# Python 3 Programming

2nd Edition

JUSSI PEKKA KASURINEN

2021

# Python 3 Programming

2nd Edition

**JUSSI PEKKA KASURINEN**



This work is also a translated, revised and updated version of the book “Python 3 ohjelmointi”, ISBN 978-951-0-35273-1, originally published by WSOYPro/Docendo, 2009.

# Table of Contents

# Preface

The book right now on your screen, since nobody uses paper anymore, is a book about the fundamentals of Python programming language, and in a larger scale the first step into the world of programming. In this book, we discuss of all the basics and details of the introductory things, that you will need to understand the basic concepts and continue from that by learning yourself. This book is starter-friendly, but it is not meant as a manual; you will need to test the examples yourself, study also by yourself and at the best-case scenario, have an interactive course in which you can use the things this book talks about.

The examples of this book are designed so that they are easy to follow, yet show in detail the needed aspects. As the book advances, the examples will become more complex, but in the design of this book the most problems are addressed based on my personal experiences from teaching programming for the last fifteen years.

Since this book has been out of print for almost a decade, there are obviously some things that have changed, or might no longer be as an acute problem as they were before. However, all parts of the 2nd edition have been tested, and if you find errors or out-of-date information, you should contact me via email, so we that we can update everything up to the snuff now that this book is once again available.

In any case, hopefully you will find this manual useful, and I hope that you enjoy reading and using it!

As for the last words for this preface, I would kindly like to thank WSOY, WSOYPro and Docendo, the previous publishers of this book for their cooperation and support for developing the materials so many years ago, and the current publisher Viope Solutions for allowing a limited release of this manuscript for the use of the LUT Software Engineering department.

In Kotka, Finland, 14.6.2021

Jussi Pekka Kasurinen

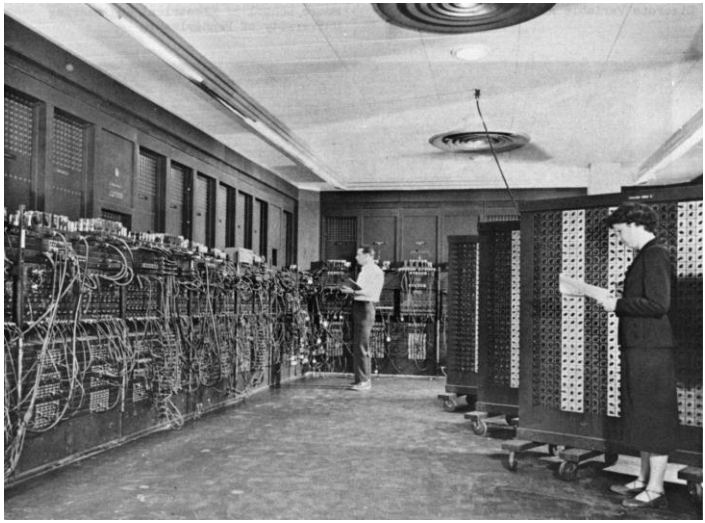
# Chapter 1: Introduction to Programming

Welcome to the introduction to the Python-programming! During this course, we will take a look into the fundamentals of Python 3 programming, covering the basic components and structures of the Python language. The course material focuses on introducing the concepts of Python 3, and by presenting code examples, easing the process of understanding how the Python works.

If you are eager to jump on actually doing stuff, you can skip to the page 29, and start doing the actual programming!

## *Historical starting points*

The history of programming and programmable computers started in the 1940's, when the static computing machines were becoming advanced enough to warrant more usage than just one given, purpose-built tasks. The supercomputers of that decade, ENIAC, Zuse Z3 and Colossus, were the first programmable computers in their actual sense; the programmer could give the system new purposes and tasks, which were beyond the initial designated tasks for the systems. To do this, it was obvious that a they needed a set of rules, tasks and commands, which would tell the computer what to do next, since it no longer was hardwired to the systems brains. By making a set of these commands for various usages, and how they were implemented, was the starting point for a programming as a concept.



ENIAC-supercomputer from the early 1950's. Notice the part of the systems "keyboard" at the left wall.

However the ENIAC, Zuse Z3 and Colossus were massive installations with very limited actual capabilities, and the thought of selling one of those for private, or even commercial use, was considered completely ridiculous, and there was no realistic way of mass produce any of them. It actually took a better part of 20 years in the advantages in the microchip design, microprocessor advantages and electronics miniaturization to reach the point, where the PC workstations and other similar systems to what we know, first appeared.

The era of workstations started with Intel corporation's 4004-microprocessor, and its advanced version 8080 in the early years of the 1970's. Because of the sudden boom of relatively capable computers being available to customers in a mass-produced volume, the programming skills and the need for programming-capable engineers transformed from the mysterious art form conjured by the computation engineers to a skill that was more or less needed everywhere. During this era, the first programming languages that are still used to at least some degree, such as FORTRAN, ALGOL or COBOL were born. In addition of this, the universities started to realize that programming skills, information technology and computer science was a thing, and started to train dedicated computer science specialists.

Around the same time with the home computers the programming languages kept evolving, producing languages such as C, C# or Pascal, which even today are still in use, or at least have relatives which are being actively developed. For example, the C-language is still evolving, with the latest additions to the language being currently in discussions, and the language itself topping as #1 on the TIOBE index for

programming language usage in 2020. Against this backdrop it is safe to say, that the fundamental concepts of programming have not veered that much from the era, when the computers took enough space to fill a five-story apartment building, to the era where computers reside at our tables, pouches and palms as smartphones, laptops and work-from-home battle stations.

## *Python and the other programming languages*

Deep down in the system, the computers process every piece of information as a stream of ones and zeros, which dictate whether the computer reads or writes memory banks, calculates things, sends messages forward to the auxiliary systems or does something else. This steam of ones and zeroes is called the machine code, which in principle is a programming language composed of primitive commands and instructions used by the computer system itself. As the machine code is really cumbersome, error-prone and primitive way to express actions, it is not really used directly in programming tasks by anyone except maybe the very high-end gurus of programming. In these limitation lays the reason for programming languages to exist: to help people tell the computers what to do, by providing means to express oneself easily enough to learn how to create programs, but in a manner which the computer system also understands. Even if we take something like Assembly presentation, it really is not very “open” for understanding what is happening:

```
                .file           "hello.c"
                .text
                .section        .rodata
.LC0:
                .string         "Hello Maailma!"
                .text
                .globl          main
                .type           main, @function
main:
.LFB0:
                .cfi_startproc
                endbr64
                pushq           %rbp
                .cfi_def_cfa_offset 16
                .cfi_offset 6, -16
                movq            %rsp, %rbp
                .cfi_def_cfa_register 6
                leaq            .LC0(%rip), %rdi
                call            puts@PLT
                movl            $0, %eax
                popq            %rbp
                .cfi_def_cfa 7, 8
                ret
                .cfi_endproc
.LFE0:
                .size           main, .-main
                .ident          "GCC:   (Ubuntu   9.3.0-17ubuntu1~20.04)
9.3.0"
                .section        .note.GNU-stack,"",@progbits
                .section        .note.gnu.property,"a"
                .align 8
                .long            1f - 0f
                .long            4f - 1f
                .long            5
0:
                .string          "GNU"
1:
                .align 8
                .long            0xc0000002
                .long            3f - 2f
2:
                .long            0x3
3:
                .align 8
4:
```

Best said about this is that this is actually what the symbolic presentation of printing out Hello World, something we do with Python later in this chapter, looks like in the lowest level of abstraction.

## Applications of different languages

However, it was soon discovered that one language cannot be specific yet dynamic enough to cater all of the different uses for computers, so the development of programming languages started to head towards specialized languages. For example, PHP, Java, Python and C++ are rather well-known programming languages, which all have different strengths and weaknesses in terms of applicability. PHP is really well-suited for creating basic websites, whilst Java is excellent in building platform-independent software and C++ can beat almost any language in the pure performance and code optimization. Basically, the selection of the language is the same as using the right tool for the job: creating a C++-based browser application or an embedded Java program for hardware system is really difficult or at least laborious, because the languages aren't supposed to be used for that sort of purposes.

Besides the application domain, other way to differentiate between the different languages is the division to high-level and low-level languages, based on how much abstraction is available between the machine code presentation and programming language. In the low-level languages such as C, the commands and available actions differ only little from the pure machine code presentation. The programmer has to implement features such as memory management and dynamic structures by hand, and only little automation is usually available for tasks such as using the internet connection or using graphical interface. The

upside to this is that the code can be made really effective and quick, and the programmer knows every detail of how the program operates. In high-level programming languages such as Python, the details are left for the interpreter or compiler to

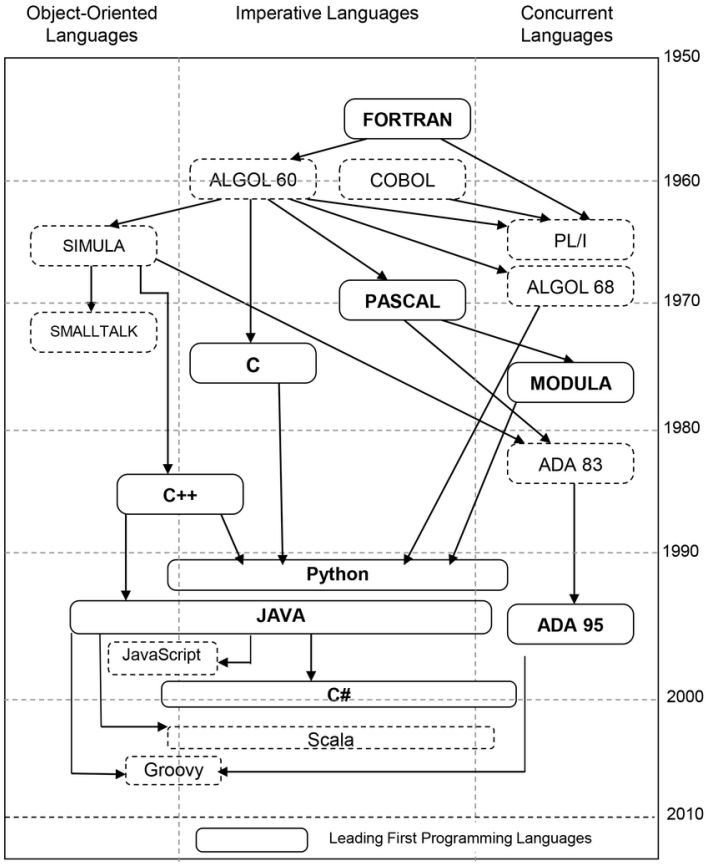



*Figure 1.1: Most programming languages are actually at least somewhat related, here are some main influences between few of the different languages. (From Farooq et al., DOI: 10.1371/journal.pone.0088941)*

worry about, while the programming tasks focus on more high-level activities, such as defining how the GUI should look, or what kind of data is searched from the database. However, because most programming languages have at least some common elements, due to most of them being at least somewhat related to each other, a skilled programmer can pick up a new language relatively quickly once they master one or two languages.

## Different programming styles

Besides high- and low-level programming the different programming languages also support different programming styles, which are also called paradigms. The programming paradigms mean the fundamental way of how the source code is implemented: in procedural programming, the source code is made of functions and variables, whereas in object-orientation the code is mostly objects and methods. In procedural programming the program functions so that there is a function which performs the action, and arguments or variables, which exchange the knowledge between the functions. In object-orientation, the activities are made inside self-contained classes, from which a group of objects are created to perform different tasks. Anyhow, there are different programming paradigms which define how the code is made; in mathematics, there is an functional paradigm which focuses on defining what is modified instead of how it is modified, and with advanced interfaces, it is common to create an event-driven code, where activities are defined as a response to the different things the user can do, like pressing a button or pushing a switch.

However, unlike the division to the high and low-level programming languages, one programming language is not tied

to only one programming paradigm. For example Python, the programming language used in this course, can be used with procedural, object-oriented, and event-driven paradigm. By expanding the command library, it can be argued that even functional programming becomes possible. This is possible because of high level of abstraction; because Python is a high-level language, the syntax and logic behind the language is not that restrictive and the source code can be made to work in several ways. In low level languages, the restrictions usually cause the language to select one programming paradigm, but implement it very well.

## *Other tools for the programming*

The programming language itself is not that much; it is a collection of structures and commands, which like the popular Danish plastic construction blocks are used to create something grander. With these commands, we can tell the computer what to do, and give instructions how the read data should be managed, but we still only have a recipe for the things to happen. We need more software and programming tools to make the system actually work.

The first tool we need, is the editor software. For each programming language, there usually is the default editor, which is also bundled with the default tools that come with the downloadable developer kit. And instead of other text editors such as Microsoft Word or LibreOffice Writer, the programmer's editor actually recognizes the used language, knows how it is supposed to work, and highlights the necessary construct parts, so that it is easier to follow and understand. Usually this happens by highlighting the keywords, using different colours to

differentiate the structure from the content, and in general, helps on completing the source code syntax. Sometimes these programs even can find logical errors, and even suggest better ways of doing programming work, or at least avoid problematic solutions.

The other tool that we absolutely need, is the compiler, or interpreter, which transforms the source code from the list of commands into actually functional program, or series of activities. This is actually the core of the programming language; a language without functional compiler or interpreter is not really a programming language, since without one the language actually cannot function. The difference of programming language, whether it is compiled of interpreted language is a technical one. The compiled languages are fully translated every time we compile the source code, and only if everything is correct enough, we can test drive the program. The interpreted languages are read line-by-line, and the system will sometimes start even with an incomplete program, but will crash the moment the system tried to do something that causes an error. This is also why compiled languages are usually faster; they already exist in the machine language form so they do not need additional layer of software such as an interpreter to function; on the other hand, they might have more difficult issues to fix since the compilers might overlook logical issues and the crash may be related to something completely different, that was done at the time. Python 3 is an interpreted language, with the interpreter that is required to run it, but for this course the performance should not be an issue. However, there are also projects which aim to wrap, or actually compile Python completely as a 2-stage compile work. This is usual if we are using tools such as .NET framework, which also supports Python but actually requires a

fully compiled system to function. However, this is advanced stuff, so the thing to remember is that native Python 3 is an interpreted language.

In addition of the editor and compiler/interpreter, there usually is also call for assisting tools such as debugging tools or unit test frameworks, or modelling tools. These are advanced tools, which can be useful on both maintenance of the existing code, or thinking about the implementation before starting to build one, or simplify the work to ensure that everything still works as intended. The IDLE editor comes with an internal debugging tool, and Python itself with its standard library has an implementation of XUnit test suite. But these are more advanced tools, which we do not need right now. But as with many other things, it is a worthwhile to mention these so that you know of their existence in preparation of the more advanced courses, where the program complexity grows into a level where simple troubleshooting, for example with list on the Chapter 8 is no longer an option.

## *On Python programming language*

Python 3 is the newest major version of the Python programming language, first released in 1991. Python 3 was released at December of 2008, and the newest subversion, 3.9.5, was launched at 3rd of May, 2021. The first thing that needs to be understood is that Python 1/2 and Python 3 are basically two different languages, albeit for obvious reasons they are very close to each other. They are relatives, but they are not the same language, and they have differences which makes them incompatible to each other, starting from even the most simplified program snippets.

*If you are seeking help from online sources such as StackOverflow, always check that you are actually reading instructions for Python 3, not Python 2! If the instructions were written pre-2012, it most likely is for Python 2 and will not work.*

The largest differences between the second and the third versions is that the Python 3 streamlines the language syntax and improves some functionalities such as input handling. However, the changes have made the Python 3 backwards incompatible with the earlier versions of the Python source code. However, the reason for breaking backwards compatibility has paid off in several ways. For example, the syntax has been cleaned for almost all of the special cases such as print-command, which now operates as a function. Similarly, the old datatype-specific input-function has been eliminated, and replaced with the raw_input. The main differences can be summarized as follows:

- Print-command functions like a proper function, and accepts additional arguments to define the end-line and division characters.
- Comparison between two different sequences (for example list and tuple) with logical operators raises TypeError, instead of returning Boolean character defined with the the arbitrary and confusing comparison rules which existed in the Python 2. In Python 3, if the operands do not have a meaningful natural ordering, they cannot be compared against each other besides with == and != operators.
- String formatting with %-operator is deprecated starting with the version 3.1. There is a new method for string formatting, operating as a format-method for string-datatypes.

- Input handling is solely done via function called input, which operates exactly as the raw_input in the Python 2. There is no separate raw_input in Python 3.
- Division operator (/) returns float, even with two interger values, if the result has non-zero decimals. The old division operator can be used with syntax "//", like results = 10//3.

The following code examples illustrate the main differences:

## *Printing*

Python 2-print command:

```
print "Hello World!",
```

Same code in Python 3:

```
print("Hello World!",end = "")
```

Or even shorter

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

The difference is that the Python 3 print is a proper function, unlike the earlier print-operator which applied its own rules and designs. Because print is now a function, it also have two arguments end and sep, which make it easier to achieve some of the layout tricks used with the printing command.

## *Inputs*

In Python 2, the input is taken as follows:

```
value = input("Please insert value: ")
```

or alternatively

```
value = int(raw_input("Please insert value: "))
```

Same command with Python 3:

```
value = int(input("Please insert value: "))
```

The old type-specific input has been eliminated and the new function operates similarly as the old raw_input.

## Python compared to other languages

In a way, Python is a peculiar language, as there really is no clear niche the language is aimed at. Based on application areas where the language is mostly used, it is close to Perl, although with much clearer syntax and better support for platform independence. However, when building stand-alone software, it can work as well as C++ or Java, and in network services, it can be supplemented with right extensions to be comparable against Ruby with on Rails, or overtake PHP on website scripts.

Originally the Python language was created as an extension for a language called ABC, with features taken from languages such as Modula-3, Haskell, Lisp and Perl. In the later development, there are distinct similarities with C and Java. The original implementation of Python interpreter, CPython, was programmed with C-language. The main features of Python can be summarized as follows:

- Instead of brackets { and }, Python uses indentation as the divider for logical source code blocks.
- Variables are dynamically typed, se their contents can be redefined without any restrictions; a string can be given as a value to a variable which previously had an integer. There also is no need to separately declare the variables, they can be created "on-the-fly" when needed within the source code.
- The syntax is primarily designed to be understandable, simple and minimalistic. This is the reason why one of the most common application areas for Python is as the first taught language in the introductory programming courses.
- Even though Python is originally an object-oriented language, the syntax is designed to allow procedural

programming and ignore the object-oriented stuff such as class declarations. In addition, with additional libraries the language can be extended to allow several other programming paradigms.

- Python is deliver with "everything including the kitchen sink": The standard installation package comes with an extensive module library and all of the needed software to get started. There is no need for separate function libraries, editor packages or other stuff to try and start with the language.
- There are several strong suites within the Python core functionalities: extensive structured data types, several data manipulation methods and especially strong string operators.

On several occasions, the Python has been said to combine the straightforwardness of scripts with the possibilities of programming languages. However, it should be noted that **Python is not a scripting language**, but rather a high-level programming language which can, and is, used to create a commercial, complete software. The main design principles of the Python language have also been expressed as a poem:

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

This poem is usually called *the Pythonin Zen* which lays the foundation for the further development of the language. This, and many other design principles and commentation on the existing features can be found at the Python Enhancement

Proposals, which are publicly available at http://www.python.org/dev/peps/.

## *Application areas for Python*

Even though it is true that Python can be used to create full programs, it is rather unusual application area for Python. In professional development, Python is usually applied as a support language to create small tool functionalities such as data generators or filters. In addition to the tool funtions, Python is also used to make small programs such as hex calculators, IRC-bots or image converters. For this area of application, the Python has a module for creating a graphical user interface with a small amount of code:

```
from tkinter import *
import sys

def endme():
    window.destroy()
    sys.exit(0)

window = Tk()

text_window = Label(window, text = "Tkinter in action")
text_window.pack()

endme_button = Button(window, text = "Quit", command =
endme)
endme_button.pack(side = BOTTOM)

window.mainloop()
```

This small snippet of source code defines a program, which has a window, a text box with a short description and a button that has a functionality, which ends the program. Also, because

Python is a platform independent programming language, this program works in all of the different operating systems (for example Windows, OSX, Linux distributions...) for which the Python interpreter and the standard module library is implemented.

Besides easy GUI generation, the ability to modify strings has been a large marketing point; the ability to edit and slice character strings and dynamic structures is exceptionally well-implemented and easy to use. For example, one fairly common exercise in string comparisons is the palindrome test. In Python, the actions required to turn the string backwards and compare it against the original makes this exercise almost trivial:

**Source code**

```
01  # -*- coding: cp1252 -*-
02
03  while True:
04      word = input("Give the tested string: ")
05
06        #test if the string backwards is same as the
original
07      if word == word[::-1]:
08         print("Given string,",word,"is a palindrome!")
09      else:
10          print("String is not a palindrome.")
11
12
13      decision = input("Test another? (Y/N): ")
14
15      if decision == ("N" or "n"):
16          print("Program has been terminated.")
17          break
```

**Result**

```
>>>
Give the tested string: oolated squigg
String is not a palindrome.
Test another? (Y/N): y
Give the tested string: racecar
Given string racecar is a palindrome!
Test another? (Y/N): n
Program has been terminated.
>>>
```

Notice, that the actual conversion and comparison for the string is implemented as a one command in the line 7. Everything else in the code is "fluff" used to define and build the program interface.

## *Using the Python interpreter*

The development and distribution of the Python interpreter is done by the non-profit organization called Python Software Foundation (http://www.python.org), from which the installer package for the interpreter and all of the basic tools is available. This installation package includes all of the essential tools for basic programming; an interpreter, debugging tools for finding errors from the software and most importantly, a source code editor called IDLE. In addition, all of the included tools are platform-independent, meaning that by learning the ropes of the tools allows programming in all of the most usual PC operating systems (Windows, Linuxes, MacOS...). This is also the reason why this course expects you to use these tools; they work regardless of the actual system you are using. However, if you are using this course with other tools, for example different editor, it is still wise to use the actual PSF-version of the Python interpreter.

Additionally, if the source code editor you are using is not IDLE, it is wise to make sure that the editor actually understands how Python syntax works. Python language has eliminated the brackets ( "(", ")", "{}" etc.) from the syntax almost completely, but uses indentation to denote the logical groupings of the code. This is discussed in more detail later, but for now remember two things: A) if in doubt, use IDLE and B) do not indent the source code unless especially told to do so.

## Installing Python on the workstation

These instructions are meant for permanent installation of the Python interpreter and the basic tools for the workstation. This is completely optional, as everything in this course can be done

with the appropriate online tools. If you do not want to, or cannot install Python to your workstation, you may freely skip forward a couple of pages to the last pages of this chapter.

The installation of the Python interpreter is rather straightforward and does not need any special skills. The Python interpreter does require administrator rights for guaranteed successful installation, so unless you are doing the installation on your own computer, you will need a person with the sufficient rights to do the installation. There also is a possibility that the installation package has changed after these instructions were written, so use these instructions only as an advice, and do what the installation asks, if the package instructions differ from these.

**On Windows workstations**, the first action is to grab the installation package from the the Python website. This website is at address *www.python.org*, and it does not need any separate registration or user account for fetching the installation packages.

On the top bar at the main page is a title “Downloads”, and subtitle "Windows ". There is also a button which instructs the browser to immediately try to load the installation package for the most up-to-date Python. Save this file to the download directory.

When the package has loaded, it should be neighbourhood of ~30 megabytes, run the package. In installation, ensure that all of the package contents are installed, and accept the default settings. This installs Python interpreter to the system drive. If necessary, reboot the system afterwards to finalize the installation.

**On most Linux systems** the Python can be installed from the package management, as Python usually is included in the standard package repository. Please ensure that the installed version is "3.-something", as in some systems the Python 2 might still remain the "default" Python.

**On Mac systems** the installation packages can be found at Python homepage *www.python.org*, following a link "Download" at the top bar of the main page. Select the most appropriate installation package depending on your hardware, and run the installation as an admin.

## Additional module libraries

The design principles of everything plus kitchen sink in Python basically mean that the installation package should be enough for most basic needs in programming. This is most obvious when looking at the installation package; it has everything needed to start writing working Python code regardless of the actual platform, with an extensive module library which allows several different special activities, such as GUI programming, using the network connection or engineering calculus.

However, the Python is also known for its extensive library of additional modules, which extends the already huge library of premade functions even more. In the following some of the most important additional Python libraries are listed. Since most of them are supported by third parties, the availability of these modules to the different versions of Python installations varies; right now you do not need to install any of these, but it is good to be aware of them for the future.

## *Pillow*

Pillow is a Python 3-compatible version of the Python Imaging Library (http://www.pythonware.com/products/pil/), which is a group of extra modules, which enables Python to create and edit basically any image format available. This module has an extensive selection of different picture manipulation tools and different filters for doing pretty much anything any good picture editing software would be able to do.

## *Pycrypto: Python Cryptography toolkit*

Python Cryptography toolkit (http://www.amk.ca/python/code/crypto) is a toolset meant for working with different cryptography algorithms and coding schemes. Somewhat hard to find on the Internet, as some countries still restrict the distribution and usage of said functions.

## *Numpy: Numeric Python*

Numeric Python (http://numpy.scipy.org/) has even more tools for scientific calculations and engineering calculus. Other notable feature is the ability to do matrix calculations, similarly as in Matlab..

## *Django*

Django (http://www.djangoproject.com/) is a large module which is almost a separate entry for programming languages itself. Django makes the Python able to create and maintain websites, something that was previously lacking from the Python language itself, but readily available in one of the most important

competitors of Python, Ruby programming language (with Ruby on Rails framework).

## Regarding the third party modules in general

When getting these packages, attention should be paid on the version number of the Python interpreter, the package is intended for. The package meant for Python 2.5 does not work in Python 3.X, even if it in some random case happens to install itself. Also, the third party modules are not always professionally made; usually they are a product of a small group of enthusiasts or non-profit group. They may have errors or certain problems in the execution or compatibility.

## The module library of standard installation

Besides third party material, the installation package also contains several useful and interesting modules which are there to help with implementing some of the more difficult tasks such as networking. The easiest way of getting to know these modules is to select the shortcut at the Start menu by selecting the shortcut *Python manuals*, or in the interpreter window by selecting from the drop-down menu Help->Python Docs (hotkey F1). This documentation contains all of the official documentation for the interpreter and modules, and is actually pretty useful, albeit very limited with the complete, functional source code examples.

The most recent and updated version of this material is also available online, in the documentation repository at address http://docs.python.org/py3k/

## *Getting to know the interpreter*

Now that the interpreter is installed, its time to try to use it for a first programming exercise. First thing to do is to start the interpreter by navigating to the Python-named folder in the Start menu, and selecting a shortcut named "IDLE (Python GUI)". When the interpreter starts, it should open the interpreter window, which is a mostly white window which has the following text:

```
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52)
[MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for
more information.
>>>
```

The easiest way of testing whether the installation was successful is to write a command **import sys** to this window. This command imports a library module called sys, which has several system-related functions. If the command is successful, no error message is given, then the interpreter should have everything in satisfactory condition:

```
>>> import sys
>>>
```

*Some older spyware detectors and firewalls may sometimes block the interpreter from starting, causing an error "Subprocesses did not connect". The best way to deal with this error is to give a program executable named "pythonw.exe" in the installation folder all the rights it asks for, for the local workstation.*

Anyway, now that the interpreter is running and installed correctly, its time to look around a bit. First of all, when the interpreter starts, it opens a new windows called "interpreter shell", and prints the current version number of the installed interpreter. In this window, there also is a command line prompt with preceding characaters ">>>". To this prompt, different Python commands can be tried out to test how the interpreter would react on it. In this course, if some code example has the preceding ">>>", it means that this command is written directly to the interpreter shell. For example, giving the print-command causes the interpreter to print the given text:

```
>>> print("Test!")
'Test!'
>>>
```

Similarly, the interpreter can also act as a calculator with numbers and the normal mathematical operators (+, -, /, * etc.)

```
>>> 3 + 5
8
>>> 45 / 6
7.5
>>>
```

The interpreter shell is useful for testing out stuff, but its not very good at writing actual programs, as the interpreter keeps always responding something when the 'Enter'-key is pressed. The real programs are written into a separate file, and they have absolutely more than one command. It would also be useful for the execution if the interpreter would not "think out loud", as it does in the interactive shell. All this can be done by opening a new editing window from the shell window.

