CHAPTER 1

Basics of Imperative Programming

Lernziele

KdP2 Du implementierst gut strukturierte *Python-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *imperativer Programmierkonzepte*.

KdP5 Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und paradigmen.

Organisatorisches

- Videolänge: ∼102 Min.
- Tutorien: 23.+24. Oktober
- Großplenum: 25. Oktober, 8–10 Uhr im Hörsaal 1B
- Lernzentrum: 23.–27. Oktober
- Großtutorium: 27. Oktober, 8–10 Uhr im Hörsaal 1B
- Abgabe der Übungsaufgaben: 27. Oktober, 14 Uhr

Contents

- Von-Neumann-Architecture
- Programming Languages: Assembly languages; High-level Languages
- Expressions
- Imperative Programming: Assignments; Input and Output; Control Flow

Übungsaufgaben

Hinweise: Aufgabe 3 ist freiwillig und wird nicht bewertet. Von Aufgabe 1 und 2 wird eine Aufgabe ausführlich korrigiert und 10 Punkte vergeben, eine Aufgabe wird nur abgehakt und 1 Punkt vergeben. Welche der beiden Aufgaben das jeweils sein wird, wird nach dem Abgabedatum entschieden.
Plagiate werden beim 1. Mal mit 0 Punkten auf dem gesamten Zettel und beim 2. Mal mit dem Verlust der aktiven Teilnahme geahndet. Bei jeder Programmieraufgabe wird erwartet, dass der Quelltext ausreichend kommentiert ist und getestet wurde.

1. **Imperatives** (10 Punkte)
   a) Schreibe einen arithmetischen Ausdruck in Python, der die Fläche eines Dreiecks mit Grundseitenlänge a und Höhe h_a berechnet.
   b) Informiert euch über die `switch-case`-Anweisung in C. Beschreibe, wie man die `switch-case`-Anweisung aus C mit Hilfe einer `if`-Verzweigung in Python simulieren kann.
   c) Erkläre, wie man eine Schleife der Form `for i in range(a,b,c): ...` in Python mit Hilfe einer `while`-Schleife nachbilden kann. Gebt außerdem ein Beispiel.
   d) Schreibe ein Python-Programm, das drei ganze Zahlen einliest und genau dann `True` auf dem Bildschirm anzeigt, wenn die Zahlen paarweise verschieden sind.
   e) Schreibe ein Python-Programm, das so lange Zahlen von der Tastatur einliest, bis eine 0 eingegeben wird. Danach soll der Durchschnitt der eingegebenen Zahlen ausgegeben werden.

2. **Noch mehr Imperatives** (10 Punkte)
   a) Schreibe einen bedingten Ausdruck in Python, der den Preis für eine Kinokarte berechnet: Kinder unter 12 zahlen 10 Euro, Jugendliche bis 18 zahlen 12 Euro, Erwachsene zahlen 14 Euro und Rentner ab 65 zahlen 12 Euro.
   b) Nehmt an, ihr habt eine Programmiersprache, die nur eine `if-then` Anweisung zur Verfügung stellt (ohne `elif`- und `else`-Zweige). Zeige, wie man trotzdem eine allgemeine `if-then-elif-else` Anweisung simulieren kann.
   c) In einigen Programmiersprachen gibt es eine Schleife der Form `do-while`. Führe eine Recherche durch und erkläre, wie diese Schleife funktioniert. Beschreibe anschließend, wie man sie mit Hilfe einer `while`-Schleife in Python nachbilden kann. Gib ein Beispiel.
   d) Schreibe ein Python-Programm, das drei ganze Zahlen einliest und auf dem Bildschirm anzeigt um wie viele verschiedene Zahlen es sich handelt.

e) Schreibe ein Python-Programm, das folgende Pyramide mit Hilfe einer `for`-Schleife ausgibt.

```
    1
   212
  32123
 4321234
543212345
65432123456
```

3. Einfach nur völlig random ... (ohne Punkte)

Betrachte den folgenden *Zufallsspaziergang*. In jedem Schritt befinden wir uns bei einer natürlichen Zahl $z \geq 0$, wobei zu Beginn $z = 0$ ist. Sind wir bei $z = 0$, so gehen wir immer zu $z = 1$. Ist $z > 0$, so werfen wir eine faire Münze. Zeigt die Münze Kopf, so gehen wir von z zu $z + 1$, zeigt die Münze Zahl, so gehen wir von z zu $z - 1$.

