Let's Code Python

Programming Fundamentals

Orientation Document

Duration: 4 days (23 & 30 April, 7 & 14 May 2022)

Time: 9am to 12pm

Delivery Method: Virtual Instructor Lead Training (VILT)

Course Credits: N/A (Certificate of completion will be awarded at the end)

Google Classroom code: svpqjll

Google Meet code: meet.google.com/xjt-nqjz-oev

Table of Contents

1. Overview

Introduction

Let's Code Python is an initiative geared towards teaching programming fundamentals to an audience that would normally not have access to such knowledge. One of the major driving forces is the shift happening within the technological space and industry worldwide.

To prepare for the fourth industrial revolution (4IR) programming has become more than a special skill but rather a fundamental core skill one needs to be able to position themselves accordingly within the 4IR.

A quick ***disclaimer*** this is not a credited python course but rather a fundamentals course to teach programming fundamentals using python because it is the easiest language to learn. By completing the course no certification will be given only a course completion certificate.

Why Learn Python?

Python is designed to be highly readable. It uses English keywords frequently whereas other languages use punctuation, and it has fewer syntactical constructions than other languages. Python is a MUST for working professionals to become great Software Engineers.

The below list is some of the key advantages of learning Python:

- **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is like PERL and PHP.
- **Python is Interactive** − You can sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** − Python supports an Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** − Python is a great language for beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Characteristics of Python

The following are important characteristics of Python Programming −

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to bytecode for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python Features

- **Easy-to-learn** − Python has few keywords, a simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode that allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries, and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

Target Audience

This course is intended for people who would like to learn to program and may or may not have programming experience.

Prerequisites

Before attending this course, students must have the following:

- Laptop / Desktop with an Internet Connection (The class will be 3 to 4 hours long)
- The Second screen to follow along can be a mobile device or another screen connected to your computer however it is not compulsory.
- Basic computer Literacy Skills
- Some programming experience (recommended but not compulsory)

Course Objectives

After completing this course, students will be able to:

- Learn how Python works and what it is good for
- Understand Python's place in the world of programming languages
- Learn to work with and manipulate strings in Python
- Learn to perform math operations with Python
- Learn to work with Python sequences: lists, arrays, dictionaries, and sets
- Learn to collect user input and output results
- Learn flow control processing in Python
- Learn to write to and read from files using Python
- Learn to write functions in Python
- Learn to handle exceptions in Python

2. How does a computer program work?

This course aims to show you what the Python language is and what it is used for. Let's start from the absolute basics. A program makes a computer usable. Without a program, a computer, even the most powerful one, is nothing more than an object. Similarly, without a player, a piano is nothing more than a wooden box.

Computers can perform very complex tasks, but this ability is not innate. A computer's nature is quite different. It can execute only extremely simple operations. For example, a computer cannot understand the value of a complicated mathematical function by itself, although this is not beyond the realms of possibility soon.

Contemporary computers can only evaluate the results of very fundamental operations, like adding or dividing, but they can do it very fast and can repeat these actions virtually any number of times.

Imagine that you want to know the average speed you've reached during a long journey. You know the distance, you know the time, you need the speed. Naturally, the computer will be able to compute this, but the computer is not aware of such things as distance, speed, or time. Therefore, it is necessary to instruct the computer to:

- accept a number representing the distance.
- accept a number representing the travel time.
- divide the former value by the latter and store the result in the memory.
- display the result (representing the average speed) in a readable format.

These four simple actions form a program. Of course, these examples are not formalized, and they are very far from what the computer can understand, but they are good enough to be translated into a language the computer can accept.
Language is the keyword.

3. Natural languages vs. programming languages

A language is a means (and a tool) for expressing and recording thoughts. There are many languages all around us. Some of them require neither speaking nor writing, such as body language; it's possible to express your deepest feelings very precisely without saying a word.

Another language you use each day is your mother tongue, which you use to manifest your will and ponder reality. Computers have their language, too, called very rudimentary machine language.

A computer, even the most technically sophisticated, is devoid of even a trace of intelligence. You could say that it is like a well-trained dog - it responds only to a predetermined set of known commands. The commands it recognizes are very simple. We can imagine that the computer responds to orders like "take that number, divide by another and save the result".

A complete set of known commands is called an instruction list, sometimes abbreviated as IL. Different types of computers may vary depending on the size of their ILs, and the instructions could be completely different in different models.

Note: machine languages are developed by humans.

No computer is currently capable of creating a new language. However, that may change soon. Just as people use several very different languages, machines have many different languages, too. The difference, though, is that human languages developed naturally.
Moreover, they are still evolving, and new words are created every day as old words disappear. These languages are called natural languages.