To open a new editing window, simply select File->New Window from the drop down menu (or press Ctrl-N). To this window, all of the source code written will be executed immediately, and only the explicitly printed values and text is shown on the interpreter shell.

## *The First Python program*

So the last thing in this chapter is making the first actual Python program. Lets open a new editing window, and on that screen write the following to commands. Pay attention to the details, the commands must be written exactly as shown here:

```
print("This is our first Python program,")
```

and

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

After this, the source code has to be saved by selecting a drop-down menu "File" and "Save as". In this window the name of the new source code file is given, for this example "hello-code.py" will do nicely. Please pay attention to the file extension name ".py", which tells the system that this file contains Python code, similarly as ".docx" would tell that the file is Microsoft Word document, of ".png" that the file is a picture. Also, if the IDLE opens up a dialog window, where it suggests adding a line which is something similar to "# -*- coding: UTF-8 -*-" to the file, select "edit my file". This is triggered by the IDLE identifying non-ASCII or special characters in the source code file, making it want to add a comment to the file telling the interpreter that which code page the code should use. This does not change the behavior of the program in any way, and it is only meant to help the interpreter to print these special characters, (Ä,É,Ö for example) correctly.

*If the interpreter does not understand the character it is supposed to print, it may come out garbled mess, like "&?a" or "â£"*

*for "Ä". There is no easy cure for this problem, but this is not in any way harmful for the program, only an aesthetic nuisance. Since most of the world has converted to using UTF-8 encoding, this is exceedingly rare problem. It still might happen every now and then, especially with the older systems, so it is useful for you to be aware of it.*

The last thing to do is to ask the interpreter to execute the program and run the written source code through the interpreter. However, before doing this, ensure one last time that the written code is exactly like this, without any indentation and character by character verbatim to the following:

```
print("This is our first Python program,")
print("Hello World!")
```

In Python - and really in any programming language - the commands have to be always written correctly for them to work. The Python commands are always written only in small letters - not Print, PRINT or pritn but print - as the letter size is a meaningful difference for the interpreter, and the interpreter does not guess when encountering a typo, it gives an error message and terminates the program. Also, the quotation marks and brackets have to be at the both ends of both commands.

If everything is in order, select "Run" and "Run Program" from the drop-down menu (hotkey F5). If the interpreter or IDLE gives an error at this point, something is written incorrectly. Otherwise following printout should appear on the interpreter shell:

```
>>>
This is our first Python program,
Hello World!
```

```
>>>
```

## *Example 1-1: The first Python program*

### Source code

```
01  print("This is our first Python program,")
02  print("Hello World!")
```

### Output

```
>>>
This is our first Python program,
Hello World!
>>>
```

The next task is to try this on your own, by doing a similar exercise on the IDLE editor and interpreter, and then at the next chapter, start learning with the actual Python programming stuff.

# Chapter 2: Fundamentals of the Python Language

In this chapter we will discuss the fundamental concepts of the Python language, such as the naming conventions of the variables, different types of comments and such. This chapter is rather long, but the things introduced in this chapter will be used thruout the course, and the only way to be able to meaningfully discuss the more advanced structures is by understanding the basics. Obviously, there are some things which may not be immediately relevant, but it is OK to return to this chapter every now and then to remind oneself for example how the string slicing was done.

## *Comments*

In this chapter, the aim is to teach the basic things about Python syntax before advancing to the more complex structures. Rather funnily, the first thing discussed has nothing to do with the actual programming, at least in theory. This obviously is the commenting and comments in general.

In some cases, it is important to be able to write random notes and reminders to the code to describe how the program should be working, or why some part of the program is implemented as it is. Especially when the program size starts to grow beyond the about hundred lines of code which is easy to remember, and it becomes impossible to manage all of the details at once, or there are other people contributing to the software, the comments are important way of understanding how the stuff is supposed to work.

### *Comments in source code*

Commenting in programming means the task of writing instructions and notifiers to the source code to remind oneself and possible other programmers on the details of the implemented code. In general, the comment lines are meant for other humans, so they are open, non-formal, freely typed text, which is intended to just convey knowledge. For this same reason, the comments do not affect the way the source code is executed, as in normal cases the interpreter simply bypasses the lines which start with the comment sign.

In Python, the while line is marked as a comment by starting it with the sign # (Shift-3). If this character is the first character in

a new line, it means that the interpreter simply skips forward to the begining of a next source code line. For example, line

# This here is simply comment, the code to come does stuff

would be completely bypassed. Even in a case where the comment line has source code such as this

# pritn("This here should be printed."

the line is bypassed without any error, even though the line is obviously code, has a typo in the print-command and is missing the closing parenthesis ")", because the line is marked as a comment. In fact, this is one of the applications for comments; taking out source code which is no longer needed but still too valuable to completely erase. It can be simply bypassed by "commenting it out".

## *In-line-commenting*

Besides separate comments, the comment can be placed after the working line. For example, the command

print("This is important!") #Comment!

actually has both, command and comment on the same line. The interpreter simply first reads the command normally, and then skips to the next line when the comment character '#' is encountered. This way of adding comment to the end of normal command line is called in-line-commenting, which can be used to emphasize single command or denote part which really needs

attention. However, when using the in-line comments the actualy programming command in the line has to be functional; for interpreter the line with inline comment is same as any other normal line of code.

In Python, there is one exception to the rule "comments do not affect how the program is executed": the first line and code page declaration. In fact, this exception has been used in the examples of the last chapter! The very first line of code in the source code file tells the interpreter on what characters the source code was written in, affecting the way the interpreter works with the non-English letters such as A-umlaut (Ä) or o-umlaut (Ö), which are popular in the Nordic languages.

## *Example 2-1: Commments in the source code*

**Source code**

```
01  # -*- coding: cp1252 -*-
02
03  print("This line will be printed.")
04  #This is a comment, interpreter cannot see me!
05
06  print("This is the second printed line.")
07  #print("This print-command is commented away.")
08
09  #On the next line is inline-comment:
10  print("This is the third printed line.") #Com-
ment!
11
12  # This comment is divided
13  # to several line of code. The interpreter does
not
14   # care for this as long as every line starts
with "#".
15
16  print("This is the last printed line.")
```

**Output**

```
>>>
This line will be printed.
This is the second printed line.
This is the third printed line.
This is the last printed line.
>>>
```

The most important thing to remember is that the lines starting with #-sign are not source code, but instructions for the programmer. In the latter examples in this course, the comments are only used to remind on what the code is doing, they do not affect the way the program works in any way.

## *Variables*

Other fundamental concept in programming are the variables. The variables can be considered as a container or "a box", which can be used to save data such as a name or value for later use. Making a variable is easy: first the variable is given a name, and then the initial value with operator "=". Unlike some other programming languages, in Python the variable does not have to be introduced separately, nor is it bound to a certain type of data. When the variable is created, the content type can be changed whenever and whatever during the program execution. In the following example this is demonstrated:

```
>>> value = 100
>>> print(value)
100
>>> value = "FluffyTheTerrible"
>>> print(value)
FluffyTheTerrible
>>>
```

As demonstrated by the code, the given value is saved to the variable, which can be then used as a substitute for the actual written value in command. The contents of the variable can also be rewritten by simply assigning the new value.

## *Making a variable*

A variable can be made in any part of the source code, unlike in programming languages such as C, where every variable has to be introduced in the beginning of the code. Obviously in some cases the variable has to exist before it is used, such as in using it as a selection variable, because the interpreter cannot just guess some value for them and therefore cannot know if the variable is more or less than five, or true or false, if it does not know the variable beforehand.

```
>>> stringline = "Teapot"
>>> number = 1100
>>> print(stringline)
Teapot
>>> print(number)
1100
>>>
```

In this code, two variables named stringline and number are made. To the variable stringline a character string "Teapot" is saved, and number gets the value 1100 as an initial value. Make notice of the citation marks, they are used to denote a string in the Python, as string can have whitespace characters (line breaks, spaces and tabulations) without any restrictions. With numbers the citation marks are not needed; the interpreter understands that a group of numbers denote one value. This is discussed in more detail later.

## *Using the variables with the command*

The printing command print reacts to the variables as it would to the normal text. The interpreter sends the contents of the variable to the print-command, meaning that the print actually gets the contents, not the name of the variable to work with. This is also the reason why variables work in programming; they are more or less "placeholders" for values, and the interpreter takes care of seeing that the values the variables represent are used and saved correctly. This also is a reason why variables can substitute, and even be compared to solid, static values!

```
>>> variable = "Coffeemug"
>>> print(variable)
Coffeemug
>>> stuff = variable
>>> print(stuff)
Coffeemug
>>> number + 150
1250
>>>
```

Besides substituting static values, the variables can be used for example in calculations. In the last example, the variable value is assigned to another variable, and the variable value is modified by adding another number. However, when doing this there is a possibility for a problem: if the command is

```
>>> name + 500
```

causes an error message

```
Traceback (most recent call last):
File "", line 1, in
```

```
name + 500
TypeError: Can't convert 'int' object to str implic-
itly
>>>
```

because the interpreter cannot add numbers to a string and causes a type error, which then ends the program. This is discussed more later.

### *Example 2-2: Creating and using the variables*

**Source code**

```
01  # -*- coding: cp1252 -*-
02
03  message = "I am the harbinger of doom."
04  print(message)
05
06  numbervalue = 42
07  print(numbervalue)
08
09  word = "blueberries."
10  print("I would like to have ice cream and",word)
```

**Output**

```
>>>
I am the harbinger of doom.
42
I would like to have ice cream and blueberries.
>>>
```

## Variable naming conventions

There are some restrictions on the names that can be given to the variables. The first rule is that the name cannot be anything from the list below, as these names are reserved for the program language syntax. The syntax is the vocabulary of the programming

language, and the syntax words define the programming structures so to avoid confusion, these names are made unavailable.

| | | | |
|---|---|---|---|
| and | elif | import | return |
| as | else | in | try |
| assert | except | is | while |
| break | finally | lambda | with |
| class | for | not | yield |
| continue | from | or | |
| def | global | pass | |
| del | if | raise | |

*Table 2.1 Reserved names in the Python language*

Otherwise the name has to be composed of normal, lower or uppercase letters of the English dictionary (a,b,c...,z, A,B,C...,Z) or numbers (0...9). In addition, underscore ( _ ) can be used to separate different parts of the name, as the variable name cannot have whitespace characters such as space (' '). Other characters, such as dollar signs ($) or percents (%) cannot be used, and although non-English letters (Å, Ä, Æ, Ü ...) might work on some systems they should not be used since they can break the backwards compatibility and cause weird behavior with 3rd party libraries. Additionally, the first character cannot be a number, even if they are otherwise allowed. There are examples of good and bad names on the Table 2.2.

| Variable name | Will it work? |
|---|---|
| flag | Yes it will, it fulfills all of the requirements. |
| conTROl | Yes it will, there are no technical restrictions on mixing lower and uppercase characters. |
| setting_2 | Yes it will, numbers are allowed as long as they are not the first character, as is the underscore. |
| _sign | Yes it will, underscore can be the first character. |
| 10th_item | No it won't, the name cannot begin with number. |
| two-parter | No it won't, the normal dash is not allowed. |
| %checksum | No it won't, there is non-allowed character (%). |
| hääyöaie | Might work, but should not be used; the letters should be from English vocabulary. |

*Table 2.2 Examples for Python naming convention*

Something to remember are also the names which start with two underscores (like __doc__). These variables are internal, automatically generated names which are usually related to the objects and classes. These variables are usually meant only for the internal use, and should not be edited manually. Unless there is a real need to do so, these variables should be left alone.

## *Different printing techniques*

## The print command

So far we have been using the print-command to print out stuff, but have not discussed the actual application methods of this command. The way we have been working so far has been the normal

```
print("Im printing stuff!")
```

which in addition of printing the written text, also produces an automatic end line-character to the print, meaning that the following commands

```
print("Im printing stuff!")
print("Solid!")
```

are printed on the different lines like this:

```
Im printing stuff!
Solid!
```

Similarly, if the printed line has a variable, it is simply defined within the printed line, separated from static text with colons:

```
>>> variable = 4
>>> print("There are",variable,"lights.")
There are 4 lights.
```

The interpreter is nice enough to automatically separate the variable from the text by adding a space to the both sides of the variable:

```
There are 4 lights.
```

This becomes annoying feature when the added space-character makes the punctuation incorrect, like when a exclamation mark is added::

```
>>> variable = "F"
>>> print("You got a grade",variable,"!")
You got a grade F !
>>>
```

If the printed variable is a string, one possibility is always to add the exclamation mark with +-operator, like this:

```
>>> yell = "Go team"
>>> print(yell+"!")
Go team!
>>>
```

but this wont work with numbers as the interpreter tries to process the +-operator as an addition, and causes a problem. Also the type conversion route to string and theh adding the exclamation is an option, but this is only a way to tiptoe around the interpreter, as there are more easier and straightforward options available.

This is the exact reason why print-command can take additional parameters, which can be used to tell the command on what to do. With print, there is a possibility of defining two additional parameters, end and sep, which tell on to what character the string will be ended, and what character is used to separate two different parts of the printed line. These parameters are given in a format [parameter]= [value], like this:

```
print("This line will not have automated line break.
",end="")
print("So this becomes right after that last one.")
```

This prints text

```
>>>
This line will not have automated line break. So
this becomes right after that last one.
>>>
```

Notice, that the given parameter is separated from the rest of the printed text with colon. The parameter sep works similarly:

```
>>> number = 4
>>> print("In a quartet, there are",number,"play-
ers.", sep="")
In a quartet, there are4players.
>>>
```

It is obvious from the example above, but if the standard separator is replaced with an empty string (""), the separating spaces have to be defined in the code. As a final note, it should be mentioned that these operators can be any string, they are not limited to one character:

```
>>> print("Well eat soup.", end = " And spam!")
Well eat soup. And spam!
>>>
```

## Other notifications on presenting data

In addition of print-command parameters, there are some additional functions available for modifying the presented data. For example, if the string has modification characters such as line change (\n) or indentation (\t), and the program is requested to

present these characters without actually using them, the line can be modified to show these characters with a function repr():

```
>>> text = "This text has a line change.\n"
>>> print(text)
This text has a line change.

>>> rawtext = repr(text)
>>> print(rawtext)
'This text has a line change.\n'
>>>
```

Function repr modifies the string to present the exact characters in the string, meaning that it shows the possible layout-editing characters such as \n or \t.

Other function related to representing data is the function round(). This function takes a floating point number - a decimal number - and an integer as a parameter. The function rounds the given decimal to the amount of meaningful numbers as expressed by the integer number:

```
number1 = 3.5
number2 = 4.123123
number3 = 1234.1231513

number1 = round(number1)
number2 = round(number2,2)
number3 = round(number2,4)
print(number1, number2, number3)

>>>
4 4.12 1234.1232
>>>
```

Function works like presented above; the first parameter is the number which will be rounded, and the other parameter defines how many meaningful numbers are saved. If round() is not given the second parameter, a value of 0 is used. In this case, the rounding function returns an integer, as there is no decimal part.

## Presenting strings

With Python-programming language there are some fundamental concept which should be discussed before moving on to the more advanced topics, and the different details on presenting strings is one of them. First of all, in the earlier examples we have been rather close to the maximum width of one printing line; secondly, the different layout characters have been discussed, but not really discussed. These all and some other stuff are covered in this section.

### *Dividing the source code*

In Python programming language, there are cases where the practical width of line becomes cumbersome to maintain. In Python language, the minimum width for line any editor should be able to support is 80 characters, while the maximum is not set. However, at some point the width gets annoying, and at that point it is useful to be able to divide one command to span over several lines. For example, lets consider this overly long command:

```
print("This is a very big and long, even annoying command which takes way too much space and is irritating to handle.")
```

Were Python a normal text editor, the line could be divided like this:

```
print("This is a very big and long, even annoying
command
 which takes way too much space and is irritating to
handle.")
```

but obviously this does not work as the upper line print-command brackets are not closed, and the second line text is not a proper command or comment. In these cases it can be said that one logical line spans over two physical lines. So anyway, the interpreter needs to know that these two lines are connected. This can be done with a connector character "\". The connector character is used as the last character of the cut line, indicating that the line continues at the next physical line:

```
print("This is a very big and long, even annoying \
command which takes way too much space and is irri-
tating to handle.")
```

Now the interpreter undestands that the command is cut in half, and prints out normally:

```
>>>
This is a very big and long, even annoying command
which takes way too much space and is irritating to
handle.
>>>
```

Connector character can be placed at any place, where interpreter would allow space, like between parameters or selection arguments. Its also worthwhile to notice that this is one of the rare occasions in which the indentation is ignored; if the line is

indented normally, the indentation is replicated in the printed text:

```
if True:
    print("This is a very big and long, even annoy-
ing\
    command which takes way too much space \
    and is irritating to handle. And now its full \
    of holes.")
```

This looks pretty nice in the source code editor, but unfortunately it is full of ugly holes when printed:

```
>>>
This is a very big and long, even annoying   command
which takes way too much space     and is irritating
to handle. And now its full     of holes.
>>>
```

With connector character, the indentation should be ignored completely, starting the new line nonindented:

```
if True:
    print("This is a very big and long, even annoy-
ing \
command which takes way too much space \
and is irritating to handle. And now its full \
of holes.")
```

This looks pretty nice in the source code editor, but unfortunately it is full of ugly holes when printed:

```
>>>
This is a very big and long, even annoying ommand
which takes way too much space and is irritating to
handle. And now its full of holes.
```

```
>>>
```
The next line starts always without indentation. Because this may be somewhat misleading, this practice should be avoided at first. However, later when the indentation level starts to build up because of structures within structures, dividing commands to span several physical lines becomes somewhat unavoidable evil.

## *Predefined string*

Besides "\"-character, there is also another option on dividing one line of code to several physical lines, although it only works with strings. This is the so-called predefined string, which can be defined by using triple quotations:

```
"""This thing here is predefined"""
```
or optionally
```
'''predefined strings can be also made with citation marks.'''
```

Triple quotation marks can be used when defining long text such as help file, example code or usage instructions. Python interpreter understands the triple quotation so that everything after triple quotation is a part of the same string until another triple quotation. Unlike normally, this also includes line breaks, indentation and syntax characters:

```
>>> print("""There are several things that should be
checked when
buying a car:
-maintenance record
-rust spots
-possible needs for maintenance
Also one should:
-ensure that the car sale is legit
```

```
-insurance is ok
-the car "feels" normal on a test drive""")
```

This prints out things correctly:

```
There are several things that should be checked when
buying a car:
-maintenance record
-rust spots
-possible needs for maintenance
Also one should:
-ensure that the car sale is legit
-insurance is ok
-the car "feels" normal on a test drive
>>>
```

As observable, the interpreter does understand everything as a part of the string until new (""") is given. The predefined string also conserves everything, including manually made indentations and such.

## *Escape sequence for layout characters*

In the topic few pages ago a connecter character was introduced to allow dividing one line of code to several physical lines, and even earlier a character combination that produces a line break. However, these things raise a stupid concern; what about the situations where "\"-character or "\n"-combination is needed to print on screen?

```
>>> print("So are we supposed to see \
'\\'-this or what? What does this \ do?")
So are we supposed to see '\'-this or what? What
does this \ do?
>>> print("\n")
```

```
>>> print("\\n")
\n
>>>
```

The line connector character "\" has also other use besides allowing the user to divide source code to several lines, it also tells the interpreter to disregard the following layout-affecting function of the character combination, or bypass the next quotation mark and continue the string. Even if citation and quotations marks are interchangeable to allow using one or another on a string:

```
>>> print("Timmy O'Toole")
Timmy O'Toole
>>>
```

or possibly

```
print('He said "Dish is best served cold."')
He said "Dish is best served cold."
>>>
```

There are only so many occasions this can avoid, as either one or another has to be omitted. Its easier to stict to one character, the default is the quotation mark ("), bypass the offending characters with "\":

```
>>> print("Man cried \"One does not simply walk into
Berlin!\"")
Man cried "One does not simply walk into Berlin!"
>>>
```

Last thing to understand with the layout characters is that the interpreter sees them as a one character, and there can be as many of them as neccessary:

```
>>> print("\n\n\nLine breaks!\n\n\n")


Line breaks!


>>>
```

There are no restrictions on how many of these characters are put together. When combined with \t, which makes tabulator-spaced indentations, allows making pretty-looking printouts.

```
>>> print("\t1\t2\n\t3\t4\n\t5\t6")
        1       2
        3       4
        5       6
>>>
```

From this point onwards, these topics are discussed in the book Chapter 3.

## *Handling inputs*

## Reading a user input, input()

In the earlier versions of Python language, getting an input from user was done with two different functions, input() and raw_input(). In Python 3, this situation has been streamlined by making the old Python 2 input() obsolete, and renaming the raw_input() as a new input(). So unlike Python 2, in Python 3 every user input is read as a string, and has to be manually converted to other types, such as numbers.

Input quite similarly to the print command. The command input() takes only one parameter, which indicates the string printed before reading the user input. This string is usually used to tell the user what to do - "select one (1-5)", "enter VAT percentage (0-100):" etc. - and it has no other technical function. When user inputs something and presses enter, the input returns this value to the interpreter. If the command is paired to a variable with an assignment operator, the user-given value is stored to the variable as a value.

```
>>> givenvalue = input("Write something: ")
Write something: Hello Python
>>> print(givenvalue)
Hello Python
>>>
```

As demonstrated in this example, the parameter in input was printed when requesting an input from user. User wrote "Hello Python" and pressed enter. Because of the assignment, the interpreter saved the value to the variable "givenvalue".

## *Observations regarding input*

There are some things that are good to know regarding the input. First of all, when the program uses the input-command, it waits until the user presses enter. If user wrote something before enter, that is given to the input as a value. Other thing is that this value can be anything; the interpreter handles it as a string and does not care for what the user intended:

```
>>> height = input("Give your height in cm: ")
Give your height in cm: Monkeys?
>>> print(height)
Monkeys?
```

```
>>>
```

This means that making the program to take only usable inputs is something that programmer has to manually implement. However, lets take an example on taking an input and using it.

## *Example 2-4: Taking string as an input*

**Source code**

```
01  # -*- coding: cp1252 -*-
02
03 # creates an variable, and saves the user-written
04  # text as a value to the variable
05  nimi = input("What is your name?: ")
06
07  print("Hello",nimi+"!")
08  print("Nice to see you.")
```

**Output**

```
>>>
What is your name?: John
Hello John!
Nice to see you.
>>>
```

String is a useful tool in these cases, as it can take any value, regardless of what the final type is supposed to be. This includes integers, decimal numbers, words, full sentences and such, as they can always be presented as a group of characters. This is excellent stuff when usings data such as written text, but when the input is needed for calculation, it needs a type conversion before it can be used. This is discussed next.

## *Type conversions, str(), int() and float()*

As said earlier, input only takes data as strings. This is nice when working with text, but when using numeric values the input causes an error:

```
>>> value = input("Give some number: ")
Give some number: 313
>>> 42 + value
Traceback (most recent call last):
  File "", line 1, in
    42 + value
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
>>>
```

The interpreter uses the input as a string even if the given input consisted only numbers. Because the interpreter is insistent that the input is a string of characters, it cannot use it in a calculation as is, even though that probably was what the programmer wanted. To fix this error, a type conversion is needed.

Type conversion functions are functions which changes the type of the given input. For example, if there is a string which only has numbers, like this:

```
>>> value = "42"
>>> print(value)
42
>>>
```

it superficially looks like a number, but when used in a calculation, causes some severe problems:

```
>>> value + 10
Traceback (most recent call last):
  File "", line 1, in
```

```
    value + 10
TypeError: Can't convert 'int' object to str implic-
itly
>>> value * 5
'4242424242'
>>>
```

This happens because the contents of the value is not fourtytwo for the interpreter, but rather a string with characters four and two, "four"-"two". This causes the addition to fail, and multiplication to simply repeat the numbers.

To get rid of the problem, a type conversion command is used to manually set the variable type. Type conversion functions only take one parameter, a variable, which is then converted to another type. In the example above, the number starts to work when type converter int() is used:

```
>>> value = int(value)
>>> value + 10
52
>>> value * 5
210
>>>
```

The value starts to work, as now the interpreter understand that the value is an integer. The type conversion functions all work the same way, they are given one variable and they return the value as the new data type. It should be noted, that the new value has to be saved to a variable with assignment operator in order for the conversion to work; just using the conversion command does not save the changes:

```
>>> number = "300"
>>> int(number)
```

```
300
>>> number + 100
Traceback (most recent call last):
  File "", line 1, in
    number + 100
TypeError: Can't convert 'int' object to str implic-
itly
>>> number = int(number)
>>> number + 100
400
>>>
```

## Other type conversion functions

The type conversion function int() returns an integer value of the given number. However, the int() is not the only type conversion function, there actually exists one for every basic datatype, including some of the rarer datatypes, such as complex numbers. The conversion functions for the most common types are listed in this Table:

| Name | Explanation |
|---|---|
| int(variable) | Returns the variable as an integer, causes ValueError if the conversion is not possible. Does not round the number correctly, just drops the decimal part. |
| str(variable) | Returns the variable contents as a string. |
| float(variable) | Returns the variable as an floating point number (decimal), causes ValueError if the conversion is not possible. |

Table 2.3 The most common type conversion functions

Type conversion functions are somewhat vulnerable, as the conversion cannot "guess" values in case the conversion is not

straightforward. For example, if the system tries to convert letters to a number value, the result is ValueError:

```
>>> int("Whatnow?")
Traceback (most recent call last):
File "", line 1, in
int("Whatnow?")
ValueError: invalid literal for int() with base 10:
'Whatnow?'
>>>
```

In addition, unlike in most calculators and for example in Excel, the decimal point is period, not comma. Also, when converting decimal numbers to integers, the type conversion does not round the number properly, it only cuts away the decimal part:

```
>>> number = 4.9995
>>> int(number)
4
>>>
```

## *Editing strings*

Unlike most programming langauges which allow low-level programming, Python has two very powerful tools for editing strings. The first of these is the string slicing, which allows programmer to easily pick parts of the string for further use, and the second, string methods, allow easy access to activities which are useful in modifying and testing the strings.

Things discussed here are also discussed in the book chapter 7.

## String slicing

Slicing a string in practice means cutting a part out of the original string. This means that it is possible to take only one word, or a few characters from the string, or cut out the last character, or rearrange the string backwards. To understand what slicing is, an example string is needed:

```
>>> bigstring = "auxiliaryemergencyfireprevention-
system"
>>> len(bigstring)
38
>>>
```

Notice the function len(), it returns the length of a string as an integer. However, it should be noticed that while len returns the actual length of the string, the last character is always in place length - 1. More on this below.

Slicing also works with dynamic structures such as list to manage a group of items, so this technique should be learned and practiced well.

