Introduction to Informatics for Students from all Faculties

Prof. Dr. Ingo Scholtes
Chair of Machine Learning for Complex Networks
Center for Artificial Intelligence and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
Würzburg, Germany
ingo.scholtes@uni-wuerzburg.de

Julius-Maximilians-UNIVERSITÄT WÜRZBURG

Lecture 04
Programming Languages

November 12, 2024

Notes:

- **Lecture L04: Programming Languages** 12.11.2024
- **Educational objective:** We introduce high-level programming languages and explain the difference between compiled and interpreted languages.
  - OS User Interfaces
  - Machine Instructions and Assembly Language
  - High-Level Programming Languages and Compilers
  - Interpreted Languages: Python
- **Exercise Sheet 3** due 26.11.2024

Motivation

- we have taken a **top-down approach** to study the hardware/software interface
- we investigated how programs are executed at the **level of machine code**
- we introduced key functionality of **operating systems** and discussed the abstraction of **processes**
- we discussed how multi-tasking allows to execute **multiple processes simultaneously**

open questions

- how can **humans interact with the operating system**?
- how can we write programs that **solve actual problems**?
- how can we translate **code that is understandable for humans** to instructions that can be executed by the CPU?

```
64a:   55                     push   %ebp
64b:   48                     dec    %eax
64c:   89 e5                  mov    %esp,%ebp
64e:   48                     dec    %eax
64f:   83 ec 10               sub    $0x10,%esp
652:   48                     dec    %eax
653:   8d 05 ab 00 00 00      lea    0xab,%eax
659:   48                     dec    %eax
65a:   89 45 f8               mov    %eax,-0x8(%ebp)
65d:   48                     dec    %eax
65e:   8b 45 f8               mov    -0x8(%ebp),%eax
661:   48                     dec    %eax
662:   89 c6                  mov    %eax,%esi
664:   48                     dec    %eax
665:   8d 3d a7 00 00 00      lea    0xa7,%edi
66b:   b8 00 00 00 00         mov    $0x0,%eax
670:   e8 ab fe ff ff         call   520 <printf@plt>
675:   b8 00 00 00 00         mov    $0x0,%eax
67a:   c9                     leave
67b:   c3                     ret
67c:   0f 1f 40 00            nopl   0x0(%eax)
```

A simple Hello World program in machine code

Notes:

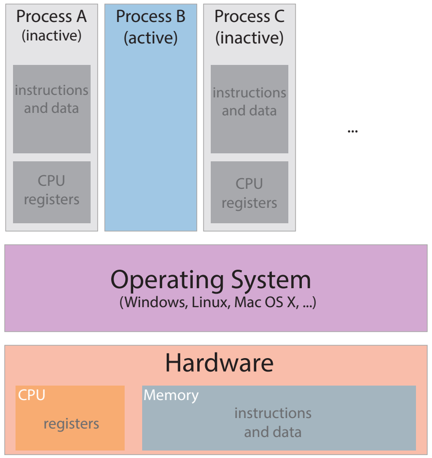
Reminder: Multi-Tasking

- OS can use **multi-tasking** to execute **multiple processes concurrently** (even on a single CPU)
- every few milliseconds, OS performs **context switch** between running processes
- context switch from process A to B requires to **switch execution context**

context switch from process A to B

1. interrupt execution of program by CPU
2. save current values in CPU registers (incl. PC) to memory, which fully determine execution state of process A
3. restore previously saved CPU registers of process B from memory
4. continue execution of program by CPU

- **OS scheduler** fairly allocates CPU time
- **preemptive scheduler** forces context switches



