# Course Introduction

*1DV501: Introduction to programming*

Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

The slides are available in Moodle

August 30, 2020

# Agenda (Today ...)

- Course information
- Computer systems
- The Python programming language
- Getting started with Python

**Important**
Please register yourself using the Lnu online system reachable from MyMoodle. Contact me (Jonas.Lundberg@lnu.se) and provide name, Swedish ID, Lnu username, if you run into problems.

**Reading Instructions:** 1.1-1.3 in book by R.L. Halterman

# Course Information

- 1DV501: Introduction to Programming (7.5 credits)
- A beginners course in Python programming
- Teacher: Jonas Lundberg (Examiner)
- Literature: Fundamentals of Python Programming, Richard L. Halterman, Draft March 29, 2019
  **Available as pdf in Moodle**.
- Moodle – a web based course information system
  - News
  - Lecture slides
  - Reading instructions
  - Assignments
  - Assignment submission system
  - Chat forum (Slack)
- Moodle is accessed from `http://mymoodle.lnu.se`
- **Important:** Mail me your Name + Swedish ID number + LnU username if you can't access Moodle.

# Course Setup

**Notice**: The course is given in 2 versions: In English and in Swedish. Also, we have a group of students in Kalmar $\Rightarrow$ **make sure to follow correct instructions/schedules**

- 10 lectures, 2h/lecture
  - In English: Monday and Thursdays
  - In Swedish: Monday and Wednesday
- Tutoring sessions: 2 meetings each week
- No written exam!
- Pass requires: Pass all assignments + Pass the mini-project + Pass Python Test
- 3 opportunities to pass assignments and Python Test (October, November, December)
- Final grade: Mini-project(60%) + Assignments(40%)
- Assignment 1 and 2 are graded as Fail or Pass,
  Assignment 3 and Mini-project are graded using ECTS grades: A-F
  ($A$ = Brilliant!, ..., $E$ = Sufficient to Pass, $F$ = Fail)

# Lectures

- 10 lectures, 2h/lecture
  - In English: Monday and Thursdays
  - In Swedish: Monday and Wednesday

- Teachers
  - Växjö: Jonas Lundberg (jonas.lundberg@lnu.se)
  - Kalmar: Fredrik Ahlgren (fredrik.ahlgren@lnu.se)

**Important**

- Limited lecture room capabilities in Växjö due to corona restrictions
  $\Rightarrow$ All students can not attend all lectures $\Rightarrow$ about 60% of all students will not be able to attend the lectures
- **You must sign up on a list in Moodle to attend Växjö lectures**
- Limited capabilities $\Rightarrow$ first come, first served
- A new "I would like to attend the lecture" list is published before each lecture
- **In Kalmar**: no such problems $\Rightarrow$ all students are welcome
- However, all lectures will be streamed (live), recorded, and made available in Moodle $\Rightarrow$ all students are given access to the lecture material

# Tutoring Groups

Each study program forms a tutoring session group with an individual time schedule. Certain programs are further split in a Swedish and an English version.

- Computer Engineering, TGI1D, Supervisor: Jonas Lundberg and Simon Bäcklund
- Electrical Engineering, TGI1E, Supervisors: Jonas Lundberg and Simon Bäcklund
- Software Engineering, TGI1V, Supervisors: Fredrik Ahlgren, Amer Hodzic and David Mozart
- Master of Science in Engineering: Software Engineering, CIDMV, Supervisor: Morgan Ericsson
- Network Security (English), NGDNS, Supervisors: Ola Flygt and Måns Regin
- Network Security (Swedish), NGDNS, Supervisors: Ola Flygt and Måns Regin
- Software Technology (English), NGDPV, Supervisors: Tobias Andersson-Gidlund and Albin Ljungdahl
- Software Technology (Swedish), NGDPV, Supervisors: Tobias Andersson-Gidlund and Albin Ljungdahl
- Applied Mathematics, NGMAT, Supervisors: Jonas Lundberg and Simon Bäcklund
- Others, all handled remotely, Supervisors: David Mozart and Amer Hodzic

# Tutoring Sessions

- We have divided all students into separate groups
- Each such group has a separate course schedule
  $\Rightarrow$ read the time-plan carefully.
- **Attending the tutoring sessions is very important**
- Assignments 1 and 2 will be graded during the sessions $\Rightarrow$ you must show up
  - Attending meetings $\Rightarrow$ 80% will pass the course in November
  - Work at home $\Rightarrow$ 50% will pass the course in November