### *Referencing to one character*

The function len returned the length of the example string, and said that it was 38 characters. To reference to one character in a string, a slicing can be defined as follows:

```
[string name][place of the character]
```

Meaning that the name of the string is immediately followed by square brackets, where the place number of the wanted character is given. However, it should be noted that the first character is

on the place [0], second in [1] and the last in the place [length-1]:

```
>>> bigstring[0]
'a'
>>> bigstring[1]
'u'
>>> bigstring[37]
'm'
>>> bigstring[38]
Traceback (most recent call last):
  File "", line 1, in
    bigstring[38]
IndexError: string index out of range
>>>
```

In fact, slice reference stringname{length] causes an IndexError. This is a really common type of error, in fact probably one of the most common programming errors the beginning Python programmers keep making, especially when using iteration. Anyway, the slicing also works with negative numbers. The numbers are counted from the last character, last being at place [-1]. Referencing to place [-0] returns the first character (as -0 for interpreter is same as 0):

```
>>> bigstring[-1]
'm'
>>> bigstring[-2]
'e'
>>> bigstring[-0]
'a'
>>>
```

## *Slicing several characters at once*

Slicing one character from string is rather straightforward. The usefulness of the slicing mechanics becomes apparent when the slice is expanded to cover more than only one character. The easiest approach is to use two values divided by a colon inside the square brackets, like this:

```
[stringname][startingpoint:endingpoint]
```

The startingpoint obviously tells the first character which is included in the slice. The ending point tells the place where the slice is cut; if the intention is to get the tenth character into the slice, the ending point is 10, as the place numbering starts from 0.

```
>>> bigstring[0:15]
'auxiliaryemerge'
>>> bigstring[3:19]
'iliaryemergencyf'
>>>
```

Slice can start from any point in the string, and end at any point. Its also possible to combine positive and negative numbers in the slice definition:

```
>>> bigstring[20:-1]
'repreventionsyste'
>>> bigstring[-30:-10]
'yemergencyfirepreven'
>>> bigstring[6:-10]
'aryemergencyfirepreven'
>>> bigstring[8:-10]
'yemergencyfirepreven'
>>>
```

It is also possible to define a third number in the slice. This number defines the step size taken between characters. Then the syntax is like this:

```
[stringname][startingpoint:endingpoint:stepsize]
```

Changing the step size to 2 only shows the every second character, step size 3 every third.

```
>>> bigstring[0:37:2]
'axlaymrecfrpeetosse'
>>> bigstring[0:37:3]
'aiaernfeenoye'
>>>
```

A special feature in the step length is the step size -1. This is an easy way to turn the string around:

```
>>> bigstring[38:0:-1]
'metsysnoitneverperifycnegremeyrailixu'
>>>
```

its also possible to use varibles to define the size of the slice:

```
>>> bigstring[number1:number2]
'mergencyfi'
>>> bigstring[1:number2]
'uxiliaryemergencyfi'
>>>
```

Its also possible to use default values when using the slices. If the start point is not given, the interpreter sets it to the first character. Similarly if the ending point is not given, the interpreter uses the last character. The default for step is 1. To apply default values, it simply is left away from the slice:

```
>>> bigstring[10:]
```

```
'mergencyfirepreventionsystem'
>>> bigstring[:15]
'auxiliaryemerge'
>>> bigstring[:30:2]
'axlaymrecfrpeet'
>>> bigstring[10::2]
'mrecfrpeetosse'
>>> bigstring[::-1]
'metsysnoitneverperifycnegremeyrailixua'
>>>
```

The slice [::-1] is especially nifty feature. It turns the entire string around, showing also the first character, which was missing from the slice [38:0:-1].

## *Observations on string slices*

As said earlier, the error IndexError is pretty common with the slices. This happens, when the slice references to a character place beyond the string size:

```
>>> bigstring[313]
Traceback (most recent call last):
  File "", line 1, in
    bigstring[313]
IndexError: string index out of range
>>>
```

However, there is one expection to the rule: if the slice is defined so that it has start and end points, this error is deprecated, even if both of the points are outside string size. In these cases, the interpreter returns the part that was in the defined area, or if the slice was completely outside string, an empty string "":

```
>>> bigstring[313:939]
```

```
''
>>> bigstring[30:101243]
'onsystem'
>>>
```

The slicing should be practiced well, as it will be used also with the dynamic structures such as list. Here's an example with several different types of slices:

## *Example 2-5: Collection of different string slices*

**Source code**

```
# -*- coding: cp1252 -*-

bigstring = "Damn the torpedoes, full speed ahead!"
length = len(bigstring)
print("The       length       if       the       string
is",length,"characters.")


slice_1 = bigstring[:15]
slice_2 = bigstring[15:]
slice_3 = bigstring[::2]

slice_4 = bigstring[1]
slice_5 = bigstring[5:26]
slice_6 = bigstring[::-1]

slice_7 = bigstring[-10:]
slice_8 = bigstring[:-10]
slice_9 = bigstring[4:30:2]

print("slice_1: ",slice_1)
print("slice_2: ",slice_2)
print("slice_3: ",slice_3)
print("slice_4: ",slice_4)
print("slice_5: ",slice_5)
```

```
print("slice_6: ",slice_6)
print("slice_7: ",slice_7)
print("slice_8: ",slice_8)
print("slice_9: ",slice_9)
```

**Output**

```
>>>
The length if the string is 37 characters.
slice_1:  Damn the torped
slice_2:  oes, full speed ahead!
slice_3:  Dm h opde,fl pe ha!
slice_4:  a
slice_5:  the torpedoes, full s
slice_6:  !daeha deeps lluf ,seodeprot eht nmaD
slice_7:  eed ahead!
slice_8:  Damn the torpedoes, full sp
slice_9:   h opde,fl pe
>>>
```

## String methods

The string slices are not the only thing that make the strings a powerful tool. However, when combining the strings to the methods, the strings become very efficient way of manipulating data. In the previous examples the strings have been shown to do different things. However, how would the strings work, for example, when changing the letters from one string to capital letters (ABCD...)? Or how should the string be tested if it has other characters besides letters and numbers? These things are accomplished with string methods, which will be discussed next.

### *Tools for strings*

In the Python, there are several tool functions called methods, which are useful when editing strings. These methods are a type

of functions within string datatype, and they are called with syntax

```
[string or variable name].[methodname]()
```

for example, if changing the letters from a string to capital letters:

```
>>> text = "whatnow?"
>>> text.upper()
'WHATNOW?'
>>> "really?".upper()
'REALLY?'
>>>
```

or when testing if the string has only numbers (isdigit) or letters (isalpha), saving the test result into a variable:

```
>>> text = "whatnow?"
>>> result = text.isdigit()
>>> result
False
>>> text = "testing"
>>> result = text.isalpha()
>>> result
True
>>>
```

Methods work so that they return either the string in the requested format, or Boolean result False or True whether the test result is positive or negative. In the following table are the most common string methods:

| Name | Explanation | Other |
|---|---|---|
| string_a.startswith(string_b) | Returns True if the string_a starts with the given string_b. | The string_b can be anything, number or string as the result is matched based on characters. |

| Name | Explanation | Other |
|---|---|---|
| string_a.endswith(string_b) | Returns True if the string_a ends with the given string_b. | The string_b can be anything, number or string as the result is matched based on characters. |
| string_a.find(string_b) | Returns the place of the first character in string_a, from whereon the string_b first appears. | Returns a value -1 if the string_b cannot be found from string_a |
| string.isalnum() | Returns True, if all characters in the tested string are either letters or numbers. | abs123 returns True, Hep-122 returns False, because dash is neither letter nor number |
| string.isalpha() | Returns True, if all characters in the string are letters. | |
| string.isdigit() | Returns True, if all characters in the string are numbers. | |
| .lower() | Returns the string with all characters in lower-case letters. | |
| .upper() | Returns the string with all characters in upper-case letters. | |
| .replace(char1,char2) | Replaces all occurances of the char1 in the string with char2. | `>>>"hello".replace("l","z")`<br>`'hezzo'`<br>`>>>` |

*Table 2.4 Some of the most useful string methods*

Its worth remembering that this table is not conclusive, but only the most useful and common ones. These methods are useful later when creating arguments for conditionals or iterations, which are the next two main topics. A complete list of methods

can be found for example here: http://docs.python.org/py3k/library/index.html, following link "Build-in types"->"Sequence types..." from point "String methods" onwards.

# Chapter 3: Conditional structures (if-else)

In this chapter we discuss the conditional structures of the Python language. The conditional structures make it possible to choose the next source code segment to be executed, or give an ability to skip some unneccessary or irrelevant parts of the program. Unlike several other programming languages, Python actually has only one conditional structure, which is if-elif-else. This design decision has been reached because of the Python principles; to aim for simple syntax, and because this conditional structure can be made to fullfil the same needs than others, for example case-select in C, anything else would be redundant.

## *If-else*

So the Python language only has a one conditional structure and that is if-elif-else. Technically the sturcture operates around the selection argument, which decides on which of the selectable segments is ran. The structure allows any number of segments, which is Python are designated by using the indentation to separate them from each other.

However, even if there is only one structure, there are several ways of using it. The only mandatory part in the if-elif-else-structure (from here onwards if-structure) is the if-segment, meaning that a complete selection structure can be made only with this one segment, which is either executed or not, based on the selection argument. The conditional structure can be supplemented with an else-segment, which will be executed if the selection argument in if-segment is false. Additionally, unlimited amount of elif-structures can be added to the structure for additional possibilities for selection. In the following segments, all these options are explained.

## *Example 3-1: Conditional structure If and the indentation*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  color = input("What is your favorite color?: ")
04
05  #Lets define the selection argument and see if it was
"blue"
```

```
06    #Notice the new indentation level inside the
structure
07  if color == "Blue":
08       print("Blue is nice.")
09
10   print("This line follows the if-structure and is
always executed.")
```

## Output

```
>>>
What is your favorite color?: Blue
Blue is nice.
This line follows the if-structure and is always
executed.
>>>
What is your favorite color?: Red
This line follows the if-structure and is always
executed.
>>>
```

In this example the selection argument

```
color == "Blue"
```

and the application of indentation in Python are shown. The if-line itself is not indented, similarly as the other source code lines which create a new code segment.

The if-segment by itself does not make the structure very useful, even if it allows the program to skip some parts of the code. However, the bigger issue is that now the code only reacts to the possibilities it is programmed to recognize (like the input "Blue" above), but otherwise it does not do anything. The selection which is made when all of the selection arguments are not true

can be implemented with the segment else. The else-segment is always selected in a case where if is not:

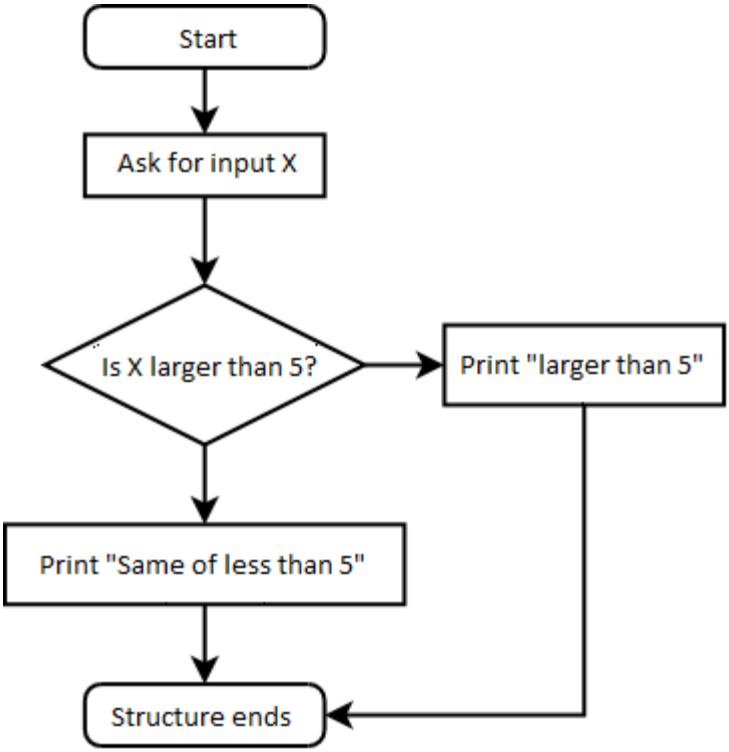



Figure 3.1: Conditional structure, which has if- and else-segments.

As illustrated in the Figure, the else is selected if the if-segment is not possible. In one conditional structure, only one else-segment can exist, and it is always the last segment defined. However, now is the perfect time to discuss the indentation in Python in more detail.

## Indentation

The indentation rules in Python are something which is somewhat uncommon feature in programming languages, and at the same time most profilic and the most despised feature of the language. The Python interpreter divides the source code to a group of logical segments based on the indentation level of the source code. For example, in code

```
01  if mode == "free":
02      print("Its free, go ahead.")
03      print("What are you waiting for?")
```

The second print-command in line 3 is executed only if the if-segment is executed, as the indentation sets it inside the if-segment. If this code is changed to this way

```
01  if mode == "free":
02      print("Its free, go ahead.")
03  print("What are you waiting for?")
```

the second print is executed always, as now it is outside the if-segment, in fact being the next logical command after the if-structure is resolved.

This is the reason why in Python the indentation is important and it should be edited carefully. By changing the indentation, the way the interpreter understands the code can alter radically. This is also the reason why it usually is a really bad idea to edit the Python source code with an editor, which either (A) does not understand how indentation works in Python or (B) does automatic indentation based on what looks nice. The additional trouble is that the tabulator key (left from 'Q' in the keyboard) and

space bar produce different kind of invisible, white space, characters. In Python, the tabulator should be handled as a group of four spaces, but some editors do not understand this. For now, the best idea is to stick with the IDLE editor, which is distributes along the interpreter package. As the tabulator and space are both invisible characters, it is impossible to tell if the editor actually differates between the two. This is the reason why the indentation should always be done with the space bar, not tabulator.

*Always use space bar to indent the source code in Python, unless you are 100% certain that your editor supports tab-indentations and handles them correctly.*

The other thing to remember is to always use the indentation levels of 4 spaces per level. Besides being the "Python standard", it also makes the source code much more understandable:

```
01  if word_1 == "Oolated":
02      if word_2 == "Squigg":
03          if word_3 == "Soup":
```

and

```
01  if word_1 == "Oolated":
02   if word_2 == "Squigg":
03    if word_3 == "Soup":
```

are technically exactly the same source codes. The only difference is in that the latter has only one space difference between the logical segments, making it virtually impossible to differentiate between two levels. Also, the indentation levels cannot be

started sporadically, only certain commands like the "if [selection argument]", which starts the conditional structure, can start a new level. For example, this code

```
01  print("This is on the first level.")
02      print("This line is not.")
03  print("The interpreter will never see me.")
```

causes an error on line 2, because print cannot start a new indentation level as it does not define a new code segment. However, these words of advice are not as critical as it may seem; in practice, with an editor that does understand Python syntax, the indentation problems are usually minimal.

When constructing conditional structure, it should be noted that there cannot be anything between the different segments of the structure. If-structure and else-structure should always be immediately connected to each other like this:

```
if [selection argument]:
    [if-segment code here]
    [if-segment code here]
else:
    [else-segment code here]
    [else-segment code here]
```

This presentation is a form of pseudo-code. It is meant to explain on a fundamental level what the code should look like, but without all of the additional details which may clutter the code and distract focus. Anyway, the code here works because if-segment and else-segment are logically next to each other (on a same indentation level and else follows if). If there would be something in between, like here

```
if [selection argument]:
    [if-segment code here]
    [if-segment code here]
variable = 100
print("Whatnow?")
else:
    [else-segment code here]
    [else-segment code here]
```

the program would cause an errror, because from the viewpoint of the interpreter, the else-segment is completely separate and in no way connected to the if-segment above it.

## *Example 3-2: Conditional pair If-Else*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  color = input("What is your favorite color?: ")
04
05
06  #Make note of the indentation in the structure
07  if color == "Blue":
08      print("Blue is also my personal favorite.")
09
10  else:
11      print(color,"is also nice.")
12
13  print("This print is after the if-structure and
is always printed.")
```

### Output

```
>>>
What is your favorite color?: Blue
Blue is also my personal favorite.
```

```
This print is after the if-structure and is always
printed.
>>>
What is your favorite color?: Red
Red is also nice.
This print is after the if-structure and is always
printed.
>>>
What is your favorite color?: Green
Green is also nice.
This print is after the if-structure and is always
printed.
>>>
```

This example is simple, yet it reacts to all inputs the user can come up with. If the user inputs "Blue", a specified response is used, and in all of the other cases, more generic else-segment is applied. This is all nice, but how about a case where there should be several options?

## If-elif-else

Its also very normal to be in a situation, where there should be several possibilities for conditional structure to select from. In Python, the if-else-selection can be supplemented with additional conditions by defining elif-segmentes to the structure. These elif-segments work like if-segment; they are given their own selection argument, which dictates whether the segment is used or not. The elif-segments are added as follows:

```
if [selection argument]:
    [if-segment code here]
    [if-segment code here]
elif [2. selection argument]:
    [elif-segment 1 code here]
```

```
        [elif-segment 1 code here]
elif [3. selection argument]:
    [elif-segment 2 code here]
        [elif-segment 2 code here]
else:
    [else-segment code here]
    [else-segment code here]
```

If the selection argument in the if-segment is not true, then the interpreter tests whether the 2. selection argument is true, and so on goes through all of the elif-structures, executing the first segment, on which the selection argument is true. If every selection argument fails, else-segment is executed.

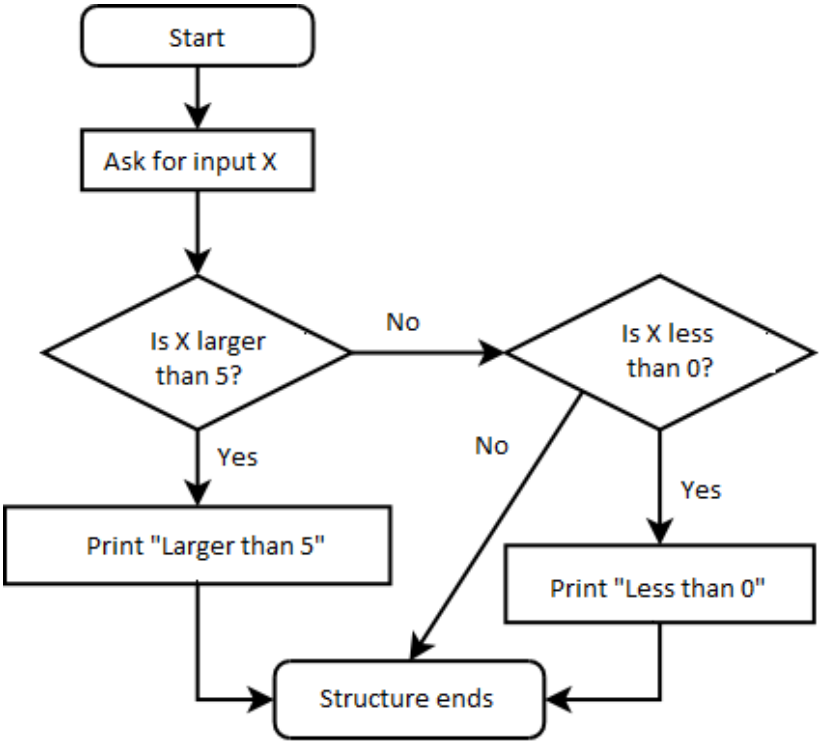


Image 1.2: If-structure, with added elif-segment.

In this conditional structure, there is one elif-segment in addition of if-segment. If the user input is 5, neither of the selection arguments is true, and nothing is executed. As else-segment is always optional, it has not been defined in this example, and therefore the code would do nothing.

## *Example 3-3: Conditional structure If-Elif-Else*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  color = input("What is your favorite color?: ")
04
05  if color == "Blue":
06      print("Blue is also my personal favorite.")
07
08  elif color == "Red":
09      print("Red looks good on a sports car.")
10
11  elif color == "Salmon":
12     print("Most would say that salmon is a fish,
but I guess it also counts as a color.")
13
14  else:
15      print(color,"is also nice.")
```

### Output

```
>>>
What is your favorite color?: Salmon
Most would say that salmon is a fish, but I guess it
also counts as a color.
>>>
```

## Using several parameters in the selection

Besides several options for outcome, it is rather normal to have conditions where the selection should be based on several arguments. Because of this, it is possible to combine several conditions to a one selection argument by using the keywords "not", "and" and "or". For example, lets take two variables named value_1 and value_2:

```
01  value_1 = 10
02  value_2 = "Engage"
```

Lets assume, that the if-strucuture can be only executed if both of the variables have the original value, 10 and "Engage". This can be required by combining two conditions:

```
03  if (value_1 == 10) and (value_2 == "Engage"):
04      print("Its alive!")
```

This would obviously print out text

```
Its alive!
```

The conditional keywords (and, or, not) can be used to "glue" together several arguments, which all are assessed before the decision is made. With the keyword "and", all of the given criteria has to be True in order to execute the code segment. If even one of the arguments is False, the segment is bypassed:

```
03  if (value_1 == 314) and (value_2 == "Engage"):
04      print("Its alive!")
05  else:
06      print("Nope, nothing happened.")
```

Would print text

```
Nope, nothing happened.
```

The keyword "or" allows more freedom. With or, it is sufficient that either of the arguments is True:

```
03  if (value_1 == 10) or (value_2 == "Run away!"):
04      print("Its alive!")
```

Would print

```
Its alive!
```

If-segment is executed even if the second argument is False, because the first argument was True. The third keyword, "not", turns the result around, requiring that some condition is not True:

```
03  if not (value_2 == "Run Away!"):
04      print("Its alive!")
```

prints result

```
Its alive!
```

Basically this works because the argument is False, and "not False" is True. Similarly, if the argument would be True, the "not True" would be False and therefore the selection argument would fail.

Obviously the amount of parameters is not restricted to two either. By combining the keywords "and" and "or" (with healthy dose of parenthesis to define in what order the arguments should

be tested) it is possible to define as many arguments as are needed:

```
03  value_3 = "Testing"
04
05  if (value_1 == 10) and (value_2 == "Engage") and
\
06   (calue_3 == "Testing"):
07       print("It works!")
08  else:
09       print("Nothing happened.")
```

Prints out text

```
It works
```

because all of the three arguments were True. In addition, it is possible to decide the order in which the arguments are tested by using the parenthesis "(" and ")" similarly as in calculations. The innermost parenthesis are always resolved first, and the arguments on the same "level" are resolved from left to right. Obviously, by applying the parenthesis the way the arguments is resolved changes, so it is important to pay attention on what exactly the interpreter is combining:

```
01  if (True == 1 and False == 1) or (False == 0):
02       print("It works!")
```

produces an output

```
It works!
```

In this example, the and fails because False is not 1 but 0. However, as the "or"-argument is resolved after the "and", the argument becomes False or (False == 0), meaning that the code is executed. In this scenario, the and-argument was designed poorly, as it has no relevance on whether or not the code was executed. This is the reason why it is really important to actually design the conditional structure well, as only the first segment with the selection argument having value True is executed.

## Logical expressions and Boolean values

The next topic in this chapter are the logical expressions and Boolean values. With conditional structures, the logical expressions are used to determine the segment, which will be executed. The logical expressions are simple; the operands - things which are for example compared - are combined to the operators, which define how the operands are compared. The result of the logical expression then results either True or False, which dictates whether the code segment is used. Basically, the logical expression can be like this:

```
>>> 5 > 3
True
>>>
```

The expression forms an argument that five is more than three. In this case, the operands are number three and five, while the operand is the larger than-operator. As five is obviously larger than three, the result is True. If the expression would be in fault

```
>>> 4 < 1
False
>>>
```

The result is correspondingly False. This basically defines the if-structure operations; if the selection argument is True, the segment is used, and if False, next selection argument is tested, else-segment is used or the structure is simply bypassed. The name of this True/False-result is also called a Boolean value, and both True and False are both reserved syntax words. In source code, these results are also interchangeable with 1 (for True) and 0 (for False):

```
>>> 1 == True
True
>>> 0 == False
True
>>>
```

However, in expressions and selection arguments it is better to use only True and False. Even if the arguments

```
>>> (5 * 100 < 1000) == 1
True
>>>
```

and

```
>>> (5 * 100 < 1000) == True
True
>>>
```

are technically identical, the latter is much more intuitive, as it clearly indicates that the code intends to test a calculation value, not define anything.

## Operators in Python

To construct a selection argument, the Python programming language has several different operators to enable testing a value, comparisons and much more. In the following tables, all of the

Python operands are explained with a short example on how they should be used.

| Operator | Explanation | Example |
|---|---|---|
| X or Y | Or-operator; if X or Y is True, the results is also True. | A = 1 and B = 0, A or B == 1 |
| X and Y | And-operator; If X and Y are both True, the results is also True. | A = 1, B = 0, A and B == 0 |
| not X | Not-operator; If X is True, the result is False. | A = 1, not A == 0 |
| X = 1 | Assignment operator; Sets the value of X to the given value. | X = 15; X == 15 is True |

*Table 3.2 Logical operators*

| Operator | Explanation | Example |
|---|---|---|
| X == Y | Equality operator; If X is same as Y, the result is True. | A == 0 and B == 1; A == B is False |
| X != Y | Inequality operator; If X is not Y, the result is True. | X = 1 and Y = 0; X != Y is True. |
| X < Y | Smaller than -operator: If X is less than Y, the result is False. | X = 1 and Y = 3; X < Y is True. |
| X <= Y | Smaller or same than-operator; If X is same or less than Y, the result is True. | X = 1 and Y = 1; X <= Y is True. |
| X > Y | Larger than: If X is larger than Y, the result is True. | X = 5 and Y = 3; X > Y is True. |
| X >= Y | Larger or same than; If X is the same or larger than Y, the result is True. | X = 2 and Y = 2; X >= Y is True. |
| X is Y | Object comparison; If X is same object than Y, the result is True. | X = "2" and Y = 2; X is Y is False. |
| X in Y | Item test; If X is equal or same as any item in the structure Y, the result is True. | X = 2 and Y = [1,2,3,4] X in Y is True. |

*Table 3.3 Comparison operators*

| Operator | Explanation | Example |
|---|---|---|
| X + Y | Addition; Returns a float point number if any of the numbers is a decimal, otherwise integer. | X = 2 and Y = 5; X + Y == 7 |
| X - Y | Substraction; Returns a float if any of the operands is a decimal number, otherwise integer. | X = 5 and Y = 2; X - Y == 3 |
| X * Y | Multiplication: Returns a float if any of the operands is a decimal number, otherwise integer. | X = 5 and Y = 2; X * Y == 10 |
| X / Y | Division; Always returns a float, even if both numbers are integers and the division has no remainder. | X = 6 and Y = 2; X / Y == 3.0 |
| X // Y | Quotient; Returns the times the Y goes to X. | X = 13 and Y = 5; X // Y == 2 |
| X % Y | Remainder; Returns the remainder from the division of X/Y. | X = 5 and Y = 2; X % Y == 1 |
| X ** Y | Exponentation; Returns a float if any of the operands is a decimal number, otherwise integer. | X = 4 and Y = 2; X ** Y == 16 |
| -X | Negative; Returns the negative equivalent of operand, same as multiplying with -1. | X = 10; - X == -10 |
| +X | Positive; does not actually do anything, equivalent of multiplying with 1. | X = 10, Y = -10; +X == 10 +Y == -10 |

*Table 3.4 Mathematical operators*

## *Source code execution order*

With operators, there are some concerns over the execution order of the actions. In introductory calculus, the first thing everyone learns is that the different mathematical actions take certain turns, like starting with the innermost brackets, multiplications

and divisions before additions etc. But does the Python interpreter understand this?