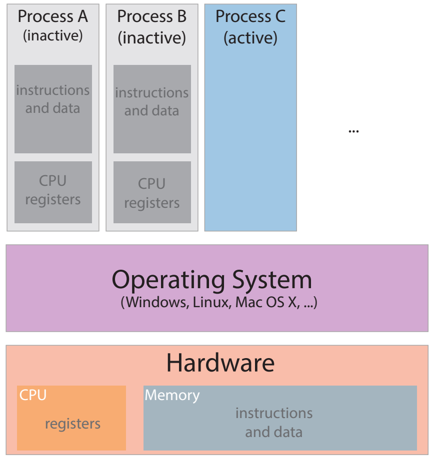
Launching a process

- we can use OS to **launch a new process** that executes a program
- reminder: process = **one instance of program** executed by CPU

launching a proces

1. OS reads **"executable file"** from hard drive/SSD and copies it into main memory (RAM)
2. "executable file" contains **machine instructions and data**
3. OS sets **program counter of CPU** to address of first machine instruction in main memory
4. OS **transfers control to CPU** (until next context switch)

- how can we tell OS to launch a new process?



Notes:

1. The opposite of preemptive scheduling is called coooperative scheduling. This means that a context switch can only happen if a process "voluntarily" surrenders the CPU periodically, such that another process can take over. Early operating systems like Windows (before Windows 95) or Mac OS (before Mac OS X) in the 1990s used cooperative scheduling, which intreoduced the problem that the whole computer freezes if a single process is implemented badly.

Notes:

Graphical User Interfaces (GUI)

- modern operating systems provide an intuitive and human-friendly **graphical user interface (GUI)**
- key functions of OS (e.g. launching a process) can be accessed in an intuitive way (e.g. by double-clicking program icon with the mouse)
- OS provides special program (e.g. file explorer or finder) to **manage files on permanent storage** (hard drive, SSD) or network shares
- multi-tasking is typically represented by **multiple program windows or icons** that represent **running processes**

definition

A **graphical user interface (GUI)** provides access to the functions of a program or OS by allowing the user to manipulate visual icons and indicators, typically by means of a touch pad, touch screen, or mouse.

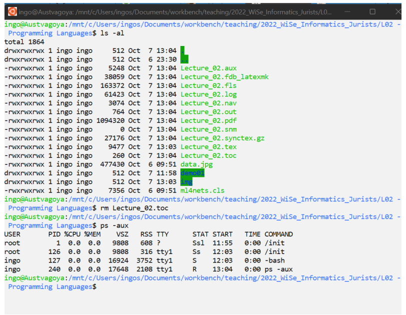
Command line Interfaces

- in addition to GUI, all major operating systems provide **text-based command line interfaces** (CLI)
  - **Windows**: command line/PowerShell
  - **Linux/Mac OS X**: terminal
- CLI provides **full access to all functions** of an OS

examplary commands (Linux-based OS)

command	meaning
`cd`	change directory
`ls`	list files in current directory
`rm`	remove file or directory
`mv`	move/rename file or directory
`ps`	list running processes
`./<executable>`	launch new process for program

- **command-line interpreter** executes commands
- CLI can be **programmed via “scripts”** (commands in text file)



Command line interface of Ubuntu Linux

definition

A **command-line interface (CLI)** accepts text-based commands to launch and manage processes, manage files, or update system settings.

Notes:

Notes:

Practice Session

- we locate the **command line interface** (CLI) of our OS
- we use the CLI to launch a **process that executes a simple `HelloWorld` program**
- we use GUI- and CLI-based tools to **monitor and kill running processes**
- we use the Linux-based CLI-tool `objdump` to inspect **machine code instructions** contained in an executable file

```
Contents of section .text:
 0530 31ed4989 d15e4889 e24883e4 f050544c  1.I..^H..H...PTL
 0540 8d058a01 0000488d 0d130100 00488d3d  ......H......H.=
 0550 e6000000 ff15860a 2000f40f 1f440000  ........ ....D..
 0560 488d3da9 0a200055 488d05a1 0a200048  H.=.. .UH.... .H
 0570 39f84889 e5741948 8b055a0a 20004885  9.H..t.H..Z. .H.
 0580 c0740d5d ffe0662e 0f1f8400 00000000  .t.]..f.........
 0590 5dc30f1f 4000662e 0f1f8400 00000000  ]...@.f.........
 05a0 488d3d69 0a200048 8d35620a 20005548  H.=i. .H.5b. .UH
 05b0 29fe4889 e548c1fe 034889f0 48c1e83f  ).H..H...H..H..?
 05c0 4801c648 d1fe7418 488b0521 0a200048  H..H..t.H..!. .H
 05d0 85c0740c 5dffe066 0f1f8400 00000000  ..t.]..f........
 05e0 5dc30f1f 4000662e 0f1f8400 00000000  ]...@.f.........
 05f0 803d190a 20000075 2f48833d f7092000  .=.. ..u/H.=.. .
 0600 00554889 e5740c48 8b3dfa09 2000e80d  .UH..t.H.=.. ...
 0610 ffffffe8 48ffffff c605f109 2000015d  ....H....... ..]
 0620 c30f1f80 00000000 f3c3660f 1f440000  ..........f..D..
 0630 554889e5 5de966ff ffff5548 89e54883  UH..].f...UH..H.
 0640 ec10488d 059b0000 00488945 f8488b45  ..H......H.E.H.E
 0650 f84889c7 e8b7feff ffb80000 0000c9c3  .H..............
 0660 41574156 4989d741 5541544c 8d254607  AWAVI..AUATL.%F.
 0670 20005548 8d2d4607 20005341 89fd4989   .UH.-F. .SA..I.
 0680 f64c29e5 4883ec08 48c1fd03 e857feff  .L).H...H....W..
 0690 ff4885ed 742031db 0f1f8400 00000000  .H..t 1.........
 06a0 4c89fa4c 89f64489 ef41ff14 dc4883c3  L..L..D..A...H..
 06b0 014839dd 75ea4883 c4085b5d 415c415d  .H9.u.H...[]A\A]
 06c0 415e415f c390662e 0f1f8400 00000000  A^A_..f.........
 06d0 f3c3                                 ..
```

practice session

see directory 04-01 in `gitlab` repository at

→ https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks

Programming in machine language?

- machine code is designed to **make execution by CPU as fast as possible**
- machine code is **not optimized to be written or read by humans**
- requires us to manually **address registers**, store values at **addresses in memory**, remember **cryptic machine instructions**, etc.
- machine code is specific to CPU architecture, i.e. programs in machine code are **not portable**

```
64a:   55                      push   %ebp
64b:   48                      dec    %eax
64c:   89 e5                   mov    %esp,%ebp
64e:   48                      dec    %eax
64f:   83 ec 10                sub    $0x10,%esp
652:   48                      dec    %eax
653:   8d 05 ab 00 00 00       lea    0xab,%eax
659:   48                      dec    %eax
65a:   89 45 f8                mov    %eax,-0x8(%ebp)
65d:   48                      dec    %eax
65e:   8b 45 f8                mov    -0x8(%ebp),%eax
661:   48                      dec    %eax
662:   89 c6                   mov    %eax,%esi
664:   48                      dec    %eax
665:   8d 3d a7 00 00 00       lea    0xa7,%edi
66b:   b8 00 00 00 00          mov    $0x0,%eax
670:   e8 ab fe ff ff          call   520 <printf@plt>
675:   b8 00 00 00 00          mov    $0x0,%eax
67a:   c9                      leave
67b:   c3                      ret
67c:   0f 1f 40 00             nopl   0x0(%eax)
```

A simple Hello World program in machine code

challenges

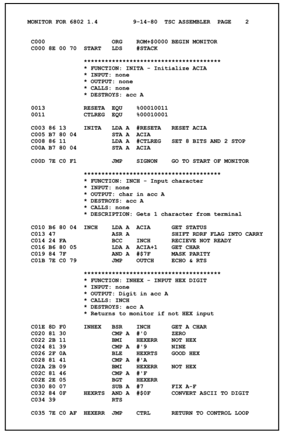
1. how can we make programming **simple and (actually) enjoyable** for human programmers?
2. how can we write **portable programs** that are independent of the processor architecture?