**The Python Test**

- A practical programming test
- 2-3 exercises, 2 hours
- Supposed to be simple for everyone having handled Assignments 1-3.
- Preliminary date: Friday October 23

Part of course examination $\Rightarrow$ you must pass to pass the course

# Assignment Rules

- **The assignments are individual. Each student submit their own set of solutions**
- Pass requires solutions (a good attempt at least) on all required exercises.
- Certain exercises in Assignment 1 and 2 are marked as **If time permits**. These exercises are not mandatory (but recommended).
- **You must submit your assignments before the deadline**
- Certain exercises in Assignment 3 are marked as **VG-exercises**. These exercises are not mandatory but required to get the grades A or B.
- **You must submit your assignments before the deadline**
  - Not presenting Assignment 1 solutions $\Rightarrow$ not active $\Rightarrow$ you will be removed from the course
  - Not complete submission $\Rightarrow$ your exercises will be corrected $\Rightarrow$ you will have to resubmit a complete solution at a later time
- Exceptions can be given if you, within a reasonable time *before* the actual deadline, contact your assignment supervisors and give a reasonable explanation.
- Exchange ideas, not solutions. **Plagiarism** $\Rightarrow$ students who copy (parts of the) programs from colleagues or from elsewhere, or get others (friends, relatives, hired skilled persons) to complete their assignments for them, fail the assignment automatically.
- **Read Assignment Rules in Moodle carefully!**

# Slack and Moodle

The Moodle website for 1DV501/1DT901 is our main source for publishing information ⇒ one-way communication

- Assignments
- Lecture slides
- Lecture videos
- Pinned news (e.g. news about an upcoming exam)

Slack is our chat forum ⇒ two-way communication

- **Slack as our main channel of communication in this course**
- Assignment or course related questions/discussions
- Visited daily by teachers/assistants ⇒ expect a fast response
- Students are encouraged to help each other
  (but please don't post assignment solutions)
- We want you to react ("like") to all pinned posts from the teachers
- Use the function "Reply in thread". Do not start a new thread for all questions

**Do not ask course related questions via email.** Ask them in Slack. Only personal questions in e-mail.

# Questions?

Any questions about the course setup?

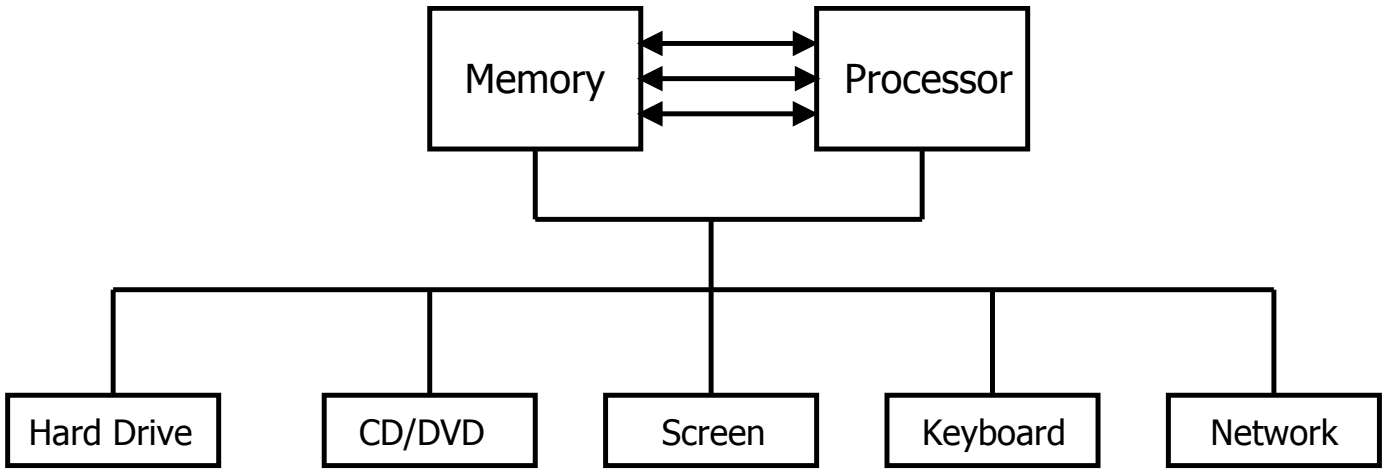
# Computer Systems

Most *Computer Systems* consists of

1. **Hardware**
   - Physical parts you can "touch"
   - Processor
   - Primary memory
   - Hard drive
   - Memory card
   - Buses (cables inside computer)
   - Keyboard
   - Screen
2. **Software**
   - Software = programs and their data
   - Operating System
   - Other programs (applications)