```
>>> 15 - 3 * 2
9
>>>
```

Surprisingly the interpreter had no problem with the given operators. Everyone knows that the multiplication has to be done before substraction, and so did the interpreter. But how does it do it?

This is possible because the interpreter has a priority list on all possible actions; calculations, slicing strings, function calls and such. Because the multiplications and divisions are above addition and substraction in the priority, the operators in the above example were resolved in the correct order. If the operations have the same priority, they are resolved from right to left.

| Command | Explanation |
|---|---|
| Slicing operator, [x:y:z] | Slicing a string or list |
| Selection of an item, listvariable[x] | Selecting item or character |
| Negativity: -value | Turns value to negative |
| Multiplication and division: *, /, //, % | All operators related to multiplication and division |
| Addition and substraction: +, - | Addition and substraction operators |
| Comparisons: <, <=, >, >=, !=, == | Comparison operators in a conditional argument |
| Operators is, is not | Operators is and is not in a conditional argument |
| Operators in, not in | Operators in and not in in a conditional argument |
| Operator not | Operator not in a conditional argument |
| Operator and | Operator and in in a conditional argument |
| Operator or | Operator or in a conditional argument |

*Table 3.5 Execution order of different programming commands, above are resolved before lower*

In the list above, most of the important commands and operators are listed. The mathematical calculations work because the prioritization is designed to support this. The actions above in the list are done before lower ones; if the prioritization is same between the operators, they are executed from right to left. All operations are listed and prioritized, but the complete list also includes stuff that is irrelevant to this course. Full list is available for example in http://docs.python.org/py3k/

This prioritization list is internally fixed for the interpreter and cannot be permanently changed. However, as with mathematics, the order can be temporarily changed with brackets:

```
>>> (5 + 6) * 10
110
>>>
```

The so-called "normal" brackets can be used to manually change the order the commands are executed. The interpreter always starts with the innermost brackets, and works from there outwards following the Table 3.5 execution order.

## Simple conditional structure

The last thing covered in this chapter is an ecxeption, or rather, addition to the selection structure syntax. Besides the normal conditional structure taught at the beginning of this chapter, there also exists other, "simple", structure for achieving the same functionality. The limitation of this simple if is that it only allows one command, but on the other hand it fits in its entirely to a one line of code:

if [selection argument]: [executed command]

For example

```
value = 1
if value == 1: result = True
```

This simple form is a fully functional conditional structure, and it can even take an else-segment:

```
value = 1
if value == 1: result = True
else: result = False
```
The simple form only allows one command, and it has to be separated from the selection argument with colon. If there is need for more than one command, the "full" structure is needed.

# Chapter 4: Iteration and Iterative Structures

In this chapter we discuss the iteration, where we can set the program to run a section of the code as many times as we need to, without the need to repeat the code all over again.

## *Iteration and Iterative structures*

In programming the iteration, repeating something again and again, is useful. In fact, without iteration there would be only so little that could be done: reading a file line by line, or asking a multitude of inputs from user would be rather ackward without the possibility to tell the computer to run some part of the code multiple times. With the iterative structures the creation of source code parts which are executed a multitude of times becomes a possibility.

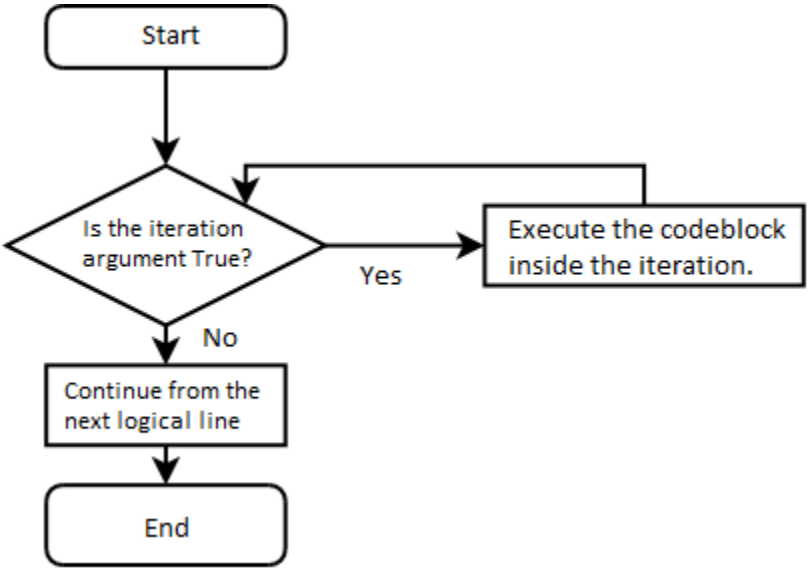



Figure 4.2: The principles behind the iterative structures

The main concept of iterative structure are the iteration argument which defines the requirements to end (or continue) the iteration, and the iteration segment, which has the code which will

be repeated as long as the iteration argument is true. The iteration argument is exactly the same as the selection argument; it consists of operands and operators, and based on the logical result (true of false) the iteration either takes another round or is terminated. In Python, as long as the iteration argument is True, the iteration starts another round, if False, the iteration immediately skips the iteration structure and continues from the next logical command in the source code.

Topics discussed in this chapter are also discussed in the book chapter 5.

## *While-iteration*

While-iteration is the Pythons' open, more simple iteration structure. The while-structure can be used either with a starting iteration argument or closing argument, with known or unknown amount of iterations, depending on the actual implementation. Basically, the while-structure is constructed as follows:

```
while [iteration argument]:
    [while-segment source code]
        [while-segment source code]
        [while-segment source code]
        [while-segment source code]
    ...
```

The most simple way of using while-structure is when it uses opening argument, and the amount of repetitions is calculated beforehand. In the next example the code asks the amount of repetitions from the user, and then calculates and prints the requested amount of lines.

## *Example 4-1: While-iteration*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  #Requests the amount of repetitions
04  totalrounds = int(input("How many rounds?: "))
05  nowround = 0
06
07  #define the iteration argument;
08  #when the amount of done rounds becomes
09  #higher than given total, the iteration ends.
10
11  while nowround < totalrounds:
12      print("Now is the round",nowround)
13
14      #The amount of done rounds adds 1
15      nowround += 1
```

### Printout

```
>>>
How many rounds?: 4
Now is the round 0
Now is the round 1
Now is the round 2
Now is the round 3
>>>
```

The important part of the last example was the last line of code within the iteration segment. This code calculates the amount of completed rounds in the code, and is crucial for the program to know when to move forward. In while-structure, there is no separate method for calculating the amount of rounds, it always has to be implemented by the programmer.

Please keep in mind that "round" is the name of the rounding function in the Python language. Do not use it as a calculator variable for the amount of completed rounds.

```
01  donerounds = 0
02
03  while donerounds < 6:
04      print("Iteration!")
05
06      if donerounds == 5:
07          print("Last lap!")
08
09      donerounds = donerounds + 1
```

Will produce a printout

```
>>>
Iteration!
Iteration!
Iteration!
Iteration!
Iteration!
Iteration!
Last lap!
>>>
```

This example also covers all of the basics. The iteration argument is now set to continue as long as the donerounds is less than six, meaning that the program will execute six iterations as the variable was set to 0 at the beginning of the program. Because the variable donerounds is added one on every round, it causes the iteration argument to become False at the sixth round. There also is an additional segment which prints the line "Last lap!" when the donerounds is 5. Because the iteration argument will become False at the beginning of the next iteration - donerounds

is no longer less than six - that will be the last line which will be printed.

## *For-iteration*

For is the second iteration structure in Python. For has more definite characteristics than while, and it is used for more profilic activities than while. For-iteration always operates with an opening argument, and the amount of turns has to be calculatable beforehand. For this reason, the for-iteration is mostly used for manipulating lists or other dynamic structures, where the amount of items present the amount of executed loops. Other usual way of using for is by combining the turn calculator with the function range(). The "turn indicator" in the abstract syntax below accepts only a dynamic structure, like list, and the amount of repetitions is calculated from the length (i.e. the amount of items) of the said structure. Other way is to use range() to create a fixed-item list, which is only used as a turn indicator. Anyway, the for-structure works as follows:

```
for [variable] in [turn indicator]:
    [for-segment]
    [for-segment]
    ...
```

The for structure itself is rather simple. When the for-structure is presented a variable, which is used as a kind of turn calculator and a sequence, the code within the for-segment is run the number of times. The for-structure may seem somewhat difficult, but in fact it is not that difficult to use. Lets exam this with the following example:

## *Example 4-2: Number nultiplier with For*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  #Ask the amount of iterations from the user
04   series = int(input("How many rounds are calcu-
lated?: "))
05
06  result = int(1)
07  for turn in range(1,series+1):
08      result = result*turn
09     print("At turn",turn,"the result is",str(re-
sult)+".")
10
11  print("The final result is", result)
```

### Output

```
>>>
How many rounds are calculated?: 5
At turn 1 the result is 1.
At turn 2 the result is 2.
At turn 3 the result is 6.
At turn 4 the result is 24.
At turn 5 the result is 120.
The final result is 120
>>>
```

As observable from the example, the for-structure does not need a separate system to ensure that the amount of rounds is calculated. For-structure has a round calculation variable, which in this example is named turn, which can be used as a normal var-

iable in the for-segment. Obviously this variable - and the structure indicating the amount of turns - can be altered to make an indefinate repetition, but the for should not be used this way. If the amount of turns cannot be expressed definitely, either by providing the manipulated data as a dynamic structure, or by calculating the length with range(), while should be used.

When compared to while, the for easily seems like a complicated way to achieve the same functionality as with the while. This is not strictly true, there are uses for for-structure, as it automatically goes through the dynamic structures and does not need separate calculation method. One pretty useful tool for the for-iteration is range(), which will be talked next.

## range()

As mentioned earlier, the amount of iterations with for-structure is defined as a length of the list or other structur which is given as the iteration indicator. To create these lists easily and effectively, Python has the command range(), which is helpful in creating these types of lists. Range is used to that it is given the first position and last position, and the step length, similarly as in slicing:

range(first,last, step)

Where the first and step are additional parameters. If range is given only one number, it assigns it to the "last", if two, to "first and last" and finally with three, to their according places. When used with for-structure, the idea becomes more apparent:

```
>>> for i in range(5):
```

```
print(i)

0
1
2
3
4
>>>
```

Or similarly

```
>>> for i in range(104,108):
print(i)

104
105
106
107
>>>
```

and when using a separate variable as a parameter for range:

```
>>> number = 3
>>> for i in range(number):
print(i)

0
1
2
>>>
```

It is worth noticing, that range never returns an error. Even if the last place is before the first, at least an empty list is returned. Also, as demonstrated in the examples above, for-structure always has a turn indicator variable, which either saves the value of the item currently used, or if using range-generated list, the

turn number, starting with 0. The usual name for this turn indicator is "i", followed by "j" and "k" if nested loops are needed. There is no restriction on the name, but this naming convention has become somewhat a tradition amongst the programmingers.

## *Manipulating the iteration*

With iteration, it is common occurance to not exactly know on which turn the repetition is supposed to end. In these cases, it is worthwhile the consider building the program so, that the iteration argument can be controlled from within the repeated segement. This method is called infinite loop. Unlike the error state where loop is executed infinetily by mistake, in this case the name comes from the fact that the loop ends only when the programmer intends it to do so, continuing infinately until ending criteria is achieved.

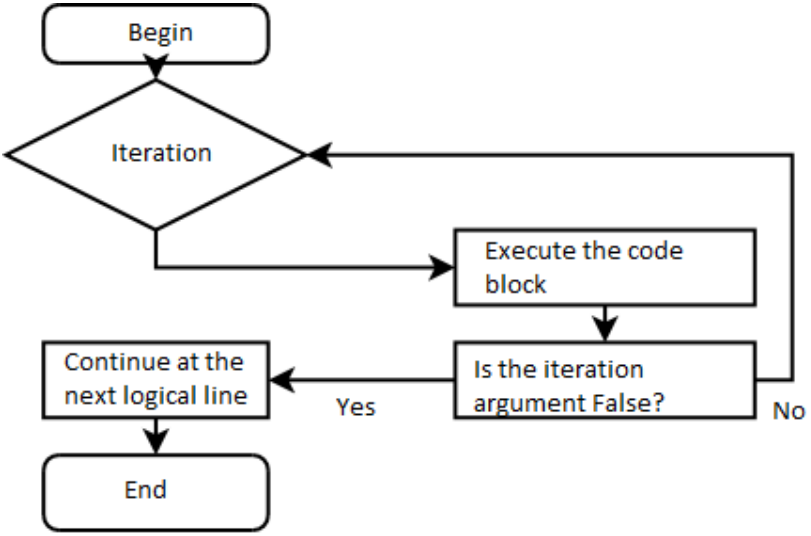


Figure 1.2: Infinite loop with separate ending criteria.

Especially the while-structure supports this approach. With while, the structure needs an iteration argument, which decides whether the loop continues to the next round. New round is started when the argument is True, and structure is terminated when the argument is False. However, the iteration argument does not have to be a expression, function call or operation, it can simply be a variable, holding value of True or False. When the variable is True, the iteration keeps going, and when it is turned False, the execution ends after the ongoing round:

## *Example 4-3: Infinite loop with ending criteria*

### Source code

```
01  keepgoing = True
02
03  while keepgoing:
04      userwrote = input("Write something: ")
05
06      if userwrote == "End":
07          keepgoing = False
08      else:
09          print(userwrote)
```

### Printout

```
>>>
Write something: Test?
Test?
Write something: Echo!
Echo!
Write something: End
>>>
```

The example here is very simple example of the infinite loop structure. It has no way of calculating how many times the loop will be executed, but for the entire duration of the program, everything is under control and the program can be terminated by the user at will. When the user gives a correct input, the variable that keeps the while-structure going is changed to False, and the program ends normally. However, the infinite loop can also get out of control, for example code

```
01  rounds = 0
02  keepgoing = True
03  while keepgoing:
04      print("Iteration!")
05      if rounds == 100:
06          keepgoing = False
```

gets to the infinite loop as the variable rounds is never updated, so the ending criteria is never reached. To get the program to terminate normally, it needs an addition, code segment to calculate the current round:

```
01  rounds = 0
02  keepgoing = True
03  while keepgoing:
04      print("Iteration!")
05      if rounds == 100:
06          keepgoing = False
07      else:
08          rounds = rounds + 1
```

This fixes the problem, as the variable rounds is always added by one in every turn. This eventually leads to the situation where the variable keepgoing gets a value False, and the iteration, and subsequently the entire program, terminates successfully.

If the program gets stuck on a run-away infinite loop, a keyboard shortcut Ctrl-C will kill the execution of the current program. However, it should be noted that runaway-loops may cause a huge memory load or take a huge chunk of computing time from the processor, so the computer may be a bit sluggish for a while. Everything returns back to normal after the automated memory management flushes the memory allocation tables; this usually takes a minute or two.

## Ending iteration, Break

The iteration structures can be terminated at will by using the trick of putting a single variable to the iteration argument. However, there actually exists even more direct way of accomplishing the same result, and that is the first of the iteration control commands, break. This command takes no parameters or arguments, and as its only function is to terminate iterations, it works only inside iteration structure. In fact, using it outside iteration causes a SyntaxError:

```
>>>break
SyntaxError: 'break' outside loop (, line 1)
>>>
```

The control command break works so that when it is encountered within the loop, the innermost loop is immediately terminated, and the next executed line of code is the first logical line after that structure.

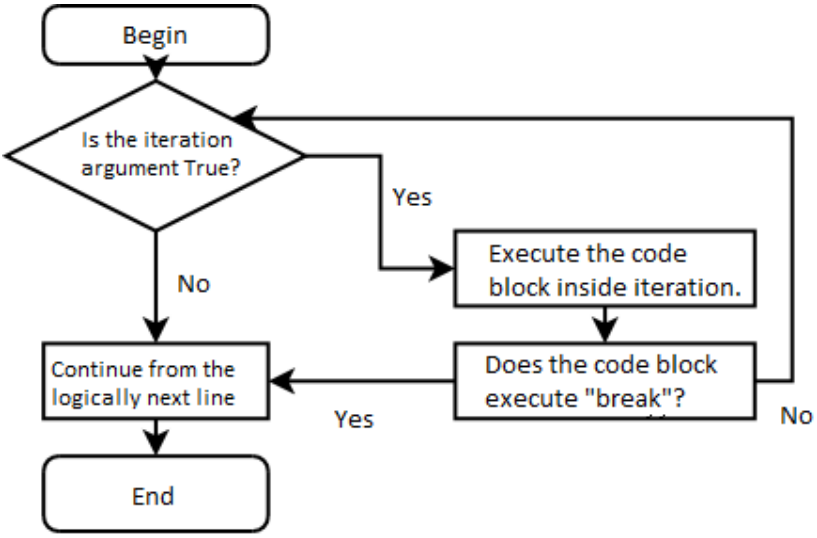



Figure 4.3: Break-command in iteration

As illustrated in the Figure above, the break-command does not give a damn about the actual iteration argument or anything else. If the break-command is encountered, the iteration is terminated. If there are several nested iterations, loop within a loop, the innermost loop is ended.

## *Example 4-4: Break-command in action*

### Source code

```
startpoint = int(input("Give the starting point: "))

while True:
    if startpoint % 13 == 0:
        print("We found a number divisible by 13!")
        break
```

```
    else:
        print("Currently we are at",startpoint)
        startpoint += 1
```

**Output**

```
>>>
Give the starting point: 8
Currently we are at 8
Currently we are at 9
Currently we are at 10
Currently we are at 11
Currently we are at 12
We found a number divisible by 13!
>>>
...
>>>
Give the starting point: 24
Currently we are at 24
Currently we are at 25
We found a number divisible by 13!
>>>
```

The code is not very complex. The structure keeps going onwards from the user-given number one by one until it finds a number which is divisible by 13 (number % 13 == 0). When the number is encountered, the program executes the if-segment and encounters break-command, terminating immediately. Otherwise the else-segment is executed, and the program continues to another round. Also pay attention to the iteration argument, which is hardcoded to be "True". That means that the only way out of this iteration is to get to the break-command, as there is no variable to control the iteration argument.

## Skipping turn , Continue

Like the previous command break, the continue is also one of the iteration control commands. The idea of continue is that it skips the rest of the iteration before starting a new round. Also like break, continue cannot exist outside iterative loop:

```
>>>continue
SyntaxError: 'continue' not properly in loop (, line
1)
>>>
```

As said, the continue is used to skip the rest of the turn. When program encounters continue-command within the iteration, it skips rest of the turn and starts next turn from the beginning of the iteration.

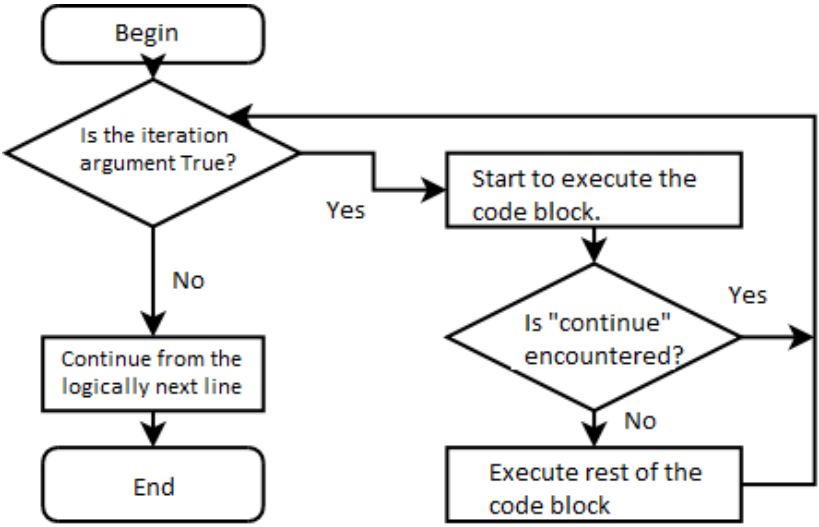



Figure 4.2: The principle of continue-command

The main functionality of continue is in speeding up the iteration. If the programming is searching for something particular, it can skip rest of the loop if the first conditional structure is not true. In some cases this speeds up the process considerably by cutting away excess tests and computing cycles.

## *Example 4-5: Continue in an iteration*

### Source code

```
total = 0
i = 0
rounds = int(input("How many rounds?: "))

while i < rounds:
    i += 1
    #If the i is not round number, skip it
    if i % 2 != 0:
        continue

    print("Added ",i,".", sep="")
    total = total + i

print("The sum was ",total,".", sep="")
```

### Printout

```
>>>
>>>
How many rounds?: 20
Added 2.
Added 4.
Added 6.
Added 8.
Added 10.
Added 12.
```

```
Added 14.
Added 16.
Added 18.
Added 20.
The sum was 110.
>>>
```

This example starts from 0 and continues to the number the user gives. If the current round is even, the round number is added to the total, otherwise just skipped. If the round number is odd, the first if-clause causes the interpreter to encounter continue, and the rest of the turn is skipped. Also pay attention to the calculation of i; it is in the beginning of the iteration to ensure that the number is always added up by one. If this command were after the continue, the program would end up in an infinite loop.

## Bypass segment, Pass

Unlike the other iteration control commands break and continue, pass is not exactly used to control anything, as it by design does not do anything. When the interpreter encounters a pass-command, it simply bypasses the structure completely, continuing from the next logical command.

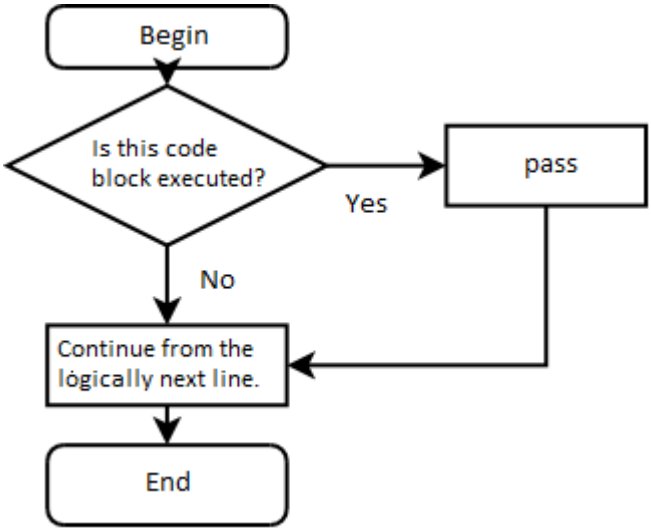



Figure 4.3: Pass-command as a placeholder for the actual activity.

However, the command pass is not merely a comedy option or useless trinket, it actually has a special use. The Python interpreter requires, that every command which begins a new logical block (new indentation level) must have at least one command within the block. For example, if there is a huge, complicated if-structure within the core system, and the if-structure happens to become outdated, it is somwtimes safer to just delete any functionalities from the executed block than start revising the entire structure. In this case, pass can serve as a standy or placeholder, so that the segment is technically still in, but it simply does not do anything:

```
if name == "Jeeves":
    pass
else:
```

```
    print("Now executing else-segment")
```

This works, because the if-segment still exists, although it does not do anything. In graphical user interface (GUI) definitions with Tkinter-module, the pass also has another function; it serves as a stub for action. With Tkinter, and also in general, if the program needs to have a function, but there is no functionality in it, a stub like

```
def superfunction(parameters):
    pass
```

is actually useful tool, as it allows the program to currently refer to a function, which at this state does not exist, but does not prevent the program from executing at the interpreter. This is especially important with Tkinter, as all named actions have to exist before the program is even able to start.

## Naturally ended iteration, else-segement

The basic idea of iteration and loops in general is to do serial tasks like do calculations, print lists, or search through data. To enable more control over the resulting outcome, the iterations also a bit surprisingly allow using else-segment in the structure.

In iteration, the else is defined as the last code segment, in the same "level" as the while (or for)-command, just like with conditional structures. However, unlike conditionals, the else-segment is not automatically executed when the iteration is skipped. With iterations, the execution of else-segment is tied to the break-command; if the iteration is ended with the control command break, the else is not executed. If the iteration expires "naturally" by ending with the iteration argument going False, the

else is used. The logic behind this is in the application of break. If break is used, it can be assumed that the loop served its purpose as it was terminated prematurely. If the iteration ends with the iteration argument closing, the loop did not find what it was looking for, and the else-segment is used instead.

## *Example 4-6: Else in an iteration*

### Source code

```
# -*- coding: cp1252 -*-

start = int(input("Enter starting position: "))
end = int(input("Enter ending position: "))

options = range(start,end)

for i in options:
    if i == 42:
        print("We found 42!")
        break

#Notice that this else is connected to for, not if
else:
    print("Seems that there was no answer in
there.")
```

### Output

```
>>>
Enter starting position: 10
Enter ending position: 50
We found 42!
>>>
...
>>>
```

```
Enter starting position: 50
Enter ending position: 1024
Seems that there was no answer in there.
>>>
```

The example code looks for a number 42 from the range defined by the user. If the number is found, the loop terminates with the break, and else is not executed. If the loop ends "naturally" by running out of numbers, the program executes the else-segment connected to the for-segment, commenting that the number was not found.

# Chapter 5: Handling files

At some point we also want to expand our programs to start using external files or locations for data so that we do not need to manually insert everything every time. This chapter discusses about this topic.

## External files in Python

Over time, it is common that the size of the programs keeps rising. At some fairly early point it becomes apparent that the best way to save stuff like data or user preferences, is to have a nice external file to which read and write the data. Obviously the external file would also be an ideal way of inserting a large amount of data to the program, so it is rather easy to understand why file are a really popular tool even for a small program.
From here on, the programming concepts are also discussed in a book chapter 8.

## Using the files

The most natural way of learning how to use the external files is probably to learn how to read one. Unlike editing a text file in an editor such as Notepad or OpenOffice Writer, in programming the first thing to know is to wheter the intention is to read or write the file. This is important, as the first thing to do when using a file is the creation of a file handle, which requires the intended usage of a file as a parameter, while the name of the used file is obviously the other parameter. Lets assume that there is a file example.txt on the harddrive, and that it exists in the same directory as the source code file, having the following text:

```
First line!
Second line!
Last in line!
```

This file can be used by the Python program residing in the same directory, simply by giving a command

```
sourcefile = open("example.txt","r")
```

This command creates a new file handle, basically a variable with the methods of accessing the file it is attached to, in a reading mode, which is denoted by the second parameter "r". When using the command open, the first parameter should always be a string, which gives the full name of the file, which is going to be used, including the possible name extensions such as .txt, .doc or .py. The second parameter tells the mode, in which the file is opened to. In this example, the mode is read, which also is the default selection, so technically in this example it was unneccesary, but file mode nonetheless.

If the given file name does not match any file in the directory, the interpreter gives an IOError. This also is the reason, why file operations should usually be written inside an error handler, if the user can in any way affect the name of the file by for example, giving the name as an input.

## .read, .readline and .readlines

Now that the file is open, its contents can be read. This can be done with three different file handle methods:

- read(length) returns the entire contents of the file as a one string. There also is an optional parameter length, which takes an integer value and denotes the maximum amount of characters, which will be read from the file.
- readline() returns one line from the file. The interpreter considers the one line to be the part from the beginning of the file to the first line break (\n), from one line break to another, or from line break to the end of file.

- readlines() returns the entire contents of the file as a one list, where the lines are separated to list items following the same logic as in the command readline().

These methods are used in the same way as the string or list modification methods, with the syntax

```
readtext = filehandle.readingmethod()
```

Each of these reading methods always return something if they arent in the end of the file. If there is an empty string in the middle of the file, they return the line break "\n"; only if they are in the absolute end of file, they return empty string "". Obviously this is not a concern for reading the file with "readlines" or "read" without parameters, as they allways return the entire content of the file.

## *Example 5-1: Opening and reading a file*

### Source code

The file example.txt has the following contents:

```
First line!
Second line!
Last in line!
```

The actual source code file:

```
01  # -*- coding: cp1252 -*-
02
03  readfile = open("example.txt","r")
04  content = readfile.readlines()
05
```

```
06  print(content)
07  for i in content:
08      print(i)
09
10  readfile.close()
```

**Result**

```
>>>
['First line!\n', 'Second line!\n', 'Last in
line!\n']
First line!