Implementiere diesen Zufallsspaziergang in Python und verwende dein Programm, um die folgenden Fragen zu beantworten.

a) Wie viele Schritte sind durchschnittlich nötig, bis der Zufallsspaziergang die 10 erreicht? Wie viele Schritte sind es für die 20? Die 30? Die 50? Könnt ihr eine allgemeine Formel für die durchschnittliche Anzahl der Schritte raten, bis n erreicht ist?

b) Wo befindet sich der Zufallsspaziergang nach 100 Schritten? Wo nach 1000 Schritten? Versuche, die Verteilung empirisch zu ermitteln und graphisch darzustellen.

1.1 Von-Neumann-Architecture

A *computer* is a universal machine that executes one or many *programs*. A program is a sequence of *instructions* that are usually executed in the given order. Nowadays, most computers are based on the *von-Neumann architecture*[1], which basically consists of the following four components:

1. **Main Memory/RAM.** The random access memory consists of cells of equal size. These memory cells are numbered consecutively (*address*) and include programs and data. The programs and the data are represented binary.
2. **Central Processing Unit.** The CPU includes the *Control Unit* which interprets instructions, the *Arithmetic Logical Unit* (ALU) which evaluates logical and arithmetical operations, the *Program Counter* which contains the address of the next instruction, and the *clock generator* which controls the execution by giving the calculation cycle.
3. **Input/Output.** It provides mechanisms for communication between humans and machines, e.g., mouse, keyboard, screen output, USB, and many more.
4. **BUS system.** The BUS system provides for communication of data, addresses and clock signal between the other three components.

Selbsttest: Schaut euch das Video 001 an. Welche der folgenden Aussagen sind korrekt?

1. Der Taktgeber befindet sich in der RAM.
2. Die ALU ist dazu da, um Zahlen zu speichern.
3. Der Befehlszähler beinhaltet den nächsten Befehl.
4. Der Befehlszähler beinhaltet die Adresse des nächsten Befehls.

Given these components we can describe a mechanism how a computer executes an instruction. This mechanism is called *machine command cycle* and works in five steps:

(i) The control unit loads the address of the next instruction from the program counter.

(ii) The instruction passes from the main memory to the CPU.

(iii) The control unit interprets the instruction.

(iv) The control unit initiates corresponding actions, e.g., computing in ALU, reading/writing, comparing, input/output.

(v) The control unit updates the program counter.

Selbsttest: Schaut euch das Video 002 an. Was passiert, wenn die nächste Anweisung nicht die Addition von zwei Zahlen ist, sondern die Anweisung lautet „Springe zu der Anweisung mit der Adresse 4711“?

[1] This architecture was first described by the mathematician John von Neumann in 1945.

The *machine language* consists of very simple instructions, for example arithmetic and logic operations, comparisons, branches, data manipulation, or input and output. The instructions highly depend on the concrete hardware and can vary widely between systems and architectures. They are encoded numerically, e.g., `10 12 05 13 F2 0D` might be a valid machine language instruction. The first computers were programmed directly using this simple machine language. It can be very difficult and cumbersome to learn, to use and to maintain them.

1.2 Programming Languages

Higher level programming languages are designed for humans. We distinguish these languages by their degree of abstraction.

1.2.1 Assembly languages

Assembly languages are close to machine languages. They provide simple textual representations of machine language. These languages introduce *mnemonics*: short names for machine language instructions, such as `ADD`, `SUB`, MUL, CMP, BNZ, IN, OUT, and many more. The source code is written as a simple text file and a computer program called *assembler* translates this text file into the numerical machine language. This is a partial example of an assembly language program:

```
MOV  EBX, EAX
SUB  EAX, 25
JMP  foo
```

The first line moves the data in the memory cell `EBX` to the cell `EAX`. After that it subtracts `25` from the cell `EAX` and finally it jumps to the line with the label `foo`.
Some assemblers provide additional functionality to simplify the coding task (e.g., macros, naming of memory locations, resolution of addresses). However, assembly language is still very specific to the concrete hardware and architecture.

