))
```

Question 2: What is the output of the following snippet?

```
from datetime import datetime

dt1 = datetime(2020, 9, 29, 14, 41, 0)
dt2 = datetime(2020, 9, 28, 14, 41, 0)

print(dt1 - dt2)
```

Check Answers

TWENTY-FOUR – THE CALENDAR MODULE – CALENDAR-RELATED FUNCTIONS

In addition to the `datetime` and `time` modules, the Python standard library provides a module called `calendar` which, as the name suggests, offers calendar-related functions.

One of them is of course displaying the calendar. It's important that the days of the week are displayed from Monday to Sunday, and each day of the week has its representation in the form of an integer:

This table shows the representation of the days of the week in the `calendar` module. The first day of the week (Monday) is represented by the value *0* and the `calendar.MONDAY` constant, while the last day of the week (Sunday) is represented by the value *6* and the `calendar.SUNDAY` constant.

DAY OF THE WEEK	INTEGER VALUE	CONSTANT
Monday	0	`calendar.MONDAY`
Tuesday	1	`calendar.TUESDAY`
Wednesday	2	`calendar.WEDNESDAY`
Thursday	3	`calendar.THURSDAY`
Friday	4	`calendar.FRIDAY`
Saturday	5	`calendar.SATURDAY`
Sunday	6	`calendar.SUNDAY`

Your first calendar

You will start your adventure with the `calendar` module with a simple function called `calendar`, which allows you to display the calendar for the whole year. Let's look at how to use it to display the calendar for 2020. Run the code and see what happens.

```
1  import calendar
2  print(calendar.calendar(2020))
3
```

The result displayed is similar to the result of the *cal* command available in Unix. If you want to change the default calendar formatting, you can use the following parameters:

- `w` – date column width (default 2)
- `l` – number of lines per week (default 1)
- `c` – number of spaces between month columns (default 6)
- `m` – number of columns (default 3)

The calendar function requires you to specify the year, while the other parameters responsible for formatting are optional. We encourage you to try these parameters yourself.

A good alternative to this function is the function called *prcal*, which also takes the same parameters as the `calendar` function, but doesn't require the use of the `print` function to display the calendar. Its use looks like this:

```
1  import calendar
2  calendar.prcal(2020)
3
```

Calendar for a specific month

The `calendar` module has a function called `month`, which allows you to display a calendar for a specific month. Its use is really simple, you just need to specify the year and month – check out this code.

```
1  import calendar
2  print(calendar.month(2020, 11))
3
```

The example displays the calendar for November 2020. As in The `calendar` function, you can change the default formatting using the following parameters:

- `w` – date column width (default 2)
- `l` – number of lines per week (default 1)

NOTE You can also use the `prmonth` function, which has the same parameters as the `month` function, but doesn't require you to use the `print` function to display the calendar.

The `setfirstweekday()` function

As you already know, by default in the `calendar` module, the first day of the week is Monday. However, you can change this behavior using a function called `setfirstweekday`.

Do you remember the table showing the days of the week and their representation in the form of integer values? It's time to use it, because the `setfirstweekday` method requires a parameter expressing the day of the week in the form of an integer value. Take a look at this example.

```
1  import calendar
2
3  calendar.setfirstweekday(calendar.SUNDAY)
4  calendar.prmonth(2020, 12)
5
```

The example uses the `calendar.SUNDAY` constant, which contains a value of *6*. Of course, you could pass this value directly to the `setfirstweekday` function, but the version with a constant is more elegant. As a result, we get a calendar showing the month of December 2020, in which the first day of all the weeks is Sunday.

The `weekday()` function

Another useful function provided by the `calendar` module is the function called `weekday`, which returns the day of the week as an integer value for the given year, month, and day. Let's see it in practice. Run the code to check the day of the week that falls on December 24, 2020.

```
1  import calendar
2  print(calendar.weekday(2020, 12, 24))
3
```

Result:

```
3
```

The `weekday` function returns 3, which means that December 24, 2020 is a Thursday.

The `weekheader()` function

You've probably noticed that the calendar contains weekly headers in a shortened form. If needed, you can get short weekday names using the `weekheader` method. The `weekheader` method requires you to specify the width in characters for one day of the week. If the width you provide is greater than 3, you'll still get the abbreviated weekday names consisting of three characters. So let's look at how to get a smaller header. Run the code.

```
1  import calendar
2  print(calendar.weekheader(2))
3
```

Result:

```
Mo Tu We Th Fr Sa Su
```

NOTE If you change the first day of the week, for example, by using the `setfirstweekday` function, it'll affect the result of the `weekheader` function.

How do we check if a year is a leap year?

The `calendar` module provides two useful functions to check whether years are leap years. The first one, called `isleap`, returns *True* if the passed year is leap, or *False* otherwise. The second one, called `leapdays`, returns the number of leap years in the given range of years. Run the code.