Second line!

Last in line!

>>>
```

Noteworthy in this example is also the break line-character, which is read from the file. When this break line is combined with the automatic break line added by the print-command, the resulting output has unneccessary extra lines. However, for this unwanted problem is an easy remedy; the print-command takes a parameter end, which denotes the character which ends the printed line. Under normal circumstances this would be the extra line break "\n", but as it now causes problems, it can be set to null string by changing the print command to

```
print(variable, end = '')
```

Other possibility is to slice the last character from the read string with slicing command

```
readline = readline[:-1]
```

Obviously this solution requires the read string to actually end with line break. The slicing takes away the last character in any case, so this approach needs attention. Third thing to remember is that just like the input, all of the reading methods allways return character strings, even when the entire file is made out of numbers. Lets take another example, on which the line break issue is attended:

```
# -*- coding: UTF8 -*-


handle = open("example.txt","r")
filetext = handle.read()
print(filetext)
handle.close()
```

This prints the correct form:

```
First line!
Second line!
Last in line!
```

In this example the read-method was used. This method is rather straightforward and is useful tool when used correctly, but in practice the most useful reading method is the readline. In read, the problem is that it really can return everything in the file; accidentally reading a multi-gigabyte file can really cause nasty side-effects, when the interpreter tries to cram the entirety of a DVD image into the system memory at once.

## Handling and closing the files

The last thing to do when using any file is to close the file. This is done by using the file handles close-method. This method

does not take any parameters, it simply ends the connection and releases the file from the handle. When the file is no longer needed, the closing is done simply by giving the command

```
handle.close()
```

and the file is no longer open. One good way to remember to add error handling and file closing to the file operations is frapping the thing inside a function. This function open the file, returns the file contents and closes the file in one, simple function:

```
01  def readfile(name):
02      try:
03          readfile = open(name,'r')
04          content = readfile.read()
05          readfile.close()
06          return content
07      except IOError:
08          return False
```

This way the files are not accidentaly left open. However, if the file is accidentally left open, the interpreter takes care of the problem and notifies OS at some point that the file is no longer needed. However, this can take a while, and is especially irritating if the file prevents the OS from deleting an extra folder or such. That, and the fact that it is a common courtesy to close the used files, is a reason why files should always be closed after usage.

## *File pointer*

When talking about reading files, the file pointer is something that is worth mentioning. Unlike pointers in another languages,

this pointer does not have anything to do with memory management, but it is closely related to the reading action. The file pointer is a "bookmark", which tells the interpreter on where in the file the reading (or writing) action was taking a place. This pointer is the system that allows the program to read one line at a time without repetition; when one line is read, the file pointer is advanced to the beginning of a next line. Obviously, methods like read() or readlines() put the file pointer right at the end of the file, as the entire file is read at once. The place the file handler points can be manually altered:

- seek(place) -function moves the file handler to the parameter-defined place on the file. The parameter place is an integer value, indicating how many characters from the beginning of the file the pointer is placed.
- tell() tells the amount of characters from the beginning of file the file pointer has advanced so far.

## *Example 5-2: Moving the file pointer*

### Source code

```
01  readfile = open('example.txt','r')
02
03  content = readfile.readline()
04  location = readfile.tell()
05 print(content[:-1]+"; The pointer is now at",lo-
cation)
06
07  print("Return to character number 10:")
08  readfile.seek(10)
09  content = readfile.read()
10  print(content)
11
```

```
12  readfile.close()
```

**Results**

```
>>>
First line!; The pointer is now at 13
!
Second line!
Last in line!
>>>
```

## Different file modes

Reading and writing to the files does not differ that much in the Python. The first action is to declare on what will be done, the file is connected to a file handle, file manipulation is done by using the file methods and finally the file is closed. In Table 8.1 the different file modes are displayed. The last two of these modes, bit-mode reading and bit-mode writing, are discussed more in the latter part of this chapter.

| Mode | Definition |
|---|---|
| r | Read mode: if the file name does not match any known file, raises IOError. Handles the data as characters. |
| w | Write mode; If the file name does not match any known file, creates a new file. If it does, empties the file before writing. Handles data as characters. |
| a | Adding mode; If the file name does not match any known file, creates a new file. If it does, adds the new text to the end of the existing file. Handles data as characters. |

| Mode | Definition |
|---|---|
| rb | bit-read; if the file name does not match any known file, raises IOError. Handles the data as bit values. |
| wb | bit-write; If the file name does not match any known file, creates a new file. If it does, empties the file before writing. Handles data as bit values. |

Table 5.1 File handling modes

The only problem related to the reading files is reading the numerical values and using them. As the file reading methods only return character strings, lets take an example on how the numberical values are retrieved and used after reading:

## *Example 5-3: Using the read numerical value*

### Source code

```
01  # -*- coding: UTF8 -*-
02
03  number = 1024
04
05  readfile = open("numberfile.txt","w")
06  readfile.write( str(number) )
07  readfile.close()
08
09  readnumber = 0
10  readfile = open("numberfile.txt","r")
11  readnumber = int(readfile.readline())
12  readfile.close()
13
14  print("A number",readnumber," was read and converted
to a number:")
15  readnumber = readnumber *2
```

```
16  print(readnumber)
```

**Printout**

```
>>>
A number 1024 was read and converted to a number:
2048
>>>
```

Once again, the example itself is not that useful, but it has all the neccessary parts needed for using numbers with files. It also includes the first example of writing to a file, a topic which will be discussed next:

## *Writing to a file*

Reading a file is an optiomal method of loading a large amount of data to a program at once. However, it also naturally means that the answer or output would be ideal to store for a later use into a file. To make this possible, the Python language allows the creation of new files and writing data to an external file.

Eriting to a file is practically similar to the reading action. The file is opened to a file handle, albeit now with mode "w" or "a", data is written with a handle method, and finally the file is closed. As in file reading, in writing there are selections in the action. The first selection is wether the program should destroy the existing data in a file or not, as the file mode "w" (write) does create a handle, but also empties the file contents. The file mode "a" (append) continues the file by writing the stuff to the end of the existing file. Both modes also automatically create a new file, if the given name does not match to any existing file name. In this sense the writing operation is safer than reading; the file is always available, so at least IOError does not happen. Well, at least if the access rights are OK and there actually is enough storage space in the media where the file is written, but these are not a concern of an introductory course.

In any case, the actual file writing is done with a method .write(text), which only accepts character strings, not numerical values. With simple data types this restriction is trivial, as everything can be converted to a string with the conversion function str(), but with dynamic types this can become somewhat of a pain. With dynamic structures such as list, the items have to be "deconstructed" to a string. One way of doing this is saving all of the list items as a string, and denoting the cutting point with a

character combination, which is uncommon in normal usage, such as €$€ (euro-dollar-euro), like this:

```
01  def delist(mylist):
02      stringdata = ""
03      for i in range(0,len(mylist)):
04              stringdata = stringdata + "€$€" +
str(mylist[i])
05
06      return stringdata
```

The other option would be using the module pickle, but this is discussed later. As for now, lets take an example on writing to a file:

## *Example 5-4: Creating and writing to a file*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  myfile = open("writings.txt","w")
04
05  mytext = "First line!\nSecond line!\nLast in line!"
06  print(mytext)
07
08  myfile.write(mytext)
09  myfile.close()
```

### Printout

Tulkin ikkunaan Printoutttiin kirjoitettavaksi tarkoitettava teksti:

```
>>>
```

```
First line!
Second line!
Last in line!
>>>
```

The same text now exists within the file "writings.txt". This file is automatically created to the same folder as was the executed source code.

Writing to the file is rather straightforward. The filename is declared in the opening command, so it can be created if no such file exists. Also, as the example used the file mode "w", the file was emptied in the opening procedure. If the example would have used the mode "a", the possible existing content would have been kept, and the file pointer would have been set at the end of the file. This is demonstrated in the next example, where a few additional lines are written to the file.

## *Example 5-5: Appending existing file*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  myfile = open("writings.txt","a")
04
05  addedtext = "Added line!\n This goes to writings.txt!
\
06  which was created earlier.\n"
07
08  print(addedtext)
09
10  myfile.write(addedtext)
11  myfile.close()
```

**Printout**

Program prints the following text:

```
>>>
Added line!
This goes to writings.txt which was created earlier.
>>>
```

The additions are also present in the file writings.txt itself:

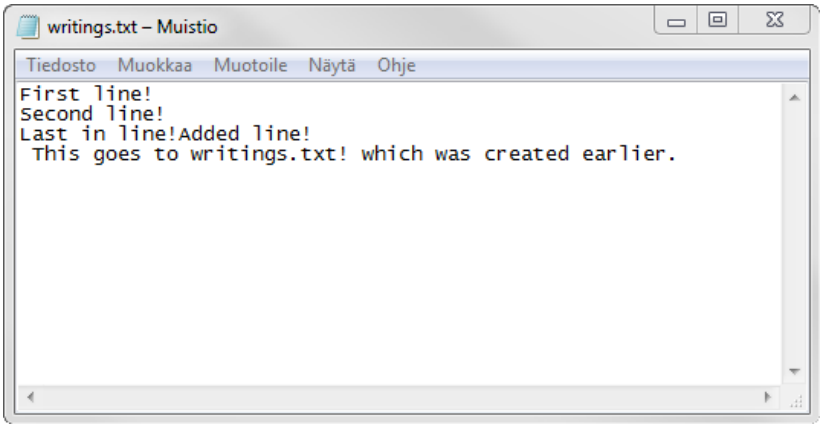



*Figure 5.1: Example file; the appending was successful as the earlier text is still present.*

## *Handling data in a bit-state*

In the previous data the files have been operated as characters. However, inside the computer the data is not on the disks as a characters (a, X, e, V, ?,$ etc...) but as numeric values, stored as bits in binary form. When discussing binary form, it means that

the numbers 1 (True) and 0 (False) represent different values which add up to represent different values:

| | |
|---|---|
| 01 | is 0*2 + 1*1 meaning 1 |
| 10 | is 1*2 + 0*1 meaning 2 |
| 11 | is 1*2 + 1*1 meaning 3 |
| 100 | is 1*4 + 0*2 + 0*1 meaning 4 |
| 110 | is 1*4 + 1*2 + 0*1 meaning 6 |
| 1011 | is 1*8 + 0*4 + 1*2 + 1*1 meaning 11 |
| 10011 | is 1*16 + 1*2 + 1*1 meaning 19 |

*Table 5.2 binary numbers*

The value is defined by summing the powers of two by adding those powers which are 1 and discarding those which are 0. Longer the binary number is, the larger it also gets. Generally the current computing technology applies eight bit bytes, meaning that the saved value is between 0 (00000000) and 255 (00000000).

*It is a common mistake for example in marketing to mix bits and bytes between each other. One bit is only one 0 or 1, able to represent only these two values. In byte, there are eight bits, like "11110000" or "00010011", being able to represent numbers between 0-255. Traditionally in computing, one character in stored on a disk was represented by one byte, as defined by the ASCII chart.*

## *ASCII-chart*

Basically the concept of hard drive is to store bits, which in binary numbers represent the different characters we currently use

in language. To create a some sort of concensus for how the bits represent different characters, an ASCII-chart was designed in the 60's to define a binary value which would correspond to different numbers, alphabets and syntax characters like comma or exclamation mark. For example, a binary value 01000001 is 65 in decimal value, meaning a character "A" based on the ASCII chart. Similarly, "B" would be 66 and "C" 67. This is also the reason why Python has hard time understanding the connection between lowercase and uppercase letters; uppercase letters from A-Z are values 65-90, whereas lowercase a-z are 97-122. Other special characters (#,?,%,& ...) also have a numeric value based on the ASCII. In fact, that numeric value is the method the Python uses as a backbone in sorting and comparison operations between characters.

For non-english-speaking world, the ASCII-chart has one huge weakness: it only covers purely English letters, not having any of the more uncommon letters from Scandinavian, German or Cyrillic alphabet. This also is the reason why the ASCII table was later revised with supplemental Extended ASCII charts, which were meant to be localized based on the user needs. This also is the fundamental reason why it was (and in many places still is) hard to get the special characters to show properly.

## *Selecting the code page*

In the extended ASCII chart there are 256 different additional characters, which technically take the places 256-512 in the character sheet and cover all the special alphabet characters like (å, ä, ö, æ) if it is the northern Europe code page, cyrillic characters if it is eastern Europe code page and for example greek

letters if Greek code page. This also is the reason why some Python codes start with the cryptic message like this:

```
# -*- coding: cp1252 -*-
```

This tells the interpreter, that the current code applies codepage 1252, which in Windows operating system means the extended ASCII-chart for western and northern Europe. This enables the system to correctly show characters like Å or Ö, which are common in for example Swedish and Finnish. This also is the cause for error characters which are sometimes seen; not all systems support the extended code pages, which leads to the different error characters.

There also is a fix for this problem, called UTF-8. This new codepage technology allows over eight million individual characters, allowing all different character sets from Greek, Cyrillic, Arabic, Japanese, Korean and Chinese alphabets. This is also meant to end once and for all, all of the problems with the different characters. The Python language already understands this codepage system, and it can be used by giving the code page declaration

```
# -*- coding: UTF8 -*-
```

Unfortunately, many operating systems still in use do not fully support UTF8, with older Windows systems being the most influential offenders. However, the newer Windows-systems do seem to be at least better with the problem, and hopefully the problem dies completely in the next ten to fifteen years.

# Chapter 6: Functions and workflow

A function, in a programming language context, means a small partial program, which does one activity within one program. They become very useful the second, when we have a program that we want to maintain and extend later.

## *Functions and subfunctions*

A function, in a programming language context, means a small partial program, which does one activity within one program. Usually one of the functions is also called the main function, which is responsible for starting, running and ending the program, and a group of subfunctions, which are specialized to do one action of the program.

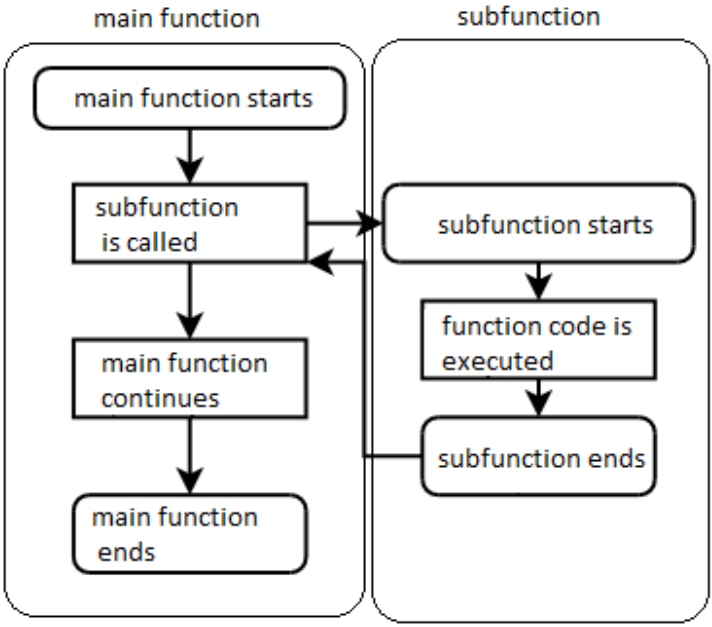



Figure 6.1: A program with a function, which calls a subfunction

As illustrated in the figure, the functions enable the program to be divided to smaller, logical segments. The main function directs the program and serves as a center which calls the different subfunctions and delievers the messages between them. For example a program which manages personal records in a database, there may be the main function which controls the entire program, a function for adding data, one function for searching the database, one for editing and so on. This also makes it easier to develop the program; features can be added later on by creating new functions and connecting them to the user interface.

## Basics of functions

In Python programming language a function is created by simply using the syntax word def and giving the function a name:

```
def converter():
```

creates a new function called converter. Similarly as iterative structures or conditions, this also starts a new logical block of code, meaning that all source code inside this new function is written to the lower indentation level. Its also worth noticing the parenthesis; these are mandatory and tell the interpreter the amount of parameters the function takes. Nothing in between the parenthesis obviously means that the function does not take parameters, it is simply called.

Inside the function is defined the code, which will be run when the function is called. The function can be called from any other part of the code, even from within the function itself (this is called recursion and techniques related to it are beyond the top-

ics of this course), so the function code can be used at any occasion. Obviously, if the program is doing something that occurs every now and then, its much more sensible to write it once inside a function, and call it when needed. Other option would be to manually copy the code to several locations several times, and if changes would be needed, fixed on every single occurrence once and again. Basically, the principle of function is something like with the iterations, but the function is not necessarily repeated several times at a time, but every now and then.

## *Example 6-1: Creating a function*

### Source code

```
01  # -*- coding: UTF8 -*-
02
03  #Lets make a new function
04  def hellofunction():
05      print("This print is from the function!")
06
07  print("This print is from main code!")
08
09  #Lets call the function
10  hellofunction()
11  print("Again, the main code butting in!")
12  #Lets call the function again.
13  hellofunction()
```

### Output

```
>>>
This print is from main code!
This print is from the function!
Again, the main code butting in!
This print is from the function!
>>>
```

As observable from the example, the interpreter jumps to the function every time it is called. The "main code" - that is, the code outside functions written on the source code - is put on hold for the duration of the function hellofunction to execute. Anyway, the function call always is like this:

```
[function name]( [parameters] )
```

For example in the last example

```
hellofunction()
```

As there were no parameters. If we want to have parameters, like in the type conversion functions, its not a problem, and is a simple addition to the function call. This will be discussed next.

## Parameters in functions

The ability to just create a function, which is called to perform some action every now and then is useful, but pretty soon it starts to become limited. It would be nice to be able to give the function some instructions, parameters, to tell what to do or give input to manipulate. Because of this, functions have been designed to be able to take parameters.

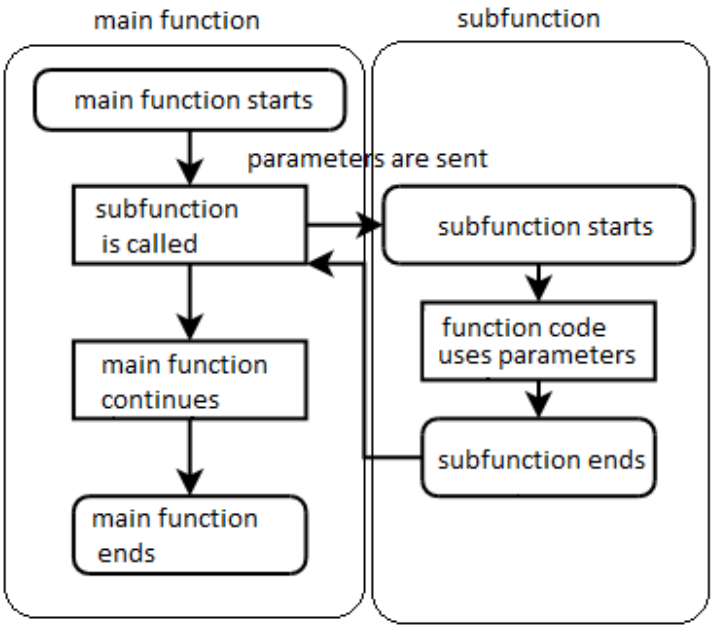



Figure 6.2: A function which accepts parameters

Using the subfunctions in code becomes much more convenient when the functions are able to take data as parameters. The amount of parameters is defined in the creation of the function simply by writing the names of the parameters inside the parenthesis in the function name, separated by comma:

```
def functionname(first, second, third, fourth, so_one)
```

This would define a function with five parameters. The defined parameters work inside the function as normal variables, and

there are no restrictions beyond practicality on how many parameters one function call can take. Lets take an actual example:

```
01  def Comparison(number_1, number_2):
02      """This function takes two integers."""
03
04      if number_1 == number_2:
05          print("The numbers are equal.")
06      elif number_1 > number_2:
07          print("The first number is larger.")
08      else:
09          print("The second number is larger.")
```

The example creates a function called Comparison, which takes two parameters, number_1 and number_2. Like with variables, there are no restrictions on what kind of data is given to the parameter; this is something that the function has to take into account. Anyway, parameters can be used inside the function just like any other variable. To help the users on how to use the function, the first line of the function is dedicated to a special case of using the """-string; this string is the documentation string of the function, and is the text what is shown when using the interpreter command help(). Anywhere else this separate string would cause an error, but if the string is first thing in the function, it is bypassed similarly as normal comments. Anyway, lets try the function by calling it:

```
>>> value1 = 10
>>> value2 = 12
>>> Comparison(value1,value2)
The second number is larger.
>>> Comparison(5,5)
The numbers are equal.
>>> Comparison(value1, 0)
The first number is larger.
>>>
```

The given parameter can be a fixed value like 5 or 10 or "testme", but more commonly it is a variable from the part of the program which was calling the function. The interpreter makes sure that in every function call every parameter is given at least some value, so that is something that the function does not have to worry about. In fact, if there is a parameter left without a suitable value, it causes an error:

```
>>> Comparison(100)
Traceback (most recent call last):
  File "", line 1, in
    Comparison(100)
TypeError: Comparison() takes exactly 2 arguments (1
given)
```

Here is also the help()-command to illustrate the document string in action:

```
>>> help(Comparison)
Help on function Comparison in module __main__:

Comparison(number_1, number_2)
    This function takes two integers.

>>>
```

## *Example 6-2: Parameters, main function and subfunction*

### Source code

```
# -*- coding: cp1252 -*-

#Lets define a subfunction
def printerfunction(word1,word2):
```

```
    print("We got parameters",word1,"and",word2)

#This is the main function
def main():
    string_1 = "Blues record"
    string_2 = "Artichoke"

    #Lets call the subfunction here
    printerfunction(string_1, string_2)

#This code tells the interpreter the name
#of the main function which starts the program.

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

**Printout**

```
>>>
We got parameters Blues record and Artichoke
>>>
```

This example shows for the first time the division of code to two main components; to the main function which starts and operates the program, and subfunction, which exists to provide a functionality. Inside the main function all of the "main level" code, which earlier was written on the root of the file without indentation, can be written. The program then starts with the if-structure as defined in the end of the example:

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

This tells the program that if the code is executed as a program, the program starts with the function main, and this structure calls that function.

*Unlike in some languages such as C, there actually is no restriction on what the name of the main function should be. However, using the name "main" for the main function is probably the clearest one, and definitely makes it easier for the non-Python-expert to understand the code.*

In any case, the example program is executed from the main function, which calls for the subfunction called printerfunction. This function takes two parameters, and uses them inside the function similarly as a normal variable. Like mentioned earlier, all of the parameters always get some value; the interpreter simply refuses to allow a function call that does not define all parameters in one way or another. Even if the other parameter was not in use, calling that function with only the used parameter causes an error:

```
def printerfunction(string1, string2):
    print("We got",string1)
```

called with only the first parameter defined would still cause an error:

```
>>> printerfunction("Lighthouse")
Traceback (most recent call last):
File "", line 1, in
printerfunction("Lighthouse")
TypeError: printerfunction() takes exactly 2 posi-
tional arguments (1 given)
>>>
```

Besides defining the parameters in call, they can be given default values, similarly as with the print-function parameters sep and end. This will be discussed later.

## Return value

Besides receiving data with parameters, the functions are also capable of returning data to the main function with a return value. Lets consider this example:

```
01  def change_price(value):
02      value = 250 + value
03
04  value = 100
05  price = change_price(value)
06  print(price)
```

This code has a definite problem. The idea is sound; to create a function which would change the content of a variable in the main function. However, currently the changes to not get recorded, in fact the results are surprisingly bad:

```
>>> value = 100
>>> price = change_price(value)
>>> print(price)
None
>>>
```

The result is not even the 100 which would have been assumed, but None, as the functino does not return anything. In fact, None is not even False, as None literally means absolutely nothing, whereas False is 0.

```
>>> None == False
False
>>>
```

To correct this problem, we need the return value, which would make the function to return a new price which was calculated based on the given value.

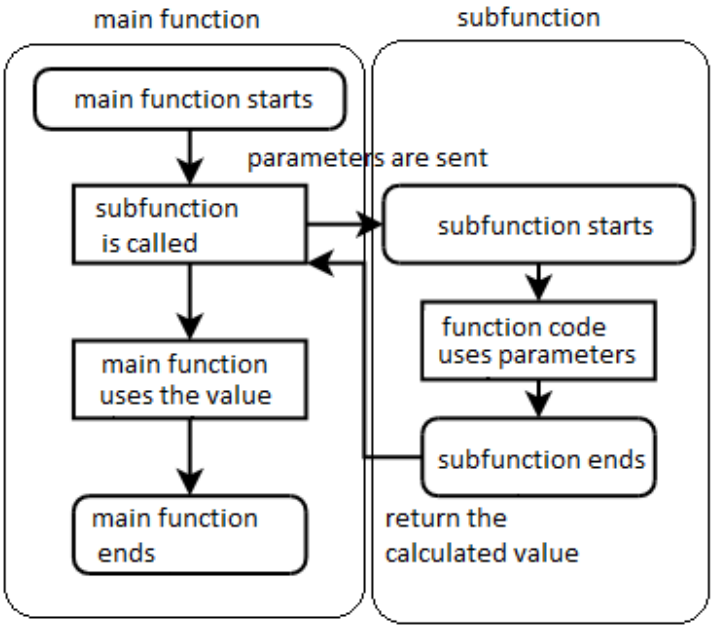



Figure 6.3: A subfunction with return value and parameters

Return value is defined by using the keyword "return" and defining the value which is returned. If the value is a fixed value like "True" or 0 the funvtion always returns this value, if it is a

variable from inside the function, then the contents of the variable is returned. In function, it is also possible to have several return clauses; even though the function always ends to the first encountered return, it is possible to have several return commands, for example inside different code segments in the if-structure. For example, if the last example is extended with a functional return value, it looks like this:

```
01  def change_price(value):
02      price = 250 + value
03      return price
```

Now the function actually works as inteded, as it is able to tell the return value to the main program:

```
>>> pricenow = 100
>>> pricenow = change_price(pricenow)
>>> print(pricenow)
350
>>>
```

Now the function change_price() returns the new value to the variable which was assigned to the function call, similarly as with the type conversion functions or string methods. Besides returning a value or result to a function, it can also work as a stopgag to prevent problems. If the interpreter encounters a return value, the function is immediately terminated. By making several return values it is possible to avoid errors, like in the following example on division by zero:

```
01  def divider(number1,number2):
02      if number2 == 0:
03          return False
04      else:
```

```
05          result = number1/number2
06          return result

>>> results = divider(100,0)
>>> print(results)
False
>>> results = divider(100,20)
>>> print(results)
5.0
```

Normally the return-value is used to pass an answer from a sub-function to the main function. However, it is not the only usage, in fact it is quite common to have several return values. For example, when testing the strings it is usual that the method returns either a value 1 (True) if the requested action was successful and 0 (False) if the action was unsuccessful. Its also common to return 0 (False) when the result was negative, like when looking for an item which does not exist, from a list. In the next example this concept is further demonstated by defining a string testing function:

```
01  def getlength(testme):
02      if len(testme) < 42:
03          return 0
04      else:
05          return 1
06
07    result  =  getlength("ohgodwhydoesthisstringh\
avetobesolongicanteverrememberthis")
08  if result == True:
09      print("This string is long enough.")
10  else:
11      print("This string is too short.")
```

Will produce a result

```
This string is long enough.
```

This obviously indicates that the string was longer than the 42 characters which was the given limit. Basically this example is a bit artificial, but it demonstrates the purpose of the return value very well. The subfunction takes care of one of the activities, and returns the result of the action to the main function. The main function still knows what is going on in the subfunction, but leaves everything to the separate function. Lets take one more example, this time with actual usage; a program that calculates travel expenses with car:

## *Example 6-3: Return value with subfunctions*

### Source code

```
# -*- coding: cp1252 -*-

def calculator(distance,gas,mpg):
    price = gas*(distance/mpg)

    price = int(price)
    return price

def main():
    gasprice = float(input("How much is one gallon
of gas?: "))
    tripdistance = int(input("How many miles will be
driven?: "))
    averagempg = float(input("How many mpg does the
car get?: "))

    total_sum      =      calculator(tripdistance,
gasprice,averagempg)
    print("The trip will cost",total_sum,"euroes.")
```

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

**Printout**

```
>>>
How much is one gallon of gas?: 1.5
How many miles will be driven?: 100
How many mpg does the car get?: 10
The trip will cost 15 euroes.
>>>
...
>>>
How much is one gallon of gas?: 2.2
How many miles will be driven?: 80
How many mpg does the car get?: 6.5
The trip will cost 27 euroes.
>>>
```

In this example the calculation itself is done in the subfunction, while the main function takes care of asking the values from the user, and prints the results after calculations.