Selbsttest: Schaut euch das Video 003 an. Was spricht dagegen, alle Programme in Assembler zu schreiben? Was spricht dafür?

1.2.2 High-level Languages

High-level languages completely *focus on humans* and not on machines. Here the goal is to develop a formal artificial language that allows humans to develop and execute solutions for algorithmic problems in a simple and efficient manner. The languages have a *higher level of abstraction* than assembly languages. They provide more structure and are easier to understand than machine-level code. Moreover, we obtain less dependency of a concrete hardware, i.e., high-level programming languages can be used on computers

with different hardware and architecture. Moreover, high-level languages provide more ways of detecting and avoiding mistakes in computer programs.

There are several different approaches when designing a high-level programming language. We call these approaches *programming paradigm*.

Imperative programming languages. These languages are still close to the hardware but easy to read: An imperative program is a sequence of *instructions* that changes the *state* of the computer. Most widely used languages are imperative: C, C++, Java, C#, Pascal, Python, Rust, and many other more.

Declarative programming languages. These languages completely abstract from the hardware. A declarative program typically describes what should be computed, but the details of how this is done are left to the machine. These languages usually correspond to some mathematical abstraction. There are different variants like *logic programming* (PROLOG), *database programming* (SQL), or *functional programming* (Haskell, Scala).

Multi-paradigm programming languages. Older languages like Haskell, C, Pascal, or Prolog only fit to one of the paradigms. However, many languages – especially the recent ones – are multi-paradigm languages and provide concepts of different paradigms. For example, the languages Python and Scala are functional, imperative and object-oriented programming languages.

We will consider imperative programming in *Python 3* and functional and object-oriented programming in *Scala 3* in this class.

1.3 Expressions

Almost every high-level programming language, irrespective of the underlying paradigm, has a notion of *expression*. Every expression has a *value* and a *type*. The *type* of an expression determines which values are possible and which operations are allowed. We discuss the different types later in class. A basic task in every programming language is to determine the value of a given expression, i.e., to *evaluate* the expression. There are different kinds of expressions:

- *Constant expressions*. Consist of a single value. For example:

```
0                              # Number 0
1.5                            # Number 1.5
"Konzepte der Programmierung"  # Text, character sequence, String
True                           # truth value
```

- *Arithmetic expressions*. Consist of basic numerical operations. For example:

```
1 + 2 * (5 - 7)     # value is -3
1.5 - 2/3           # value is 0.8333333
```

- *Boolean expressions.* Result in either True or False. Can be obtained by combining other truth values or by evaluating a predicate. For example:

```
3 <= 5              # value is True
7 == 2              # value is False
True and not False # value is True
True or False       # value is True
3 <= 5 and 7 == 2   # value is False
```

- *Conditional expressions.* Obtained by first evaluating a Boolean expression and then proceeding according to the result. For example:

```
0 if 3 >= 5 else 10       # value is 10
10 if 12*5 == 60 else 20 # value is 10
```

- *Function calls.* Python has built-in functionality that is represented as *functions*. Functions can be used in expressions. We will talk much more about functions later in this class.

```
len("Hallo")    # value is 5
abs(-1)         # value is 1
int(2.7)        # value is 2
```

Different kinds of expressions can be nested, as long as they fit together.

Selbsttest: Schaut euch das Video 004 an. Ist `False and not "Hello"` ein korrekter Python-Ausdruck? (Probierts aus!)

1.4 Imperative Programming

We will now talk about imperative programming in *Python*. Python is a relatively recent language that has become very popular in recent years. It is typically called a *scripting language*: it focuses on quick solution of problems, tries to avoid many constructs from other languages that are aimed at enforcing structure and discipline in larger software projects. It has many libraries and tools and is popular in scientific contexts.

The imperative programming paradigm closely follows the model of the actual computing hardware, but at a higher level of abstraction.

An imperative program consists of a sequence of *instructions* that are executed in sequence. Every instruction has an *effect*, changing the *state* of the machine. The state of the machine consists of the contents of all memory locations, the current instruction, and the positions of all input/output devices.