```
1  import calendar
2
3  print(calendar.isleap(2020))
4  print(calendar.leapdays(2010, 2021))  # Up to but not including 2021.
5
```

Result:

```
True
3
```

In the example, we obtain the result 3, because in the period from 2010 to 2020 there are only three leap years (note: 2021 is not included). They are the years 2012, 2016, and 2020.

Classes for creating calendars

The functions we've shown you so far aren't everything the `calendar` module offers. In addition to them, we can use the following classes:

- `calendar.Calendar` – provides methods to prepare calendar data for formatting;
- `calendar.TextCalendar` – is used to create regular text calendars;
- `calendar.HTMLCalendar` – is used to create HTML calendars;
- `calendar.LocaleTextCalendar` – is a subclass of The `calendar.TextCalendar` class. The constructor of this class takes the *locale* parameter, which is used to return the appropriate months and weekday names.
- `calendar.LocaleHTMLCalendar` – is a subclass of The `calendar.HTMLCalendar` class. The constructor of this class takes the locale parameter, which is used to return the appropriate months and weekday names.

During this course, you've already had the opportunity to create text calendars when discussing the functions of the `calendar` module. Time to try something new. Let's take a closer look at the methods of the `calendar` class.

Creating a `Calendar` object

The `Calendar` class constructor takes one optional parameter named `firstweekday`, by default equal to 0 (Monday). The `firstweekday` parameter must be an integer between 0-6. For this purpose, we can use the already-known constants – look at the following code.

```
1  import calendar
2
3  c = calendar.Calendar(calendar.SUNDAY)
4
5  for weekday in c.iterweekdays():
6      print(weekday, end=" ")
7
```

The program will output the following result:

```
6 0 1 2 3 4 5
```

The code example uses the `Calendar` class method named `iterweekdays`, which returns an iterator for week day numbers. The first value returned is always equal to the value of the `firstweekday` property. Because in our example the first value returned is 6, it means that the week starts on a Sunday.

The `itermonthdates()` method

The `Calendar` class has several methods that return an iterator. One of them is the `itermonthdates` method, which requires specifying the year and month. As a result, all days in the specified month and year are returned, as well as all days before the beginning of the month or the end of the month that are necessary to get a complete week. Each day is represented by a `datetime.date` object. Take a look at the example.

```
1  import calendar
2
3  c = calendar.Calendar()
4
5  for date in c.itermonthdates(2019, 11):
6      print(date, end=" ")
7
```

The code displays all days in November 2019. Because the first day of November 2019 was a Friday, the following days are also returned to get the complete week: 10/28/2019 (Monday) 10/29/2019 (Tuesday) 10/30/2019 (Wednesday) 10/31/2019 (Thursday). The last day of November 2019 was a Saturday, so in order to keep the complete week, one more day is returned 12/01/2019 (Sunday).

Other methods that return iterators

Another useful method in the `Calendar` class is the method called `itermonthdays`, which takes year and month as parameters, and then returns the iterator to the days of the week represented by numbers.

Take a look at the following example.

```
1  import calendar
2
3  c = calendar.Calendar()
4
5  for iter in c.itermonthdays(2019, 11):
6      print(iter, end=" ")
7
```

You'll have certainly noticed the large number of `0`s returned as a result of the example code. These are days outside the specified month range that are added to keep the complete week. The first four zeros represent 10/28/2019 (Monday)

10/29/2019 (Tuesday) 10/30/2019 (Wednesday) 10/31/2019 (Thursday). The remaining numbers are days in the month, except the last value of 0, which replaces the date 12/01/2019 (Sunday). There are four other similar methods in the `Calendar` class that differ in data returned:

- `itermonthdays2` – returns days in the form of tuples consisting of a day of the month number and a week day number;
- `itermonthdays3` – returns days in the form of tuples consisting of a year, a month, and a day of the month numbers. This method has been available since Python version 3.7;
- `itermonthdays4` – returns days in the form of tuples consisting of a year, a month, a day of the month, and a day of the week numbers. This method has been available since Python version 3.7.

For testing purposes, use the previous example and see how the return values of the described methods look in practice.

The `monthdays2calendar()` method

The `Calendar` class has several other useful methods that you can learn more about in the documentation (https://docs.python.org/3/library/calendar.html). One of them is the `monthdays2calendar` method, which takes the year and month, and then returns a list of weeks in a specific month. Each week is a tuple consisting of day numbers and weekday numbers. Look at this code.

```
1  import calendar
2
3  c = calendar.Calendar()
4
5  for data in c.monthdays2calendar(2020, 12):
6      print(data)
7
```

Note that the day numbers outside the month are represented by 0, while the weekday numbers are a number from 0-6, where 0 is Monday and 6 is Sunday. In a moment, this method may be useful for you to complete a laboratory task. Are you ready?

LAB: The `calendar` module

During this course, we took a brief look at the `Calendar` class. Your task now is to extend its functionality with a new method called `count_weekday_in_year`, which takes a year and a weekday as parameters, and then returns the number of occurrences of a specific weekday in the year. Use the following tips:

- Create a class called `MyCalendar` that extends the `Calendar` class;
- Create The `count_weekday_in_year` method with the year and weekday parameters. The weekday parameter should be a value between 0-6, where 0 is Monday and 6 is Sunday. The method should return the number of days as an integer;
- In your implementation, use the `monthdays2calendar` method of the `Calendar` class.

The following are the expected results:

Sample arguments

```
year=2019, weekday=0
```

Expected output

```
52
```

Sample arguments

```
year=2000, weekday=6
```

Expected output

```
53
```

Check Sample Solution

Section Summary

1. In the `calendar` module, the days of the week are displayed from Monday to Sunday. Each day of the week has its representation in the form of an integer, where the first day of the week (Monday) is represented by the value 0, while the last day of the week (Sunday) is represented by the value 6.

2. To display a calendar for any year, call the `calendar` function with the year passed as its argument, e.g.:

```
import calendar

print(calendar.calendar(2020))
```

NOTE A good alternative to this function is the function called `prcal`, which also takes the same parameters as the `calendar` function, but doesn't require the use of the `print` function to display the calendar.

3. To display a calendar for any month of the year, call the `month` function, passing year and month to it. For example:

```
1  import calendar
2  print(calendar.month(2020, 9))
3
```

NOTE You can also use the `prmonth` function, which has the same parameters as the `month` function, but doesn't require the use of the `print` function to display the calendar

4. The `setfirstweekday` function allows you to change the first day of the week. It takes a value from 0 to 6, where 0 is Sunday and 6 is Saturday.

5. The result of the `weekday` function is a day of the week as an integer value for a given year, month, and day:

```
1  import calendar
2  print(calendar.weekday(2020, 9, 29)) # This displays 1, which means Tuesday.
3
```

6. The `weekheader` function returns the weekday names in short form. For the `weweekheader` method, you must specify the width in characters for one day of the week. If the width is greater than 3, you'll still get the abbreviated weekday names consisting of only three characters. For example:

```
1  import calendar
2  print(calendar.weekheader(2)) # This display: Mo Tu We Th Fr Sa Su
3
```

7. A very useful function available in the `calendar` module is the function called `isleap`, which, as the name suggests, allows you to check whether the year is a leap year or not:

```
1  print("Hello world!")
2
```

8. You can create a `calendar` object yourself using the `Calendar` class, which, when creating its object, allows you to change the first day of the week with the optional `firstweekday` parameter, e.g.:

```
1  import calendar
2
3  c = calendar.Calendar(2)
4
5  for weekday in c.iterweekdays():
6      print(weekday, end=" ")
7  # Result: 2 3 4 5 6 0 1
8
```

The `iterweekdays` returns an iterator for weekday numbers. The first value returned is always equal to the value of the `firstweekday` property

Section Quiz

Question 1: What is the output of the following snippet?

```
import calendar
print(calendar.weekheader(1))
```

Question 2: What is the output of the following snippet?

```
import calendar


c = calendar.Calendar()


for weekday in c.iterweekdays():
    print(weekday, end=" ")
```

Check Answers

APPENDICES

APPENDIX A – Section Quiz ANSWERS

PART 1

Chapter 1 – Introduction to modules in Python

Question 1

```
mint.make_money()
```

Question 2

```
make_money()
```

Question 3

```
# sample solution
from mint import make_money as make_more_money
```

Question 4

```
make_money()
```

Back

Chapter Two – Selected Python modules (math, random, platform)

Question 1

```
True
```

Question 2

... the pseudo-random values emitted from the random module will be exactly the same.

Question 3

The `processor()` function.

Question 4

```
3
```

Back

Chapter Three – Modules and Packages

Question 1

```
import sys

if __name__ == "__main__":
    print("Don't do that!")
    sys.exit()
```

Question 2

```
import sys

# note the double backslashes!
sys.path.append("D:\\Python\\Project\\Modules")
```

Question 3

```
import abc.def.mymodule
```

Back

Chapter Four – Python Package Installer (PIP)

Question 1

It's a reference to an old Monty Python's sketch of the same name.

Question 2

When Python 2 and Python 3 coexist in your OS, it's likely that `pip` identifies the instance of `pip` working with Python 2 packages only.

Question 3

`pip --version` will tell you that.

Question 4

You have to ask your *sysadmin* – don't try to hack your OS!

Back

PART 2

Chapter Five – Characters and Strings vs. Computers

Question 1
BOM (Byte Order Mark) is a special combination of bits announcing encoding used by a file's content (e.g. UCS-4 or UTF-B).

Question 2
Yes, it's completely internationalized – we can use UNICODE characters inside our code, read them from input and send to output.

Back

Chapter Six – The nature of strings in Python

Question 1
1

Question 2

```
['t', 'e', 'r']
```

Question 3

```
bcd
```

Back

Chapter Seven – String Methods

Question 1

```
ABC123xyz
```

Question 2

```
of
```

Question 3

```
Where*are*the*snows?
```

Question 4

```
It is either hard or possible
```

Back

Chapter Eight – String in action

Question 1

1, 3 and 4

Question 2

```
 are
```

Question 3

The code raises a `ValueError` exception.

Back

Chapter Ten – Errors, the programmer's daily bread

Question 1

```
Let's try to do this
We failed
We're done
```

Question 2

```
zero
```

Back

Chapter Eleven – The anatomy of exceptions

Question 1

```
zero
```

Question 2

```
arith
```

Question 3

```
some
```

Back

Chapter Twelve – Useful exceptions

Question 1

```
KeyboardInterrupt
```

Question 2

```
BaseException
```

Question 3

```
OverflowError
```

Back

PART 3

Chapter Thirteen – The Foundations of OOP

Question 1

Snake, reptile, vertebrate, animal – all these answers are acceptable.

Question 2

Indian python, African rock python, ball python, Burmese python – the list is long.

Question 3

No, you can't – `class` is a keyword!

Back

Chapter Fourteen – A short journey from procedural to object approach

Question 1

```
class Python(Snakes):
```

Question 2

The `__init__()` constructor lacks the obligatory parameter (we should name it `self` to stay compliant with the standards).

Question 3

The code should look as follows:

```
class Snakes
def __init__(self):
```

```
        self.__venomous = True
```

Back

Chapter Fifteen – OOP: Properties

Question 1

`population` and `victims` are **class** variables, while `length_ft` and `__venomous` are **instance** variables (the latter is also **private**).

Question 2