- Hardware and software are useless without each other
- This course is about **software development**
- We present a minimum of hardware

# Hardware




- **(Micro)Processor** (also *CPU = Central Processing Unit*)
  A chip performing simple computations very fast
- **(Primary) Memory** (also *Main Memory or RAM*)
  Memory where program/data are stored while processed. Small with fast access.
- **Hard drive**
  Permanent storage for programs/data not currently used. Large and slow.
- The hard drive is a **secondary memory** where information can be stored permanently $\Rightarrow$ can survive without electricity
  Other types of secondary memory: SSD, CD, DVD, USB Memory stick, etc

# Execution = Running a program

- Execution ⇒ read a program from hard drive to primary memory and start to process its instructions.
- The processor consists of
  - a **arithmetic/logic unit** processing instructions
  - a **register**: very small/fast memory where intermediate data/results are stored
- An executable program = a sequence of instructions in **binary form**

```
C = A + B                                  Binary form
**********                                 ***********
1. Read value of A to R(1)                 1. 01010101 10110010 11000110 01001100
2. Read value of B to R(2)                 2. 00100010 11100111 00101010 10001111
3. Compute R1+R2 and save in R(3)          3. 10010010 11100100 11001001 11010100
4. Assign C value in R(3)                  4. 10101011 00100101 11000011 10010001

Where R(N) = Register number N
```

- Binary form ⇒ instructions are sequences of 0 and 1 that only (almost) a computer can understand.
- Different types of processors have different set of instructions

# Digital information

- Digital comes from the word *digit* = 0,1,2,3,4,5,6,7,8,9
- All information on a computer are stored as integers
- Apart from instructions, it also holds for:
  - floats (e.g. decimals like 2.56 or $-0.0045$)
  - text
  - sound
  - pictures
  - videos
- Each picture pixel is stored as three integers (red,green,blue).
- Text: Each character is stored as an integer.
- Computers store all integers in **binary form**.
  For example, 14 is stored as 1110.

# Number Systems

## Decimal Numbers

- Ordinary (decimal) numbers have the *base* 10.
- We use the digits 0-9 to describe a number
- $234 = 2 * 100 + 3 * 10 + 4 * 1 = 2 * 10^2 + 3 * 10^1 + 4 * 10^0$
- Each digit (2,3 or 4) has a specific *weight*.
- Decimal $\Rightarrow$ we use weights $..., 10000, 1000, 100, 10, 1$
- $\Rightarrow$ the $n$:th digit from the end has weight $10^{n-1}$

## Binary Numbers

- The binary numbers have the *base* 2.
- We use only the digits 0 an 1 to describe a number.
- $1110_2 = 1*2^3+1*2^2+1*2^1+0*2^0 = 1*8+1*4+1*2+0*1 = 14_{10}$
- That is, 1110 in base 2 equals 14 in base 10.
- Binary weights: $\ldots, 16, 8, 4, 2, 1$
- $\Rightarrow$ the $n$:th digit from the end has weight $2^{n-1}$

# Binary Numbers

- A binary digit (0 or 1) is called a *bit*
- 8 bits forms a *byte*
- Using a byte, we can represent 256 integers between 0 (0000000) and 255 (11111111)
- In general: Using $n$ bits we can represent $2^n$ integers between 0 and $2^n - 1$
- Negative numbers: The first bit gives the sign ($1 = +$, $0 = -$), the remaining bits give the size.

**Why binary numbers?**

- Hardware rules! Much easier, faster, and safer to transport and store information in binary format
- In cables (transport): Voltage on $= 1$, voltage off $= 0$
- In memories (storage): Magnetization up $= 1$, magnetization down $= 0$
- Memory: 1 GB (GigaByte $= 2^{30}$ bytes, more than a billion bytes)
- Hard drives: 1 TB (TeraByte $= 2^{40}$ bytes, more than 1000 billion bytes)
- Furthermore : 1KB $=$ 1 KiloByte $= 2^{10} = 1024$ bytes, 1MB $=$ 1 MegaByte $= 2^{20} = 1048576$ bytes,

# Example: Text Encoding

- Each character (alphabet + others) is represented by an integer
- In most cases we use 1 byte/character $\Rightarrow$ 255 possible characters
- There are a few different encoding standards. Most common is ASCII.

```
Hello Sweden!

H=72, e=101, l=108, l=108, o=111, " "=32,

S=83, w=119, e=101, d=100, e=101, n=110, !=33. linebreak=10
```

- Also non-visible characters like space, tab and line break (Enter) have an integer encoding.
- A text file is a long sequence of bytes where each byte represents a character.
- A text editor (like Wordpad) converts between characters and integers whenever a file is read/saved.

# Software

The computer software is divided in two parts:

1. **Operating System:** The program that controls the computer
   - Starts the computer
   - Distributes processing power between programs
   - Provides a GUI (Graphical User Interface) for the user
     $\Rightarrow$ We can control the computer by "mouse clicks"
   - A software layer between hardware and other programs.
     $\Rightarrow$ gives other programs controlled access to hardware
     $\Rightarrow$ protects hardware
   - Common OS: Windows, Mac OS, Unix, Linux
2. **Applications:** All other programs
   - Word processing
   - Games
   - Software controlling a car engine
   - MP3 players, anti-virus software, email client, ...
   - Internet Explorer, Visual Studio Code, Excel, ...
   - Solves a specific task by utilizing the hardware accessed through the OS.

# Programming Levels

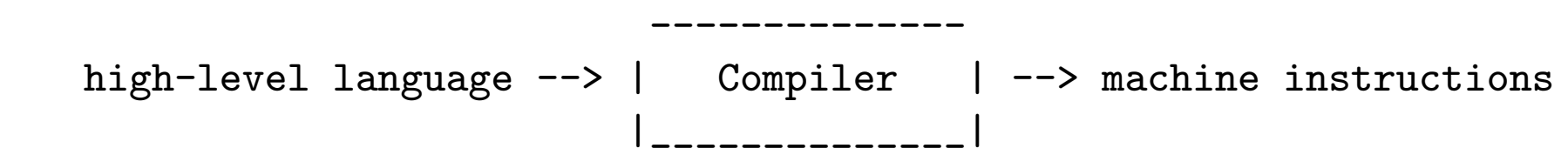
- The processor only understands binary instructions
  (also called *machine code* or *machine instructions*)
- Humans have a hard time understanding binary instructions
- 1950s: Assembler code

```
Assembler (C = A + B)      Binary form
=========                  ==========
1. Load A R1               1. 11100111 00101010 10001111
2. Load B R2               2. 11100111 11001001 11010100
3. Add  R1 R2 R3           3. 10101011 10001111 11010100 00100010
4. Save R3 C               4. 00100101 00100010 10010001
```

- Assembler: Give each binary instruction a name.
- Simple to translate to binary form (one bit assembler $\Rightarrow$ one instruction)
- Assembler is still in use to control hardware

# High-level Languages

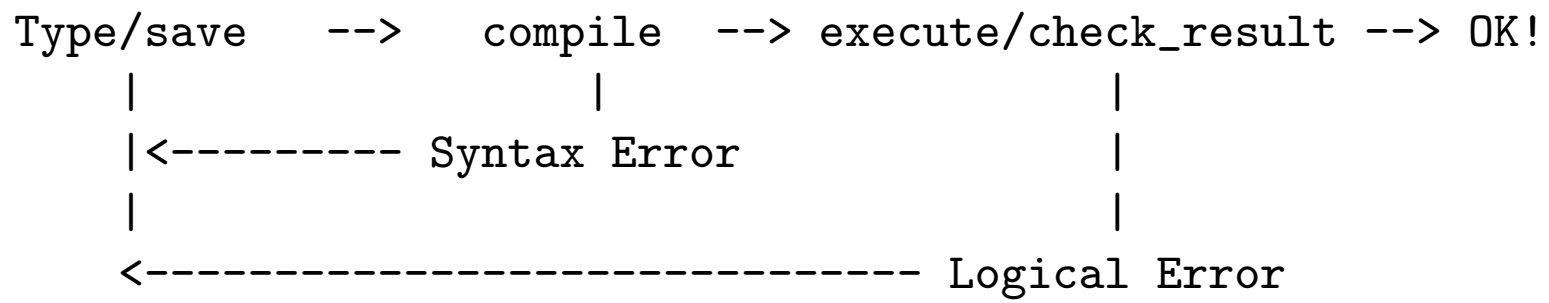
- Assembler is easier than binaries, but still difficult/slow for humans
- The programmer must know everything about the hardware
- 1960s: High-level languages where each instruction represents many machine instructions.
- Common HL languages: C, Ada, Pascal, C++, C#, Java and **Python**
- Converted to machine instructions by a **compiler**
  $\Rightarrow$ a program translating high-level language to machine instructions




- A compiler is specific for: one high-level language, and one type of machine instructions
- The compiler knows about machine instructions $\Rightarrow$ we don't have to.
- The compiler start by checking if the input program is correct
  $\Rightarrow$ `Error: Missing semi-colon on line 234`

# High-Level Programming

- A **programming language** has a given **syntax**.
- Syntax are rules about:
  - which words and symbols that can be used in a program
  - how to combine words/symbols into correct **statements**
- A program is in principle a sequence of statements
- Programming (in theory)
  1. Type (edit) the program as ordinary text
  2. Save the program in a file
  3. Compile the program ⇒ generate machine code
  4. Execute the program
- Programming in practice