4. What makes a language?

We can say that each language (machine or natural, it doesn't matter) consists of the following elements:

- Alphabet: a set of symbols used to build words of a certain language
  (e.g., the Latin alphabet for English, the Cyrillic alphabet for Russian, Kanji for Japanese, and so on)

- lexis: (aka a dictionary) a set of words the language offers its users
  (e.g., the word "computer" comes from the English language dictionary, while "cmoptrue" does not; the word "chat" is present both in English and French dictionaries, but their meanings are different)

- Syntax: a set of rules (formal or informal, written or felt intuitive) used to determine if a certain string of words forms a valid sentence
  (e.g., "I am a python" is a syntactically correct phrase, while "I a python am" isn't)

- Semantics: a set of rules determining if a certain phrase makes sense
  (e.g., "I ate a doughnut" makes sense, but "A doughnut ate me" doesn't)

The IL is, in fact, the alphabet of a machine language. This is the simplest and most primary set of symbols we can use to give commands to a computer. It's the computer's mother tongue. Unfortunately, this mother tongue is a far cry from a human mother tongue. We both (computers and humans) need something else, a common language for computers and humans, or a bridge between the two different worlds.

We need a language in which humans can write their programs and a language that computers may use to execute the programs, one that is far more complex than machine language and yet far simpler than natural language.

Such languages are often called high-level programming languages. They are at least somewhat like natural ones in that they use symbols, words, and conventions readable to humans.
These languages enable humans to express commands to computers that are much more complex than those offered by ILs.

A program written in a high-level programming language is called a source code (in contrast to the machine code executed by computers). Similarly, the file containing the source code is called the source file.

Compilation vs. interpretation

Computer programming is the act of composing the selected programming language's elements in the order that will cause the desired effect. The effect could be different in every specific case – it's up to the programmer's imagination, knowledge, and experience.
Of course, such a composition has to be correct in many senses:

- alphabetically – a program needs to be written in a recognizable script, such as Roman, Cyrillic, etc.
- lexically – each programming language has its dictionary, and you need to master it; thankfully, it's much simpler and smaller than the dictionary of any natural language.
- syntactically – each language has its rules, and they must be obeyed.
- semantically – the program must make sense.

Unfortunately, a programmer can also make mistakes with each of the above four senses. Each of them can cause the program to become completely useless.
Let's assume that you've successfully written a program. How do we persuade the computer to execute it? You must render your program into machine language. Luckily, the translation can be done by a computer itself, making the whole process fast and efficient.

There are two different ways of transforming a program from a high-level programming language into machine language:

COMPILATION - the source program is translated once (however, this act must be repeated each time you modify the source code) by getting a file (e.g., a .exe file if the code is intended to be run under MS Windows) containing the machine code; now you can distribute the file worldwide; the program that performs this translation is called a compiler or translator.

INTERPRETATION - you (or any user of the code) can translate the source program each time it has to be run; the program performing this kind of transformation is called an interpreter, as it interprets the code every time it is intended to be executed; it also means that you cannot just distribute the source code as-is, because the end-user also needs the interpreter to execute it.

Due to some very fundamental reasons, a particular high-level programming language is designed to fall into one of these two categories. There are very few languages that can be both compiled and interpreted. Usually, a programming language is projected with this factor in its constructors' minds - will it be compiled or interpreted?

What does the interpreter do?

Let's assume once more that you have written a program. Now, it exists as a computer file: a computer program is a piece of text, so the source code is usually placed in text files.

Note: it must be pure text, without any decorations like different fonts, colors, embedded images, or other media. Now you must invoke the interpreter and let it read your source file.

The interpreter reads the source code in a way that is common in Western culture: from top to bottom and from left to right. There are some exceptions - they'll be covered later in the course. First, the interpreter checks if all subsequent lines are correct (using the four aspects covered earlier). If the compiler finds an error, it finishes its work immediately. The only result, in this case, is an error message.

The interpreter will inform you where the error is located and what caused it. However, these messages may be misleading, as the interpreter isn't able to follow your exact intentions and may detect errors at some distance from their real causes. For example, if you try to use an entity of an unknown name, it will cause an error, but the error will be discovered in the place where it tries to use the entity, not where the new entity's name was introduced.

In other words, the actual reason is usually located a little earlier in the code, for example, in the place where you had to inform the interpreter that you were going to use the entity of the name. If the line looks good, the interpreter tries to execute it (note: each line is usually executed separately, so the trio "read-check-execute" can be repeated many times - more times than the actual number of lines in the source file, as some parts of the code may be executed more than once). It is also possible that a significant part of the code may be executed successfully before the interpreter finds an error. This is normal behavior in this execution model.

You may ask now: which is better? The "compiling" model or the "interpreting" model? There is no obvious answer. If there had been, one of these models would have ceased to exist a long time ago. Both have their advantages and their disadvantages.

5. What does this all mean for you?

Python is an interpreted language. This means that it inherits all the described advantages and disadvantages. Of course, it adds some of its unique features to both sets.
If you want to program in Python, you'll need the Python interpreter. You won't be able to run your code without it. Fortunately, Python is free. This is one of its most important advantages.
Due to historical reasons, languages designed to be utilized in the interpretation manner are often called scripting languages, while the source programs encoded using them are called scripts.