Notes:

Notes:

Assembly language

- assembly language is a **low-level language** that simplifies writing of machine code
- different from machine code, assembly language allows **symbolic labels, directives, and comments**
- **assembler** (software) translates assembly program to machine instructions
- strong but not strict **correspondence between assembly language and machine instructions**
- developer maintains **control over machine** instructions, i.e. programs are (potentially) very fast
- but: assembly code is still **not portable**



```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2

 C000                   ORG    ROM+$0000 BEGIN MONITOR
 C000 8E 00 70  START   LDS    #STACK

                ***************************************
                * FUNCTION: INITA - Initialize ACIA
                * INPUT: none
                * OUTPUT: none
                * CALLS: none
                * DESTROYS: acc A

 0013           RESETA  EQU    %00010011
 0011           CTLREG  EQU    %00010001

 C003 86 13     INITA   LDA A  #RESETA   RESET ACIA
 C005 B7 80 04          STA A  ACIA
 C008 86 11             LDA A  #CTLREG   SET 8 BITS AND 2 STOP
 C00A B7 80 04          STA A  ACIA

 C00D 7E C0 F1          JMP    SIGNON    GO TO START OF MONITOR

                ***************************************
                * FUNCTION: INCH - Input character
                * INPUT: none
                * OUTPUT: char in acc A
                * DESTROYS: acc A
                * CALLS: none
                * DESCRIPTION: Gets 1 character from terminal

 C010 B6 80 04  INCH    LDA A  ACIA      GET STATUS
 C013 47                ASR A            SHIFT RDRF FLAG INTO CARRY
 C014 24 FA             BCC    INCH      RECIEVE NOT READY
 C016 B6 80 05          LDA A  ACIA+1    GET CHAR
 C019 84 7F             AND A  #$7F      MASK PARITY
 C01B 7E C0 79          JMP    OUTCH     ECHO & RTS

                ***************************************
                * FUNCTION: INHEX - INPUT HEX DIGIT
                * INPUT: none
                * OUTPUT: Digit in acc A
                * CALLS: INCH
                * DESTROYS: acc A
                * Returns to monitor if not HEX input

 C01E 8D F0     INHEX   BSR    INCH      GET A CHAR
 C020 81 30             CMP A  #'0       ZERO
 C022 2B 11             BMI    HEXERR    NOT HEX
 C024 81 39             CMP A  #'9       NINE
 C026 2F 0A             BLE    HEXRTS    GOOD HEX
 C028 81 41             CMP A  #'A
 C02A 2B 09             BMI    HEXERR    NOT HEX
 C02C 81 46             CMP A  #'F
 C02E 2E 05             BGT    HEXERR
 C030 80 07             SUB A  #7        FIX A-F
 C032 84 0F     HEXRTS  AND A  #$0F      CONVERT ASCII TO DIGIT
 C034 39                RTS

 C035 7E C0 AF  HEXERR  JMP    CTRL      RETURN TO CONTROL LOOP
```

Motorola 6800 assembler program

image credit: Wikipedia, public domain

High-Level Languages

- idea: use **programming language** with **higher-level abstractions** that are easy to understand by humans
- high-level languages typically provide (at least) the following **abstractions**
  - **symbolic variables** (with data types), e.g. `int k = 42`
  - **complex types and data structures** (text, list, queue, etc.)
  - **control structures** to influence **control flow** in a program
  - **functions or routines** that can be called for code reuse
- **compiler** (software) **translates program in high-level language** to simpler machine instructions
  - original program = **source code**
  - compiled program = **executable** or **binary**
- many compilers can generate **binaries for multiple processor architectures** (cross-compilation)

```
int k = 1;
int l = 1;

for (int i=0; i<10; i++) {
  int t = k + l;
  k = l;
  l = t;
}

char* text = "Result: %s\n";
printf(text, l);
```


Notes:

Notes:

Variables vs. registers or memory addresses

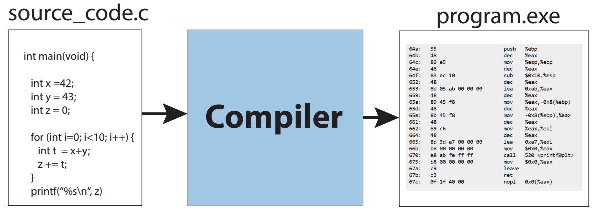
- in machine code, we use **registers** and **addresses in main memory** to store data
  1. need to manually move values between registers and main memory
  2. need to specify registers/memory based on address (i.e. register R2 or $0x4a2f$)
- high-level languages allow to **store values in variables**
- we use **assignment operator =** to assign value to variable, i.e. contents can change during runtime
- variable can refer to **address in memory** or **CPU register** (decided by compiler)
- in statically-typed languages, variables have **types** (e.g. 32-bit integer or list of 8-bit characters)

```
int k = 1;
int l = 1;

for (int i=0; i<10; i++) {
  int t = k + l;
  k = l;
  l = t;
}

char* text = "Result: %s\n";
printf(text, l);
```

definition

In high-level programming languages, a variable is a **symbolic name for an abstract storage location**, i.e. it is a “named container” that can hold a value that can change during the runtime of a process.

From source code to executables . . .



advantages

1. massively simplifies programming: **increases productivity and reduces errors**
2. makes it easier to **maintain complex software systems**
3. allows to perform **automatic optimizations** at the level of machine code
4. facilitates writing of **source code that is portable across processor architectures**
5. distribution of executables **hinders access to source code** (e.g. for copyright/security reasons)

disadvantages

1. **no direct correspondence** between high-level and machine instructions
2. **lack of control** which specific instructions are executed
3. hinders **manual optimization** of machine instructions
4. possible introduction of errors/security issues, i.e. we **need to trust the compiler**
5. distribution of executables **hinders access to source code** (i.e. requires to trust executable)

Notes:

Notes:

The C programming language

- **general-purpose programming language** created by Ritchie and Thompson in 1972 as successor to language B
- one of the most **important and widely-used** programming languages
- **statically-typed language**, i.e. we must specify type of variable
- C compilers support **virtually any processor architecture**

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
  char* text = "Hello World!";
  printf("%s\n", text);
  sleep(5);
}
```

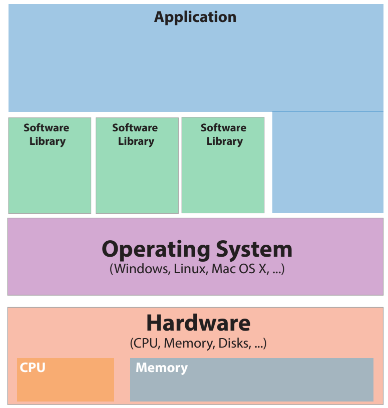
limitations of C

- error-prone dynamic allocation/release of memory
- lack of object-oriented abstractions

- basis for object-oriented "successors" C++ (1979) and Objective-C (Apple, 1984)

Software libraries

- **self-contained programs** must implement all functions that are needed by the software that we want to develop
- analogy: if you write a book, you can rely on (and refer to) **common knowledge** published by other authors
- **software libraries** contain common functionality that can be reused by other programs
  1. binary libraries with machine instructions
  2. library with reusable source code
- most high-level programming languages provide **standard libraries for common tasks**
  - complex mathematical operations
  - reading/writing from/to files
  - network communication
  - graphics and visualization



Notes:

Notes:

Application Programming Interfaces

- software library provides **application programming interface (API)** that enables us to access common functions
- analogy: table of contents in a book, which gives page number for each "topic"
- API specifies details that are required to **call function**
  - **name of function**
  - number, type and semantics of **parameters** that caller must provide
  - semantics and type of **return value** that is returned by the function
- example 1: C library `stdio` provides function `printf` that **outputs text via CLI**
- example 2: `python` module `math` provides function `sqrt` that returns **square root of given value**