## Default values in parameters

Earlier in this chapter it was demonstrated that the program raises an error if all of the parameters are not defined. This is rather cumbersome, as there really is no reason to always define all of the parameters, one such example being the sep and end in the print command. So how can this be achieved in programs? The answer are the default values of parameters.

As mentioned, the function parameters are a kind of variables, and they indeed work like variables inside their functions. This expression is actually really accurate, the default values of parameters are simply values assigned to the parameters in the function definition:

```
def printstuff(charline = "Defaults!"):
    print(charline)
```

when this function is called, it can be called without any given parameters as every parameter has a value:

```
>>> printstuff("Testing my function!")
Testing my function!
>>> printstuff()
Defaults!
>>>
```

This example works whether a parameter value is given or not, as in every case the value is either "Defaults!" or whatever the function call defined. In addition of default values in general, if there are several parameters with a default value, it is also possible to decide which parameters are given:

```
01  def printstuff(parameter1 = "First", parameter2
= "Last"):
02      print(parameter1+"----"+parameter2)
```

Now it is possible to define just some of the parameters, and in any order, by addressing which parameter gets which value:

```
>>> printstuff(parameter2 = "Stonewall")
First----Stonewall
>>> printstuff(parameter1 = "Moat")
Moat----Last
```

```
>>>
```

It is possible to define a parameter value with the syntax [parameter name] = [value], [parameter name] = [value] ... and so on. Obviously as with normal function calls, all of the parameters without a default value still need something or else the program fails. Anyway, with this assignment syntax the ordoer of parameters is irrelevant:

```
01    def printstuff(value = 1024, parameter1 =
"First", parameter2 = "Last"):
02        print(parameter1+"----"+parameter2)
03        print(value)

>>> printstuff(parameter2 = "Black", parameter1 =
"Red", value = 256)
Red----Black
256
>>>
```

Lets take one more example on the topic and then move on:

## *Example 6-4: Default values for parameters*

### Source code

```
# -*- coding: cp1252 -*-

def square(width= float(5.0), height = float(8.0)):
    area = width*height
    return area

def main():
    #Since we now have default values,
    #we can leave some of the parameters out.
    area1 = square()
```

```
    area2 = square(4.0,3.0)
    area3 = square(10.0)
    area4 = square(height = 11.0)

    print("Four different ways of calling our \
function...")
    print("And they all work:")
    print(area1,area2,area3,area4)

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

## Printout

```
>>>
Four different ways of calling our function...
And they all work:
40.0 12.0 80.0 55.0
>>>
```

## *Observations regarding the subfunctions in use*

## Visibility of the variables and global variables

So far it has been established that there are variables inside subfunctions and the main function. However, there has been no discussion over the visibility of these variables inside, outside and between the functions. In this case, the term we should discuss in more details are the namespaces, in other words the areas of source code where the different variable names are seen. Lets take an example:

```
01  def printer(stringstuff):
02      stringrow = stringstuff
03      print(stringrow)
04
05  stringrow = "Defined in the main level."
06  printer(stringrow)
```

Simply prints the line

```
Defined in the main level.
```

as there is only one declaration and assigment for a variable stringrow value, even though there is a variable stringrow both in the subfunction and the main level code. If the code is changed to the following:

```
01  def printer(stringstuff):
02      stringrow = "Defined in the subfunction."
03      print(stringrow)
04
05  stringrow = "Defined in the main level."
```

```
06  printer(stringrow)
```

now produces an answer

```
Defined in the subfunction.
```

as in this code the variable stringrow is redefined in the subfunction which does the printing. However, things get interesting in the next example:

```
01  def printer(stringstuff):
02      stringrow = "Defined in the subfunction."
03      print(stringrow)
04
05  stringrow = "Defined in the main level."
05  printer(stringrow)
06  print(stringrow)
```

which prints the following:

```
>>>
Defined in the subfunction.
Defined in the main level.
>>>
```

Even though the subfunction did redefine the variable stringrow in the function, the changes did not get saved! This is because the subfunctions exist in a different namespace as the main level code; they both have their own variable named stringrow, meaning that without proper return value system in place the changes in the subfunction do not get saved anywhere which would be visible to the code outside the function itself. This is somewhat inconvenient, as the interpreter does not give any indication on

which namespace is used, and they are not exactly easy to understand.

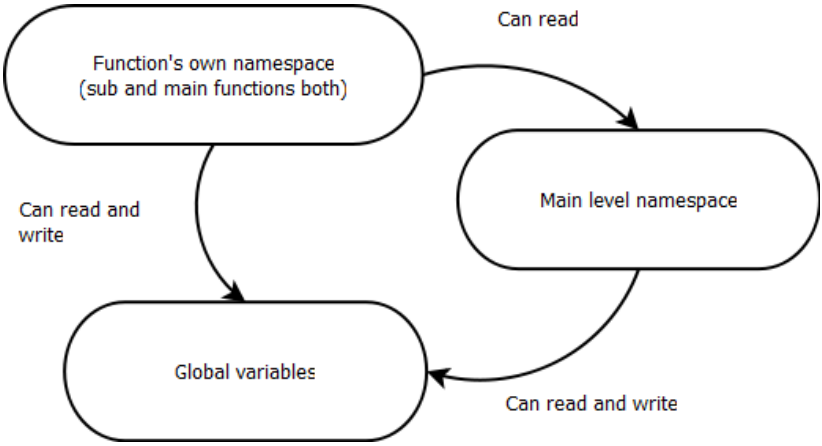



*Figure 6.3: Variable namespaces*

Other thing which surely adds to the confusion is the interpreters ability to see outside functions. Take for an example following code; Like the example earlier, this surely fails?

```
01  def print_price():
02      print(price)
03
04  price = 100
05  print_price()
```

However, this does not fail, and it actually works exactly as someone without any idea on namespaces would have assumed. So what happended here?

This feature is one of the interpreter features - and on a personal opinion of the author - the one of the few somewhat odd decisions on Python syntax. The upper code works, because the subfunction print_price() does not define a new variable price, nor does it take a parameter of the said name. In this special case, the interpreter is able to see the namespace of the main level code, and read the variable from the main level. However, this works only that way; the function can read the variable from the main level. The variable cannot be changed with the code inside function, only read. The functions are also not able to see variables from anywhere else besides the main level; not from the calling function, or main function if that is declared, only from the main level.

## *Example 6-5: Getting variable value from the main level*

### Source code

```
01  def printlabel():
02      if postbox == "30":
03          print("P.O. Box 30")
04
05  postbox = "30"
06  printlabel()
```

### Printout

```
>>>
P.O. Box 30
>>>
```

If there is some reason to be able to manipulate the variable values in the main level, it is possible to use another special case of

variables, a global variable. The application of global variables is usually a really bad idea in the long run, so they in general should be avoided. However, sometimes it may be the only bandage fix to get some throwaway code to work, so next we will look into this usually very negative attention collecting method.

## *Global variables*

The main idea in a global variable is that it can be accessed from anywhere in the program, from the main level, main function or any of the subfunctions to read and store data. In Python, a global variable is first defined by creating a variable into the main level:

```
commonvariable = ""
```

Although it is not customary in Python to define or introduce variables, with global variables this is needed as there needs to exist a variable to declare global. This is done by starting the function with a following command

```
global [variable name]
```

meaning that the interpreter is told that the variable using this name is not a function's own variable but a global variable, taken from the main level of the source code.

## *Example 6-6: Global variables*

**Source code**

```
01  def printlabels():
02      global postnumber
03      if postnumber == "00102":
04          print("Parliament house: 00102")
05      postnumber = "99999"
06
07  postnumber = "00102"
08  printlabels()
09  if postnumber == "99999":
10      print("North Pole:",postnumber)
```

**Printout**

```
>>>
Parliament house: 00102
North Pole: 99999
>>>
```

The problem with the global functions is in the details; the first impression on them is usually that they are really nifty and easy tool to send data from one function to another, but that also is the problem; when one function sets the variable to one position, it not only overwrites the data set by some other function, but also may affect the way the other functions which rely on the global variable work. Sooner or later this usually ends up in a synchronization problem; one function resets the value or overwrites it at the wrong occasion or does not declare the variable global and accidentally looses the data etc until the program goes seemingly insane. Especially if there are simultaneously acting functions the global variables are a really bad idea. Experience with the global variables and programming in general has thaught many programmers to solely rely on parameters and return values, they are the only thing that definately work.

## Creating and using the main function

Unlike in other languages such as Java or C, ín Python there is no syntactic reason to create any functions or objects which would act like a main function. However, as this practice is generally worthwhile and avoid a number of problems with the reuse of code in the latter parts of this course, lets take a moment to discuss the main function in Python in detail. The first thing understand is the division between the code in main function, main level and subfunctions:

```
01  # -*- coding: cp1252 -*-
02  #This is MAIN LEVEL code
03  print("Main level code!")
04
03  def subfunction():
04      #This is SUBFUNCTION code:
05      print("Subfunction code!")
07
08  def main():
09      #This is MAIN FUNCTION code:
10      print("Main function code!")
11
12  #This if-structure tells the interpreter
13  #which of the functions is the main function
14  if __name__ == "__main__":
15      main()
```

As shown in the code above, there are three distinguishable types of code; the main level, main function and subfunctions. The main level code is all of the code which is written on the "root" of the source code file, without indentation outside all function stuctures. The subfunction code is the code inside any function besides main function, which is one of the functions in the source file, defined by the special if-structure similar to the one

above. The main function is also the one which is responsible for starting and terminating the program in a controllable manner.

Unlike in languages which have mandatory main functions, in Python the main function does not have any special properties. The only thing main function does differently than other functions is that it is called in the program startup with this structure:

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

which basically means that if the program is started as the file which is executed (the __main__ -thing in the conditions) instead of imported, the function called main() is executed. If the file is imported as a module (discussed more in the next chapter) the program does not start autonomously, so the subfunctions can be used in the other program. Otherwise the if-structure is completely normal conditional structure; the reason why it is left last is practicality; if there is any code in the main level after this call, it is executed after the main function closes, usually meaning after the program has basically ended. If there was something like a declaration for global variable after this structure, it will not be available for the main function, or any function the main function starts.

## *Lambda-functions*

The last thing discussed in this chapter are the lambda-functions. These functions are a kind of miniature functions, which enable the programmer to do small changes to a variable with small

modifiable function call. This action can be basically any operator, and it is defined in the following way:

```
def [functionname]([parameter]):
    return lambda [2. parameter]: [2. parameter]
[operator] [parameter]
```

for example like this:

```
def adder(n):
        return lambda total: total + n
```

which is created and used like this:

```
>>> smalladder = adder(5)
>>> smalladder(10)
15
>>> smalladder(20)
25
>>>
```

# Chapter 7: Modules

There is a common saying in the programming that one should not spend time reinventing the wheel. Adding modules is more or less an act of installing pre-made wheels to your program.

## *Modules in Python*

There is a common saying in the programming that one should not spend time reinventing the wheel. This saying illustartes the the point that usually it is easier to find an existing solution and apply it in a program, than try to create the functionality from the scrap. For example, using network connection or implementing clock in the program are such problems; it definately is easier to just request operating system to give time or send data through the network connection, than to create everything needed by hand.

Obviously, in this matter the Python is no exception. The basic installation package includes a large array of modules, which can be used to get helpful functionalities and use them in a program. In fact, the large array of different types of modules in the module library is one of the strong points of Python; it simply covers several topics such as GUI building, mathematics, randomization, using internet connection and even using audio files.

## 1.1 Using the modules

A module from the module library is taken into use, imported to the program, with syntax keyword import. Importing the library modules is usually done as the first action in the source code, right after the character encoding declaration:

```
import random
```

This simple command informs the interpreter that the program will use a module called random. The module random is a library

module, which includes several different functions to pick random selections or random numbers or elements. Probably the easiest way to understand how import works is to use it in an example, so here is a source code for a small program which picks a random number between zero and ninetynine. Example 7-1: Importing library module, random

**Source code**

```
01  # -*- coding: cp1252 -*-
02
03  import random
04
05  #lets pick a random number between 0 and 99
06 #the easiest way is to use random-module function
randint
07  number = random.randint(0,100)
08
09  print("Program picked a number",number)
```

**Result**

```
>>>
Program picked a number 89
>>>
Program picked a number 95
>>>
Program picked a number 36
>>>
Program picked a number 85
>>>
```

As can be observed from the example, the functions within the modules imported with import command are used with syntax `modulename.functionname()`

This notation is used to avoid situations, where the two modules have a function that has the same name. For example, if the program already uses another module which has the function named randint, or has a defined with the same name such as this:

```
def randint(int a, int b):
    print("This function does not do a thing.")
```

The problem is, that the interpreter would not know which of these three functions it has to call. Therefore the notation uses the prefix with the module name. Related to the naming conventions, is a command dir([module name]), which can be used to browse module contents and help([function name]) which prints the document strings and additional instructions within the functions. This command pair is also a nice tool for browsing the module library, as the functions within the standard library modules are rather well documented and explained. Heres and ecample on how the dir()-command works:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__pack-
age__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb',
'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd',
'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'is-
nan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan',
'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians',
'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'tau', 'trunc', 'ulp']
>>>
```

## Python standard module library

The standard installation of Python interpreter includes a vast selection of module libraries and functions, which can be used to create functionality with relative ease. In the following table, the most useful and common of these libraries are listed, along with a short description summarizing the library abilities.

| Name | Summary |
|---|---|
| random | Random numbers, random selection |
| math | Mathematical functions, matrix |
| pickle | A support module used to read and write the dynamic structures such as list or class to a file. |
| sys | Functions related to the operating system services. |
| zipfile | Module which can be used to create and manipulate the zip-archive files. |
| tkinter | Creation of graphical user interfaces (see book chapter 11) |
| winsound | module which can play sound files (Only available for Windows-operating systems.) |
| time | Timing, calendar and time calculation functions |
| os | Separate module for the additional operating system-related functions. |

*Table 7.1 Some of the more useful modules of the Python standard module library*

The complete list of modules delievered with the Python 3 interpreter is available at the documentation, available online in address http://docs.python.org/3.0/library/, from point "Numeric and Mathematical Modules" onwards. Besides this, point "Generic Operating System Services" may also be a topic of interest.

As mentioned earlier, the command help() can be really helpful when browsing the module library. With this command, it is possible to get the instructions written to the module itself. The command works with the syntax

```
help([module name].[function name])
```

for example

```
>>> import random
>>> help(random.randint)
Help on method randint in module random:

randint(self, a, b) method of random.Random instance
    Return random integer in range [a, b], including
both end points.

>>>
```

One of the most common functions needed in the programs is available in the module sys. From this module, the function exit is extremely useful, as it ends the program immediatly when the command is encountered. The function exit() takes one optional parameter, which should be and interger value, and it indicates the reason why the program was terminated. However, as we are not building a OS-integrated, or parallelly processed program, this value can be left at 0, which also is the default value.

By applying this function, the program can be designed so that it does not need to be bypassable, for example by creating a huge loop which can be exited with a break-command. In addition, in Tkinter-built GUI the exit-command is needed for a clean exit.

## *Example 7-2: Ending program, sys-module*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  import sys
04
05  startprice = int(input("Please input the price:
"))
06  if startprice < 0:
07      print("Please, no negative numbers.")
08      sys.exit(0)
09  else:
10      tax = int(input("Please insert the VAT % (0-
100): "))
11  if tax < 0:
12      print("VAT cannot be less than 0.")
13      sys.exit(0)
14                      print("Final          price
is",startprice*(tax/100)+startprice)
```

### Result

```
>>>
Please input the price: 100
Please insert the VAT % (0-100): 21
Final price is 121.0
>>>
Please input the price: -1150
Please, no negative numbers.
Traceback (most recent call last):
File "C:/Users/Jussi/Desktop/koodit/9eSys.py", line
8, in
sys.exit(0)
SystemExit: 0
>>>
```

When the program advances to a sys.exit(0)-command, the program is immediately terminated. If this happens, the interpreter passes a notification which resebles an error message. As mentioned earlier, this is not an error, but rather a notification for reason the program terminated. In fact, the general error class Exception does not catch this message.

Lets take a few more examples on applying the library modules: First, we make a program which simply uses the math-module for a few calculations, and the then a larger program, which randomly picks the lottery numbers.

## *Example 7-3: Mathematical funtions, math-module*

### Source code

```
01  # -*- coding: cp1252 -*-
02  import math
03
04  side_1 = 1.5
05  side_2 = 3.5
06
07  print(math.sin(1.5 / 3.5))
08  print(math.pi)
```

### Result

```
>>>
0.415571854993
3.14159265359
>>>
```

This example is very minimalistic. The program imports the library module math, and by applying two of its funtions calculates a sin value from the given sides, and prints the defaults pi value stored in the fixed value. Unlike other methods, the math.pi is not a function or method, but more like a fixed variable. It can be used as a normal variable in the calculations, with the exeption that it should not be altered. However, if the fixed value is altered anyway, it may compromise the stability of the program, as there usually are other functions which rely on these types of fixed values. The value resets when the interpreter is restarted, so no permanent damage is done.

## *Example 7-4: Picking lottery numbers*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  import random
04
05  numbers = []
06  #picks 7 random numbers from 1 to 39
07  while True:
08      if len(numbers) == 7:
09          break
10      pick = random.randint(1,39)
11      if pick not in numbers:
12          numbers.append(pick)
13
14  numbers.sort()
15  print("The program picked the following\
 numbers:")
16
17  for i in numbers:
18      print(i, end = ' ')
```

**>Result**

```
>>>
The program picked the following numbers:
1 14 16 22 27 28 38
>>>
The program picked the following numbers:
8 21 26 30 34 37 38
>>>
```

This program picks lottery numbers by using the randint-function from the module random. The picked number is saved to a list numbers if it has not been picked earlier. When the amount of numbers saved in the list numbers reaches seven, the list is sorted with the list method sort, and printed.

## *Special import methods, from import*

For importing modules there are some special cases. For example, if the size of the final program is limited, or the program only needs one functionality from a huge module, it is also possible to import individual funtions instead of the entire module. This can be done with syntax

```
from [module name] import [function name]
```

for example

```
from math import sqrt
```

which would mean that in this program, the square root calculating function sqrt is now available similarly as normal funtions such as input or print:

```
[function name]([parameters])
```

for example

```
sqrt(1024)
```

This is possible because now the interpreter knows that the call means exactly this function from the module math. In the next example this method is used to import the function randint from the module random in a game, where user tries to guess the right number:

## *Example 7-5: Importing individual function*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  from random import randint
04  import sys
05
06  def numbergame():
07      number = randint(0,100)
08      while True:
09
10          guess = int(input("Give a number between
0-100: "))
11           if guess < 0:
12               print("Bad guess!")
13               print("Program is terminated.")
14               sys.exit(0)
15           if guess == number:
16               print("You guessed correctly!")
17               break
18           elif guess < number:
19                 print("The number is larger than
that.")
20           else:
21                print("The number is smaller than
that.")
```

```
22
23  if __name__ == "__main__":
24        numbergame()
```

**Result**

```
>>>
Give a number between 0-100: 50
The number is larger than that.
Give a number between 0-100: 75
The number is larger than that.
Give a number between 0-100: 87
The number is larger than that.
Give a number between 0-100: 95
The number is smaller than that.
Give a number between 0-100: 90
The number is larger than that.
Give a number between 0-100: 92
You guessed correctly!
>>>
```

As can be seen from the example, the included funtion now operates direcly with its name, without any reference to the module name. In addition, this way of importing names can be extended even further: If it is absolutely certain that there will be no naming conflicts, the entire module can be imported to operate directly without name reference with command

```
from <module name> import *
```

This way of importing the entire module is usually done with the modules which are large, complete overhauls which alter the entire programming paradigm. From the standard module library, the GUI-producing module Tkinter is one of this type of modules. It changes the programming style to a degree where it is

plausible to argue that the tkinter funtionalities become the program core, so importing it this way is acceptable. Other, less imposing possibility would be a custom-build module, which includes several programmer-made general purpose functions. This option is discussed next.

## *Creating an own module*

Another really important feature related to the modules is the ability import functionalities such as class definitions or functions from the other source code files.

The concept of reusability has been discussed earlier in the sections regarding the modular structure of the source code and in introduction of functions. This was emphasized because in Python, any function written into any source code file can be used in another source file simply by importing it. For example, lets assume that there is a file called "calculator.py" in the harddrive, and it has the following code:

```
01  def average(int a, int b):
02       """Calculates the average between the two
integers."""
03      ave = (a + b) / 2.0
04      return ave
```

This function is a self-contained, complete piece of source code. Now, lets assume that there is another source code, in which this function is needed. To be able to use this function, we simply need to copy this file to the same directory as the new source code file so that the interpreter finds the correct source file, and add to the new file a import-command

```
import calculator
```

which basically is the the name of the imported source file without the .py-file extension. After this command, the function average can be used in the new file with name calculator.average(). It is worth mentioning that both of the source code files has to

be in the same directory; only the module library modules, and registered extension modules (PIL, py2exe etc.) can be found automatically from their home directories. In the next example, this operation is practiced:

## *Example 7-6: Importing own modules*

### Source code

mymodule.py-source code:

```
01  def taxcalculator(salary,percentage):
02       """Parameters salary (int) and percentage
(0-100)
03       returns the final salary after taxes as a
integer """
04
05     final = (salary * ((100 - percentage) / 100))
06      return final
```

The actual example source code:

```
01  # -*- coding: cp1252 -*-
02
03  import mymodule
04
05  def main():
06      salary = int(input("Give monthly salary: "))
07        tax = int(input("Give tax percentage (0-
100): "))
08
09      sum = mymodule.taxcalculator(salary,tax)
10
11      print("You'll get", sum,"euroes.")
12
13  if __name__ == "__main__":
14      main()
```

### Tuloste

```
>>>
Give monthly salary: 2250
Give tax percentage (0-100): 31
You'll get 1552.5 euroes.
>>>
```

The approach used here is the way which can be used to divide the code to the different source code files. Obviously the benefits of this practice become obvious only after the amount of code growths to several hundred lines of code. Earlier, the code was broken into several functions, now the functions are divided into several modules.

At this point the difference between the main level and main function becomes apparent: in the main level, the code cannot be used in any meaningful way, but with main function, the module can be started by simply calling the main function. However, there is one exception to this rule: the variables defined in the main level can be used as fixed values:

Lets assume, that there is a source code called mymodule.py, which has the following code:

```
01  fixedvalue = 100
02  def printout():
03      print("Fixed value is:",fixedvalue)
```

fixedvalue-variable can be used by importing the module mymodule, where it can either be used directly (as was math.pi in the earlier example) or by using the module funtion printout:

```
01  import mymodule
02
03  print(mymodule.fixedvalue)
```

```
04  mymodule.printout()
```
This produces the output
```
>>>
100
Fixed value is: 100
>>>
```

The interpreter ability to search for an appropriate value for the variables beying the function itself works here wonderfully. The function printout is able to determine the value of fixedvalue even if the variable is not defined in the function itself. When the source code is used as a module, this somewhat controversial feature is actually useful: the functions within one module see their own variables, and the module "fixed variables" in the main level, and can use them without separate delievery system or parameter.

This is also the reason why the module fixed values (such as math.pi) should never be modified: The values can be used within the module itself. By changing the value of math.pi, it is probable that every single calculation, which has anything to do with the value of pi will give wrong answers. In this case, it would probably only cause wrong answers, but in some modules, it can cause instability or altogerher crash the program.

```
>>>
```

# Chapter 8: Exception handling

At some point, every user makes a mistake. Even worse, even the computer might have a problem, because the resource it was trying to use was reserved. What to do at those situations, is handled with the exceptions.

## *Catching Exceptions*

During this course, everyone has probably seen at least one error when trying to run a program. For example, if the program tries to add an integer to a string, the interpreter gives out the following error message:

```
>>> 3 + "string"
Traceback (most recent call last):
File "", line 1, in
3 + "string"
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
>>>
```

The unfortunate truth with the programming is that there is no way to be prepared against all different possibilities for errors in a practical and meaningful manner. One reason for this situation is that the programmer has no way of being sure what the end user decides to do, and that the programs end immediately when they encounter a situation which cannot be resolved. When writing the source code, the main way to prepare for user-caused problems is to think of all the possibilities which could happen, and create expection handlers or instructions for all of these situations.

Even though it is not realistic to expect the programmer to be able to pick out all of the possible problems, especially when the program size grows, it is a good practice in development to at least try to fix all errors and prevent possible problems. Actually,

in Python this is also rather easy, because the language is interpreted, it can actually pinpoint errors rather accurately, usually highlighting the cause, not a symptom of the error.

Here onwards the topics are also covered in the book chapter 6.

## Understanding the error messages

From the error message itself it is possible to make some observations. The interpreter constructs the error message so, that it actually tells the programmer what happened, where, and what was the offending error that caused the program to crash. Lets consider the above example, and see what we can assess from the error itself, even without the actual source code which was executed.