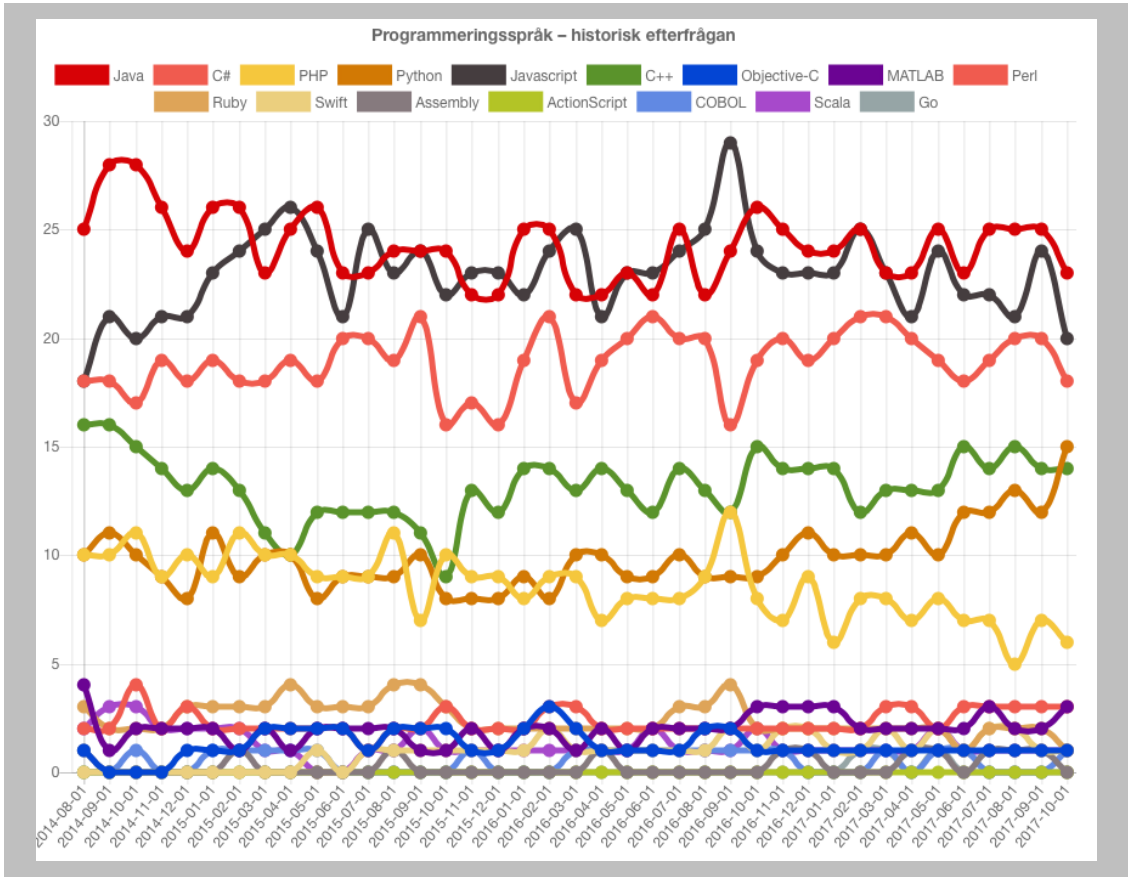


# Why learn how to program?

- Computer ⇒ an extremely powerful problem solving tool
- Programming ⇒ take control over the computer
- Non-programmers ⇒ must rely upon other people's program
- Little programming (less than 1000 lines of code)
  - short scripts for web pages
  - simple device drivers (to control machines)
  - rather complex scientific computations
  - understand what programming is all about
- Much programming (millions of lines of code)
  - develop commercial software, for example
  - games
  - administrative tools program
  - missile system
  - search engine
- **Notice:** Develop robust and user-friendly programs for non-experts require a lot of programming.

# No programming language is the best but ...

No language is the best but some languages are in higher demands than others




Q: And which language should you learn then

A: Learn to program, to understand important principles, and to solve problems!

# A 10 minute break!

zzzzzzzzzzzzz ...

# The Python Programming Language

- Created by Guido van Rossum and first released in 1991
- Python's design philosophy emphasizes code readability
- Python 2.0 was released in 2000, and Python 3.0 in 2008
- Current version is Python 3.8
- Python is run by the Python Software Foundation (at `python.org`)

- Python is a high-level, interpreted and general-purpose language ...
- ... that focuses on code readability.
- The syntax rules in Python helps the programmers to do coding in fewer steps as compared to Java or C++.
- Python is less difficult to learn compared to Java, C#, ...
  $\Rightarrow$ Python is suitable for a beginners course in programming

- **My opinion** (and many agree)
- Use Python for small programs and prototypes
- Use Java, C#, ... for larger software systems

# Java vs Python

Programs in Java and Python that prints `Hello` on the screen

**Hello.java**

```
public class Hello {

    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

Each Java program requires a class declaration (`class Hello`) and a main method declaration to work. Printing is done using the `System.out` stream.

**hello.py**

```
print("Hello")
```

No additional structure is needed. Just insert statements that you would like to execute. In this case `print("Hello")`

# The structure of a (simple) Python program

We use a text editor and writes the following code in a file named `simple.py`

```
# Add two integers and print the result
a = 10
b = 15
sum = a + b  # Compute sum of a and b
print(sum)   # Print result
```

- All Python programs must be saved in a file named "something" `.py`
- `#` $\Rightarrow$ a comment reaching to the end of the line
- `a, b, sum` are variables
- `print(...)` is a library function used to print results to the screen

Just an example, we will start to learn Python in Lecture 2.

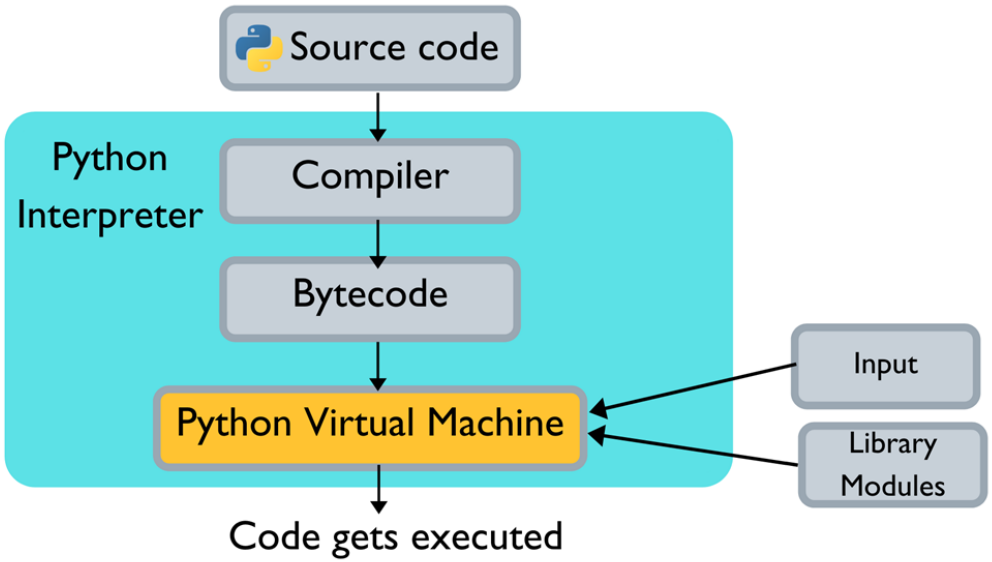
# Syntax and Semantics

- The **syntax** rules define how we can combine keywords, identifiers, and other language constructs into a correct program.
- Syntax errors are discovered by the compiler.