The instructions are characterized by the precise effect that they have on the state. This differs from programming language to programming language, but the general types of instructions are always the same. There are:

- **Assignments** (change the content of the memory)
- **Input and Output** (communication with outside world)
- **Control flow** (change the program counter)

Selbsttest: Schaut euch das Video 005 an. Was gehört alles zum Zustand einer Maschine?

1.4.1 Assignments

Assignment instructions In Python (but also in other languages) are typically of the form:

```
name = expression
```

Some languages use `:=` instead of `=` and some languages need a semicolon at the end of the line. The expression is evaluated, and the resulting value is assigned to a variable that is identified by `name`. The precise meaning of this varies from programming language to programming language.

In Python, an assignment instruction works as follows: after evaluating `expression`, Python creates a new *data object* and stores it in the memory.[2] This data object contains the information about the value and the type that were obtained from `expression`.

After this, the identifier `name` is *associated* with the data object.

Later, we can use `name` in other expressions to obtain the value of the object that `name` is associated with. It is important to note that the type is a property of the data object in the memory and not of the identifier `name`. This means that throughout a Python program, the same identifier `name` can be associated with different data objects of completely different types. It is up to the programmer to keep track of what kind of data objects `name` is currently associated with. We say that Python is *dynamically typed*. This is in contrast to other programming languages (e.g., Scala, Java, C, Haskell) that are *statically typed*. Here, we must declare a type for any identifier `name` that we would like to use, and we may name associated `name` only with data objects that conform to the type.

A particular case is the Python assignment:

```
name1 = name2
```

Here, the identifier `name1` is associated with the same data object that `name2` is associated with. That is, the same data object is associated with two different identifiers and can be accessed through two different identifiers.

Most data objects in Python are *immutable*: their value cannot be changed. For such data objects, this does not cause problems. However, some data objects in Python are

[2] This is only partially correct. Sometimes Python uses data objects that exist already in memory to represent the value of an expression. For example, for Boolean expressions, there are only two fixed data objects that represent `True` and `False` and that are used for every Boolean expression.

mutable, and their value can be changed through later instructions. In this case, the change is visible under all identifiers that are associated with this data object. We will see an example later, when we talk about lists.

Selbsttest: Schaut euch das Video 006 an. Erkläre deinem·r Übungspartner·in die Begriffe *dynamisch getypt* und *immutable*.

1.4.2 Input and Output

Input/Output instructions allow the program to communicate with the outside world (e.g. user, file system, network, screen, loudspeaker). The precise details of input/output instructions vary from programming language to programming language and depend also on the precise nature of the outside world: in some languages, the instructions for communicating with the user might look quite different from the instructions that communicate with the file system (e.g., in Python). Other programming language try to handle all kinds of input/output in a uniform way, irrespective of whether it may be appropriate or not (e.g., in Java).
For now, we consider two input/output instructions in Python:

- `print(arg1, arg2, ..., argk)` receives an arbitrary number of arguments. Each argument is evaluated and converted to a string. The strings are concatenated, with separating white-spaces in between. The result is then displayed onto the screen. In the process, `print` interprets certain *control sequences* that may appear in the argument strings, e.g., `\n` creates a new line, or `\t` performs a tab-operation.

  ```
  print("Konzepte", "der", 6*7) # shows "Konzepte der 42" on the screen
  ```

  Alternatively, we can call `print` with a single string argument that we prepare ourselves. Then, we have more control over the layout of the string, but we must convert non-string data manually to a string, typically by using the function `str(·)`.

  ```
  print("Konzepte " + "der " + str(6*7))
  ```

- `input(string)` outputs `string` onto the screen and reads an input from the keyboard, terminated by the return key. The input is stored as a string object and can be assigned to a variable.

  If we want to receive as input a data object of another type (e.g., a number), we must convert the input string manually to this type (e.g., by using the functions `int(·)` or `float(·)`).

Attention: When dealing with input/output operations, we must always expect errors. For example, the user input might contain mistakes (we would like to receive a number, but we do not receive one), hardware might fail, the network may be off-line, someone

may have deleted the file that we would like to access, etc. We must be ready to handle such errors in our code. We will talk later about how this is done.