```
int      getopt(int, char * const[], const char); (LEGACY)
char    *gets(char *);
int      getw(FILE *);
int      pclose(FILE *);
void     perror(const char *);
FILE    *popen(const char *, const char *);
int      printf(const char *, ...);
int      putc(int, FILE *);
int      putchar(int);
int      putc_unlocked(int, FILE *);
int      putchar_unlocked(int);
int      puts(const char *);
int      putw(int, FILE *);
int      remove(const char *);
int      rename(const char *, const char *);
void     rewind(FILE *);
int      scanf(const char *, ...);
void     setbuf(FILE *, char *);
int      setvbuf(FILE *, char *, int, size_t);
int      snprintf(char *, size_t, const char *, ...);
int      sprintf(char *, const char *, ...);
int      sscanf(const char *, const char *, int ...);
char    *tempnam(const char *, const char *);
FILE    *tmpfile(void);
char    *tmpnam(char *);
int      ungetc(int, FILE *);
int      vfprintf(FILE *, const char *, va_list);
int      vprintf(const char *, va_list);
int      vsnprintf(char *, size_t, const char *, va_list);
int      vsprintf(char *, const char *, va_list);
```

excerpt of API of C Standard Library `stdio`

Practice Session

- we write a simple program in the **high-level language C**
- we use two **library functions** to print text and to pause the program execution
- we use the compiler `gcc` to **compile the source code to an executable program**

```c
#include <stdio.h>
#include <unistd.h>

int main(void) {
  char* text = "Hello World!";
  printf("%s\n", text);
  sleep(5);
}
```

practice session

see directory 04-02 in `gitlab` repository at

→ https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks

Notes:

Notes:

Compiled vs. interpreted languages

- **compiler translates program** in high-level language to machine code **before** it can be executed
  - compiled binaries are not portable
  - users may need to compile source code
  - each change requires to recompile source code
- **interpreter can directly execute instructions** in a high-level programming language
- interpreter is program that reads and executes source code, i.e. process = **instance of interpreter that executes code in a file**
- no need for (re)compilation, no non-portable binaries
- interpreted languages are **typically slower than compiled languages** (but not necessarily)

definition

An **interpreter** is a software that directly executes instructions written in a programming language, without requiring its prior compilation to machine code.

Introducing Python

- `python` is the most **popular interpreted programming language**
- widely-used for **data processing, analytics, and machine learning**
- object-oriented with **automatic memory management**, i.e. memory is automatically allocated and released
- dynamically-typed language, i.e. types of variables are automatically inferred (and can change) at runtime
- user-friendly, great for **beginners in programming**
- **rich ecosystem of software libraries** (modules) that implement almost any imaginable functionality

Guido van Rossum, developer of `python`

image credit: Wikpedia, Doc Searls, CC BY-SA 2.0

Notes:

Notes:

Basic `python` syntax

- python programs are stored in text files (typically with extension `.py`)
- **one line in text file = one instruction**

key `python` statements

- assignment (=) used to **assign value to a variable**
- `def` used to define a **function**
- `import` statement used to import functions from modules
- `if` and `else` used to **conditionally execute instructions**
- `for` and `while` used to **repeatedly execute instructions in a loop**

- "blocks" of instructions grouped by **indentation level**
- `python` is **whitespace-sensitive**, i.e. placement of newline, space or tab characters changes semantics
- `python` enforces meaningful formatting of code, making programs easy to read for humans

```
import time

def main():
  for i in range(5):
    text = "Hello World!"
    print(text)
    text = 42
    print(text)
  sleep(5)
```


Practice Session

- we install the Open Source `python` **distribution Anaconda**
- we write a simple "Hello World" program in python
- we use the python interpreter to execute our program
- we **inspect running processes** during the execution of our program

```
import time

def main():
  text = "Hello World!"
  print(text)
  text = 42
  print(text)
  sleep(5)
```

practice session

see directory 04-03 in `gitlab` repository at

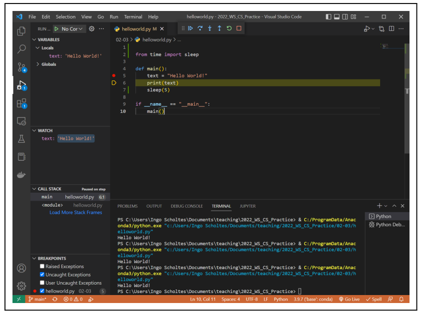
→ https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks

Notes:

Notes:

Integrated Development Environment (IDE)

- all we need to write `python` program is **text editor** and **python interpreter** (i.e. executable `python.exe`)
- sufficient for small single-file programs
- what about **complex software with hundreds of files and millions of lines in code?**
- integrated development environments (IDEs) are specialized tools to **support and simplify development of complex software**
- IDEs provide advanced functions to edit and format code, semantically highlight/color keywords, compile and/or execute program, and find errors



Open Source IDE Visual Studio Code

definition

An **Integrated Development Environment (IDE)** is a software that simplifies the programming of computers. It minimally provides functions to **edit source code files**, compile and/or execute programs, and **find errors at compile- and run-time**.

Practice Session

- we use the **integrated development environment** (IDE) Visual Studio Code to write and execute a simple `python` program
- we use VS Code to **rename variables and refactor code**
- we use the **debugger of Visual Studio Code** for a step-wise execution of python statements