```
version_2._Python__venomous = not version_2._Python__venomous
```

Question 3

```
hasattr(version_2, 'constrictor')
```

Back

Chapter Sixteen – OOP: Methods

Question 1

```
class Snake:
    def __init__(self):
        self.victims = 0

    def increment(self):
        self.victims += 1
```

Question 2

```
class Snake:
    def __init__(self, victims):
        self.victims = victims
```

Question 3

```
Python is a Snake
Snake can be Python
```

Back

Chapter Seventeen – OOP Fundamentals: Inheritance

Question 1

```
Collie says: Woof! Don't run away, Little Lamb!
Dobermann says: Woof! Stay where you are, Mister Intruder!
```

Question 2

```
True False
False True
```

Question 3

```
True False
2
```

Question 4

```
class LowlandDog(SheepDog):
    def __str__(self):
        return Dog.__str__(self) + " I don't like mountains!"
```

Back

Chapter Eighteen – Exceptions once again

Question 1

```
3.0
fine
```

Question 2

```
inf
the end
```

Question 3

```
Enemy warning! Red alert! High readiness!
```

Back

PART 4

Chapter Nineteen – Generators, iterators, and closures

Question 1

```
a e i o u y
```

Question 2

```
list(map(lambda n: n | 1, any_list))
```

Question 3

```
And*Now*for*Something*Completely*Different
```

Back

Chapter Twenty – Files (file streams, file processing, diagnosing stream problems)

Question 1

`"wt"` or `"w"`

Question 2

Permission denied: you're not allowed to access the file's contents.

Question 3

```
absent
```

Back

Chapter-One – Working with real files

Question 1

An empty list (a zero-length list).

Question 2

It copies the file's contents to the console, ignoring all vowels.

Question 3

```
image = bytearray(stream.read())
```

Back

Chapter Twenty-Two – The os module – interacting with the operating system

Question 1

```
posix
```

Question 2

```
['hello']
```

Back

Chapter Twenty-Three – The `datetime` module – working with time- and date-related functions

Question 1

```
14:53:00
```

Question 2

```
1 day, 0:00:00
```

Back

Chapter Twenty-Four – The `calendar` module – working with calendar-related functions

Question 1

```
M T W T F S S
```

Question 2

```
0 1 2 3 4 5 6
```

Back

APPENDIX B: LAB HINTS

Your own split

```
1  def mysplit(strng):
2      # return [] if string is empty or contains whitespaces only
3
4      # prepare a list to return
5
6      # prepare a word to build subsequent words
7
8      # check if we are currently inside a word (i.e., if the string starts with a
   word)
9
10     # iterate through all the characters in the string
11
12         # if we are currently inside a string...
13
14             # ... and the current character is not a space...
15
16                 # ... update the current word
17
18                 # ... otherwise, we've reached the end of the word so we need to
   append it to the list...
19
20                 # ... and signal the fact that we are outside the word now
21
22             # if we are outside the word and we've reached a non-white
   character...
23
24                 # ... it means that a new word has begun so we need to remember
   it and...
25
26                 # ... store the first letter of the new word
27
28     # if we've left the string and there is a non-empty string in the word, we
   need to update the list
29
30     # return the list to the invoker
31
32
33 print(mysplit("To be or not to be, that is the question"))
34 print(mysplit("To be or not to be,that is the question"))
35 print(mysplit("   "))
36 print(mysplit(" abc "))
37 print(mysplit(""))
38
```

Back

An LED Display

```
digits = [ '1111110',   # 0
       '0110000',   # 1
       '1101101',   # 2
       '1111001',   # 3
       '0110011',   # 4
       '1011011',   # 5
       '1011111',   # 6
       '1110000',   # 7
       '1111111',   # 8
       '1111011',   # 9
       ]


def print_number(num):
    # Write the function here.


print_number(int(input("Enter the number you wish to display: ")))

```

Back

Anagrams

```
str_1 = input("Enter the first string: ")
str_2 = input("Enter the second string: ")

# This is what we're going to do with both strings:
# - remove spaces
# - change all letters to upper case
# - convert into list
# - sort the list
# - join list's elements into string
# and finally, compare both strings.
# Let's do it!

```

Back

The Digit of Life

```
date = input("Enter your birthday date (in the following format: YYYYMMDD or
YYYYDDMM, 8 digits): ")

# Write the if-else branch here.

    # While there is more than one digit in the date...

        # ... calculate the sum of all the digits...

        # ... and store it inside the string.

    print("Your Digit of Life is: " + date)

```

Back

Find a word!

```
word = input("Enter the word you wish to find: ").upper()
strn = input("Enter the string you wish to search through: ").upper()

found = True
start = 0

# Write the rest of the code here.

```

Back

Sudoku

```
# A function that checks whether a list passed as an argument contains
# nine digits from '1' to '9'.
def checkset(digs):
    return sorted(list(digs)) == [chr(x + ord('0')) for x in range(1, 10)]


# A list of rows representing the sudoku.
rows = [ ]
for r in range(9):

ok = True

# Check if all rows are good.
for r in range(9):


# Check if all columns are good.
if ok:
    for c in range(9):


# Check if all sub-squares (3x3) are good.
if ok:
    for r in range(0, 9, 3):


# Print the final verdict.
if ok:
    print("Yes")
else:
   print("No")

```

Back

APPENDIX C: LAB SAMPLE SOLUTIONS

Your own split

```
1  def mysplit(strng):
2      # return [] if string is empty or contains whitespaces only
3      if strng == '' or strng.isspace():
4          return [ ]
5      # prepare a list to return
6      lst = []
7      # prepare a word to build subsequent words
8      word = ''
9      # check if we are currently inside a word (i.e., if the string starts with a
   word)
10     inword = not strng[0].isspace()
11     # iterate through all the characters in the string
12     for x in strng:
13         # if we are currently inside a string...
14         if inword:
15             # ... and the current character is not a space...
16 if not x.isspace():
17                 # ... update the current word
18                 word = word + x
19             else:
20                 # ... otherwise, we've reached the end of the word so we need to
   append it to the list...
21                 lst.append(word)
22                 # ... and signal the fact that we are outside the word now
23                 inword = False
24         else:
25             # if we are outside the word and we've reached a non-white
   character...
26             if not x.isspace():
27                 # ... it means that a new word has begun so we need to remember
   it and...
28                 inword = True
29                 # ... store the first letter of the new word
30                 word = x
31             else:
32                 pass
33     # if we've left the string and there is a non-empty string in the word, we
   need to update the list
34     if inword:
35         lst.append(word)
36     # return the list to the invoker
37     return lst
38
39
40 print(mysplit("To be or not to be, that is the question"))
41 print(mysplit("To be or not to be,that is the question"))
42 print(mysplit(" "))
43 print(mysplit(" abc "))
44 print(mysplit(""))
45
```

Back

An LED Display

```
digits = [ '1111110',   # 0
       '0110000',   # 1
       '1101101',   # 2
       '1111001',   # 3
       '0110011',   # 4
       '1011011',   # 5
       '1011111',   # 6
       '1110000',   # 7
       '1111111',   # 8
       '1111011',   # 9
       ]


def print_number(num):
    global digits
digs = str(num)
lines = [ '' for lin in range(5) ]
for d in digs:
      segs = [ [' ',' ',' '] for lin in range(5) ]
      ptrn = digits[ord(d) - ord('0')]
      if ptrn[0] == '1':
            segs[0][0] = segs[0][1] = segs[0][2] = '#'
      if ptrn[1] == '1':
            segs[0][2] = segs[1][2] = segs[2][2] = '#'
      if ptrn[2] == '1':
            egs[2][2] = segs[3][2] = segs[4][2] = '#'
      if ptrn[3] == '1':
            segs[4][0] = segs[4][1] = segs[4][2] = '#'
      if ptrn[4] == '1':
            segs[2][0] = segs[3][0] = segs[4][0] = '#'
      if ptrn[5] == '1':
            segs[0][0] = segs[1][0] = segs[2][0] = '#'
      if ptrn[6] == '1':
            segs[2][0] = segs[2][1] = segs[2][2] = '#'
      for lin in range(5):
            lines[lin] += ''.join(segs[lin]) +
for lin in lines:
      print(lin)


print_number(int(input("Enter the number you wish to display: ")))

```

Back

Improving the Caesar cipher

```
# Input the text you want to encrypt.
text = input("Enter message: ")

# Enter a valid shift value (repeat until it succeeds).
shift = 0

while shift == 0:
    try:
        shift = int(input("Enter the cipher shift value (1..25): "))
        if shift not in range(1,26):
            raise ValueError
    except ValueError:
        shift = 0
    if shift == 0:
        print("Incorrect shift value!")

cipher = ''

for char in text:
    # Is it a letter?
    if char.isalpha():
        # Shift its code.
        code = ord(char) + shift
        # Find the code of the first letter (upper- or lower-case)
        if char.isupper():
            first = ord('A')
        else:
            first = ord('a')
        # Make correction.
        code -= first
        code %= 26
        # Append the encoded character to the message.
        cipher += chr(first + code)
    else:
        # Append the original character to the message.
        cipher += char

print(cipher)

```

Back

Palindromes

```
text = input("Enter text: ")

# Remove all spaces...
text = text.replace(' ','')

# ... and check if the word is equal to reversed itself
if len(text) > 1 and text.upper() == text[::-1].upper():
    print("It's a palindrome")
else:
    print("It's not a palindrome")

```

Back

Anagrams

```
str_1 = input("Enter the first string: ")
str_2 = input("Enter the second string: ")

strx_1 = ''.join(sorted(list(str_1.upper().replace(' ',''))))
strx_2 = ''.join(sorted(list(str_2.upper().replace(' ',''))))
if len(strx_1) > 0 and strx_1 == strx_2:
    print("Anagrams")
else:
    print("Not anagrams")

```

Back

The Digit of Life

```
date = input("Enter your birthday date (in the following format: YYYYMMDD or
YYYYDDMM, 8 digits): ")
if len(date) != 8 or not date.isdigit():
    print("Invalid date format.")
else:
    while len(date) > 1:
        the_sum = 0
        for dig in date:
            the_sum += int(dig)
        print(date)
        date = str(the_sum)
    print("Your Digit of Life is: " + date)

```

Back

Find a word!

```
word = input("Enter the word you wish to find: ").upper()
strn = input("Enter the string you wish to search through: ").upper()

found = True
start = 0

for ch in word:
    pos = strn.find(ch, start)
    if pos < 0:
        found = False
        break
    start = pos + 1
if found:
    print("Yes")
else:
print("No")

```

Back

Sudoku

```
# A function that checks whether a list passed as an argument contains
# nine digits from '1' to '9'.
def checkset(digs):
    return sorted(list(digs)) == [chr(x + ord('0')) for x in range(1, 10)]


# A list of rows representing the sudoku.
rows = [ ]
for r in range(9):
    ok = False
    while not ok:
        row = input("Enter row #" + str(r + 1) + ": ")
        ok = len(row) == 9 or row.isdigit()
        if not ok:
            print("Incorrect row data - 9 digits required")
    rows.append(row)

ok = True

# Check if all rows are good.
for r in range(9):
    if not checkset(rows[r]):
        ok = False
        break

# Check if all columns are good.
if ok:
    for c in range(9):
        col = []
        for r in range(9):
            col.append(rows[r][c])
        if not checkset(col):
            ok = False
            break

# Check if all sub-squares (3x3) are good.
if ok:
    for r in range(0, 9, 3):
        for c in range(0, 9, 3):
            sqr = ''
            # Make a string containing all digits from a sub-square.
            for i in range(3):
                sqr += rows[r+i][c:c+3]
            if not checkset(list(sqr)):
                ok = False
                break

# Print the final verdict.
if ok:
    print("Yes")
else:
    print("No")

```

Back

Reading `int`s safely

```
1  def read_int(prompt, min, max):
2      ok = False
3      while not ok:
4          try:
5              value = int(input(prompt))
6              ok = True
7          except ValueError:
8              print("Error: wrong input")
9          if ok:
10             ok = value >= min and value <= max
11         if not ok:
12             print("Error: the value is not within permitted range (" + str(min)
    + ".." + str(max) + ")")
13     return value;
14
15
16 v = read_int("Enter a number from -10 to 10: ", -10, 10)
17
```

Back

Counting stack

```
1  class Stack:
2      def __init__(self):
3          self.__stk = []
4
5      def push(self, val):
6          self.__stk.append(val)
7
8      def pop(self):
9          val = self.__stk[-1]
10         del self.__stk[-1]
11         return val
12
13
14 class CountingStack(Stack):
15     def __init__(self):
16         Stack.__init__(self)
17         self.__counter = 0
18
19     def get_counter(self):
20         return self.__counter
21
22     def pop(self):
23         self.__counter += 1
24         return Stack.pop(self)
25
26
27 stk = CountingStack()
28 for i in range(100):
29     stk.push(i)
30     stk.pop()
31 print(stk.get_counter())
32
```

Back

Queue aka FIFO

```
class QueueError(IndexError):
    pass
class Queue:


def __init__(self):
        self.queue = []

    def put(self, elem):
        self.queue.insert(0, elem)

    def get(self):
        if len(self.queue) > 0:
            elem = self.queue[-1]
            del self.queue[-1]
            return elem
        else:
            raise QueueError


que = Queue()
que.put(1)
que.put("dog")
que.put(False)
try:
    for i in range(4):
        print(que.get())
except:
    print("Queue error")

```

Back

Queue aka FIFO: part 2

```
class QueueError(IndexError):
    pass


class Queue:
    def __init__(self):
        self.queue = []
    def put(self,elem):
        self.queue.insert(0,elem)
    def get(self):
        if len(self.queue) > 0:
            elem = self.queue[-1]
            del self.queue[-1]
            return elem
        else
            raise QueueError


class SuperQueue(Queue):
    def isempty(self):
        return len(self.queue) == 0


que = SuperQueue()
que.put(1)
que.put("dog")
que.put(False)
for i in range(4):
    if not que.isempty():
        print(que.get())
    else:
        print("Queue empty")

```

Back

The Timer class

```
1  def two_digits(val):
2      s = str(val)
3      if len(s) == 1:
4          s = '0' + s
5      return s
6
7
8  class Timer:
9      def __init__(self, hours=0, minutes=0, seconds=0):
10         self.__hours = hours
11         self.__minutes = minutes
12         self.__seconds = seconds
13
14     def __str__(self):
15         return two_digits(self.__hours) + ":" + \
16                 two_digits(self.__minutes) + ":" + \
17                 two_digits(self.__seconds)
18
19     def next_second(self):
20         self.__seconds += 1
21         if self.__seconds > 59:
22             self.__seconds = 0
23             self.__minutes += 1
24             if self.__minutes > 59:
25                 self.__minutes = 0
26                 self.__hours += 1
27                 if self.__hours > 23:
28                     self.__hours = 0
29
30     def prev_second(self):
31         self.__seconds -= 1
32         if self.__seconds < 0:
33             self.__seconds = 59
34             self.__minutes -= 1
35             if self.__minutes < 0:
36                 self.__minutes = 59
37                 self.__hours -= 1
38                 if self.__hours < 0:
39                 self.__hours = 23
40
41
42 timer = Timer(23, 59, 59)
43 print(timer)
44 timer.next_second()
45 print(timer)
46 timer.prev_second()
47 print(timer)
48
```

Back

Days of the week

```
class WeekDayError(Exception):
    pass


class Weeker:
    __names = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

    def __init__(self, day):
        try:
            self.__current = Weeker.__names.index(day)
        except ValueError:
            raise WeekDayError

    def __str__(self):
        return Weeker.__names[self.__current]

    def add_days(self, n):
        self.__current = (self.__current + n) % 7

    def subtract_days(self, n):
        self.__current = (self.__current - n) % 7


try:
    weekday = Weeker('Mon')
    print(weekday)
    weekday.add_days(15)
    print(weekday)
    weekday.subtract_days(23)
    print(weekday)
    weekday = Weeker('Monday')
except WeekDayError:
    print("Sorry, I can't serve your request.")

```

Back

Points on a plane

```
import math


class Point:
    def __init__(self, x=0.0, y=0.0):
        self.__x = x
        self.__y = y

    def getx(self):
        return self.__x

    def gety(self):
        return self.__y

    def distance_from_xy(self, x, y):
        return math.hypot(abs(self.__x - x), abs(self.__y - y))

    def distance_from_point(self, point):
        return self.distance_from_xy(point.getx(), point.gety())


point1 = Point(0, 0)
point2 = Point(1, 1)
print(point1.distance_from_point(point2))
print(point2.distance_from_xy(2, 0))

```

Back

Triangle

```
import math


class Point:
    def __init__(self, x=0.0, y=0.0):
        self.__x = x
        self.__y = y

    def getx(self):
        return self.__x

    def gety(self):
        return self.__y

    def distance_from_xy(self, x, y):
        return math.hypot(abs(self.__x - x), abs(self.__y - y))

    def distance_from_point(self, point):
        return self.distance_from_xy(point.getx(), point.gety())


class Triangle:
    def __init__(self, vertice1, vertice2, vertice3):
        self.__vertices = [vertice1, vertice2, vertice3]

    def perimeter(self):
        per = 0
        for i in range(3):
            per += self.__vertices[i].distance_from_point(self.__vertices[(i +
1) % 3])
        return per


triangle = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))
print(triangle.perimeter())

```

Back

Character frequency histogram

```
from os import strerror

# Initialize 26 counters for each Latin letter.
# Note: we've used a comprehension to do that.
counters = {chr(ch): 0 for ch in range(ord('a'), ord('z') + 1)}
file_name = input("Enter the name of the file to analyze: ")
try:
    file = open(file_name, "rt")
    for line in file:
        for char in line:
            # If it is a letter...
            if char.isalpha():
                # ... we'll treat it as lower-case and update the appropriate
counter.
                counters[char.lower()] += 1
    file.close()
    # Let's output the counters.
    for char in counters.keys():
        c = counters[char]
        if c > 0:
            print(char, '->', c)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))

```

Back

Sorted character frequency histogram

```
from os import strerror

counters = {chr(ch): 0 for ch in range(ord('a'), ord('z') + 1)}
file_name = input("Enter the name of the file to analyze: ")
try:
    file = open(file_name, "rt")
    for line in file:
        for char in line:
            if char.isalpha():
                counters[char.lower()] += 1
    file.close()
    file = open(file_name + '.hist', 'wt')
    # Note: we've used a lambda to access the directory's elements and set reverse to get a valid order.
    for char in sorted(counters.keys(), key=lambda x: counters[x], reverse=True):
        c = counters[char]
        if c > 0:
            file.write(char + ' -> ' + str(c) + '\n')
    file.close()
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))

```

Back

Evaluating students' results

```
1  # A base exception class for our code:
2  class StudentsDataException(Exception):
3      pass
4
5
6  # An exception for erroneous lines:
7  class WrongLine(StudentsDataException):
8      def __init__(self, line_number, line_string):
9          super().__init__(self)
10         self.line_number = line_number
11         self.line_string = line_string
12
13
14 # An exception for an empty file.
15 class FileEmpty(StudentsDataException):
16     def __init__(self):
17         super().__init__(self)
18
19
20 from os import strerror
21
22 # A dictionary for students' data:
23 data = { }
24
25 file_name = input("Enter student's data filename: ")
26 line_number = 1
27 try:
28     f = open(file_name, "rt")
29     # Read the whole file into list.
30     lines = f.readlines()
31     f.close()
32     # Is the file empty?
33     if len(lines) == 0:
34         raise FileEmpty()
35     # Scan the file line by line.
36     for i in range(len(lines)):
37         # Get the i'th line.
38         line = lines[i]
39         # Divide it into columns.
40         columns = line.split()
41         # There shoule be 3 columns - are they there?
42         if len(columns) != 3:
43             raise WrongLine(i + 1, line)
44         # Build a key from student's given name and surname.
45         student = columns[0] + ' ' + columns[1]
46         # Get points.
47         try:
48             points = float(columns[2])
49         except ValueError:
50             raise WrongLine(i + 1, line)
51         # Update dictionary.
52         try:
53             data[student] += points
54         except KeyError:
55             data[student] = points
56     # Print results.
57     for student in sorted(data.keys()):
58         print(student,'\t', data[student])
59
60 except IOError as e:
61     print("I/O error occurred: ", strerror(e.errno))
62 except WrongLine as e:
63     print("Wrong line #" + str(e.line_number) + " in source file:" +
   e.line_string)
64 except FileEmpty:
65     print("Source file empty")
66
```

Back

The os module: LAB

```python
import os

class DirectorySearcher:
    def find(self, path, dir):
        try:
            os.chdir(path)
        except OSError:
            # Doesn't process a file that isn't a directory.
            return

        current_dir = os.getcwd()
        for entry in os.listdir("."):
            if entry == dir:
                print(os.getcwd() + "/" + dir)
            self.find(current_dir + "/" + entry, dir)


directory_searcher = DirectorySearcher()
directory_searcher.find("./tree", "python")

```

Back

The datetime and time modules