Selbsttest: Schaut euch das Video 007 an. Implementiere ein Programm, welches eine Zahl mit `input` einliest und anschließend diese Zahl drei Mal auf dem Bildschirm anzeigt.

1.4.3 Control Flow

Using expressions, assignments and input/output instructions, we can already write interesting imperative programs, e.g., for performing simple computational tasks or for implementing simple numerical algorithms. However, to obtain more powerful programs, we need instructions that allow us to adjust the working of our program to the input data. These instructions affect the *control flow* of the imperative program, i.e., the order, in which the instructions of the program are executed. Control flow instructions change the program counter in the CPU. There are several types of control flow instructions.

Jumps. Jumps are the most direct way to affect the control flow in an imperative program.

We write a *label* in front of an instruction, and at another point in the program, we write `goto` or `jump` and the name of the label. When the goto-instruction is executed, the imperative program will continue with the instruction after the given label.

Even though jumps seem to be simple, they can lead to unstructured code that is hard to understand, to maintain, and to debug. Thus, most modern imperative languages follow a *structured programming* approach and do not allow full-fledged `goto`-instructions. Jumps are allowed only in very limited ways, e.g., in the context of exceptions (see below) or in the form of `break` or `continue` instructions in the context of loops (see below).

Attention: Although jumps are technically possibly in many languages, it is bad programming style to use them. Hence, do not use any jumps.

Conditional execution. In conditional execution, we can indicate that a certain sequence of instructions (called a *block*) should be executed only if a certain condition is met. We can test for several conditions, and we can also provide a block for the case that no condition is met.

In Python, conditional execution is handled through the `if`-construct. Is has the following form:

```
if condition1:
    Block1
elif condition2:
    Block2
elif condition3:
```

```
    Block3
...
else:
    Blockn
```

First, `condition1` is tested. It is a Boolean expression that is evaluated. If the value is `True`, Python executes `Block1`, and afterwards continues with the instruction after the `if`-construct. If the value is `False`, Python evaluates `condition2`, which is again a Boolean expression. If the value is `True`, Python executes `Block2` and afterwards continues with the instruction after the `if`-construct. This continues until the first condition that evaluates to `True`. If no condition evaluates to `True`, the block after the final `else:` is executed.

There can be an arbitrary number of `elif`s (also none). The `else:`-part can be omitted, in which case the execution continues after the `if`-construct, if no condition is `True`.

A block can consist of one or more instructions. In Python, a block is marked by *indentation*, i.e., the whitespaces that precede the instructions.

Selbsttest: Schaut euch das Video 008 an. Implementiere ein Programm, welches eine Zahl mit `input` einliest und `"Eins"`, `"Zwei"`, `"Drei"` oder `"Was anderes"` auf dem Bildschirm ausgibt.

Loops. Loops allow us to execute a sequence of instructions several times. The number of times the sequence of instructions is executed is often controlled by some condition. There are two kinds of loops:

- `for`-loops allow us to iterate over all elements of a given domain or range. In a `for`-loop there is typically an index variable that successively goes through all the possible values of the range, in order.

  In Python, a `for`-loop looks as follows:

  ```
  for variable in range:
      Block
  ```

  Here, a range can be any set of elements, we will see some examples below.

  Successively, variable is associated with every element of the range, and the Block is executed for each of them. It is very common that `range` consists of numbers. In Python, such a range is described by `range(a,b,s)`, which represents the sequence of numbers that is obtained by starting with `a` and repeatedly adding `s` until we meet or pass `b`. The step size `s` may be negative, which means that we count downwards. If we omit `s`, then it defaults to 1. In this case, `range(a,b)` consists from all numbers from `a` (included) to `b` (excluded). We can also use `range(a)`, which represents all numbers from 0 to $a - 1$.

```python
for x in range(0,10,2):
    print(x)
# 0 2 4 6 and 8 are shown on the screen
```

- Conditional loops/`while`-loops. Conditional loops consist of a condition and a block. The number of times the block is executed is controlled by the condition.