```
File "", line 1, in
```

This line tells that the error happened in the python shell (interpreter window), on line 1. The next line is the offending source code, which caused the error, and the next line the actual error, which was raised:

```
3 + "string"
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
```

These lines tell that the command which caused the error was the command 3 + "string", where a TypeError occurred. In addition of this, a short description on what happened is given: unsupported operand type(s) for +: 'int' and 'str', which obviously means that the addition operand (+) caused the error beccause it was given two values, which it could not add up, and that these

values were of type integer and string. Obviously in this case the reason for error is simple, but in any case, this data can be later used in creation of error handling mechanism.

## *Try-except*

Catching and handling an error in Python is basically done with structure try-except. In this logically two-phased structure the vulnerable code is written into the try-segment, which is followed by the except-segments, which define the actions taken, if that type of error occures.

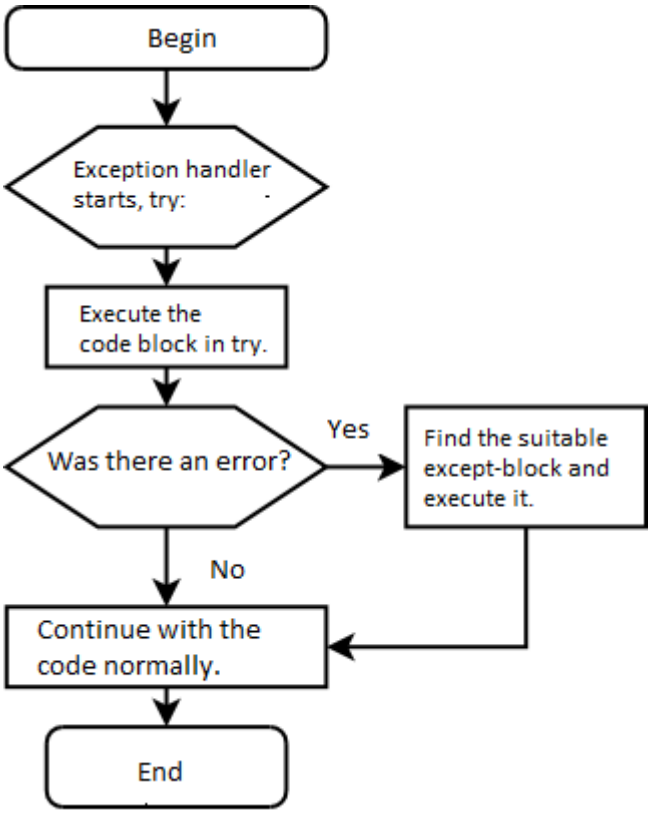



Figure 8.1: Program with an exception handling

The program executes normally, until it reaches the beginning of the exception handler, where the interpreter starts to go through the try-segment. If this segment goes through the interpreter without any errors, the exception handler is exited and the program continues at the next logical line after the handler. If there were an error in the try-sefment, the interpreter starts to look for a suitable handler. The execution of the source code at try-segment ends at the line, which caused the problem, and the interpreter moves to the beginning of the first suitable except-segment. When except-segment is done, the program continues at the next logical line of source code after the exception handler. Also, if the exception handler does not have a suitable except-segment, the interpreter raises a normal error message, similarly as in the situation where no exception handler structure was used.

## *Example 8-1: Catching an error*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  #Lets ask a value for mynumber
04  mynumber = input("Give a numeric value: ")
05
06  #convert the value to integer
07  try:
08      mynumber = int(mynumber)
09      print("You gave a number",mynumber)
10
11  #In case any error happens, the Exception
12  #-named exception segment is run
13  except Exception:
14      print("That was not a number!")
```

**Outcome**

```
>>>
Give a numeric value: Monkey
That was not a number!
>>>
Give a numeric value: 1643
You gave a number 1643
>>>
```

As illustrated in the example, the exception handler is constructed by inserting the vulnerable code to the try-segment. In this case, the code inserted into the try-segment is a typical case of vulnerable code, a type conversion from the user-given input to a numberic value. That part is vulnerable, as the code tries to do something, which can cause error if the user has not followed the instructions or deliberately tries to fool the system. Other such common cases are the file opening, division with user-given inputs (chance to divide by 0), and going through dynamic structures, where the size of the structure can change between the iterations. These activities should always be within an exception handler.

## Catching different types of errors

After the try-segment, the except-segments are implemented. In the except-segments the different types of errors, which are sought to be caught, are identified. In the last example the errors were caught with an exception class Exception:

```
except Exception:
```

This error class (Exception) acts as an umbrella, which catches every other error type, but the user-raised keyboard interruption

Ctrl-Z and the SystemExit, which is raised by the sys.exit()-function.

Naturally there are many different errors, which can be caused by a relatively harmless-looking action, and all of the errors require different type of corrective action to take place. Lets take an example of a source code, where several different errors are catched and corrected:

## *Example 8-2: Handling several types of errors at once*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  def getnumber():
04      mynumber = input("Give a numeric value: ")
05      return mynumber
06
07  def main():
08      number1 = getnumber()
09      number2 = getnumber()
10      try:
11          result = int(number1) / int(number2)
12
13      except ZeroDivisionError:
14           print("Its not possible to divide with
0.")
15
16      except (TypeError, ValueError):
17            print("Its not possible to calculate
letters.")
18
19      else:
20          print("The result is",result)
21
```

```
22  if __name__ == "__main__":
23      main()
```

**Printout**

```
>>>
Give a numeric value: 42
Give a numeric value: 0
Its not possible to divide with 0.
>>>
Give a numeric value: 42
Give a numeric value: Dolphin
Its not possible to calculate letters.
>>>
Give a numeric value: 42
Give a numeric value: 13
The result is 3.23076923077
>>>
```

This program reacts to the different types of errors, and in fact is capable of telling the user the exact reason why the given value was rejected. Also, pay attention to the else-structure in the exception handler; in exception handler, if no except-segment was executed, the else-segment is.

## Error classes in Python

In Python-programming language exists 35 different "real" error types, and 9 different warnings which are a kind-of semi-errors, which the interpreter can handle itself. In the following Tables, these error types and warnings are explained:

| Error class | Definition |
|---|---|
| ZeroDivisionError | An error caused by division with zero. |
| AttributeError | Errors caused by the faulty calls of methods and attributes. |
| ImportError | Errros associated to importing external modules and library modules. |
| NameError | Errors caused by a reference to a variable that does not exist. |
| TypeError | Errors caused by incompatible variable types. |
| ValueError | Errors caused by the unsuitable values of variables. |
| IOError | Errors associated to the file handling. |
| IndexError | Errors caused by a direct reference to outside the dynamic structure; for example reference listitem[300], when listitem has only 20 items. |
| Exception | General error class, which is used to catch all "common" error types. |
| UnboundLocalError | The varible is not defined and it cannot be created at this point; for example defining the iterator variable first time at the iteration argument. |

*Table 8.1 The common error types of Python*

| Warning class | Definition |
|---|---|
| DeprecationWarning | Program applies a syntax or structure which will be deprecated (i.e. removed) from the Python language in the upcoming version updates. |
| SyntaxWarning | Source code structure is ambiguous, sometimes caused by mixing tabulator and space in the indentation. |
| RuntimeWarning | Program causes instability in the interpreter. |
| FutureWarning | Program applies a syntax or structure which will change its behavior in the future version update. |

*Table 8.2 Python warning classes*

Warnings are relatively rare, and are usually only encountered in special circumstances. However, the DeprecationWarning and FutureWarning are worth mentioning; they usually indicate that the offending feature should be replaced with something more suitable, as the program may stop working at the next interpreter update, or at least in a near future.

As for errors, the Name, Type and ValueError are relatively common as they are related to working with the variables. When working with dynamic structures such as lists, the IndexError is relatively common sight, usually indicating that the starting or ending point of an iterator is miscalculated. Most other error classes are somewhat specific; this is actually a good thing as that way the certain types of errors can be easily separated for their own exception handler.

In these tables, only the most common errors are listed. A full list of Python errors and warnings can be accessed at http://docs.python.org/3.0/library/exceptions.html.

## *Else-segment*

As mentioned earlier, the exception handler also allows an else-segment. In the exception handler, the else-segment is defined as the last segment, after all of the except-segments. If the handler does not execute any of the except-segments, meaning that the try-segment was done without errors, the else-segment is executed.

### *Example 8-3: Else-segment in an exception handler*

**Source code**

```
01  # -*- coding: UTF8 -*-
02
03  number = input("Give a number value: ")
04
05  try:
06      number = int(number)
07  except Exception:
08      print("That was not a number.")
09  #else happens if no except is used
10  else:
11      print("You gave a number",number)
```

**Printout**

```
>>>
Give a number value: 50
You gave a number 50
>>>
```

```
Give a number value: Sguigg
That was not a number.
>>>
```

As observable from the example above, the else is used when no except-segment is activated. The else is most suitable in the situations, where the program has to do something with the data defined in the try-segment; If an error happens and it is caught, the data changes in the try-segment is never recorded, as if the code never happened. In that situation, a print-command after the exception handler would cause an error.

## *Controlled takedown, Try-Finally*

Other exception handling mechanism within the Python is the try-finally. Unlike try-except, try-finally is not meant to prevent the program from crashing, but to offer a controlled method for ending the program on an error. This gives the program an option to shut all open files or inform the host end of the program that the client program is closing and the host should shut the connection.

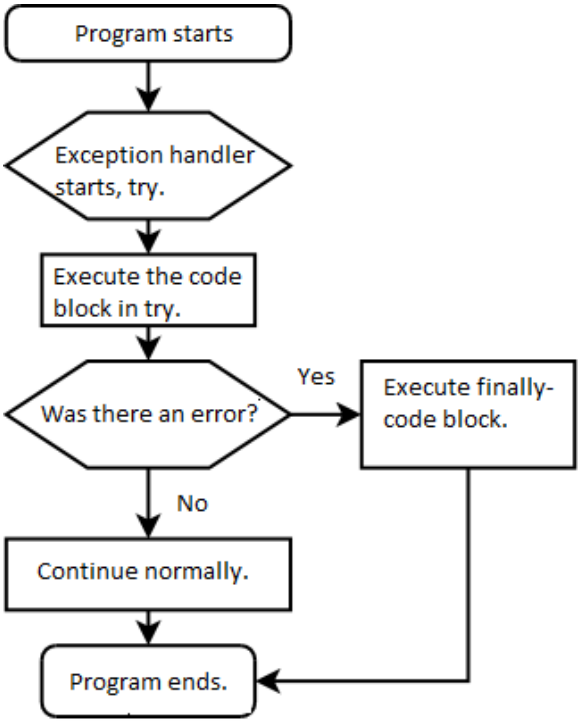



*Figure 8.2: A program, which will be ended in a controlled manner if the try-segment causes an error.*

In try-finally-structure, the finally-segment comes after the try-segment and it does not take any arguments or parameters:

## *Example 8-4: Ending on an error*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  #lets ask for an input
04  number = input("Give a value: ")
05
06  #lets try to convert the input to an integer
07  try:
08      number = int(number)
09      print("You gave a value",number)
10
11  #if anything happens, finally is executed
12  finally:
13      print("There was an error in the program!")
```

### Printout

```
>>>
Give a value: Whatnow?
There was an error in the program!
Traceback (most recent call last):
File  "C:/Users/Jussi/Desktop/koodit/6e2.py",  line
8, in <module>
number = int(number)
ValueError: invalid literal for int() with base 10:
'Whatnow?'
>>>
```

In this case the example code does not do anything else besides informing the user that there was an error. Obviously the user would also see this from the error message itself, but if there would have been something to do at the ending procedures, it could have been done at the finally-segment. Lets take one more example at the exception handling:

## *Example 8-5: Asking a numeric value*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  def numbergetter():
04      while True:
05          try:
06              numbervalue = input("Give a numeric
value: ")
07              numbervalue = float(numbervalue)
08              return numbervalue
09          except Exception:
10             print("Erroneous input, try again.")
11
12  def main():
13      print("Please type in the full salary")
14      salary = numbergetter()
15      print("Input the amount of taxes (0-100)")
16      taxes = numbergetter()
17      leftover = salary * ((100 - taxes) / 100)
18     print("You are left with",leftover,"euroes.")
19
20  if __name__ == "__main__":
21      main()
```

### Printout

```
>>>
Please type in the full salary
Give a numeric value: Hawk
Erroneous input, try again.
Give a numeric value: 1600
Anna veroprosentti (0-100)
Give a numeric value: Dove
Erroneous input, try again.
Give a numeric value: 25
You are left with 1200.0 euroes.
>>>
```

The worthwile part of this example is the function which gets the user input; it is a really strong method of ensuring that the user gives an usable input. The only way to not give an input is to terminate the program with the escape sequence Ctrl-C, which raises a KeyBoardInterrupt-error, which by default is not catched by the default except Exception-handler.

## *Self-made error classes and generating the error*

In this segment, some of the discussed topics include classes and objects. Unless you have a prior knowledge on programming, it is advicable to go through the chapter 10 before advancing.

## Defining an error class

In the Python language, every error type is inherited from the class Exception. In practice this means that to define an error class, the programmer simply needs to make a following class type:

```
class MyError(Exception):
        def __init__(self,value):
                self.value = value
        def __str__(self):
                return repr(self.value)
```

This code creates an error class MyError, which is initialized to take the given error message as an argument and used if the error is raised. If the program confronts the error the self-defined error message should be used, it can be raised with the following command:

```
>>> raise MyError("There are four lights.")
Traceback (most recent call last):
  File "", line 1, in
    raise MyError("There are four lights.")
MyError: 'There are four lights.'
>>>
```

## raise, Manually raising an error

In the above example the method of manually raising the error is mentioned. Obviously, this is done with a syntax

```
raise [error type]
```

For example

```
>>> raise SyntaxError
Traceback (most recent call last):
  File "", line 1, in
    raise SyntaxError
  File "", line None
SyntaxError:
>>>
```

This is a quick way of taking the program down; if the program ends up in a situation, where the program cannot proceed, but no error is actually caused, the program can raise the error manually. For example, in the more advanced programs this situation can happen if the other processes time out or do not respond, but no error is raised. This happens fairly regularly for example when doing low-level network programming with sockets.

## *The most common errors made by the beginners*

This list includes all of the most common errors done by the beginner-level programmers. If you encounter a problem that you simply cannot find, please use this checklist to see that everything is as you intended it:

| Error | Explanation | Example |
|---|---|---|
| Opening file | You tried to read a file that does not exist, or was accidentally emptied by writing it with mode "w". Is the name spelled correctly, it is case sensitive and needs to have full extension. Is the name given as variable name or string? | `open(file)`<br>`or`<br>`open("file")` |
| Iteration works incorrectly | Is the start and the end point counted correctly, or are you going over by one, or ending one place too soon? Usually you are trying to process the item after the last one. | `range(len(mylist))`<br>`or`<br>`range(len(mylist)+1)` |
| Function call | Are you accidentally trying to give variable name as string, or vice versa? | `functionname(myvalue)`<br>`or`<br>`functionname("myvalue")` |
| Parenthesis mismatch | Are all parenthesis, even the nested ones, actually closed by the end of the line? | `print("Text!",str(arvo )`<br>`int(input("Write:")` |
| Missing colon (:) | Is each new indented block started by a line with colon as the last character? | `While True`<br>`  print("teksti")` |
| Else-segment in the wrong place | Is else-segment indented correctly? Else-segment can connect to several different structures, so are you connecting it to the iteration or the condition? | `while True:`<br>`  if reqvar == True:`<br>`  break`<br>`else:` |
| Citation and quotation signs mismatch | Are all citations closed? If you copy-pasted the code, are the citation or quotation marks transferred correctly? For example, compare the citations marks around the example's word Killer. | `"James 'Killer'`<br>`Strongface"` |

*Table 8.2 Most common errors in beginner-level programs*

# Chapter 9: Advanced data structures

In this chapter we go through the more refined and advanced ways to manage our data with the Python-language.

# Why more datastructures?

One of the very common topics in Python, which so far has not been discussed in many details, are the advanced and dynamic data types. In the different programming languages these datatypes are called either lists, tables, structures and arrays, depending on the program lingo. In Python, the most common of these advanced types is list, which has already been used in several occasions, but there also exists several others. The concept of these advanced types is that they have items, basically meaning that their content can be varied either by defining what the advanced datatype has, or just by adding things to the existing structure. In layman's terms, this would mean that if a character string is one thing like "a bag of apples", the list is the entire shopping cart, including all of the stuff that was supposed to be bought.

## *List*

As mentioned in the introduction, the list is probably the most common advanced datatype in Python. The list basically is a normal variable, but instead of one item at a time, it can have several items, which can be refered to with a syntax similar to single characters in a string. Otherwise the list is used as any other variable, and initialized with a command

```
[name of the list] = []
```

This command creates an empty list, which has zero items. After the list is created, it can get new values as items with method .append() and remove existing items with command .remove():

```
>>> mylist = []
>>> mylist.append("oneitem")
>>> mylist.append(123131)
>>> mylist
['oneitem', 123131]
>>> mylist.remove("oneitem")
>>> mylist
[123131]
>>>
```

List items can be added and removed with list modifying methods. However, the method .remove() is a bit troublesome, as it requires the content of the item as an argument. Better method for removing items is .pop(), which only requires the item place number insted of content to remove it from the list. As with string slices, the first item is at place 0, so listname[0] refers to the first item in the list, and will be removed with command listname.pop(0).

```
>>> mylist = ["First", "Second", "Third"]
>>> mylist.pop(1)
'Second'
>>> mylist
['First', 'Second']
>>>
```

If method .pop() is not given any argument, it removes the last item in the list. The method also returns the value of the deleted item as a return value, so it can be also used to "check out" one item from the list to a variable.

## *Example 9-1: Creating and using a list*

**Source code**

```
01  # -*- coding: cp1252 -*-
02
03  #List as a variable
04   #Create mylist, notice the line break in the
definition
05  mylist = ["Apples","Milk",
06            "Beer","Squigg"]
07  print(mylist)
08
09  #Lets add one item
10  mylist.append("Pineapple")
11  print(mylist)
12
13  #Lets remove the item 1
14  mylist.pop(1)
15
16  #This prints the mylist
17  for i in mylist:
18    print(i)
```

**Output**

```
>>>
['Apples', 'Milk', 'Beer', 'Squigg']
['Apples', 'Milk', 'Beer', 'Squigg', 'Pineapple']
Apples
Beer
Squigg
Pineapple
>>>
```

In this example, the line break in line 5 seems to be in a rather peculiar place. With list, and actually any other definition of stuctured data, the line can be changed after the dot between the items, as illustrated here. This makes it easier to define and understand how the list is structured.

References to the individual items in the list are done with a syntax similar to the string slicing. In fact, most of the slicing techniques also work with list items:

```
>>> mylist = [1,2,3,4,5,6,7,8]
>>> mylist[2:5]
[3, 4, 5]
>>> mylist[:6]
[1, 2, 3, 4, 5, 6]
>>> mylist[::-1]
[8, 7, 6, 5, 4, 3, 2, 1]
>>> mylist[3]
4
>>>
```

The contents of the list items can be tested with several methods, such as index and count. Index tells the first instance of the item, which has the same value as the given argument. Count tells how many such items are in a one list.

```
>>> mylist = ["Shovel", "Sledgehammer", "Drill",
"Sledgehammer"]
>>> mylist.index("Drill")
2
>>> mylist.count("Sledgehammer")
2
>>> mylist.index("Screwdriver")
Traceback (most recent call last):
  File "", line 1, in
    mylist.index("Screwdriver")
ValueError: list.index(x): x not in list
>>>
```

With index-method it should be mentioned that if there is no item, which corresponds to the given argument, a ValueError is raised. Anyway, index and count are not the only list methods,

on the next Table, a group of the most common and useful list methods are described.

| Name | Definition | Observations |
|---|---|---|
| .append(X) | Adds value X as the last item in the list. | |
| .insert(I,X) | Adds argument X as the item number I. | |
| .remove(X) | Removes the first list item, which has the value X. | |
| .pop(I) | Removes the list item number I. | |
| .index(X) | Returns the item number of the first item, which has the value X. | Raises a ValueError if none of the items match X. |
| .count(X) | Returns the amount of times the X appears as a list items. | |
| .sort() | Sorts the list items based on their value. | Values are primarily dictated by the list character ASCII-value. |
| .reverse() | Reverses the list items, first becomes the last, last the first. | |

*Table 9.1 Some of the most common list methods.*

The full list of different list methods can be accessed at the Python documentation online, at address http://docs.python.org/3.0/library/stdtypes.html#sequence-types-str-bytes-bytearray-list-tuple-range starting from the point "Mutable Sequence Types".

One easy way to test if some value is within the list is to use the operator in. This operator is useful, as it is simple, easy and safe way to test whether some value is in the list or not:

```
>>> basket = ["Apples", "Orange","Kiwifruit","Ba-
nana"]
>>> "Cauliflower" in basket
False
>>> "Apples" in basket
True
>>>
```

With list, and other advanced datatypes with items, for-iteration becomes very usedful tool. For-structure has the limitation that it wants the amount of iterations calculated beforehand, which usually requires the programmer to use len() and range() to create this action. Hoever, with lists this range(0,len(something))-monster can be simply replaced with the list itself:

```
>>> mylist = [1,2,3,4]
>>> for i in mylist:
        print(i)

1
2
3
4
>>> newlist = []
>>> for i in mylist:
        newlista.append(i+10)

>>> newlist
[11, 12, 13, 14]
>>>
```

This technique limits the ability to refer to a certain item as list[i], as now the turn calculator has the item value, not place number as its content. However, it makes changing the all of the list items at once really quick and easy.

List as a datatype is repeatedly used in the file handling examples and other important actions, so it should be understood well. However, list is not the only advanced datatype in Python, even though it is by far the most common. Next, lets take a look at the other types, which are more or less special cases of the list-type.

## *Dictionary*

Another type of structured datatype in Python is the dictionary. The dictionary is a special type of list, where the items can be given names instead of just numbers, meaning that it is possible to, for example, couple a string "Apples" to a item name "A".

```
>>> dictio = {"A":"Apples", "O":"Oranges"}
>>> dictio["A"]
'Apples'
>>>
```

As demonstrated by the example, in dictionary it is possible to give a name to the item, instead of just using the item number. The dictionary is a useful tool for example in saving settings data, or when the system needs to systematically replace or look out data based on input. These advances are probably most easy to demonstrate with an example. In this example, a translator for Morse code is created, with the intention of turning text strings into dashes and dots, similarly as in the Morse langauge.

## *Example 9-2: Morse characters and dictionary*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  def morsecoder(word):
04      #Lets define some letters
```

```
05      alphabet = { 'a' : '.-', 'b' : '-...',
06                   'c' : '-.-.', 'd' : '-..',
07                   'e' : '.', 'f' : '..-.',
08                   'g' : '--.', 'h' : '....',
09                   'i' : '..', 'j' : '.---',
10                   'k' : '-.-', 'l' : '.-..'}
11
12      result = ""
13      for i in range(0,len(word)):
14          result = result + alphabet[word[i]]+"/"
15
16      print("Word",word,"in Morse code is")
17      print(result)
18
19  worde_1 = "cliff"
20  worde_2 = "bach"
21  morsecoder(worde_1)
22  morsecoder(worde_2)
```

**Output**

```
>>>
Word cliff in Morse code is
-.-./.-../../..-./..-./
Word bach in Morse code is
-.../.-/-.-./..../
>>>
```

The need for dictionaries in source code is really sporadic. The dictionary has a few big advantages, and a few application areas where it is irreplaceable, but in everyday programming it really is uncommon feature. As dictionary also does not have the editing methods of a list - a restriction it shares with tuple - it really becomes mostly a special case of lists.

## *Tuple*

In addition of dictionary, there is another special case of list in Python called Tuple. When dictionary was a special case in a sense that it allowed naming the different items, the special feature in tuple is that it is completely static. This means that once the tuple is defined, it can be used as a list in references, but unlike list, it is not possible to add, remove or change the individual items in the structure without redefining the entire tuple.

```
>>> place = (124,124)
>>> place
(124, 124)
>>> place[0]
124
>>> place[0] = 200
Traceback (most recent call last):
  File "", line 1, in
    place[0] = 200
TypeError: 'tuple' object does not support item as-
signment
>>>
```

As demonstrated above, the tuple is created in a same way than list, with the sole exception of using the "normal" parenthesis "(" and ")". It is possible to reference only one tuple item, but the items cannot be changed or removed. In fact, only two of the list methods are available in tuple, count and index.

Tuple is an interesting structure, and at first sight it may seem rather unnecessary. However, with multiple assignments this structure offers a easy way of delivering parameter values to a subfunction:

```
>>> part1 = "It"
>>> part2 = "Is"
>>> part3 = "Glorious"
>>> tuplelist = part1,part2,part3
>>> tuplelist
('It', 'Is', 'Glorious')
>>> one, two, three = tuplelist
>>> three
'Glorious'
>>>
```

This increases the security of delievering parameters, and ensuring that the values do not accidentally change. As tuple is kind of a write-proctection for values it has its uses. This is especially true in programs, which has intertwined or parallelly executed code, or are communicating via unreliable network carrier where the entire dataset needs to come through fully, or not at all.

## *Example 9-3: Tuple in a program*

### Source code

```
01  def tupleprint(data):
02    part1, part2, part3 = data
03    print(part1+":")
04    print(part2+" :: "+part3)
05
06  name = "Old Jolly"
07  address = "Mountaintop 1, FI-99999 Korvatunturi"
08  phone = "555-1234567"
09
10  datatuple = (name, address, phone)
11
12  tupleprint(datatuple)
```

**Output**

```
>>>
Old Jolly:
Mountaintop 1, FI-99999 Korvatunturi :: 555-1234567
>>>
```

It is true, that on first sight there does not seem to be much going on for a tuple. However, the ability of delivering the entire group of parameters as a one bundle, which also is write-protected is nice from the maintenance perspective as it ensures that the data includes all expected parts, and only the expected parts, in the required format.

## *Set*

The last structure in this chapter is Set, which also is an extension, or special case, of list. A Python set is similar to the sets familiar from the logic. A set can be created from any group of items, such as strings or numbers, and it includes all of the unique items, as in set there cannot be two items which are identical. How the set actually works is probably easiest to demonstrated with an example:

```
>>> basket = {"apple","carrot","apple","milk",
"potato"}
>>> print(basket)
{'carrot', 'apple', 'milk', 'potato'}
>>> names = ["John", "Paul","George",
"Stuart","Pete","Paul","John"]
>>> players = set(names)
>>> print(players)
{'Pete', 'Paul', 'John', 'George', 'Stuart'}
>>> letters = set("acrabadabra")
>>> print(letters)
```

```
{'a', 'c', 'r', 'b', 'd'}
>>> moreletters = set("emphasis")
>>> letters - moreletters
{'c', 'r', 'b', 'd'}
>>> letters | moreletters
{'a', 'c', 'b', 'e', 'd', 'i', 'h', 'm', 'p', 's',
'r'}
>>> letters & moreletters
{'a'}
>>>
```

This example just shows the different ways of making a set. The first possibility is to use the "wave" parenthesis ("{" and "}"), which work similarly to the way the normal list is defined. Other way is to use a dedicated type conversion function set(), which takes a list or string and makes a set out of it. In both cases, multiple instances of a same item are deleted from the resulting set.

Set also has a few special operators, which allow certain data-centric special actions. In the table below, all the special operators for set are listed:

| Operator | Explanation |
|---|---|
| A - B | Set complement; leaves items, which are in A but not in B. |
| A \| B | Set union; Leaves all items, which are in either A or B or both |
| A & B | Set intersection; Leaves items, which are both in A and B |
| A^B | Negative set intersection; Leaves items, which are in either A or B, but not in both. |
| X in A | Item belonging; Returns True, if X is a item of a set A. |

*Table 9.2: Set operators*

Set is really useful for combining two lists or collecting only those items, which exist in both lists.

## *pickle-module*

In the chapther 5, there was a discussion on writing a file which mentioned the possibility of just dumping the dynamic data structures to a file, without having to manually deconstruct them for example to a CSV (comma separated value) format. The last thing discussed in this chapter, the module pickle, is created for just this operation. The pickle is included normally with a command

```
import pickle
```

To be able to use pickle, the file has to be opened to a bit-writing mode bw. This is important, as pickle cannot use normal character-files! If the pickle gives a following error

```
Traceback (most recent call last):
File          "C:/Users/Jussi/Desktop/koodit/8ePick-
leTallenna.py", line 8, in
pickle.dump(lista,tiedosto)
File "C:\Python30\lib\pickle.py", line 1315, in dump
Pickler(file, protocol).dump(obj)
File "C:\Python30\lib\io.py", line 1421, in write
s.__class__.__name__)
TypeError: can't write bytes to text stream
```

it means that the file handle was created incorrectly. In any case, the pickle uses a dedicated function to write dynamic structure to a file handle. This function is named pickle.dump(), and it is used like this:

```
pickle.dump([structure name],[file handle])
```

This makes the pickle to write the structure and its items into the file opened to the file handle in a datagram, which can be restored to a variable with the corresponding loading function. Pickle can save all of the dynamic structures mentioned in the chapter 7, in this chapter and also the classes and objects, which are discussed in the next chapter. Here is an example on writing datagram with the pickle.

