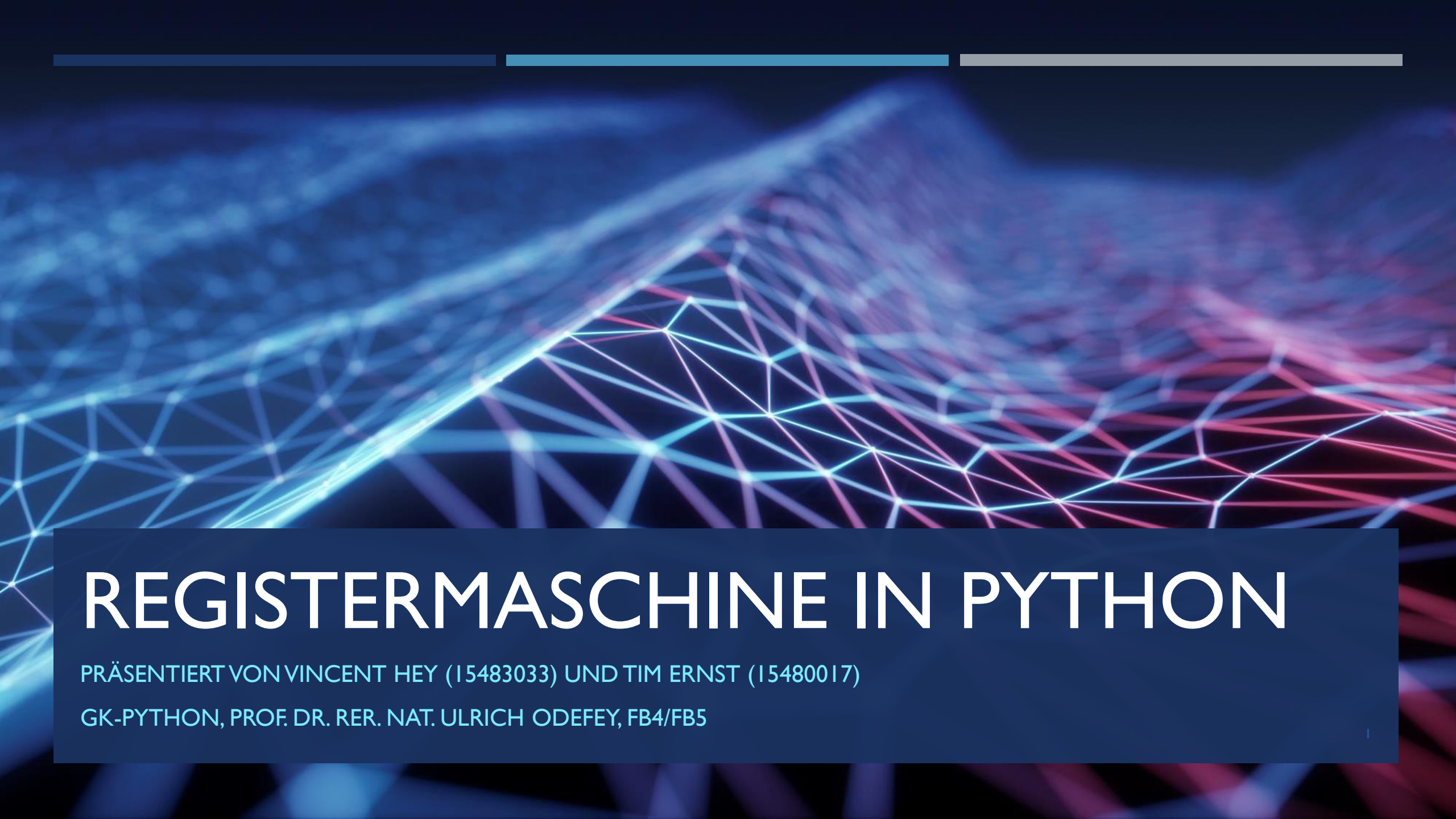


REGISTERMASCHINE IN PYTHON

PRÄSENTIERT VON VINCENT HEY (15483033) UND TIM ERNST (15480017)