  There are several variants of condition loops, depending on whether the condition is necessary for continuation (`while`-loop) or for termination (`until`-loop) and whether the condition is tested before the first iteration of the loop or after the first iteration.

  In Python, there is only one conditional loop:

```python
while conditon:
    Block
```

  `condition` is a Boolean expression. It is evaluated first. It the value is `False`, the execution continues after the `while`-construct. If the value is `True`, the block is executed and then the condition is tested again. This continues until the condition is evaluated to `False`.

```python
x = 0
while x < 10:
    print(x)
    x = x + 2
# 0 2 4 6 and 8 are shown on the screen
```

Selbsttest: Schaut euch das Video 009 an. Gib eine Schleife an, welche die Zahlen `10 7 4 1 -2 -5` in dieser Reihenfolge auf dem Bildschirm ausgibt.

Dealing with errors. Whenever we write a serious imperative program, we need to be prepared to deal with errors. Even if our own code is perfect (which is unlikely), we will need to rely on input/output operations, and these input output operations can lead to problems. For example, a user input might not have the form that we expect, we might try to access a file that does no longer exist, we might try to use a storage device that is not physically present, we might try to use a network connection that is currently down, etc.

Thus, our code needs to have a capability to handle errors. There are several ways how this can be done in imperative programs.

- Abort the program. Whenever an error occurs, the program is terminated, and we return to the operating system. All data is lost, the program needs to be restarted, in the hope that the error does not happen again.

  This is the default behaviour in Python, and in many other programming languages. Even though it is quite easy for the programmer, it is not very useful for people who use the program.

- Introduce a special value that indicates that an operation could not be executed as expected.

  For example, numbers types may have a special value, `NaN` (not a number) that can indicate that a division through 0 has occurred. If a language decides to use this feature, one would need to check after a division operation whether the resulting value is `NaN`, and if so, the program needs to react appropriately.

  Similarly, one might implement the `int(input(...))` operation in such a way that it returns `None` if the string input cannot be interpreted as an integer number. Again, the program would need to test explicitly for this `None` value to handle the error.

  This behavior is not implemented in Python, but we can find it in other languages, e.g., in C.

- Use an error handling mechanism (SIGNAL, interrupt). Somewhere in our program, we define a special error handler that is called whenever an error occurs. The task of the error handler is to determine the cause of the error and to try to fix it. After that, the program is resumed at the position where the error has occurred, and it tries to repeat the critical operation. If the error happens again, the error handler is called again, etc.

  This mechanism is implemented in many operating systems (e.g., in order to deal with a missing storage device), but it is not the default behaviour in Python.

- Exceptions offer a structured way to deal with errors that tries to separate the actual code and the error handling mechanism. They are similar to a global error handler, but work at a fine-grained level.

  In Python, exceptions use the following general syntax:

  ```
  try:
      block
  except:
      error handler
  ```

  We put the critical operation(s) into a `try`-block. If the operations are executed without problems, the execution continues after the exception-construct (the error handler is not executed). If an operation in the `try`-block causes an error, the `try`-block is aborted and execution continues with the error handler. The error handler is executed, and the program continues after the exception construct, and it is up to the program to deal with the error (e.g., the error handler might provide a default value and the program might continue, or the exception-construct might be in a loop that is repeated until the operation succeeds).

  The exception-handling mechanism in Python is much more sophisticated, we can test for specific errors, define our own errors, may nest `try`-blocks, and there are additional blocks that may be added to the exception construct. We refer the interested reader to the Python documentation for more details.

Selbsttest: Schaut euch das Video 010 an. Gib ein Programm an, welches solange eine Eingabe vom Benutzer erwartet, bis eine Zahl größer als 0 eingegeben wurde. Nutze eine Schleife und einen `try-except`-Block.

Now that we have also control-flow instructions, we have all the tools to write serious imperative programs.

Selbsttest: In Video 011 implementieren wir ein Programm, welches eine zufällige Zahl zwischen 1 und 100 würfelt. Anschließend darf der Benutzer solange diese Zahl erraten, bis sie gefunden wurde. Das Programm gibt natürlich Hinweise. ... Eure Aufgabe: Programmiert mit!