## *Example 9-4: Dumping dynamic structure to a file as a datagram*

### Source

```
01  # -*- coding: cp1252 -*-
02
03  import pickle
04
05  listexample = ["Pineapple", "Atlas", ("Shaft",
"Blade"), 1150]
06  filename = open("saveme.dat","wb")
07  print(listexample)
08  pickle.dump(listexample,filename)
09
10  filename.close()
```

### Printout

```
>>>
["Pineapple", "Atlas", ("Shaft", "Blade"), 1150]
>>>
```

In addition of printing the list to the interpreter window, the program also created a file named "saveme.dat". This file cannot be read or edited normally with editor like IDLE or Notepad, as it

has non-printable bit-mode data. If the file is opened in a normal editor, this bit data cause the uncomprehensible "scribble" in the file.

*Never, ever manually edit any file which is written in the bitmode "bw". This breaks the file completely, and puts it beyond any attempt to repair it.*

Obviously as the datagram can be written, there also exists a method of loading it. That function is pickle.load(), and it works as follows:

```
pickle.load([file handle])
```

It is worth mentioning, that the file handle here has to be opened in mode bit-read, br. Also, there can be only one dynamic structure saved as a datagram per file, meaning that it is not possible to write or read from one file more than once. If there is a need to save more than one list to one file, the lists have to be merged before and dumped as a one big list or by using several files.

## *Example 9-5: Reading datagram*

### Source code

```
01  # -*- coding: cp1252 -*-
02
03  import pickle
04
05  filename = open("saveme.dat","rb")
06  justread = pickle.load(filename)
07
08  print("Following was just read: ",justread)
09  print(justread[2],justread[3])
```

```
10  filename.close()
```

## Printout

```
>>>
Following was just read: ["Pineapple", "Atlas",
("Shaft", "Blade"), 1150]
('Shaft', 'Blade') 1150
>>>
```

# Chapter 10: Introduction to classes and what next

In this final chapter we talk about Python classes, and also about few ideas what to read next, after completing this book.

## *Object-oriented programming and Python*

In the earlier chapters the programming has been discussed based on the viewpoint of procedural programming. This means that the programs are always divided to procedures; functions, parameters and modules, which are used in a certain order, while data is saved to the variables. The procedural programming is one of the oldest approaches on programming, and still completely viable approach, while being probably one of the easiest models to understand. That is also the reason why in his course this approach is used to a very large extend. However, that is not the only approach on programming, in fact not even the only approach in the Python language. In this chapter we discuss this other very popular approach, object-orientation, and take a quick peek into what it is all about.

In the newer programming languages such as C++, Java or .NET-environment, the approach is closer to object-orientation than procedural programming. The fundamental difference between procedural and object-orientation is the application of classes and objects. Instead of functions and variables, the data is stored in objects, which are able to modify the data by themselves. In objects, there are internal variables and methods - kind of functions - and in a some sense they are dynamic structures.

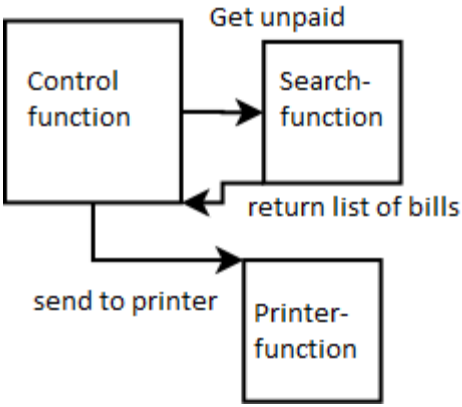



*Figure 10.1: Procedural program: activities are in functions, while data is in variables. Functions are called in order to manipulate variables.*

Object-orientation in a sense is not that new approach, as it has been around for more than 20 years. Even if it was originally created as a "revolutionary" way of combining activities with the data, using the classes and objects is not after all that different. In fact, this course has been using objects for a while; all strings, classes and file handles with their manipulating methods are actually objects. In fact, Python is a full object-oriented language, with a possibility to do "old-fashioned" procedural programming. Inside the interpreter, every variable and command are actually objects, which can also behave like functions and variables.

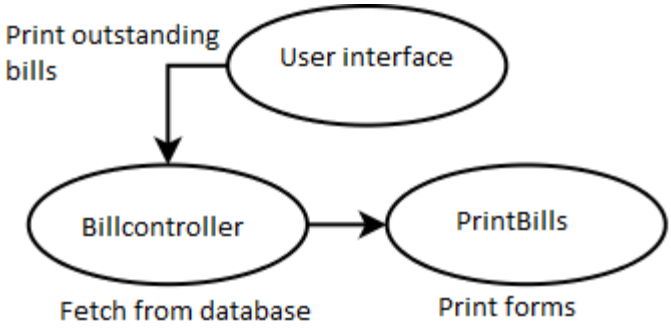



*Figure 10.2: Object-oriented program; data and methods reside in objects, which are kind of an independent units based on classes.*

The big invention in object-oriented programming are the classes. A class is a kind of dynamic structure, which has variables to save data and internal methods to manipulate these said variables. Because class is open-form structure, it can be defined from anything. In billing system, a bill can be a object, with amount and payer as data, send_invoice and mark_as_paid as methods. Similarly, a car can be an object, the color, type, owner, driven kilometers, being the internal variables, attributes.

Besides being an open datatype, which allows great adaptability, the other important feature is the class inheritance. From one class, person, the abilities can be inherited to more specific types of classes, such as employer, manager, customer. When creating a new class by inheriting an existing class, its possible to only define the new, or changed abilities, while rest of the class is copied from the old one. For example, if there already is a class "bill", which has the defined attributes and methods for normal bill, it can be inherited to a new class "repairbill". In this new

class it is sufficient to just define the methods for compiling a list of repaired items, while the other activities are taken from the old class. This is one of the reasons why classes are considered a great invention; they enable a great reuse value for existing code.

## *Creating a class*

When creating an object with attributes and methods, the first thing to do is to define a class. The class is a kind of prototype or schematics for an object, and defines the different methods and attributes the class - and subsequent object - will have. This is done by writing the class declaration, and it is done in Python with a syntax keyword class and giving the class a name:

```
class Switch:
    pass
```

this creates a class named Switch. Obviously this class currently does not have any activities as the only thing inside the class definition is the bypass keyword pass, which is required because similarly as def or any stucture, also class begins a new code segment. This type of class can be used as an open class, which can create objects which will have customly defined data. However, the more common way of creating a class is something like this:

```
class Switch:
    """Switcher-class, returns a Boolean."""
    __mode = False

    def get_mode(self):
        return self.__mode
```

```
    def switch_mode(self):
        if self.__mode == False:
            self.__mode = True
        else:
            self.__mode = False
```

This class is more like a real class declaration. The class Switch now has one internal attribute mode, and two methods which allow the program to retrieve and change the current value of the attribute mode inside the class. To test this class, the next thing needed is an object created based on the class. :

```
>>> Lamp = Switch()
```

This command creates a new object called Lamp, which basically is the functional implementation of the class Switch. The Lamp has the internal attribute mode, and the two methods get_mode and switch_mode:

```
>>> Lamp.get_mode()
False
>>> Lamp.switch_mode()
>>> Lamp.get_mode()
True
>>>
```

This example actually is a fairly common basic type of a class; it creates a private variable (mode) which cannot be modified from the outside. The only way to affect the variable mode is to use the methods which are inside the class, so there cannot be any foul play or mistakes with the variable assignments. In addition, the class also has a document string, which explains the usage of the class to the user. This feature works exactly like with the functions.

## *Class attributes*

Class attributes are a kind of variables, which exist inside the class and are are owned by the objects created from the class. They work similarly as the variables inside a subfunction, and can be used inside the class. Similarly as defining a variable in general, defining a class attribute is easy; just define a name and give it a value:

```
class [Name]:
    [attribute_1] = [default value]
    ...
    [attribute_N] = [default value]
```

For example, if the defined class is going to handle customer data such as billing information, it probably would be wise to create the class to include the customer data as a group of attributes such as name, billing method, credit card number and current bill total, something like this:

```
class Customer:
    name = "John Johnsson"
    total = 1000
    paymenttype = "Masterexpress"
    number = "1234-5678-9012345"
```

After this definition every new customer-object would have the attributes name, total, paymenttype and number, with some default value which could be used to prevent errors caused by missing data or missing attributes.

Besides normal structured classes, the Python also allows a special case of class, an open class. In this class type, the class does

not have any particular attributes or methods, but it can take new attributes simply by introducing them:

```
class Example:
    pass
```

The class Example has no defined structure. To add new attributes to the class, one can simply start stuffing the class with variables simply by defining them on-the-fly:

```
>>> Randomstuff = Example()
>>> Randomstuff.value = "I got this."
>>> print(Randomstuff.value)
I got this.
>>>
```

This creates a new object Randomstuff and gives it an attribute value, which is saved to the object and even available later to print out. However, this added attribute does not show up anywhere else besides in the object Randomstuff:

```
>>> MoreRandomstuff = Example()
>>> print(MoreRandomstuff.value)
Traceback (most recent call last):
  File "", line 1, in
    print(MoreRandomstuff.value)
AttributeError: 'Example' object has no attribute
'value'
>>>
```

Because the new objects only get the attributes and methods defined in the class. Obviously this is a bit cumbersome approach as the objects become individually different. Also, this approach uses the classes as a kind of data structure, a purpose for which

there are better options like lists or dictionaries available. However, this is important to understand, as this means that it is possible to add attributes to the objects on the go, so basically the non-internal attributes are not as save and secure as they at first might seem.

## *Example 10-1: Using class attributes*

### Source code

```
#-*- coding: cp1252 -*-

class delievery:
    """Class defines a delievery package"""
    item = ""
    name = ""
    address = ""

def addnew():
    customer = input("Customer name:")
    place = input("Delievery address:")
    stuff = input("What is the package: ")

    packet = delievery()
    packet.item = stuff
    packet.name = customer
    packet.address = place
    return packet


def main():

    round = []
    total = int(input("How many packages?:"))

    for i in range(0,total):
        deliever = addnew()
```

```
        round.append(deliever)

    print("Drop-off places:")

    for i in range(0,total):
        print(round[i].name+":"+round[i].ad-
dress+":"+round[i].item)

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

**Printout**

```
>>>
How many packages?:3
Customer name:John & Sons Inc.
Delievery address:Harbour Street 10
What is the package: Anvil
Customer name:Guitars on the go
Delievery address:Opera House Street 20
What is the package: Grand piano
Customer name:Old Mom's and Pop's Coffee shop
Delievery address:Red Light Street 25
What is the package: Undisclosed items
Drop-off places:
John & Sons Inc.:Harbour Street 10:Anvil
Guitars on the go:Opera House Street 20:Grand piano
Old Mom's and Pop's Coffee shop:Red Light Street
25:Undisclosed items
>>>
```

As observable from the example, by just using the class attributes the class is pretty much reduced to a fancy data structure. Even if this is acceptable usage for the classes, they become more interesting and definitely more useful when they are given their own methods. This will be discussed next.

## *Class methods*

The classes are useful tools, but without methods they simply lack some usability. By combining the methods and classes we start to come close to the object-oriented programming concepts, as everything for needed for working with basic classes is now discussed.

As expected, defining method is not very difficult; if attribute was a class-specific variable, then the methods are basicall class-specific functions. In fact, they are made with the same syntax, only difference being that the first parameter (and if neccessary, only parameter) is the "self", which is required by the interpreter.

Please notice that the "self" is not in fact a syntax keyword, only a naming convention similarly as the main function called "main". Techically this first parameter can have any name, but Python documentation usually recommends and uses the name self, so it is the one that should be used for this occasion.

Otherwise the methods are created just as normal functions, only inside the class definition:

```
class [Name]:
    [attribute_1] = [default value]
    ...
    [attribute_N] = [defaul value]
    def [method name](self,[parameters]):
        [function code]
```

The methods are defined after the attributes, not only because that makes the class more understandable, but also takes care of the problem of missing attributes, or accidentally overwriting an

actual value with the default. If the last example class Customer is extended to print out data form, it would be dome with a method with code somewhat like this:

```
class Customer:
    name = "John Johnsson"
    total = 1000
    paymenttype = "Masterexpress"
    number = "1234-5678-9012345"

    def printout(self):
        print("Name: ", self.name)
        print("Total: ", self.total)
        print("Payment type: ", self.paymenttype)
        print("Card/Bill number: ", self.number)
```

As observable from the example class above, the only real difference is the addition of "self" as the first parameter, and the way of addressing the class attributes with the syntax

```
self.[attribute name]
```

which basically tells the interpreter that the meant variable name is the attribute from the object being used (as in, variable in the object), not from for example global variables or main level code. At this point, lets take another larger example.

## *Example 10-2: Using class methods*

### Source code

```
01  # -*- coding: cp1252 -*-
02  class Cart:
03      """This class manages the shopping cart. """
04
```

```
05      shoppingcart = []
06      def addstuff(self):
07          esine = input("What will be added?: ")
08          self.shoppingcart.append(esine)
09
10      def checkout(self):
11          print("Following objects were added:")
12        for i in range(0,len(self.shoppingcart)):
13              print(self.shoppingcart[i], end = "
")
14
15  def main():
16      customer = Cart()
17      while True:
18            selection = input("Add more or go to
checkout?: ")
19          if selection == "checkout":
20              customer.checkout()
21              break
22          else:
23              customer.addstuff()
24
25  if __name__ == "__main__":
26      main()
```

## Printout

```
>>>
Add more or go to checkout?: add
What will be added?: Oil
Add more or go to checkout?: add
What will be added?: Squidd
Add more or go to checkout?: add
What will be added?: Banana
Add more or go to checkout?: checkout
Following objects were added:
Oil Squidd Banana
>>>
```

As shown in the example, the methods in object are called in a same manner as with string or list methods. That is because every defined string and list is actually an object, and the modifying methods are part of their classes. Outside the class declaration code, the "self"-parameter can be forgotten when calling methods, that is something that the interpreter takes care of based on the instructions given in the class declaration.

## *Private attributes*

Even in the first part of this chapter there was some talk regarding the private attributes of the Python language. So what exactly are these attributes and how do they differ from the ones discussed earlier?

The private attribute is, simply put, a attribute that is only modifiable from inside the class itself. For example, consider a program which keeps a palyer score with this type of setup:

```
01  class Score:
02      points = 0
03      def results(self):
04          return self.points
```

The class is a placeholder for the attribute which tells the current points of a player, and it has a separate method which for any reason returns the amount of points as a return value. If the attribute points is a normal attribute, it means that the points can be added like this:

```
>>> Red.points =+ 1
```

obviously requiring that there is a object called Red with this class. This adds one point to Red:

```
>>> Red.results()
1
>>>
```

But lets say that there is a minute error in the system; in one uncommon scenario, where the red team would get a point there is a code

```
>>> Red.points = 1
```

This naturally resets the points for red back to 1 instead of adding up one point. Obviously this would be easy to catch in a small program, but how about with ten thousand lines? How about over 10 million, which is in the ballpark with the length of modern games? To protect a value from outside interference, the best way to secure it in Python is to define it private.

Private attribute in Python is an attribute, which can only be changed with the classes own methods. A private attribute is defined exactly like normal attribute, but the name has to begin with at least two underscores:

```
__[attribute name]
```

Also the name cannot have more than one underscore after the name. For example names like __getsome, _____veryprivate_ and __score are ok, but names _toolittle and __failure__ are not acceptable. Anyway, if the upper example class is rewritten to this:

```
01  class Score:
02      __points = 0
03      def getpoints(self):
04          return self.__points
```

the class is protected from outside interference:

```
>>> Blue = Score()
>>> Blue.points = 5
>>> Blue.getpoints()
0
>>> Blue.__points = 5
>>> Blue.getpoints()
0
>>>
```

Obviously there is immediately a new problem. Now that the attribute is secure, it no longer can be changed simply by assigning it a value. However, this is not a problem, as the methods within the class have no problem accessing the private attribute:

```
01  class Score:
02      __points = 0
03      def getpoints(self):
04          return self.__points
05      def addpoints(self, value = 1):
06           self.__points =+ value
```

Now the class can add just one point by simply using the method, or by several points if appropriate parameter is given.

```
>>> Blue = Score()
>>> Blue.addpoints(3)
>>> Blue.getpoints()
3
>>>
```

This approach is very safe in a sense that the value cannot be overwritten. However, unlike some languages, the privatization of class attribute is not completely bulletproof. As the private attribute is simply a shorthand for little behind-the-scenes magic with interpreter, it can be directly accessed by calling it with name

```
_[classname]__[variablename]
```

but this is not somehing that can be made accidentally, or really prevented in any simple way.

## Class initialization and other class attributes

Ok, so the variable with two underscores in the beginning is a private attribute, but what are the other underscore-names visible when using commands like dir() and such? These are interpreter-generated standard data, which is generated automatically and can be of varying use. For example, if there is a reason to run a certain code block every time an object is created, its possible to define a method named __init__ (two leading and trailing underscores) and define a code block there. That method is always automatically called when new object is created, i.e. initialized. If no __init__-method is defined, interpreter simply slaps on the class definition an empty one.

Other really common automated attribute is the the __doc__. This attribute is created from the document string at the beginning of structure:

```
class Empty:
    """This class is empty."""
    pass
```

This attribute either includes the document string of the class, or is left empty. This attribute can actually be addressed:

```
>>> Keg = Empty()
>>> Keg.__doc__
'This class is empty.'
>>>
```

The third interesting automatically generated class is the __class__. This attribute stores the name of the class whgich was used to create this object. The name also stores the name of the module which created the object; being either the source file name as with the modules was explained or __main__ if the creator was the module which was executed.

```
>>> Teccco = Customer()
>>> Teccco.__class__
<class '__main__.Customer'>
>>>
```

This ability is actually also the reason why the if-structure which identifies the main function works.

## *Class inheritance*

For now, it has been possible to characterize the class as a kind of data structure which is able to perform operations on its data. Even if that is true, there is still yet one unique ability, which has not been discussed, the class inheritance. The class inheritance means that the class is able to take an existing class, copy its attributes and methods and simply redefine and add the new functionalities needed for the class. In more common terms, the new type of class inherits the abilities of the existing class.

The declaration of inheritance is simple. In the class definition, the interpreter is told that this class works exactly like the older class, with these additions and exceptions. The inherited class is given as a parameter to the class definitions, and only the new and modified attributes are written into the new class definition:

```
class [Newclassname] ([Inherited class name]):
    [New class attributes]
    ...
    [New calss methods]
    ...
```

Basically the inheritance is just declared at the definition of a new class. For example, if there is a customer, there may be regular customers, who have joined the store credit program. To create a definition of a bonus-collecting customer, it should be sufficient to just add a few attributes and methods related to the bonus points? Lets take this as an example: here is the old Customer-class defined earlier in the chapter:

```
01  class Customer:
02      name = "John Johnsson"
03      total = 1000
```

```
04      paymenttype = "Masterexpress"
05      number = "1234-5678-9012345"
06
07      def printout(self):
08          print("Name: ", self.name)
09          print("Total: ", self.total)
10         print("Payment type: ", self.paymenttype)
11          print("Card/Bill number: ", self.number)
```

and add the new, inheriting class named Regular like this:

```
01  class Regular(Customer):
02      bonuscard = "ABCD-1234"
03      bonusaccount = 0
04      def bonusdata(self):
05            print("This client has",self.bonusac-
count,"bonus points.")
```

The new object is created by using the more advanced Regular-class to create customer objects with the bonus-collecting abilities. As Regular has inherited all the attributes and methods of the Customer, this new object has both the old and the new abilities:

```
>>> Dave = Regular()
>>> Dave.name = "Dave Davidsson"
>>> Dave.printout()
Name:  Dave Davidsson
Total:  1000
Payment type:  Masterexpress
Card/Bill number:  1234-5678-9012345
>>> Dave.bonusdata()
This client has 0 bonus points.
>>>
```

With inheritance, the first instinct is probably to worry about the possible overwrites of the inherited class methods and attributes.

However, unlike with variables and namespaces, this actually is a documented feature and meant to allow greater control with the inheritance system. If there is a method which would be harmful or broken in the new class, it can be simply redefined in the inheritance. If the new class defines a method or attribute that has the same name as one of the inherited class, then the system uses the definition of a new class. For example, if the Regular-class is extended with a class VIPcustomer, whose account name is meant to be restricted, the offending printing method printout() can be simply replaced:

```
01  class VIPcustomer(Customer):
02      def printout(self):
03         print("VIP client data is confidential!")
```

When we create a new object from this class, the class has all the attributes of the original Customer, but the method printout() is replaced with the one from VIPcustomer:

```
>>> Madonna = VIPcustomer()
>>> Madonna.printout()
VIP client data is confidential!
>>> Madonna.name
'John Johnsson'
>>>
```

The object had preserved all of the attributes and their default values, but the method had changed to the new definition. Lets take one more lager example on this:

## *Example 10-3: Class inheritance and overwriting*

**Source code**

```
# -*- coding: cp1252 -*-
class Cart:
    """This class manages the shopping cart. """

    shoppingcart = []
    def addstuff(self):
        esine = input("What will be added?: ")
        self.shoppingcart.append(esine)

    def checkout(self):
        print("Following objects were added:")
        for i in range(0,len(self.shoppingcart)):
            print(self.shoppingcart[i], end = " ")

class SmallerCart(Cart):
    """This is a small cart with limited space"""
    size = 2
    def checkout(self):
        print("Following was added: ")
        for i in range(0,self.size):
            print(self.shoppingcart[i])
        if len(self.shoppingcart) > self.size:
            print("Some items were left out.")

def main():
    customer = SmallerCart()
    while True:
        selection = input("Add more or go to check-
out?: ")
        if selection == "checkout":
            customer.checkout()
            break
        else:
            customer.addstuff()

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

**Printout**

```
>>>
Add more or go to checkout?: add
What will be added?: Anvil
Add more or go to checkout?: add
What will be added?: Squigg
Add more or go to checkout?: add
What will be added?: Oil
Add more or go to checkout?: add
What will be added?: Bananas
Add more or go to checkout?: checkout
Following was added:
Anvil
Squigg
Some items were left out.
>>>
```

## Inheriting several classes

One additional feature with the classes is the multiple inheritance. This is a bit controversial technique which allows the class the inherit several classes at once. This is achieved with the syntax

```
class NewClassName (Firstinherited, Secondinherited):
    [new attributes]
    [new methods]
```

For example it is possible to combine the two customer classes Regular and VIPCustomer:

```
class VIPRegular(VIPcustomer, Regular):
    idnumber = "00001"
```

This new class and the objects created from it actually have all of the abilities of Customer, Regular, VIPCustomer and VIPRegular:

```
>>> Elton = VIPRegular()
>>> Elton.name
'John Johnsson'
>>> Elton.bonusdata()
This client has 0 bonus points.
>>> Elton.printout()
VIP client data is confidential!
>>> Elton.idnumber
'00001'
>>>
```

However, in many cases the multiple inheritance is not very good idea. Usually the multiple inheritance causes the object to grow almost uncontrollably, causing problems in the long run. For example, if there are several methods and attributes with the same name, which ones end up in the final object? Obviously as with the global variables, sometimes this is the best approach to do some maintenance, but generally this approach should be avoided.

## *Short introduction to the design patterns*

In the chapter discussing the classes and object orientation in general its good to mention a few word regarding the design patterns. Historically the design patters were developed in the 80's, when the size and complexity of the computer programs clearly surpassed the capability of individual programmers or small teams to understand thoroughly. Before this time the programming had been a profession of either individual "artist" programmers or small highly specialized groups, but now there was a

need for some plan or general design principles for making working software.

One of the reasons to come up with some sort of design plans or software architecture was that there simply was not enough programming artists to cater all needs for programmers. Also another thing were the objects, which became popular during the late 80's/early 90's. Even if the different systematic tools or design patters did not live up to the dreams , the design patterns are still interesting things which help with the classes when the programs start to become larger.

## Design patters in short

In short, design patterns are a plan for implementing classes in a pattern that is known to be feasible to implement or generally a good idea. Basically this means that a design pattern is a package solution similar to the architecture solutions for architects or builders.

Design patterns have basically three things: a problem, solution and the requirements. This means that the pattern removes a problem X with a solution Y; provided that the requirements are met. Usually the design patterns are either related to the structure of the program, behavior of the program or creation of the different data models. In short, they are a group of "good ideas" which are known to work well if implemented correctly.

Besides patterns, there also are antipatterns, which quite humorously define ideas which may at first seem great but in practice do not work. These antipatterns for example are "class with too many methods" or "yoyo-model", where class inheritances form

a loop or otherwise untraceable tract. Basically, antipattern is a model for programmers "don’t try this at home", or more precisely, "don’t try this at all.".

## *For the next topics, conclusions*

Obviously, this book can only uncover and explain the very first topics that are covered in the programming as a discipline, or even in the Python programming language. In this book, we have discussed about the following topics:

- Variables
- Iterative structures
- Logical structures
- Functions
- Standard libraries and modules
- Exception handling
- Objects and program structure
- Handling files

The good news is that these topics are basically the fundamental building blocks, that are the foundation of basically all programming languages. The bad news is, that just by learning how to do programming tasks, we have only taken the first step into the software development, and in a larger context, towards software engineering.

Usually the next topic following the fundamentals of programming with Python is to go either towards object-oriented world, since object-oriented is currently the programming paradigm that runs the software industry, or go towards hardware and learn manual data- and memory management with the C-language, since that is where the details can be manually defined. If you are reading this book because this was your course book for the first programming course at a bachelor's

program, the answer most likely will be “both”, but if you are learning things as a hobby, you might want to choose one and see how you like it.

On other ideas, there was a brief discussion over external modules way in the beginning at Chapter 1. Maybe you should try out Pillow, and create programs that can edit image data? Or take a spin at the Pycrypto and see how cryptography works in practise? One additional thing you could try out, is to learn more about network programming, and use sockets to make two computers talk to each other over the network …or ditch the simplified IDE and start developing a game with a suitable game engine, now that you know how to do programming work.

In any case, it does not matter what you choose, as long as you keep practicing. You do not need to understand everything immediately, and you can fail at getting the more advanced examples to work; it takes about a decade to become professional developer, and it takes at least 5 years to learn one programming language to the degree where you no longer need to consult the materials for details and how stuff works, when you step outside the comfort zone of doing something familiar.

You now should possess the skills to start learning about programming independently, and follow the instructions and examples that no longer start from the very beginning. Hopefully you will find this newfound skill useful, and find several interesting projects, with which you can express yourself, and your new superpowers of “computer whispering”.

# About the Author

Associate Professor Jussi Pekka Kasurinen (Doctor of Science in Software Engineering 2011, Adjunct professor of Entertainment Software Engineering 2017) is a long-standing non-fiction author and a researcher, specializing in software development, software testing, quality assurance and the organizational practices. He has more than 15 years of teaching experience from 3 different universities, over 60 international research publications, 5 books (or ebooks) including the world's first book on the esoteric programming languages, and has done industrial cooperation with more than 50 software companies.

Currently, he is working as the head of the degree programmes for LUT University's Software Engineering department.

*Image: An actual Python, not related to the author.*