GK-PYTHON, PROF. DR. RER. NAT. ULRICH ODEFEY, FB4/FB5

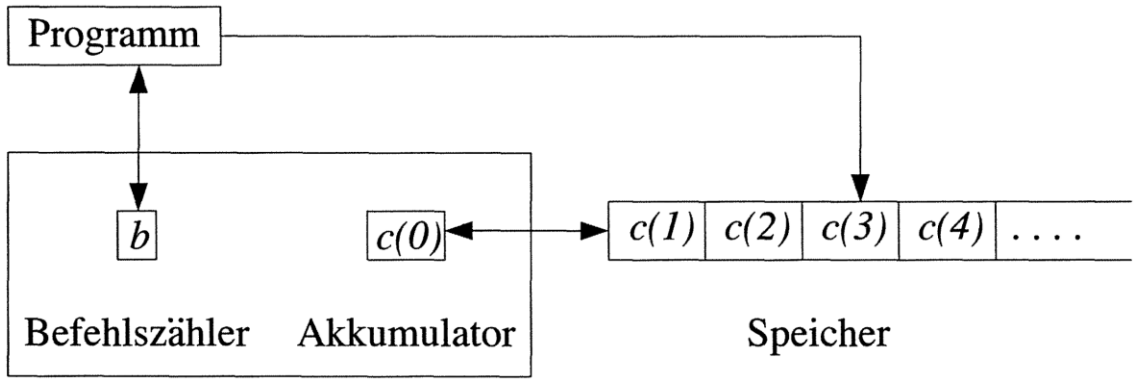
INHALT

HINTERGRUNDWISSEN

Registermaschine (RM)

- Einfaches Rechnermodell
- Basis für viele Algorithmen und Rechnerstrukturen
- Näher an realen Maschinen als Turing-Maschine (TM)
- Logisch äquivalent zur TM
  - Können sich gegenseitig simulieren

Aufbau



Aus: Ingo Wegener – Theoretische Informatik – eine algorithmenorientierte Einführung, 3. Aufl.

RM-PROGRAMMIERSPRACHE

Für unsere RM-Implementation gilt

- Das Programm für ist eine **Liste von Befehlen**
- In jeder Programmzeile ist **genau ein Befehl**
- Register 0 ist der *Akkumulator* (wie *Ans* beim Taschenrechner) und wird als impliziter 2. Operator verwendet
- Befehlsstruktur:
  - **Operand**: Befehlsname
  - **Operator**: eine Zahl

RM-BEFEHLSÜBERSICHT

Konstante Arithmetik	Register-Arithmetik
`CLOAD zahl`	`LOAD register`
`CADD zahl`	`STORE register`
`CSUB zahl`	`ADD register`
`CMULT zahl`	`SUB register`
`CDIV zahl`	`MULT register`
	`DIV register`

Indirekte Adressierung	Sprungbefehle
`INDLOAD adresse`	*NONE*
`INDSTORE adresse`	`END`
`INDADD adresse`	`GOTO zeile`
`INDSUB adresse`	`IF < <= = != >= > zahl`
`INDMULT adresse`	
`INDDIV adresse`	

Die RM kennt keine Datentypen, daher sind die Operanden allesamt Zahlen

IMPLEMENTATION IN PYTHON – PARSER

- **Parser** := System zur Zerlegung von Text in geeignetes Format zur Weiterverarbeitung
- Ablauf in unserem Falle
  - Ignorieren von Kommentaren
  - Überführung in Kleinbuchstaben → case-insensitive
  - Entfernen von Leerraum
  - Zerlegung anhand von Leerzeichen
  - Umwandlung der Teile in „Instructions“

```python
def canonicalize(string: str) -> str:
  return string.lower().replace('\\s+', ' ').lstrip().rstrip()

def from_string(source: str) -> Instruction:
  if source.startswith('#'):
    return Instruction(Operator.NONE, 0)
  parts = canonicalize(source).split(' ')
  operator = parts[0]
  part1 = parts[1]
  match operator:
    case operator if operator in SIMPLE_INSTRUCTIONS:
      operand = int(part1)
      return Instruction(Operator.from_string(operator), operand)
    case Operator.END:
      return Instruction(Operator.END, 0)
    case _:
      raise InvalidOperator(f'Operator {operator} is invalid’)

Gekürzt. Fehlerbehandlung entfallen.
Aus Platzgründen nicht wie in Python eingerückt.
```

IMPLEMENTATION IN PYTHON – BEFEHLSAUSFÜHRUNG

```
class Machine:
  def __init__(self, data_manager: DataManager):
   self.instruction_set: dict[Operator,Callable[[Machine,
int], Machine]] = {}

 # Überlädt den machine[index]-Operator (fehlend __getitem__)
 def __setitem__(self, index: int, value: int) -> Machine:
   self.memory[index] = value % Constants.REGISTER_LIMIT
   return self

 def instruction(self):
   def decorator(function: Callable[[Machine, int], Machine])
     operator = Operator.from_string(function.__name__)
     self.instruction_set[operator] = function
     return function
   return decorator
```

Gekürzt. Fehlerbehandlung entfallen.
Aus Platzgründen nicht wie in Python eingerückt.

```
def add_math(self):
  @self.instruction()
  def add(m: Machine, i: int) -> Machine:
    m[0] = m[0] + m[i]
    return m

def step(self):
  if self.get_programcounter() <= self.program.size():
    instruction = self.program[self.get_programcounter() - 1]
    try:
      self.instruction_set[instruction.operator](
        self, instruction.operand)
    except KeyError:
      raise MachineRuntimeError(
        f'Instruction {instruction.operator} is undefined‘)
    self.change_programcounter(1)
```

Gekürzt. Nur teilweise Fehlerbehandlung.
Aus Platzgründen nicht wie in Python eingerückt.

