Data analysis in R and Python

Introduction to programming

Ondrej Lexa Vojtěch Janoušek

Institute of Petrology and Structural Geology
Faculty of Science
Albertov 6

lexa@natur.cuni.cz

vojtech.janousek@natur.cuni.cz

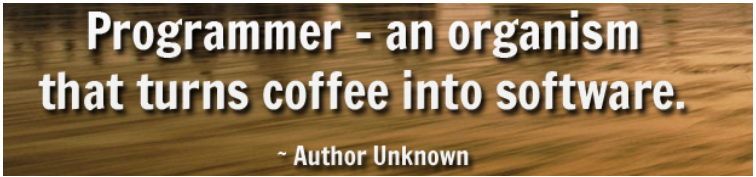
2020

Outline

Why learn programming?

Programming develops creative thinking

Programmer solve a problem by breaking it down into workable pieces to understand it better. When you start learning to program, you develop the habit of working your way out in a very structured format. You analyze the problem and start thinking logically and this gives rise to more creative solutions you've ever given.



Why programming?

Whether you want to uncover the secrets of the universe, or you just want to pursue a career in the 21st century, basic computer programming is an essential skill to learn.

– Stephen Hawking

Everybody in this country should learn how to program a computer... because it teaches you how to think.

– Steve Jobs

Computers are incredibly fast, accurate and stupid; humans are incredibly slow, inaccurate and brilliant; together they are powerful beyond imagination.

– Somebody unknown

What is programming?

Programming is the process of writing instructions that the programming language uses to tell the computer what to do.

Programming is a creative process done by *programmers* to instruct a computer on how to do a task.

Programming is a design algorithms and express algorithms precisely in programs.

Consequently, the **program** is a set of instructions written in a computer language to be carried out by a computer.

That's it, you are ready to go. Pick up a programming language and start learning. Python is recommended to start with, because it's beginner-friendly, nevertheless you will be introduced to R as well.

Programming languages

The programming language is:

- artificial language
- created by man
- to describe actions which must be done with computer

The **programming language** is just a tool to express thought (more precisely, an algorithm that represents a solution to any problem) to the computer. So it really doesn't matter much, what programming language you use; your choice generally depends on personal experience/ knowledge about the specific programming language.

The **programming language** is a system for describing computation in machine-readable and, at the same time, human-readable form.

Programming languages are easier than natural languages. Thay have fewer constructs and they do not have ambiguities.

Programming languages

Most computer programs are written in **high-level programming languages** such as Python, Java, R, C++, Ruby etc.

- High-level languages are made up of human-readable statements that make it easier for us to program
- Computers can only process instructions in the form of binary numbers. This is known as machine code and is an example of a **low-level programming language**
- Programs written in high-level languages must be converted into low-level machine code by a type of software called a **translator**

High-Level Languages

- Human-readable
- Must be translated for the CPU to execute
- Often "portable", meaning it can run on different types of CPU
- One statement represents many CPU instructions

Low-Level Languages

- Hard to read by humans
- Provides exact control over the CPU
- Will only run on one type of CPU
- Faster for computers to run

Example of high-level language - Python

```
## This is a test to  understand which is higher value
y = 4
x = 7

if y >= x:
    print('Yes')
else:
    print('No')

## In this example you are adding two test examples

x = 8
y = 8

if y > x:
    print('Y is greater')
elif y < x:
    print('X is greater')
else:
    print('They are the same')
```

Example of low-level language - Assembly language

Machine code bytes	Assembly language statements
	`foo:`
`B8 22 11 00 FF`	`movl $0xFF001122, %eax`
`01 CA`	`addl %ecx, %edx`
`31 F6`	`xorl %esi, %esi`
`53`	`pushl %ebx`
`8B 5C 24 04`	`movl 4(%esp), %ebx`
`8D 34 48`	`leal (%eax,%ecx,2), %esi`
`39 C3`	`cmpl %eax, %ebx`
`72 EB`	`jnae foo`
`C3`	`retl`

The Purpose of Translators

Translators are a special type of software that translate (convert) computer programs from a higher level language to low-level machine code that the CPU can execute (run).

Translators are essential because a CPU can only process instructions in the form of binary numbers and that are a part of its own instruction set.

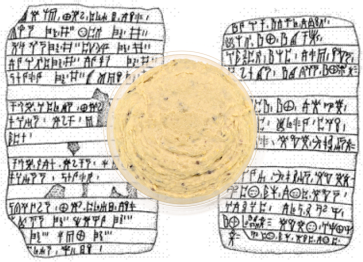
There are two types of translator, each of which works slightly differently:

- **Interpreters** translate high-level source code into low-level machine code one instruction at a time.
- **Compilers** translate the whole of a program written in a high-level language into machine code in one go.

Compilers and interpreters take human-readable code and convert it to computer-readable machine code.

Okay... but what does that actually mean?

Imagine you have a hummus recipe that you want to make, but it's written in ancient Greek. There are two ways you, a non-ancient-Greek speaker, could follow its directions.