```python
from datetime import datetime

my_date = datetime(2020, 11, 4, 14, 53)

print(my_date.strftime("%Y/%m/%d %H:%M:%S"))
print(my_date.strftime("%y/%B/%d %H:%M:%S %p"))
print(my_date.strftime("%a, %Y %b %d"))
print(my_date.strftime("%A, %Y %B %d"))
print(my_date.strftime("Weekday: %w"))
print(my_date.strftime("Day of the year: %j"))
print(my_date.strftime("Week number of the year: %W"))

```

Back

The calendar module

```python
import calendar


class MyCalendar(calendar.Calendar):
    def count_weekday_in_year(self, year, weekday):
        current_month = 1
        number_of_days = 0
        while (current_month <= 12):
            for data in self.monthdays2calendar(year, current_month):
                if data[weekday][0] != 0:
                    number_of_days = number_of_days + 1

            current_month = current_month + 1
        return number_of_days

my_calendar = MyCalendar()
number_of_days = my_calendar.count_weekday_in_year(2019, calendar.MONDAY)

print(number_of_days)

```

Back

PCAP Exam Syllabus

The exam consists of five sections:

Section 1	6 items	Max Raw Score: 12 (12%)
Section 2	5 items	Max Raw Score: 14 (14%)
Section 3	8 items	Max Raw Score: 18 (18%)
Section 4	12 items	Max Raw Score: 34 (34%)
Section 5	9 items	Max Raw Score: 22 (22%)

Section 1: Modules and Packages (12%)

PCAP-31-03 1.1 – Import and use modules and packages

- import variants: import, from import, import as, import *
- advanced qualifying for nested modules
- the `dir()` function
- the `sys.path` variable

PCAP-31-03 1.2 – Perform evaluations using the math module

- functions: `ceil()`, `floor()`, `trunc()`, `factorial()`, `hypot()`, `sqrt()`

PCAP-31-03 1.3 – Generate random values using the random module

- functions: `random()`, `seed()`, `choice()`, `sample()`

PCAP-31-03 1.4 – Discover host platform properties using the platform module

- functions: `platform()`, `machine()`, `processor()`, `system()`, `version()`, `python_implementation()`, `python_version_tuple()`

PCAP-31-03 1.5 – Create and use user-defined modules and packages

- idea and rationale;
- the `__pycache__` directory
- the `__name__` variable
- public and private variables
- the `__init__.py` file
- searching for/through modules/packages
- nested packages vs. directory trees

Section 2: Exceptions (14%)

PCAP-31-03 2.1 – Handle errors using Python-defined exceptions

- except, except:-except, except:-else:, except (e1, e2)
- the hierarchy of exceptions
- raise, raise ex
- assert
- event classes
- except E as e
- the `arg` property

PCAP-31-02 2.2 – Extend the Python exceptions hierarchy with self-defined exceptions

- self-defined exceptions
- defining and using self-defined exceptions

Section 3: Strings (18%)

PCAP-31-03 3.1 – Understand machine representation of characters

- encoding standards: ASCII, UNICODE, UTF-8, code points, escape sequences

PCAP-31-03 3.2 – Operate on strings

- functions: `ord()`, `chr()`
- indexing, slicing, immutability
- iterating through strings, concatenating, multiplying, comparing (against strings and numbers)
- operators: `in`, `not in`

PCAP-31-03 3.3 – Employ built-in string methods

- methods: `.isxxx()`, `.join()`, `.split()`, `.sort()`, `sorted()`, `.index()`, `.find()`, `.rfind()`

Section 4: Object-Oriented Programming (34%)

PCAP-31-03 4.1 – Understand the Object-Oriented approach

- ideas and notions: class, object, property, method, encapsulation, inheritance, superclass, subclass, identifying class components

PCEP-31-03 4.2 – Employ class and object properties

- instance vs. class variables: declarations and initializations
- the `__dict__` property (objects vs. classes)
- private components (instances vs. classes)
- name mangling

PCAP-31-03 4.3 – Equip a class with methods

- declaring and using methods
- the `self` parameter

PCAP-31-03 4.4 – Discover the class structure

- introspection and the hasattr() function (objects vs classes)
- properties: `__name__`, `__module__` , `__bases__`

PCAP-31-03 4.5 – Build a class hierarchy using inheritance

- single and multiple inheritance
- the `isinstance()` function
- overriding
- operators:
- not is, `is`
- polymorphism
- overriding the `__str__()` method
- diamonds

PCAP-31-03 4.6 – Construct and initialize objects

- declaring and invoking constructors

Section 5: Miscellaneous (22%)

PCAP-31-03 5.1 – Build complex lists using list comprehension

- list comprehensions: the `if` operator, nested comprehensions

PCAP-31-03 5.2 – Embed lambda functions into the code

- lambdas: defining and using lambdas
- self-defined functions taking lambdas as arguments
- functions: `map()`, `filter()`

PCAP-31-03 5.3 – Define and use closures

- closures: meaning and rationale
- defining and using closures

PCAP-31-03 5.4 – Understand basic Input/Output terminology

- I/O modes
- predefined streams
- handles vs. streams
- text vs. binary modes

PCAP-31-03 5.5 – Perform Input/Output operations

- the `open()` function
- the `errno` variable and its values
- functions: `close()`, `.read()`, `.write()`, `.readline()`, `readlines()`
- using bytearray as input/output buffer

Now that you have completed Python Essentials 2, book an exam and take the PCAP Certified Associate in Python Programming Exam.

Go to www.PythonInstitute.Org to purchase an exam voucher.