AUFBAU FRONTEND / MIDDLEWARE / BACKEND

Frontend: GUI

Nimmt Nutzeranfragen entgegen

Leitet sie an Middleware / Backend weiter

Zeigt Nutzer Rückgaben an

Middleware: DataManager

Zentraler Datenmanager

Verwaltet gemeinsam (Front- und Backend) genutzte Variablen

- Programmzähler
- Register

Backend: Instruction, Programm, Machine

Simulation der Registermaschine

TKINTER

- tkinter („Tk interface“)
  - Standard Python-Interface für das Tcl/Tk GUI toolkit
  - Einfach mit pip zu installieren
  - Plattformübergreifend
- Benutzeroberfächenerstellungsmodule
  - Textfelder, Menüs, Schaltflächen, Dateidialoge,

    sich automatisch aktualisierende Variablen

FILE-INPUT

```
def open_file(self):
    file_path = filedialog.askopenfilename(defaultextension=".ram",
                                           filetypes=[("RAM Dateien", "*.ram"),
                                                      ("Alle Dateien", "*.*")])
    if file_path:
        try:
            with open(file_path, 'r', encoding='utf-8') as file:
                content = file.read()
            self.text_area.delete(1.0, tk.END)
            self.text_area.insert(tk.END, content)
            self.current_file_path = file_path
            self.status_bar.config(text=f"Geöffnet: {file_path}")
            self.update_line_numbers()
        except Exception as e:
            self.show_exception(f"Datei konnte nicht geöffnet werden: {e}")
```

FILE-OUTPUT

```
def save_file_as(self):
    file_path = filedialog.asksaveasfilename(defaultextension=".ram",
                                             filetypes=[("RAM Dateien", "*.ram"),
                                                        ("Alle Dateien", "*.*")])
    if file_path:
        try:
            with open(file_path, 'w', encoding='utf-8') as file:
                content = self.text_area.get(1.0, tk.END)
                file.write(content)
            self.current_file_path = file_path
            self.status_bar.config(text=f"Gespeichert unter: {file_path}")
        except Exception as e:
            self.show_exception(f"Datei konnte nicht gespeichert werden: {e}")
```

SCHEDULED FUNCTION / FUNKTIONSZEIGER

```
def auto_increment(self):
    """Erhöht den Program Counter automatisch in Intervallen, bis der
       Play-Button wieder gedrückt wird."""
    if self.auto_increment_active:
        self.step()
        self.highlight_program_counter_line()
        self.root.after(1000, self.auto_increment)
```

- Durch after(…) kann eine wiederholte Ausführung erreicht werden, ohne blockierende Schleife
- Programm läuft normal weiter, bis Zeit verstrichen ist → Funktion wird aufgerufen

  Kein rundes Klammerpaar nach Funktion?

→ Funktionszeiger (Funktion als Callable-Variable übergeben)

EVENT-LISTENER

```
self.text_area.bind('<KeyRelease>', self.update_line_numbers)
```

- text_area-Widget wird an <KeyRelease>-Event gebunden
- Taste loslassen, innerhalb des Widgets → als Funktionszeiger übergebene Funktion wird aufgerufen
- Hintergrund: Zeilennummern ändern sich, wenn Zeile gelöscht oder hinzugefügt wird

TK-VARIABLEN

```
self.registers: list[tk.StringVar] =
    [tk.StringVar(master=root, value="0") for _ in range(Constants.REGISTER_COUNT)]
```

- Variablen aktualisieren sich bei Änderung automatisch in der GUI
- Verwendung bei Registern und Program Counter

LIVE-DEMO

HERAUSFORDERUNGEN UND LÖSUNGEN

- Problem: Spezielle Python-Version unter Linux installieren nicht trivial

  → Lösung: Docker, Microsoft devcontainer

  - Nachteile: Kann zu Problemen mit der Grafik unter WSL führen
- Problem: Versionierung und kollaboratives Arbeiten

  → Lösung: Git, GitHub

VIELEN DANK

TIM ERNST
VINCENT HEY