The first is if someone had already translated it into English for you. You (and anyone else who can speak English) could read the English version of the recipe and make hummus. Think of this translated recipe as the **compiled version**.

The second way is if you have a friend who knows ancient Greek. When you're ready to make hummus, your friend sits next to you and translates the recipe into English as you go, line by line. In this case, your friend is the interpreter for the **interpreted version** of the recipe.

Compiled Languages

Compiled languages are converted directly into machine code that the processor can execute. As a result, they tend to be **faster** and **more efficient to execute** than interpreted languages. They also give the developer **more control over hardware aspects**, like memory management and CPU usage.

Compiled languages need a "build" step – they need to be manually compiled first. You need to "rebuild" the program every time you need to make a change. In our hummus example, the entire translation is written before it gets to you. If the original author decides that he wants to use a different kind of olive oil, the entire recipe would need to be translated again and resent to you.

Examples of pure compiled languages are C, C++, Erlang, Haskell, Rust, and Go.

Interpreted Languages

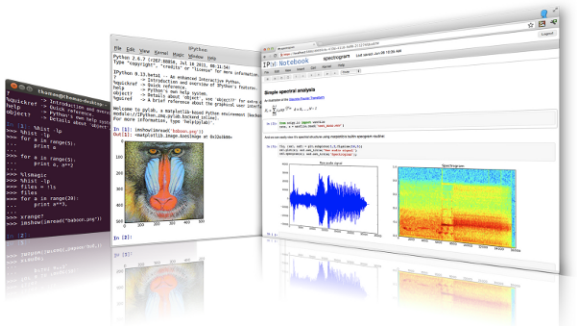
Interpreters run through a program line by line and execute each command. Here, if the author decides he wants to use a different kind of olive oil, he could scratch the old one out and add the new one. Your translator friend can then convey that change to you as it happens.

Interpreters do not produce a compiled program file, so the process of translation must be repeated every time the program is run. Interpreted languages were once significantly slower than compiled languages. But, with the development of *just-in-time compilation*, that gap is shrinking.

Examples of common interpreted languages are Python, R, PHP, Ruby and JavaScript.

Scientific programming languages

- designed for mathematical and statistical computations
- extensive use of matrices
- sophisticated graphical functions
- high-level graphical output
- e.g. ALGOL, FORTRAN, Python, R, Matlab, Julia ...



How to learn a programming language?

Anyone can start to learn a programming language, but to learn it effectively you need to learn the **building blocks of a programming language**.

- **Syntax**
- **Semantics**
- **Data types**
- **Data structures**
- **Terms**
- **Algorithms**

Syntax and semantics

- **English language**

  Syntax is the rules for how a sentence is constructed.

  Semantics is the actual meaning of statements.

- **Programming language**

  Syntax is the rules for how each instruction is written.

  Semantics is the effect the instructions have(logic).

Data types and data structures

A **data type** is a classification of data we want to store in memory. Data types can vary from one language to another. But the type of data we would like to store is common across all languages.

Most programming languages support basic data types of integer numbers (of varying sizes), floating-point numbers (which approximate real numbers), characters and booleans.

A **data structure** is how we can store, access, organize and manage the data we have created in a computer.

Terms

When it comes to a programming language there are these terms(buzzwords) you may come across in your learning journey. Some commonly used terms are **variables**, **expressions**, **statements**, **functions**, **classes** and many more.

In most languages, statements contrast with expressions in that statements do not return results and are executed solely for their side effects, while expressions always return a result and often do not have side effects at all.

You don't need to worry if you feel any difficulties, use this dictionary as a resource.
`https://hackr.io/blog/programming-terms-definitions-for-beginners`

Algorithm

Algorithm is a finite sequence of steps that solves a specific problem.

Let's say that you have a friend arriving at the Prague airport, and your friend needs to get from the airport to faculty.

The following are two different algorithms that you might give your friend for getting to faculty:

The taxi algorithm:

- Go to the taxi stop.
- Get in a taxi.
- Give the driver the faculty address.

The bus algorithm:

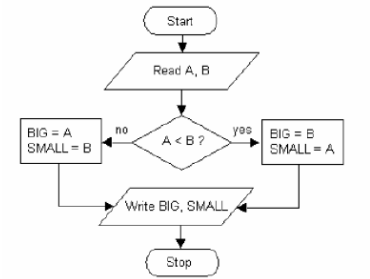
- Outside Terminal 1, take bus 191.
- Transfer to tram no. 7 at Anděl.
- Get off at Albertov station.
- Walk 200 m east to faculty.

Computer program algorithm

There are two commonly used tools to help to document computer program logic (the algorithm):

- Flowcharts - diagrammatic representation of an algorithm
- Pseudocodes - plain language description of the steps in an algorithm

Given a problem to write a program that reads two numbers and displays the numbers read in decreasing order.




```
Read A, B

If A is less than B
   BIG = B
   SMALL = A
else
   BIG = A
   SMALL = B

Write (Display) BIG, SMALL
```

The Zen of Python, by Tim Peters

```
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!
```