```
Find 5 Errors!
==============
% Add two integers and print the result
a = 10
b + 5 = 15
sum = A + b
Print(sum          # Print result
```

- The language **semantics** defines the functionality of all language constructs.
- Example: The semantics of `print("Hello")` is to print `Hello` on the screen.
- **Notice:** A syntactically correct program is not necessary logically correct.
- A computer does exactly what we *tell it* to do,
  not necessary what we *want it* to do.
- The computer (almost) never makes a mistake.
  It is we (programmers) who give it incorrect instructions.

# Python Execution




Executing `hello.py` ⇒ Start the Python Interpreter and provide `hello.py` as input.
Inside the interpreter:

1. The compiler (a program) checks if the input program is a correct Python program (and terminates with an error if something is wrong)
2. The compiler translates the source code to an internal format called bytecode
3. The Python Virtual Machine (PVM) takes the bytecode as input and executes the program
4. The PVM knows about library modules and can read input from the keyboard (or from files)

# Getting Started - Recommended Approach

1. **Buy a laptop if you don't have one!**
2. Install Python on your laptop using Anaconda (Individual Edition)

   `https://www.anaconda.com/products/individual`

   Verify that it works by writing `python` in your Terminal/Console prompt.

   ```
   jlnmsi % python
   Python 3.8.3 (default, Jul  2 2020, 11:26:31)
   [Clang 10.0.0 ] :: Anaconda, Inc. on darwin
   Type "help", "copyright", "credits" or "license" for more information.
   >>> exit()  # terminate python
   ```

3. Install Visual Studio Code (VSC) on your laptop

   `https://code.visualstudio.com`

4. Install the Microsoft Python Extension on VSC

   `https://marketplace.visualstudio.com/items?itemName=ms-python.python`

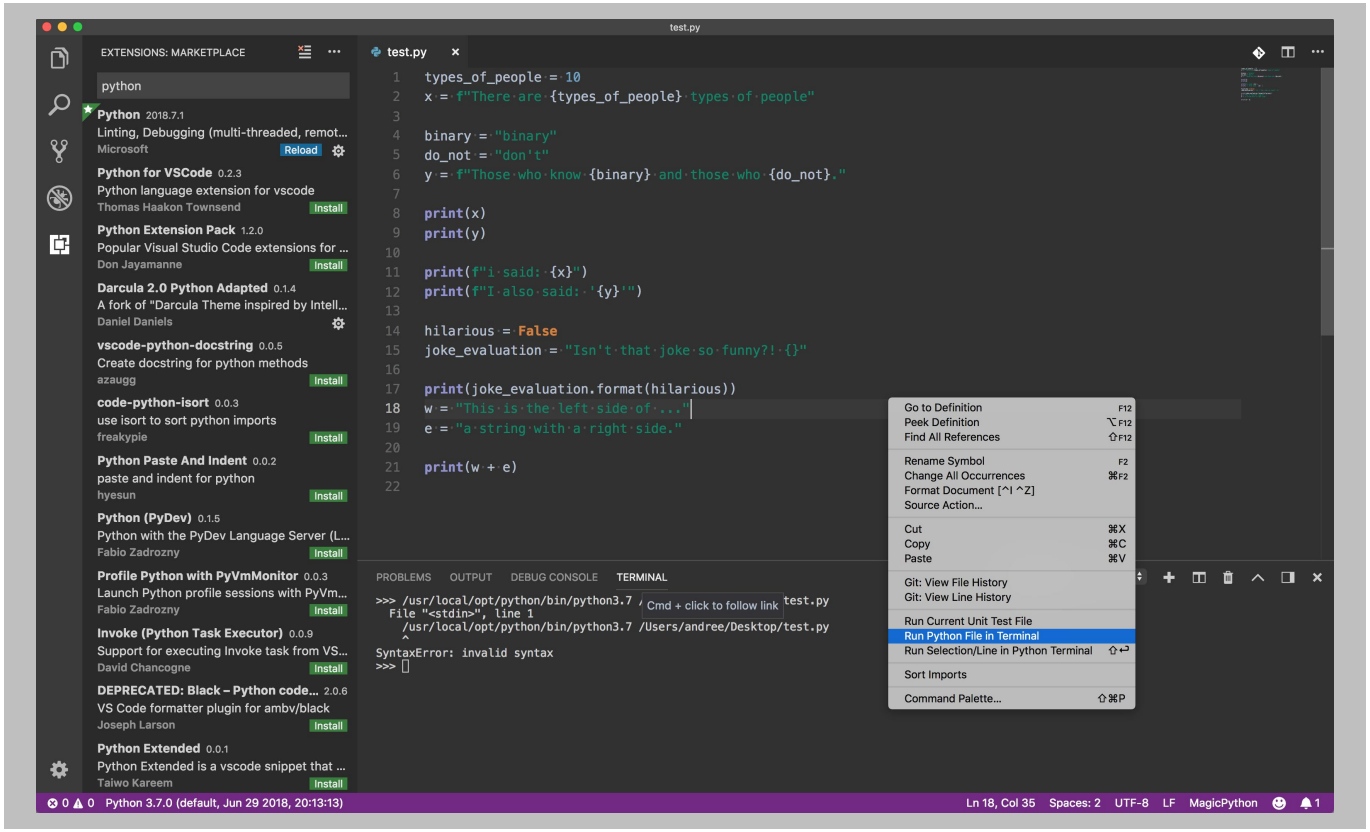
5. Setup a folder structure for Python on your laptop
6. Create a file `hello.py` and try to run it in VSC

# Anaconda

- Anaconda is a Python distribution ⇒ it contains all you need (and much more) to run a Python program ⇒ a Python interpreter and a large number of library modules (and much more)
- Advantage: Easy to download and install
- Advantage: Everything we need in this course
- Disadvantage: Much more than we need in this course
- Verify that it works by writing `python` in your Terminal/Console prompt.

```
jlnmsi % python
Python 3.8.3 (default, Jul  2 2020, 11:26:31)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()  # terminate python
```

# Visual Studio Code + Python Extension




- Visual Studio Code is an Integrated Development Environment (IDE) ⇒ a software tool to simplify programming ⇒ provides support for program execution and organizing all program files in a project (and much much more)
- We will use the Python Extension, VSC supports many other programming languages (e.g. Java).

# Python Programming - Three approaches

1. Edit a Python file `X.py`, save file, and execute. Use a tool like VSC to simplify the process
2. Write your program in the Terminal/Console window
3. Use Jupyter Notebook $\Rightarrow$ a web-based interactive computational environment for creating Python programs

**Notice**

- Edit, save, execute (Approach 1 above) is the traditional approach used in most companies and similar to the approach used in other languages
- The Console/Terminal approach only works for very small programs. Used mainly to test and try various commands and modules.
- The Jupyter approach is great for small to medium projects

We will stick to the traditional approach. It is the default and it is the way to do it in other programming languages.

The textbook by Halterman often uses the Console/Terminal approach to show simple examples. Get used to reading it but we always use the traditional approach in lectures and assignments.

# Upcoming Activities

- Next Lecture
  - English: Thursday, September 3, at 10.15
  - Swedish: Wednesday, September 2, at 10.15
  - **Remember to sign up if you plan to attend lecture**
- First tutoring session
  - Each program has their own schedule
  - $\Rightarrow$ Look it up in Moodle
  - Simple Python programs
  - The VSC Development Environment
  - Bring your laptop (if you have one)
- **Before Next Activity**:
  - Try to install Anaconda at home
  - Try to install Visual Studio Code at home
  - Follow instructions (URLs) given in Assignment 1 (in Moodle)