```
import time

def main():
    text = "Hello World!"
    print(text)
    text = 42
    print(text)
    sleep(5)
```

practice session

see directory 04-04 in `gitlab` repository at

→ https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks

Notes:

Notes:

In summary

- we inspected the **GUI and the CLI of modern operating systems**
- we motivated the use of **high-level programming languages**
- we explained the difference between **compiled and interpreted languages**
- we introduced the **popular interpreted high-level panguage `python`** and wrote a first program

```
64a:   55                      push   %ebp
64b:   48                      dec    %eax
64c:   89 e5                   mov    %esp,%ebp
64e:   48                      dec    %eax
64f:   83 ec 10                sub    $0x10,%esp
652:   48                      dec    %eax
653:   8d 05 ab 00 00 00       lea    0xab,%eax
659:   48                      dec    %eax
65a:   89 45 f8                mov    %eax,-0x8(%ebp)
65d:   48                      dec    %eax
65e:   8b 45 f8                mov    -0x8(%ebp),%eax
661:   48                      dec    %eax
662:   89 c6                   mov    %eax,%esi
664:   48                      dec    %eax
665:   8d 3d a7 00 00 00       lea    0xa7,%edi
66b:   b8 00 00 00 00          mov    $0x0,%eax
670:   e8 ab fe ff ff          call   520 <printf@plt>
675:   b8 00 00 00 00          mov    $0x0,%eax
67a:   c9                      leave
67b:   c3                      ret
67c:   0f 1f 40 00             nopl   0x0(%eax)
```

open issues

- how can we use high-level languages to **solve actual problems?**
- what are **algorithms** and how we can we implement them?
- need to develop **algorithmic thinking**, which is key to understand how computer scientists think and work.

Self-study questions

1. Explain the difference of a GUI and a CLI of an operating system. Which one is more intuitive? Which one is more powerful?
2. Explain the steps taken by an OS to launch a process that executes a `HelloWorld` program stored in an executable file.
3. Explain the difference between machine instructions and assembler code.
4. What are the advantages of high-level programming languages like C compared to assembler?
5. List abstractions provided by a high-level programming language that are not provided by machine instructions?
6. What is a variable in a high-level language?
7. What is the difference between statically- and dynamically-typed programming languages?
8. What is a compiler and what is an interpreter?
9. Explain the steps needed to write and execute a Hello World program written in the programming language C.
10. Explain the steps needed to write and execute a Hello World program written in the programming language `python`.
11. What are advantages/disadvantages of compiled and interpreted programming languages?
12. What advantages does an integrated development environment (IDE) provide?

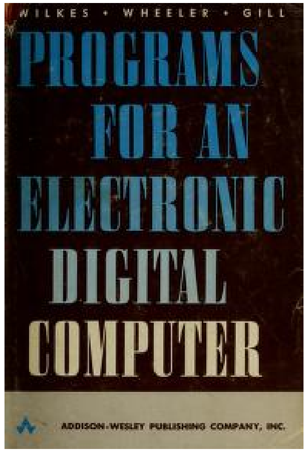
Notes:

Notes:

Literature

reading list

- W Kernighan, D Ritchie: **The C Programming Language**, Prentice Halle, 2000
- F Kaefer, P Kaefer: **Introduction to Python Programming for Business and Social Science Applications**, SAGE Publications, 2020



Notes: