Programmieren 1

Version 2.2

Inhaltsverzeichnis

1 Algorithmen

1.1 Allgemeines

Ein Computerprogramm arbeitet nach dem EVA-Prinzip.
EVA-Prinzip steht für **E**ingabe, dann **V**erarbeitung und danach die **A**usgabe von Daten.
Am Anfang der Programmierung steht immer irgendein Problem. Man möchte Daten einlesen, irgendwie verarbeiten und wieder ausgeben.
Das Einlesen der Daten, die Verarbeitung und die Ausgabe der Daten sind die Aufgabenbereiche, für die ein Programmierer Lösungen entwickeln muss.

Ein so genannter Algorithmus beschreibt für den Computer exakt, welche Arbeitsschritte erforderlich sind, um die eingehenden Daten so zu verarbeiten, dass dabei das gewünschte Ergebnis herauskommt.
Eine eindeutige Arbeitsanleitung für einen Computer bezeichnet man als *Algorithmus*.
Der Begriff des Algorithmus ist ein zentraler Begriff der Programmierung.
Solche Handlungsanleitungen bestehen aus bestimmten Anweisungen, die, wenn sie in der vorgegebenen Reihenfolge ausgeführt werden, zu dem gewünschten Ergebnis führen (→ verwandte Bedeutungen: Kochrezept, Bastelanleitung, Spielregeln).

1.2 Ein Beispielalgorithmus

Überlegen Sie folgende Aufgabenstellung →
Beispiel:
Beschreiben Sie die Tätigkeiten die erforderlich sind, um Kaffee zu kochen.
Ergebnis:
„Ich gehe zur Kaffeemaschine, fülle Kaffee und Wasser ein, schalte die Maschine ein und warte, bis der Kaffe fertig ist."
Beurteilung:
Ist dieser Algorithmus ausreichend exakt formuliert, um einem Computer klar zu machen, welche Tätigkeiten zum Kaffeekochen erforderlich sind?
Leider funktioniert dieser Algorithmus so im Computer nicht, wir haben wesentliche Dinge vernachlässigt.

Erweiterter Algorithmus:

Zur Kaffeemaschine gehen
Den Kaffeebehälter herausnehmen
Mit dem Kaffeebehälter zum Wasserhahn gehen
Den Wasserhahn aufdrehen, bis genügend Wasser herauskommt
Den Kaffeebehälter unter den Wasserhahn halten, bis dieser ausreichend gefüllt ist
Den Kaffeebehälter zurückziehen
Den Wasserhahn zudrehen
Zur Kaffeemaschine gehen
Das Wasser einfüllen
Den Kaffeebehälter in die Kaffeemaschine stellen
Zum Kaffeeschrank gehen
Den Kaffeefilter herausnahmen
Den Kaffee herausnehmen
Zur Kaffeemaschine gehen
Den Kaffeefilter einlegen
Kaffee einfüllen, bis genügend Kaffee vorhanden ist
Die Kaffeemaschine einschalten
Zum Kaffeeschrank gehen
Den Kaffee wieder hineinstellen
Warten bis der Kaffee fertig ist

Dachten Sie, dass Kaffeekochen so kompliziert ist ?
Für Computer schon, da die eine exakte Arbeitsanweisung für jeden einzelnen Arbeitsschritt benötigen. Sie haben keine Möglichkeit, bzw. besitzen nicht die Fähigkeit, Aussagen zu interpretieren oder zu deuten.

Es wurden hier sogar einige Sonderfälle weggelassen: Was ist z.B., wenn kein Wasser, kein Kaffee oder kein Filter vorhanden ist?

Im Pseudocode würde der **Kaffeekochen-Algorithmus** etwa so aussehen:

```
Zur Kaffeemaschine gehen
Die Kaffeekanne herausnehmen
Mit der Kaffeekanne zum Wasserhahn gehen
Solange bis genügend Wasser herauskommt
    Den Wasserhahn aufdrehen
Wiederholen
Solange bis die Kaffeekanne ausreichend gefüllt ist
    Die Kaffeekanne unter den Wasserhahn halten
Wiederholen
Die Kaffeekanne zurückziehen
Solange bis der Wasserhahn geschlossen ist
    Den Wasserhahn zudrehen
Wiederholen
Zur Kaffeemaschine gehen
Das Wasser einfüllen
Die Kaffeekanne in die Kaffeemaschine stellen
Einen Kaffeefilter aus Schachtel nehmen
Den Kaffeefilter in Filterbehälter einlegen
Solange bis genügend Kaffee eingefüllt ist
    Kaffee einfüllen
Wiederholen
Die Kaffeemaschine einschalten
Solange bis der Kaffee fertig ist
    Warten
Wiederholen
```

Dieser Algorithmus arbeitet schon mit wichtigen Programmstruktur-Elementen, nämlich **Schleifen**.

Die erste Schleife

```
Solange bis genügend Wasser herauskommt
    Den Wasserhahn aufdrehen
Wiederholen
```

führt z. B. die Anweisung ‚Den Wasserhahn aufdrehen' so oft wiederholt aus, bis die Bedingung, die oben im Schleifenkopf angegeben ist („Bis genügend Wasser herauskommt"), erfüllt ist.
Ist die Bedingung erfüllt, wird die nächste Anweisung nach Wiederholen ausgeführt.

Die in einer Schleife wiederholten Anweisungen - abweichend vom obigen Beispiel können das auch mehrere sein - rückt man üblicherweise etwas ein, um zu erkennen, dass diese Anweisungen zu der Schleife gehören.

Aber der Algorithmus ist, wie schon gesagt, noch nicht perfekt.
Was passiert z. B., wenn kein Wasser vorhanden ist?
Der Algorithmus bleibt in einer Endlosschleife hängen (wenigstens so lange, bis Wasser fließt). Wir müssen also noch etwas nachbessern, nämlich so, dass wir nachfragen, ob Wasser vorhanden ist:

```
Zur Kaffeemaschine gehen
Die Kaffeekanne herausnehmen
Mit der Kaffeekanne zum Wasserhahn gehen
Den Wasserhahn aufdrehen
Wenn Wasser aus dem Wasserhahn kommt dann
    Solange bis genügend Wasser herauskommt
        Den Wasserhahn aufdrehen
    Wiederholen
    Solange bis die Kaffeekanne ausreichend gefüllt ist
        Die Kaffeekanne unter den Wasserhahn halten
    Wiederholen
  .....
Sonst
    Meldung, dass kein Wasser vorhanden ist
Ende Wenn
```

Sie sehen den kleinen Unterschied:
Ich drehe den Wasserhahn vorher auf und überprüfe danach, ob Wasser herauskommt. Wenn Wasser herauskommt, wird der Wasserhahn weiter aufgedreht, bis genügend Wasser eingefüllt ist; danach wird der weitere Algorithmus abgearbeitet. Wenn kein Wasser vorhanden ist, wird nicht weitergearbeitet, sondern nur eine Meldung ausgegeben.

Diese **Verzweigung** ist, neben der Schleife, eine andere wichtige Programmstruktur.
Jede Programmiersprache enthält Verzweigungen und Schleifen.

Der komplette *Algorithmus zum Kaffeekochen*:

```
Wenn eine Kaffeemaschine vorhanden ist dann
    Zur Kaffeemaschine gehen
    Wenn eine Kaffeekanne vorhanden ist dann
        Die Kaffeekanne herausnehmen
        Wenn ein Wasserhahn vorhanden ist dann
            Mit der Kaffeekanne zum Wasserhahn gehen
            Den Wasserhahn aufdrehen
            Wenn Wasser herauskommt dann
                Solange bis genügend Wasser herauskommt
                    Den Wasserhahn aufdrehen
                Wiederholen
                Solange bis die Kaffeekanne ausreichend gefüllt ist
                    Die Kaffeekanne unter den Wasserhahn halten
                Wiederholen
                Die Kaffeekanne zurückziehen
                Solange bis der Wasserhahn geschlossen ist
                    Den Wasserhahn zudrehen
                Wiederholen
                Zur Kaffeemaschine gehen
                Das Wasser in den Wassertank einfüllen
                Die Kaffeekanne in die Kaffeemaschine stellen
                Wenn Kaffeepulver vorhanden ist dann
                    Wenn Kaffeefilter vorhanden sind dann
                        Einen Kaffeefilter herausnehmen
                        Den Kaffeefilter einlegen
                        Solange bis genügend Kaffee eingefüllt ist
                            Kaffee einfüllen
                        Wiederholen
                        Behälterdeckel schließen
                        Die Kaffeemaschine einschalten
                        Solange bis der Kaffee fertig ist
                            Warten
                        Wiederholen
                    Sonst
                        Meldung, dass kein Kaffeefilter vorhanden ist
                    Ende Wenn
                Sonst
                    Meldung, dass kein Kaffeepulver vorhanden ist
                Ende Wenn
            Sonst
                Meldung, dass kein Wasser vorhanden ist
            Ende Wenn
        Sonst
            Meldung, dass kein Wasserhahn vorhanden ist
        Ende Wenn
    Sonst
        Meldung, dass keine Kaffeekanne vorhanden ist
    Ende Wenn
Sonst
    Meldung, dass keine Kaffeemaschine vorhanden ist
Ende Wenn
```

Dieses Beispiel zeigt eines sehr deutlich:
Computer sind dumm. Sie müssen dem Computer alles, aber auch wirklich alles, sagen, damit dieser korrekt arbeitet. Sie können nicht voraussetzen, dass der Computer abstrakt formulierte Anweisungen wie ein Mensch interpretieren kann. Einem Menschen können Sie einfach sagen: »Geh Kaffee kochen«, einem Computer nicht.

Programme im Computer - Anweisungen, Befehle, Variablen, Schleifen und Verzweigungen:
Programme im Computer bestehen also aus Anweisungen, die je nach Bedarf mit Schleifen wiederholt ausgeführt und mit Verzweigungen bedingungsabhängig ausgeführt werden. Anweisungen sind Befehle (die zu einer Aktion führen) oder Zuweisungen von Berechnungen an Variablen. Variablen werden verwendet, um Zwischenergebnisse für eine weitere Verwendung zu speichern.

1.3 Weitere Beispiele

1.3.1 Verzweigung

Wenn eine Bedingung erfüllt ist, dann tue dies, ansonsten tue das.

Bsp: Fußballspiel - ein Spieler von Mannschaft A wird von einem Spieler von Mannschaft B gefoult.

Spielregel:
```
Wenn das Foul im Strafraum von Mannschaft B erfolgt
Dann pfeife Elfmeter
Ansonsten pfeife Freistoß
```

1.3.2 Schleife

Solange eine Bedingung erfüllt ist tue folgendes.

Bsp: Anleitung Mensch-Ärgere-Dich-Nicht-Spiel

```
Solange ein Spieler eine 6 würfelt
        Darf er nochmals würfeln
```

1.3.3 Programmaufbau

Im Allgemeinen bestehen also Programme aus 2 Teilen:
Einer Auflistung bestimmter Voraussetzungen bzw. Zubehör und der eigentlichen Arbeitsanweisung.

Bsp: Jenga Spiel

```
Zubehör:
        1 Spielanleitung
        45 Holzlötze
Anleitung:
        Solange die Spieler Lust haben zu spielen:
                Turm aufbauen, dabei jeweils die Klötzchen rechtwinkelig
                zueinander versetzen;
                Solange der Turm noch nicht eingestürzt ist, müssen die
                Spieler der Reihe nach folgendes tun:
                        Ein Klötzchen aus dem Turm entnehmen;
                        Das Klötzchen oben auf den Turm legen.
```

1.4 Definition des Begriffes Algorithmus

1.4.1 Ausführung von Algorithmen

Bei der Ausführung eines Algorithmus läuft ein so genannter *‚Prozess'* ab, der durch einen *‚Prozessor'* gesteuert. Erfolgt die Abarbeitung durch einen Menschen, so ist dieser der Prozessor.

Computer (als *‚Prozessor')* haben Menschen gegenüber einige Vorteile:

- hohe Rechengeschwindigkeit
- große Zuverlässigkeit (kein ermüden, relativ fehlerlos)
- gewaltige Speicherfähigkeit

1.4.2 Eigenschaften von Algorithmen

- *Eindeutigkeit*: eindeutige Beschreibung des Lösungsweges → keine widersprüchlichen Anweisungen
- *Parametrisierbarkeit, bzw. allgemeine Gültigkeit*:
  Ein Algorithmus darf nicht nur ein spezielles Problem lösen, sondern muss für alle gleichartigen Aufgaben in jeder Verarbeitungssituation anwendbar sein (z.B.: A=B+C muss für alle beliebigen Zahlen gelöst werden können, nicht nur für bestimmte Zahlen).
- *Finitheit*: Beschreibung besitzt endliche Länge
- *Ausführbarkeit*: nicht durchführbare Anweisungen (z.B. Division durch Null) dürfen nicht sein
- *Terminierung*: Algorithmen müssen nach endlich vielen Schritten terminieren, d.h. sie müssen ein Ergebnis liefern und anhalten
- *Determiniertheit*: Algorithmen müssen wenn sie mit gleichen Startbedingungen ausgeführt werden immer die gleichen Ergebnisse liefern.
- *Determinismus*: Zu jedem Zeitpunkt der Ausführung besteht lediglich eine Möglichkeit der Weiterführung.

1.4.3 Praxisrelevante Eigenschaften von Algorithmen:

Es gibt für jedes Problem viele verschiedene Möglichkeiten für einen entsprechenden Algorithmus, die alle das gleiche Ergebnis liefern. Kriterien die in der Praxis bei der Findung oder der Auswahl eines Algorithmus eine wesentliche Rolle spielen sind folgende:

- *Effizienz*: so schnell wie möglich und mit so wenig Ressourcen wie möglich zu einem Ergebnis kommen.
- *Speicherbedarf*: Die Beschreibung sollte möglichst knapp sein, aber die Lesbarkeit darf nicht darunter leiden.
- *Erweiterbarkeit*: ohne großen Aufwand an geänderte Anforderungen anpassbar.
- *Wiederverwendbarkeit*: Algorithmen sollen so formuliert werden, dass sie auch zur Lösung von Teilproblemen in anderen Zusammenhängen nutzbar sind.
- *Portabilität*: nicht auf einen bestimmten Computertyp zugeschnitten.
- *Zuverlässigkeit*: korrekte und vollständige Problemlösung.

1.4.4 Programme

Die Beschreibung eines Algorithmus mit Hilfe einer Programmiersprache wird als Programm bezeichnet.

1.4.5 Regeln für das Schreiben eines (Python-)Programms

Zum einen werden durch die Programmiersprache exakte Regeln vorgegeben, wie die korrekte Formulierung einer Anweisung ausschauen muss (z.B. in Python: "Eine Funktion wird mit dem Schlüsselwort ‚def' festgelegt.") – diese Bestimmungen nennt man die ***Syntax*** der Programmiersprache.
Diese Syntax wird um (allgemeine und spezielle) *verbindliche* ***Programmierrichtlinien*** ergänzt - sogen. ‚*Coding Conventions*':

Name = Bezeichner = Identifier

- Je Zeile maximal eine (einfache) Anweisung.
- Einrückungen immer mit 4 Leerzeichen!
- Zwei Leerzeilen vor erster Funktionsdefinition; jeweils zwei (2) Leerzeilen zwischen Funktionsdefinitionen.
- Keine Leerzeichen vor der runden Klammer von Funktionsnamen.
- Verwendung sprechender Namen (für Funktionen, Variable, Methoden, usw.) – sie müssen aussagekräftig sein, d.h., man soll vom Bezeichner auf die Wirkung/den Zweck der Funktion, den Inhalt einer Variablen, usw. schließen können. U. U. ist ein (zusätzlicher) Kommentar notwendig.
- Keine Umlaute oder Sonderzeichen in Namen (auch wenn sei von der Programmiersprache erlaubt werden).
- Funktionsnamen immer mit Kleinbuchstaben geschrieben, Unterstrich („_") zur Trennung einzelner Wörter (z.B. get_area) oder in ‚mixedCase'-Darstellung (z.B. getArea); der erste Buchstabe eines Funktionsnamens muss immer ein Kleinbuchstabe sein!
- Jede Variablendeklaration muss in einer eigenen Zeile stehen.
- Variablennamen beginnen immer mit Kleinbuchstaben.
- Innerhalb einer Anweisung gilt: Keine Leerzeichen vor Komma, Strichpunkt, Punkt oder Doppelpunkt; ein Leerzeichen nach Komma, Strichpunkt und Doppelpunkt, aber kein Leerzeichen nach einem Punkt.
- Links und rechts von jedem binären Operator genau ein Leerzeichen – z.B.: `c = a ** 2 + b ** 2`
- Keine Leerzeichen um ein =, wenn dieses ein Keyword-Argument oder einen Defaultwert bei Funktionen angibt – z.B.:

```
def sum(a, b=0, c=0):
    return a + b + c
```

- Dateinamen, Verzeichnisnamen dürfen keine Umlaute, Sonderzeichen und Leerzeichen enthalten.
- Globale Variablen nicht verwenden! (bzw. vermeiden)
- Jede Programmdatei muss einen Kommentarkopf enthalten.
- Konstante definieren. Es dürfen keine Zahlen "unmotiviert" im Code verwendet werden.
- Bei Vergleichen sollte die Konstante immer links stehen, damit nicht versehentlich eine Zuweisung entsteht.

- Importanweisungen am Anfang der Datei, jeweils in einer eigenen Zeile.
- Maximale Zeilenlänge von 79!

Beispiele und Ergänzungen zu den o.a. Richtlinien und Regeln:

➲ **Jeder Befehl steht in einer eigenen Zeile** - Keine „Befehlsschlangen" erzeugen!

➲ **Leerzeilen einfügen** - Keinen „Spaghetti-Code" erzeugen!

Um die Lesbarkeit zu erhöhen, werden Programmteile, die innerhalb einer Funktion *eine logische Einheit* bilden, durch Einfügen einer Leerzeile (engl.: blank line) optisch vom anderen Programmcode abgesetzt.

➲ **Einrückungen -** Strukturierte Programmierung

Damit man Programmstrukturen und -zusammenhänge leichter erkennen kann, werden alle Befehle eines Blocks, d.h., alle Anweisungen, die zur gleichen Programmstruktur gehören, um jeweils 4 Leerzeichen (nach rechts) eingerückt (⇒ Tabulator entsprechend einstellen).
Außerdem werden durch diese strukturierte Schreibweise die jeweiligen Verarbeitungsebenen deutlich gemacht → die „weiter rechts liegende" Verarbeitungsebene muss vollständig abgeschlossen sein, bevor die davon „links liegende" Verarbeitungsebene weiter ausgeführt wird.

➲ **Kommentare**

Ein Kommentar beschreibt in kurzen und prägnant formulierten Sätzen, welcher Aufgabenteil in einem Abschnitt – z.B. in einer Funktion - gelöst wird. Um ein Programm auch für andere Programmierer (als nur den Autor selbst) (leichter) lesbar zu machen, muss jedes [gute] Programm aussagekräftige Kommentare für alle wesentlichen Programmteile enthalten.
Jede Funktion beginnt mit einem Kommentar, in dem beschrieben wird, welche (Teil)Aufgabe von dieser Funktion ausgeführt, bzw. gelöst wird.
Der Kommentar für die Hauptfunktion (das ‚Hauptprogramm') enthält üblicherweise noch Informationen über die Programmversion und den/die Autor/en.
Schwer zu verstehende oder komplexe Abschnitte des Programms werden durch *zusätzliche Block- oder Zeilenkommentar(e)* erläutert.

Die Zeichenfolge # dient zum Erzeugen einzeiliger Kommentare. Alles, was hinter # steht, wird bis zum Zeilenende nicht als Programmanweisung interpretiert und kann somit beliebigen Text enthalten.

➲ **Gestaltung des Programmcodetextes – Formatierung**

Je Textzeile wird eine Anweisung geschrieben.
Wenn eine Programmcodezeile so lang ist, dass sie über den rechten Bildschirmrand hinaus ragt, muss sie umgebrochen werden, sodass der vollständige Programmcode ohne Hin- und Herschieben des Bildschirmfensters sichtbar ist. →

→ Das bedeutet auch, dass das Programmcodefenster (für praktisches und übersichtliches Arbeiten) auf die Größe der Bildschirmfläche aufgezogen werden soll.

Die Umbruchstelle innerhalb der Zeile sollte so gewählt werden, dass die Wortfolge der Anweisung trotzdem den logischen Zusammenhang richtig darstellt;
d.h. z.B.: Zeilenumbruch nach einem Komma; oder Zeilenumbruch vor einem Operator .
Ein expliziter Zeilenumbruch, der den inhaltlichen Zusammenhang einer Anweisung ermöglicht, wird in Python mit dem Zeichen „\" (‚backslash') erreicht.

Die *Schriftart* des Programmcodetextes darf *keine Proportionalschrift* sein, sondern eine, in der alle Zeichen dieselbe Breite haben, ansonsten erkennt man die Einrückungen nicht; also z.B. `Courier New` .

➲ **Namenskonventionen**

Python unterscheidet zwischen Groß- und Kleinschreibung.

Bei der Benennung von Funktionen (und später von Variablen und anderen Strukturen) haben sich gewisse Regeln eingespielt, die nach Möglichkeit beachtet werden sollten. Dadurch erleichtert man anderen Programmierern, fremde Programme mit geringerem Aufwand zu verstehen.
Folgende Festlegungen sind zu beachten:
-) Namen können von beliebiger Länge sein. Sie beginnen mit Buchstaben (a-z und A-Z; ohne Umlaute) und dürfen einen Unterstrich _ und Ziffern (nicht als erstes Zeichen) enthalten.

-) Namen beginnen grundsätzlich mit einem Kleinbuchstaben, z.B.: laenge, breite, zaehler, …. .
-) Zur besseren Lesbarkeit wird ein neuer Wortstamm, d.h., jedes weitere Teilwort nach dem ersten, wieder mit einem Großbuchstaben begonnen, z.B.: pruefWert, eingabeWert, geburtsDatum.
-) Ein Funktionsname sollte ein Verb enthalten, z.B.: einlesenPersonendaten() ; differenzenBerechnen()

➲ Klammernsetzung

Die (großzügige) Verwendung von (runden) Klammern, um eine Anweisung(szeile) logisch zu gliedern, erhöht im Allgemeinen die Lesbarkeit und Verständlichkeit des Programmtextes merklich.
Vor allem bei Ausdrücken mit mehreren, verschiedenen Operatoren wird erst *durch Klammernsetzung* die gewünschte *Verarbeitungspriorität definiert*.

1.5 Wie geht man beim Schreiben eines Programmes vor?

Die zu lösende Gesamtaufgabe wird derart in Teilaufgaben zerlegt, dass jede Teilaufgabe eine inhaltlich und/oder logisch abgeschlossene Einheit darstellt und sie voneinander klar abgrenzbar sind.
Jede (große) Teilaufgabe wird weiter so in kleinere Teilaufgaben zerlegt, wie sie inhaltlich und/oder logisch voneinander abgrenzbar sind.

Als „Richtschnur" (jetzt am Anfang des Programmierenlernens) gilt für das weitere Vorgehen:
jede der so ermittelten Teilaufgaben wird als je eine Unterfunktion dieses Programms erstellt, in der jetzt detailliert beschrieben wird, mit welchen Arbeitsschritten diese bestimmte Teilaufgabe ausgeführt werden soll.

Ausgangspunkt bei der Programmerstellung ist die Hauptfunktion:
in ihr wird der vollständige Verarbeitungsablauf in der Gesamtsicht beschrieben (vergleichbar mit dem Inhaltsverzeichnis eines Buches, wo der Buchinhalt durch die Kapitelbezeichnungen im Überblick dargestellt ist), die einzelnen Arbeitsschritte sind hier vor allem die Aufrufe einer entsprechenden Unterfunktion, in der dann diese Teilaufgabe im Detail beschrieben ist sowie eine Schleife zur Steuerung des Gesamtablaufs.

Die Grundstruktur eines Programms ist das *EVA-Prinzip* – daraus folgt, dass jedes Programm zumindest aus einer Unterfunktion für die Teilaufgabe ‚Eingabe von Daten' und einer Unterfunktion für die Teilaufgabe ‚Ausgabe von Daten' sowie der (immer notwendigen) Hauptfunktion besteht.
In der Hauptfunktion wird, wie oben dargestellt, die Verarbeitung der Daten mittels der erforderlichen weiteren Unterfunktionen beschrieben.

Die Vorgehensweise bei der Entwicklung der (Unter-)Funktionen ist wie folgt:
entsprechend dem geforderten Verarbeitungsablauf wird eine Funktion nach der anderen vollständig erarbeitet; d.h., dass die Funktion, die gerade erstellt wird, solange bearbeitet und getestet werden muss, bis das erwartetet Ergebnis vorliegt - erst dann darf mit der Ausarbeitung der nächsten Funktion begonnen werden.

Der Sinn dieser Arbeitsweise ist, dass auftretende Fehler somit im Prinzip nur in der gerade bearbeiteten Funktion oder in der Schnittstelle zwischen der Hauptfunktion, bzw. der aufrufenden Programmstelle und dieser Funktion liegen können und sich die Fehlersuche auf diesen überschaubaren Programmcodebereich konzentrieren kann.

→ *Exkurs:* Wie lese / interpretiere ich den Quellcode eines Programms?

Grundsätzlich wird eine Anweisungszeile eines Computerprogramms von rechts nach links gelesen – dabei muss – wie auch sonst beim Lesen eines Textes – jedes Zeichen beachtet und seine Bedeutung analysiert werden, um die Wirkungsweise des betrachteten Befehls zu verstehen.
Ausgangspunkt beim Lesen eines Computerprogramms ist die erste Zeile der Hauptfunktion und dann wird Schritt für Schritt eine Zeile (= eine Anweisung) nach der anderen analysiert.
Ist eine Anweisung ein Funktionsaufruf, dann wird zum Programmcode dieser Funktion verzweigt und wieder für jede Befehlszeile, von rechts nach links fortschreitend, die Bedeutung jedes Wortes, bzw. Zeichens ermittelt und erklärt.

2 Programmieren mit Python

2.1 Begriffe mit ihrer Bedeutung

IDE (integrated development environment) :
ist ein Anwendungsprogramm zur Entwicklung von Software.

IDLE ist die integrierte Entwicklungsumgebung von Python. Sie umfasst u.a. einen Texteditor, einen Debugger und einen Interpreter, und ist unter den Betriebssystemen Windows und Linux verwendbar.

Texteditor (lat. Editor: „Herausgeber", „Erzeuger"):
ist ein Computerprogramm zum Bearbeiten von Texten. Der Editor lädt die zu bearbeitende Textdatei und zeigt ihren Inhalt auf dem Bildschirm an. Durch diverse Aktionen können die Daten dann bearbeitet werden. Zu diesen Aktionen kann das Einfügen, Löschen und Kopieren gehören.

Debugger (von engl. „bug" im Sinne von Programmfehler):
ist ein Werkzeug zum Diagnostizieren und Auffinden von Fehlern in Programmen.

Interpreter:
ist ein Computerprogramm, das einen Programm-Quellcode einliest, analysiert und ausführt. Die Analyse des Quellcodes erfolgt also zur Laufzeit des Programms.

Shell bzw. Konsole: ist die Eingabe/Ausgabe-Schnittstelle zwischen Computer und Benutzer.

Prompt (Eingabeaufforderung):
bezeichnet eine Markierung in der Shell (Eingabekonsole), welche auf die Stelle verweist, an der man z.B. Kommandozeilenbefehle eingeben kann.

GUI (engl. „Graphical User Interface", grafische Benutzeroberfläche):
erlaubt dem Benutzer eines Computers die Interaktion mit der Maschine über grafische Symbole(z.B. Arbeitsplatz, Symbole, Papierkorb, Menü)

Objekte sind Datenelemente
- mit Eigenschaften (Variablen) und
- mit Methoden (Funktionen), die jeweils an ein bestimmtes Objekt gebunden sind und

zusammen eine Einheit bilden, die so gekapselt („eingehüllt") wird, dass nur mehr über definierte Schnittstellen mit diesem Objekt kommuniziert werden kann.

2.2 Anweisungen und Anweisungsfolgen

2.2.1 Allgemeines

Ein Programm setzt sich aus einer Menge von Anweisungen zusammen, die in einer bestimmten Anordnung einen bestimmten Algorithmus bilden. Diese Anweisungen stehen in einer vom Programmierer festgelegten Reihenfolge untereinander und werden vom Computer auch in dieser Reihenfolge verarbeitet. Diese Form der Abarbeitung wird als sequentielle Verarbeitung bezeichnet.

2.2.1.1 Einfache Anweisungen

Eine **Anweisung** ist die **kleinste ausführbare Einheit** eines Programms;
m.a.W: Eine Anweisung ist eine Verarbeitungsvorschrift.
In Python wird üblicherweise pro Zeile eine Anweisung angeschrieben.

2.2.1.2 Anweisungsfolgen

Eine **Folge von Anweisungen** wird auch **Sequenz** genannt.
Eine Sequenz ist eine Liste von Anweisungen, die nacheinander von oben nach unten (sequentiell) je einmal abgearbeitet wird. Ein solcher Ablauf stellt die einfachste Struktur eines Programms dar und wird für die Lösung einfach strukturierter Probleme verwendet.

2.3 Kommentare

Kommentare dienen dazu, ein Programm lesbarer zu gestalten.
Kommentare über mehrere Zeilen werden mit `"""` (am Beginn und am Ende) gekennzeichnet. Ein Kommentar in einer Zeile wird mit `"` (am Beginn und am Ende) gekennzeichnet.

Im Editor erscheinen Kommentare in roter Farbe.

Beispiel: Jedes Programm muss am Beginn einen Programmkopf haben.

```
"""
author: Mustermann Max
matnr:  d10123
file:   dreieck.py
desc:   Dieses Modul enthält Funktionen zur Dreiecksberechnung.
        Als solches bla bla...
date:   2010-04-26
class:  1A
catnr:  123
"""
```

Kommentar für eine Zeile:
Das Zeichen `#` leitet einen Kommentar ein. Der Rest dieser Zeile wird als Kommentar aufgefasst.

2.4 Bezeichner (Name, identifier)

Regeln für Bezeichnernamen:

- dürfen nur Buchstaben, Ziffern und Unterstrich (‚underline') enthalten
- müssen mit Buchstaben beginnen (ein Unterstrich darf nur in Ausnahmefällen das Anfangszeichen sein)
- Groß- und Kleinschreibung muss berücksichtigt werden (‚case-sensitiv')
  d.h. „Nimm" und „nimm" sind nicht der selbe Bezeichner
- sie müssen sprechend sein
- es dürfen keine reservierten Wörter (Schlüsselwörter) sein

Beispiele:

```
anzahl      #richtig
wert_2      #richtig
1Wert       #falsch, weil Ziffer an erster Stelle steht
ein Wert    #falsch, weil ein Blank enthalten ist
```

Richtlinien für Schreibweise von Bezeichnern:
Objektname (Variablenname): mit kleinem Anfangsbuchstaben
Klassenname: mit großem Anfangsbuchstaben

2.5 Datentypen

Zahlen

- Ganze Zahlen

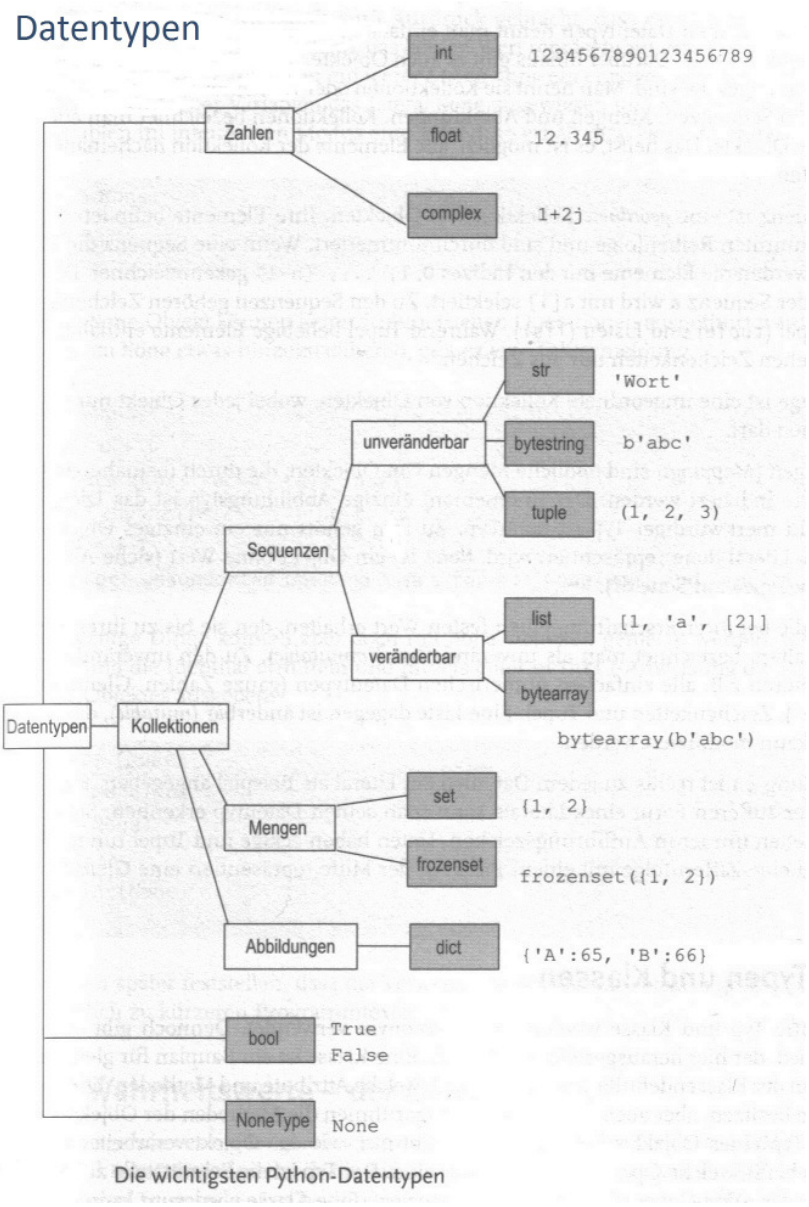
`integer` Bsp.: `12` (*Hinweis*: maximale Größe $2^{31}-1$ =2147483647)

`long` Bsp.: `1234569978934L`

- Zahlen mit Nachkommateil (float)

Bsp: `23.56` (*Achtung*: immer Dezimalpunkt)

- Komplexe Zahlen



Die wichtigsten Python-Datentypen

Zeichenketten (`String`)
werden von einfachen oder doppelten Kommas eingerahmt.

Bsp:
```
'Hallo Welt!'
"Hallo Welt!"
'"Yes," he said.'
'doesn\'t'
# \ ist das Escape-Zeichen
```

Wenn sich ein String über mehrere Zeilen erstreckt, müssen Anfang und Ende mit `"""` gekennzeichnet werden.

Bsp.:
```
Name=""" Susi
Müller"""
```

Sequenzen (sequence) nehmen eine feste Folge anderer Objekte auf. Das können bspw. auch weitere Sequenzen sein.
Die Elemente sind in einer Reihenfolge angeordnet; die gleichen Werte können mehrmals vorkommen.

Sequenzen können veränderbar oder fix sein ⇨

Listen (`list`): sind **veränderbar**e Sequenzen; durch *eckige Klammern* gekennzeichnet
Bsp.: L = [1, 3]; L.append(7); print L[2]
Ausgabe: [7]

Tupel (`tuple`): sind **unveränderbar**e Sequenzen, d.h., Tupel können, wenn sie einmal erzeugt sind, nicht mehr verändert werden (außer durch eine Neuzuweisung).
Sie sind durch *runde Klammern* gekennzeichnet.

Bsp.: T = (1, 3);
Ausgabe: (1, 3)

2.6 Ein- und Ausgabefunktionen

Ausgabe

Funktion `print(……)`: Ausgeben einer Zeichenfolge (d.h., eines String)
mit Zeilenschaltung: `print([object, ...], *, sep=' ', end='\n', file=sys.stdout)`

Bsp:
```
print("Hallo Welt!")        #Zeichenfolge wird ausgegeben
print ('Hallo Welt')
```

Bsp:
```
print("Hallo Welt!",)       #Beistrich unterbindet Zeilenumbruch
```

Bsp:
```
print()    #gibt eine Leerzeile aus
```

Bsp:
```
wert1=34     # ganze Zahl
wert2=3.5    # Gleitkommazahl
#Durch Beistriche getrennt werden Texte und Zahlen in der gewünschten
#Reihenfolge ausgegeben
 print("Wert1:",wert1,"Wert2",wert2)
#Um Zahlenwerte in einen Text einzubinden, müssen sie vorher in eine
#Zeichenkette(String) umgewandelt werden – das geschieht hier mit der
#Funktion ‚str'
 print("Wert1 : " + str(wert1) + " , Wert2: " + str(wert2))
```

Einlesen von der Konsole

Funktion `input()`: liefert immer den Datentyp ‚`string`' (von der Konsole).

Bsp: Eine Zeichenfolge wird (bis zum Zeilenumbruch) in eine Variable *name,* die vom Datentyp `string` sein muss, eingelesen: `name = input()`

Bsp: Einlesen einer ganzen Zahl - die Zeichenkette muss in den Datentyp `integer` umgewandelt werden: `zahl = int(input())`

Bsp: Einlesen einer Kommazahl, die Zeichenkette muss in den Datentyp `float` umgewandelt werden: `zahl = float(input())`

Einlesen über einen grafischen Eingabedialog: mit der Funktion `numinput()`.

Bsp: Einlesen einer Zahl (ganze Zahlen oder Nachkommazahlen) mit einem Text, der zur Eingabe auffordert: `wert5 = numinput("Wert der Seitenlänge: ")`

Formatieren von Zahlen bei der Ausgabe

```
pwd='abc'
uid=12  #eine ganze Zahl soll ausgegeben werden
print ("%s is not a good password for %d" % (pwd, uid) )

wert=12/3.4   #Zahl mit genau 4 Nachkommastellen soll ausgegeben werden,
              #bei Bedarf wird gerundet

print ("Das Ergebnis lautet %2.4f" % (wert) )

print ('%(language)s has %(#)03d quote types.' % \
        {'language': "Python", "#": 2})

#Mit dem Schlüsselwort ‚end' kann man den automatischen Zeilenumbruch
#durch ein anderes Zeichen ersetzen:
print('Susi',end = ";")  #Anstelle des Zeilenumbruchs wird ein Strichpunkt
                         #ausgegeben.

#Mit dem Schlüsselwort ‚sep' kann man ein gewünschtes Trennzeichen
#zwischen den einzelnen Teilstrings angeben:
print('Susi', 'Hansi', 'Anna', sep = ";")
```

2.7 Variablen

Eine Variable ist der Name, der auf ein (Daten-)Objekt verweist → dieser Verweis wird auch als *‚Referenz'* auf das Objekt bezeichnet und (oft) *als ‚Zeiger'* dargestellt; der Typ der Variablen wird durch die Zuweisung eines (Daten-)Werts festgelegt.

2.7.1 Lokale und globale Variablen

Alle Variablennamen werden in einem Wörterbuch eingetragen →

Globale Variablen: Beim Ausführen eines Python-Scripts werden diese ins globale Wörterbuch eingetragen.

Lokale Variablen: Jede Funktion hat ihr eigenes Wörterbuch, in dem alle Variablen, denen innerhalb der Funktion ein Wert zugewiesen wird, eingetragen werden. Das heißt, *alle Variablen in Funktionskörpern, die links vom ‚=' stehen, kommen ins lokale Wörterbuch*. Lokale Variablen werden innerhalb von Funktionen verwendet; sie sind so lange gültig, wie der Anweisungsblock gültig ist (d.h., abgearbeitet wird), in dem sie deklariert wurden. Danach werden sie aus dem Speicher entfernt.

Priorität hat ein lokales Wörterbuch vor dem globalen Wörterbuch!!

2.7.2 Initialisierung

Variablen erhalten durch eine Zuweisungsoperation einen Wert, und dadurch wird auch der (Daten)Typ der Variablen festgelegt.

ddfee

Bsp.:
```
x = 3                  #ganze Zahl → Typ ‚integer'
y = x/3.4              #Nachkommazahl → Typ ‚float'
s = "Susi"             #Zeichenkette → Typ ‚string'
z = None               #'None' ist der „Nullwert" (der ‚leere' Wert)
```

Der Typ einer Variablen kann mit der Funktion `type(<variablenname>)` abgefragt werden.

Hinweis: Konstanten, d.h. Namen, die einen unveränderlichen Wert darstellen, gibt es hier nicht.

2.8 Einfache Anweisungen und ihre Operatoren

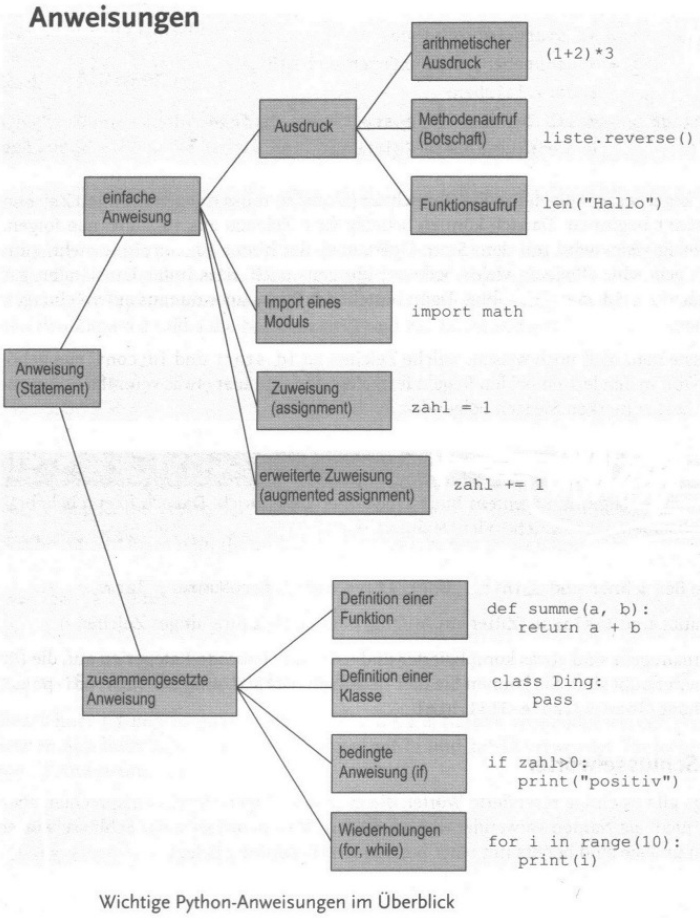
Einfache Anweisungen sind *Ausdrücke* und *Zuweisungen* → ***siehe Bild***.

Hinweis: Eine Anweisung wird durch das Zeilenende beendet.

Eine Anweisung wird jedoch fortgesetzt, wenn am Ende der vorhergehenden Zeile ein

Backslash \ steht oder
noch Klammern "offen sind."

Beispiel:
```
Erg = wert * 34
Erg = (wert
       * 34)
Erg = wert \
       * 34
```



Wichtige Python-Anweisungen im Überblick

Anweisungen, die zur selben logischen Verarbeitungsgruppe gehören, werden gleichmäßig eingerückt ⇨ *Tab-Taste*

Ein Operator ist ein Rechenbefehl, der zwei Ausdrücke miteinander verknüpft:

2.8.1 Arithmetische Operatoren

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
//	Division, wobei der Nachkommateil abgeschnitten wird - *Bsp:* 17 // 3 liefert 5
**	Potenzieren
%	berechnet den Rest einer Division - *Bsp:* 10 % 3 liefert 1

→ Das Ergebnis eines arithmetischen Ausdrucks ist ein Zahlenwert.

2.8.2 Vergleichsoperatoren

==	Gleichheit
!=	Ungleichheit
<=	kleiner gleich
<	kleiner
>	größer
>=	größer gleich

→ Das Ergebnis eines Vergleichs ist immer entweder `true` oder `false`.

2.8.3 Zusammengesetzte Operatoren

+=	Addition mit anschließender Zuweisung: `a += b` entspricht `a = a + b`
-=	Subtraktion mit anschließender Zuweisung: `a -= b` entspricht `a = a - b`
*=	Multiplikation mit anschließender Zuweisung: `a *= b` entspricht `a = a * b`
/=	Division mit anschließender Zuweisung: `a /= b` entspricht `a = a / b`
//=	
**=	
%=	

2.8.4 Boolesche Operatoren

not not-Operator ! : Dieser Operator verkehrt das Ergebnis in das Gegenteil, aus `true` wird `false` und umgekehrt.

Operand	Ergebnis von !
T	F
F	T

and und-Operator && : Dieser Operator liefert nur dann `true`, wenn beide Operanden `true` sind, sonst liefert er immer `false`. Der zweite Operator und die nachfolgenden Operatoren werden nicht mehr ausgewertet, wenn der erste bereits `false` ergibt.

Operand1	Operand2	Ergebnis von &&
T	T	T
T	F	F
F	T	F
F	F	F

or oder-Operator || : Dieser Operator liefert nur dann `false`, wenn beide Operanden `false` sind, sonst liefert er immer `true`. Ist der erste Operator bereits `true`, wird der restliche Ausdruck nicht mehr ausgewertet.

Operand1	Operand2	Ergebnis von ||
T	T	T
T	F	T
F	T	T
F	F	F

Das Ergebnis einer logischen Operation ist immer entweder `True` oder `False`.

2.8.5 Bit-string - Operationen auf Integer-Zahlen

`x << n` left shift der Inhalt von *x* wir um n Bits nach links verschoben
`x >> n` right shift der Inhalt von *x* wir um n Bits nach rechts verschoben

Hinweis: n muss eine positive ganze Zahl sein.

`&` bitweises AND
`|` bitweises OR
`^` bitweises XOR
`~` bitweises NOT [Negation(d.h. Invertierung)]

2.9 Konvertierung

Das Umwandeln von einem Datentyp in einen anderen nennt man konvertieren.
Aus einem String kann zum Beispiel eine Zahl werden, wenn der String nur Ziffern enthält.
Andernfalls wird eine Fehlermeldung beim Versuch der Konvertierung ausgegeben.

Die folgende Tabelle enthält einige Konvertierungsfunktionen:

Funktion	Konvertiert von	Konvertiert nach	Beispiel
int()	string, float	ganze Zahl	int("33")
float()	string, int	Fließkommazahl	float(1)
str()	int, float	string	str(69)
unicode()	String, Zahl	Unicode String	unicode(3.14)
ord()	Zeichen	ganze Zahl (ASCII-Codewert)	ord('A')
chr()	ganze Zahl (ASCII-Wert)	Zeichen	chr(71)

Struktogramme

Quelle: http://de.wikipedia.org/wiki/Nassi-Shneiderman-Diagramm

Entwurfsmethode für die strukturierte Programmierung (prozedurale Programmiersprachen), die 1972/73 von Dr. Isaac Nassi und Dr. Ben Shneiderman entwickelt wurde; es ist genormt nach DIN 66261.

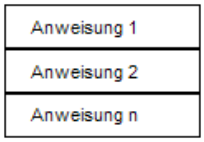
Die Methode zerlegt das Gesamtproblem, das man mit dem gewünschten Algorithmus lösen will, in immer kleinere Teilprobleme, bis schließlich nur noch elementare Grundstrukturen wie Sequenzen und Kontrollstrukturen zur Lösung des Problems übrig bleiben.
Diese können dann durch ein Struktogramm visualisiert werden. Die Vorgehensweise entspricht der sogenannten Top-down-Programmierung, in der zunächst ein Gesamtkonzept entwickelt wird, das dann durch eine Verfeinerung der Strukturen des Gesamtkonzeptes aufgelöst wird.

Für die Abbildung objektorientierter Programmkonzepte sind Nassi-Shneiderman-Diagramme ungeeignet. Hierfür wurde die Unified Modeling Language (UML) entwickelt.

Struktogramm - Sinnbilder nach DIN 66261

Die nachfolgenden Strukturblöcke können z. T. ineinander geschachtelt werden. Das aus den unterschiedlichen Strukturblöcken zusammengesetzte Struktogramm ist im Ganzen rechteckig, d. h. genauso breit wie sein breitester Strukturblock.

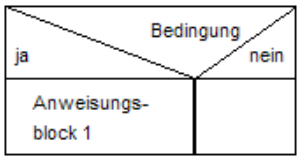
Linearer Ablauf (Sequenz)



Jede Anweisung wird in einen rechteckigen Strukturblock geschrieben.
Die Strukturblöcke werden nacheinander von oben nach unten durchlaufen.
Leere Strukturblöcke sind nur in Verzweigungen zulässig.

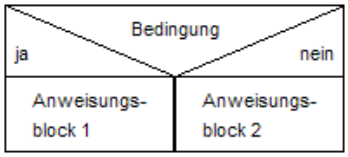
Verzweigung (Alternative)

Einfache Auswahl (bedingte Verarbeitung)(Selektion)



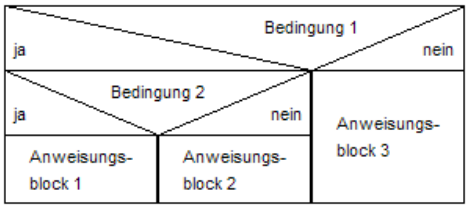
Nur wenn die Bedingung zutreffend (wahr) ist, wird der Anweisungs-block 1 durchlaufen. Ein Anweisungsblock kann aus einer oder mehreren Anweisungen bestehen. Trifft die Bedingung nicht zu (falsch), wird der Durchlauf ohne eine weitere Anweisung fortgeführt (Austritt unten).

Zweifache Auswahl (alternative Verarbeitung)



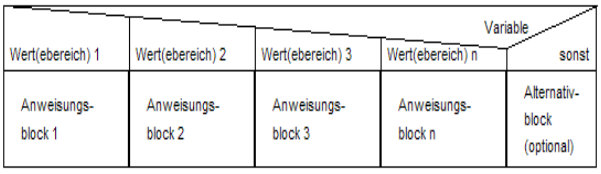
Wenn die Bedingung zutreffend (wahr) ist, wird der Anweisungs-block 1 durchlaufen. Trifft die Bedingung nicht zu (falsch), wird der Anweisungsblock 2 durchlaufen. Ein Anweisungsblock kann aus einer oder mehreren Anweisungen bestehen. Austritt unten nach Abarbeitung des jeweiligen Anweisungsblocks.

Mehrfachauswahl



Auch "verschachtelte" Auswahl genannt, da eine weitere Bedingung folgt. Die Verschachtelung ist ebenso im Nein-Fall (noch) möglich.

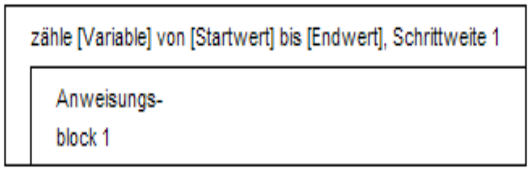
Fallauswahl



Besonders bei mehr als drei zu prüfenden Bedingungen geeignet. Der Wert von "Variable" kann bedingt auf Gleichheit wie auch auf Bereiche (größer/kleiner bei Zahlen) geprüft werden und der entsprechend zutreffende "Fall" mit dem zugehörigen Anweisungsblock wird durchlaufen. Eine Fallauswahl kann manchmal in eine Mehrfachauswahl umgewandelt werden – etwa wenn die eingesetzte Programmiersprache Fallauswahlen nicht kennt.

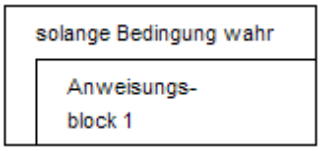
Wiederholung (Iteration)

Zählergesteuerte Schleife



Wiederholungsstruktur, bei der die Anzahl der Durchläufe festgelegt ist. Als Bedingung muss eine Zählvariable angegeben und mit einem Startwert initialisiert werden. Ebenso muss ein Endwert und die (Zähl-)Schrittweite angegeben werden. Nach jedem Durchlauf des Schleifenkörpers (Anweisungsblock 1) wird die Zählvariable um die Schrittweite inkrementiert (bzw. bei negativer Schrittweite dekrementiert) und mit dem Endwert verglichen. Ist der Endwert überschritten, wird die Schleife verlassen.

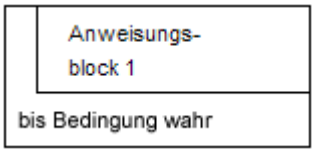
Abweisende (vorprüfende - kopfgesteuerte) Schleife



Wiederholungsstruktur mit vorausgehender Bedingungsprüfung. Der Schleifen-körper (Anweisungsblock 1) wird nur durchlaufen, wenn (und solange) die Bedingung zutreffend (wahr) ist.

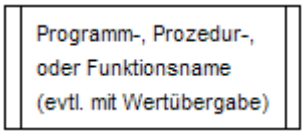
Diese Symbolik wird auch für die Zählschleife (Anzahl der Durchläufe bekannt) benutzt.

Nicht abweisende (nachprüfende - fußgesteuerte) Schleife



Wiederholungsstruktur mit nachfolgender Bedingungsprüfung für den Abbruch. Der Schleifenkörper (Anweisungsblock 1) wird mindestens einmal durchlaufen, auch wenn die Bedingung von Anfang an nicht zutreffend (falsch) war.

Aufruf



Symbolik zum Aufruf eines Unterprogramms bzw. einer Prozedur oder Funktion. Nach Durchlauf dieser wird genau zu der aufrufenden Stelle zurückgesprungen und der nächstfolgende Strukturblock durchlaufen.

Füllregeln

Allgemeingültigkeit: Struktogramme sollten keine programmiersprachenspezifische Befehlssyntax enthalten. Sie müssen so programmiersprachenunabhängig formuliert werden, dass die dargestellte Logik einfach zu verstehen und als Codiervorschrift in jede beliebige Programmiersprache umgesetzt werden kann.

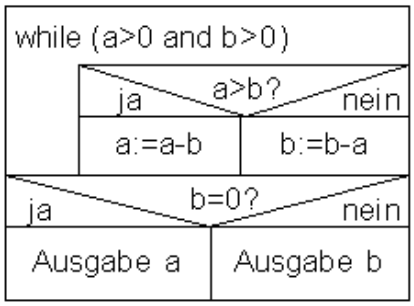
Deklaration: Die Deklaration von Variablen und Konstanten ist im ersten Anweisungsblock vorzunehmen.

Exklusivität: Jede Anweisung erhält einen eigenen Strukturblock. Selbst mehrere Anweisungen gleicher oder ähnlicher Art dürfen nicht in einem Strukturblock zusammengefasst werden.

Jede Anweisung muss aus min. einer Zuweisung bestehen (z. B. Zielvariable ← Zielvariable * AndereVariable). Eine Zuweisung wird durch einen nach links gerichteten Pfeil dargestellt. Ältere Struktogramme benutzen alternativ aus alten Pascal-Zeiten als Zuweisungszeichen den Doppelpunkt gefolgt vom Gleichheitszeichen (Zielvariable := Zielvariable * AndereVariable). Das Ziel einer Anweisung steht immer links vom Zuweisungszeichen. Rechts davon steht die Quelle. Über jedes Struktogramm gehört ein Name, um die Identifikation durch Ereignis- oder (Unter-)Programmaufrufe gewährleisten zu können.

Beispielstruktogramm: Das folgende Beispiel zeigt den Ablauf des euklidischen Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier Zahlen.

als Nassi-Shneiderman-Diagramm ...



... und in Pascal:

```
PROGRAM GGT(Input,Output);
VAR a,b: Integer;
BEGIN
ReadLn(a,b);
WHILE (a>0) AND (b>0) DO
IF a>b THEN
      a:=a-b;
ELSE
      b:=b-a;
IF b=0 THEN
WriteLn(a)
ELSE
WriteLn(b)
END.
```

2.10 (Weitere) Kontrollstrukturen

Kontrollstrukturen steuern den Ablauf der Programmverarbeitung, d.h., es wird damit die Reihenfolge der Ausführung der Anweisungen festgelegt.
Kontrollstrukturen treffen Entscheidungen, indem sie eine oder mehrere Bedingungen prüfen.

Bedingungen sind logische Ausdrücke, die wahr oder falsch sein können.
Eine (sogen.) einfache Bedingung ist ein Vergleich – das ist ein Ausdruck, der zwei Operanden mit einem Vergleichsoperator verknüpft. Das Vergleichsergebnis ist ein Wahrheitswert, der ‚True' oder ‚False' sein kann.

Einfache Bedingungen können mit logischen Operatoren (z.B. and, or, not) zu komplexen Bedingungen verknüpft werden.

In Python besitzen Objekte und Ausdrücke immer auch einen Wahrheitswert:
Numerische Objekte mit dem Wert null haben den Wahrheitswert „Falsch", Zahlen ungleich Null sind „Wahr". Alle Sequenzen (z.B. Zeichenketten oder Listen) tragen den Wahrheitswert „Falsch", wenn sie leer sind und „Wahr", wenn sie zumindest ein Element enthalten.

Beispiele:

Ausdruck	Wahrheitswert	Erklärung
127	True	Zahl ungleich null
0.0	False	Zahl gleich null
(3-9)*2	True	Ausdruck ergibt einen Wert ungleich null
8-8	False	Ausdruck ergibt null
"falsch"	True	Nichtleere Zeichenkette
""	False	Leere Zeichenkette
[1, 2, 3]	True	Nichtleere Liste
[]	False	Leere Liste
[[]]	True	Nichtleere Liste – sie enthält ein Element, nämlich eine leere Liste

2.10.1 Bedingte Anweisung (if-else - Struktur)

Eine bedingte Anweisung ermöglicht durch die *einmalige* Prüfung einer Bedingung eine Verzweigung im Programmablauf.

Einfachauswahl

Syntax:
```
if (Bedingung):
     Anweisungen
```

Wenn die Bedingung wahr ist (True), dann werden die Anweisungen des `if`-Zweiges ausgeführt.

Beispiel:
```
wert=23
if wert != 44:
      print("Falscher Wert!")
```

Zweifachauswahl

Syntax:
```
if (Bedingung):
     Anweisungen
else:
    Anweisungen
```

Wenn die Bedingung wahr ist, dann werden die Anweisungen des `if`-Zweiges ausgeführt, ansonsten werden die Anweisungen des else-Zweiges ausgeführt.

Beispiel:
```
wert=int(input("Eine Zahl eingeben: "))
if wert == 44:
      wert=wert*2
      print("richtiger Wert!")
else:
      print("falscher Wert!")
```

Geschachtelte Verzweigungen (Mehrfache Fallunterscheidungen)

Beispiel: Es soll abhängig vom Zahlenwert eine Meldung ausgegeben werden.

```
x = int(input("Bitte eine ganze Zahl eingeben: "))
if x < 0:
    x = 0
    print('Negativer Wert wurde null gesetzt.')
elif x == 0:
     print('Null')
elif x == 1:
    print('Einzeln')
else:
     print ('Mehrere')
```

2.10.2 Bedingte Wiederholung (while-Schleife)

Eine Bedingung wird geprüft und bei `'True'` werden die Anweisungen der Schleife ausgeführt, danach wird die Bedingung wieder geprüft und bei `'True'` werden die Anweisungen der Schleife ein weiters Mal ausgeführt, usw. usf., - und zwar solange, bis die Bedingungsprüfung zum ersten Mal `'False'` ergibt. Damit wird die Schleife abgebrochen, die Programmausführung setzt mit der nächsten Anweisung nach der `while`-Struktur fort.

Diese Art von Schleife kann somit null mal, ein mal oder mehrmals durchlaufen werden.

Syntax

```
while (Bedingung):
    Anweisungen
```

Beispiel: Zahlen von 1 bis 10 werden ausgegeben.

```
i = 1    # i ist die Zählvariable
  while (i <= 10):
      print(i)
      i = i + 1
```

[Quelle für das folgende Kapitel ist das Buch: Objektorientierte Programmierung mit Python 3, 4. Auflage, von Michael Weigend im mitp-Verlag 2010 sowie http://docs.python.org/py3k]

Programmierparadigmen

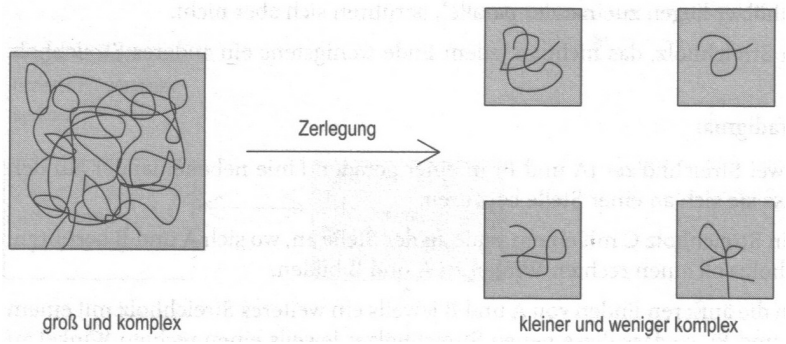
Ein Paradigma ist allgemein ein Denk- oder Handlungsmuster, an dem man sich z.B. bei der Formulierung einer Problemlösung orientiert.
Wenn man ein Programm als Algorithmus betrachtet, also als System von Befehlen, folgt man dem *imperativen* Programmierparadigma *(imperare:* lat. befehlen).
Ein imperatives Programm beschreibt, wie eine Aufgabe gelöst werden soll.
Das deklarative und das funktionale Programmierparadigma spielen bei der Software-Entwicklung heutzutage keine so große Rolle.

Auch die *objektorientierte* Programmierung (OOP) wird als eigenes Programmierparadigma beschrieben. Hier wird der Fokus auf die Beherrschung von Komplexität bei der Programmentwicklung gelegt.

2.10.3 Objektorientierte Programmierung

Strukturelle Zerlegung

Die ersten Computerprogramme wurden zur Lösung relativ kleiner, gut abgrenzbarer Probleme geschrieben und waren als Ganzes noch überschaubar. Die Situation wird ganz anders, wenn man umfangreiche Softwareanwendungen erstellen möchte, etwa ein umfangreiches Textverarbeitungsprogramm oder ein Verwaltungsprogramm für eine Bibliothek.
Solche großen Systeme lassen sich nur beherrschen, wenn man sie zunächst in kleinere, überschaubare Teile aufbricht (siehe Abbildung unten).



Die Zerlegung einer komplexen Struktur in kleine(re) Teilstrukturen ist vorteilhaft, weil:
- sich die kleineren Teile der Gesamtaufgabe einfacher programmieren lassen
- die Wahrscheinlichkeit, dass die kleineren Teilprogramme Fehler enthalten, geringer ist
- mehrere Personen diese kleineren Teile gleichzeitig und unabhängig voneinander erstellen können und damit Zeit gespart wird
- man einen Baustein, der schon früher einmal programmiert wurde, später wiederverwenden kann und damit Kosten gespart werden

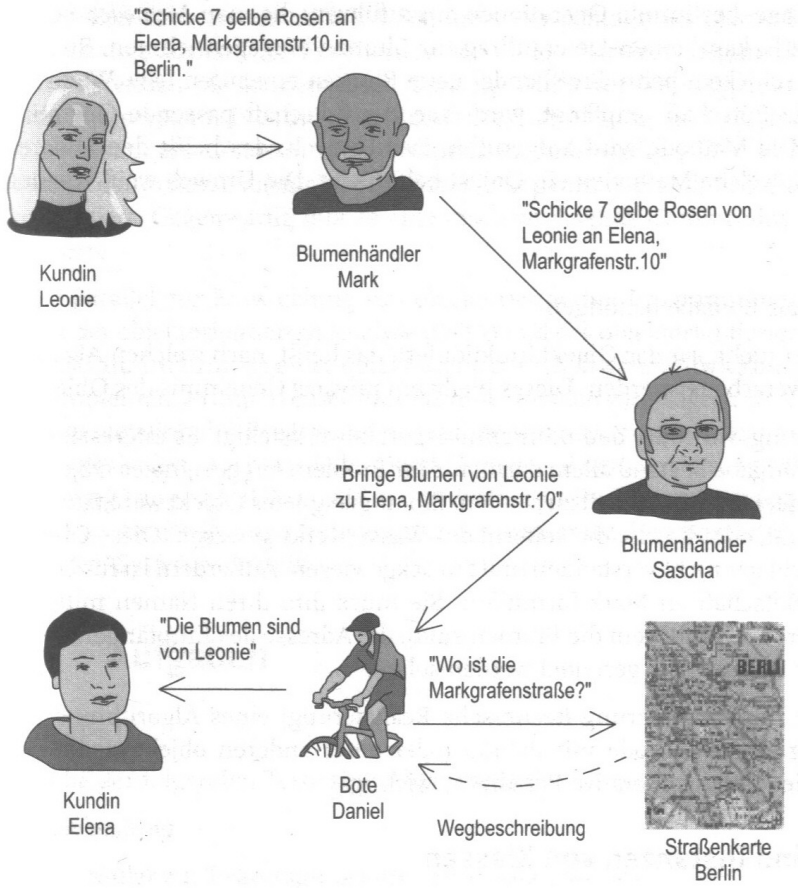
Das objektorientierte Programmierparadigma bietet hierzu ein Verfahren an, um große, vielschichtige Systeme in kleinere, weniger komplexe Teile zerlegen zu können.

Die Welt als System von Objekten

In der objektorientierten Sichtweise stellt man sich die Welt als System von Objekten vor, die untereinander Botschaften austauschen. Diese Systematik soll das u.a. Alltagsbeispiel veranschaulichen (entnommen aus dem Buch von M. Weigend: OOP mit Python 3):

Leonie in Bonn möchte ihrer Freundin Elena in Berlin einen Blumenstrauß schicken. Sie geht deshalb zu Mark, einem Blumenhändler, und erteilt ihm einen entsprechenden Auftrag. Betrachten wir Mark als Objekt. In der Sprache der objektorientierten Programmierung sagt man: Leonie sendet an das Objekt Mark eine Botschaft, nämlich: „Sende sieben gelbe Rosen

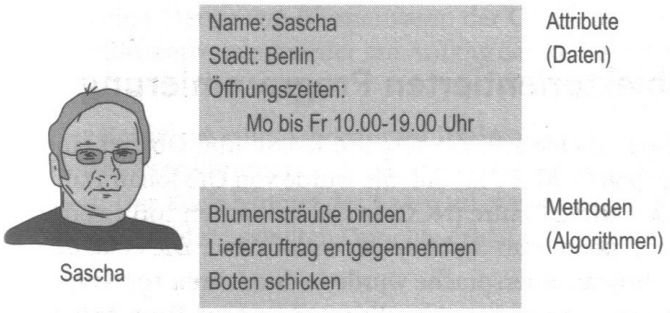
an Elena, Markgrafenstraße 10 in Berlin". Mit der Bezahlung des Preises für diese Dienstleistung hat Leonie einen Prozess in Gang gesetzt, der im Weiteren selbständig abläuft.
Es liegt nun in Marks Verantwortung, den Auftrag zu bearbeiten. Mark versteht die Botschaft und weiß, was zu tun ist. Das heißt, er kennt einen Algorithmus für das Verschicken von Blumen. Der erste Schritt ist, einen Blumenhändler in Berlin zu finden, der die Rosen an Elena liefern kann. In seinem Adressverzeichnis findet er den Floristen Sascha. Ihm sendet er eine leicht veränderte Botschaft, die nun zusätzlich noch den Absender enthält. Damit ist Mark fertig und hat die Verantwortung für den Prozess weitergegeben. Auch Sascha hat einen zur Botschaft passenden Algorithmus parat. Er stellt den gewünschten Blumenstrauß zusammen und beauftragt seinen Boten Daniel, die Rosen auszuliefern. Daniel muss nun den Weg zur Zieladresse finden und befragt seine Straßenkarte. Sie antwortet ihm mit einer Wegbeschreibung. Nachdem Daniel den Weg zu Elenas Wohnung gefunden hat, überreicht er die Blumen und teilt ihr in einer Botschaft mit, von wem sie stammen. Damit ist der gesamte Vorgang, den Leonie angestoßen hat und an dem mehrere Objekte beteiligt waren, beendet.



Objektorientiertes Modell eines Blumenversandsystems

Objekte besitzen Attribute und beherrschen Methoden

Jedes Objekt besitzt Eigenschaften oder *Attribute.* Ein Attribut eines Blumenhändlers ist z.B. die Stadt, in der sein Geschäft ist. Dieses Attribut ist für die Umwelt wichtig, um zu wissen, wo überall es einen Blumenhandel gibt. Weitere typische Attribute von Blumenhändlern sind Name, Telefonnummer, Warenbestand und Öffnungszeiten.



Weiters sind Objekte in der Lage, bestimmte Operationen auszuführen, die man *Methoden* nennt. Ein Blumenhändler z.B. kann einen Lieferauftrag für Blumen entgegennehmen, Sträuße binden, einen Boten schicken, beim Großhandel neue Blumen einkaufen usw. Wenn ein Objekt eine geeignete Botschaft empfängt, wird eine zur Botschaft passende Operation gestartet. Man sagt: Die Methode wird aufgerufen.

Der Umwelt, das heißt den anderen Objekten, ist bekannt, welche Methoden ein Objekt beherrscht und die Umwelt weiß von den Methoden

- was sie bewirken und
- welche Daten sie als Eingabe benötigen.

Die Umwelt weiß aber nicht, *wie* die Methoden eines Objekts funktionieren, das heißt, nach welchen Algorithmen die Botschaften verarbeitet werden. Dieses Wissen bleibt ein ‚Geheimnis' des Objektes.

Leonie hat keine Ahnung, wie Mark den Blumentransport bewerkstelligt - es interessiert sie auch gar nicht. Ihr Handeln (ihre Aufgabe) bestand darin, für ihr Problem ein geeignetes Objekt zu finden und ihm eine geeignete Botschaft zu senden. Ein ungeeignetes Objekt wäre zum Beispiel Hans, der Zahnarzt, oder Elisabeth, die Leiterin des Wasserwerks gewesen. Diese Objekte hätten Leonis Nachricht gar nicht verstanden, bzw. zurückgewiesen.
Wichtig für das Objekt Leonie ist es jedoch, zu wissen, wie eine Botschaft an Mark richtig formuliert wird. Sie muss ihm ihren Namen mitteilen (damit der Empfänger weiß, von wem die Blumen sind), die Adresse des Empfängers sowie Anzahl und Sorte der Blumen, die zugestellt werden sollen.

Eine Methode eines Objekts ist die Implementierung eines Algorithmus → Programmstruktur ‚Funktion' zur technische Realisierung einer (Teil)Aufgabe.
Bei der Programmierung einer Methode in einer objektorientierten Sprache wird also wieder das imperative Paradigma wichtig.

Objekte sind Instanzen von Klassen

Die Objekte des o.a. Beispiels kann man in Gruppen einteilen. Sascha und Mark sind Blumenhändler; sie beherrschen beide dieselben Methoden und besitzen dieselben Attribute (z.B. die Stadt), allerdings mit unterschiedlichen Werten. Man sagt: Sascha und Mark sind *Instanzen* der Klasse „Blumenhändler".
In der objektorientierten Programmierung ist eine Klasse die Definition eines bestimmten Typs von Objekten. Sie ist so etwas wie ein Bauplan, in dem die Methoden und Attribute beschrieben werden. Nach diesem Schema können Objekte (Instanzen) einer Klasse erzeugt werden. Ein Objekt ist eine Konkretisierung, eine Inkarnation einer Klasse. Alle Instanzen einer Klasse sind von der Struktur her gleich. Sie unterscheiden sich allein in der Belegung ihrer Attribute mit voneinander verschiedenen Werten. Die Objekte Sascha und Mark besitzen dasselbe Attribut „Stadt", aber bei Sascha trägt es den Wert ‚Berlin' und bei Mark ‚Bonn'.

So wie der Bauplan eines Hauses eine Vorlage ist, die zur Realisierung konkreter Häuser dient (im weitesten Sinne die Beschreibung einer Schablone, um viele Kopien davon anfertigen zu können), so ist in der objektorientierten Programmierwelt eine **Klasse** der 'Bauplan', nach dem ein konkretes **Objekt** (entspricht einem fertig 'gebauten Haus') erzeugt werden kann.
Liegt ein 'Bauplan' (= Klasse) vor, können damit x-beliebig viele 'Häuser' (= Objekte) erstellt werden.

Eine Klasse beschreibt einen Datentyp als eine (vom Programmierer festgelegte) Struktur, die angibt, wie Objekte dieser Klasse aussehen sollen. Ein Objekt hingegen ist die konkrete Realisierung einer Klassenbeschreibung, ein tatsächlich existierendes »Ding« im Speicher.
Ein weiterer, oft benutzter synonymer Begriff für ein Objekt ist der der *Instanz* (einer Klasse).

Ein Objekt wird durch die Merkmale beschrieben, die in der Klassendefinition festgelegt sind, von der es *abgeleitet* wurde, m.a.W., von der das Objekt *instanziert* ist. Diese Merkmale werden als **Eigenschaften** bezeichnet, die als Variablen abgebildet werden, denen der jeweilige objektindividuelle Merkmalswert zugewiesen wird.
Beispielsweise kann ein Objekt vom Typ Person durch die Eigenschaften Name, Alter, Gewicht, Haarfarbe und Geschlecht beschrieben werden.

Objekte können auch Operationen ausführen, um beispielsweise die Eigenschaften zu manipulieren oder um das Verhalten einer Person zu beschreiben.
Im objektorientierten Sprachgebrauch wird eine von einem Objekt ausgeführte Operation auch als Methode oder Elementfunktion bezeichnet. Methoden werden in einer Klassenstruktur als Prozedur oder Funktion definiert.

⇨ *Eigenschaften* beschreiben den *Zustand eines Objekts,*
Methoden charakterisieren die *Verhaltensweisen eines Objekts.*

2.10.3.1 Schlüsselkonzepte der objektorientierten Programmierung

Klassen
Datenkapselung
Vererbung
Polymorphie

Klassen

Eine Klasse realisiert einen Datentyp und ist das fundamentalste Sprachelement der objektorientierten Programmierung. Eine Klasse beschreibt, durch welche Eigenschaften sich Objekte dieses Typs voneinander unterscheiden und welche Fähigkeiten diese Objekte haben. Ein Objekt ist in der objektorientierten Begriffswelt nichts anderes als eine Variable vom Typ einer bestimmten Klasse.

Datenkapselung

Die Eigenschaften eines Objekts sind im Allgemeinen Variablen, die einen Wert enthalten.
Der Anwender kann auf diese Variablen nicht direkt zugreifen, um z.B. einen Wert zuzuweisen, denn in der Klassenbeschreibung sind sie als „private" deklariert.
Stattdessen wird die Wertzuweisung durch die Umleitung über eine Methode ausgewertet, in der die Zulässigkeit der Zuweisung überprüft wird. Eine Klasse schützt (kapselt) so ihre Daten vor dem direkten, d.h. unkontrollierten Zugriff. Das ist ein ganz wesentlicher Aspekt der Objektorientierung.

Vererbung

Objekte, die von Natur aus verschiedenartig sind, können dennoch über gemeinsame Merkmale verfügen. Deshalb liegt es nahe, eine separate Klasse zu definieren, die nur die Beschreibung der gemeinsamen Merkmale enthält.
Das Resultat dieser Überlegung ist eine Klassenhierarchie, in der es eine »Mutterklasse« gibt, von der andere Klassen abstammen, also deren Abkömmlinge sind.
Eine Klasse, die ihre Merkmale untergeordneten Klassen zur Verfügung stellt, wird als Basisklasse bezeichnet; eine untergeordnete Klasse, welche die Merkmale einer Basisklasse übernimmt, als abgeleitete Klasse oder Subklasse.
Abgeleitete Klassen übernehmen die Eigenschaften und Methoden der übergeordneten Basisklasse, sie beerben die Basisklasse und ergänzen die geerbten Merkmale durch typspezifische Merkmale.

Polymorphie

Die Implementierungsvererbung stellt die Wiederbenutzung von Programmcode einer Basisklasse durch eine abgeleitete Klasse sicher. Verwandte Objekte können aber auch eine Gemeinsamkeit haben, die sich nur in der Codeimplementierung unterscheidet.
Das bedeutet, dass das gleiche Verhalten in der Methode jeder abgeleiteten Klasse unterschiedlich beschrieben werden muss.

Dazu wird in der Basisklasse das gemeinsame Merkmal/Verhalten festlegt, das aber implementierungslos bleibt. Die Subklassen beerben die Basisklasse, müssen aber ihrerseits die erforderliche typspezifische Programmlogik bereitstellen.
Polymorphie (auf Deutsch: Vielgestaltigkeit) bedeutet, dass erst zur Laufzeit einer Anwendung bestimmt wird, welche Implementierung aufgerufen werden muss, da dies beim Kompilieren noch nicht bekannt ist.

2.10.4 Objekte in Python

Python ist in einem sehr umfassenden Sinn eine objektorientierte Programmiersprache; Daten, Funktionen und andere Sprachelemente werden durch Objekte repräsentiert.
Wenn man in der Mathematik von der Zahl 123 spricht, denkt man zunächst an einen numerischen Wert.
Der *Wert* eines Objekts wird durch *Literale* repräsentiert.

Literale sind Zeichenfolgen, die nach bestimmten Regeln aufgebaut sind. Der Wert der natürlichen Zahl 123 kann durch die Zeichenfolgen 123 (Dezimalzahl) oder 0173 (Oktalzahl) repräsentiert werden - zwei verschiedene Literale für den gleichen numerischen Wert.

Der Wert ist nur *ein* Aspekt eines Objektes. Für das Objekt mit dem Wert 123 ist auch von Bedeutung, dass es sich bei diesem Wert um eine ganze Zahl handelt – somit gehört das Objekt zu einem bestimmten *Typ,* nämlich dem Datentyp »ganze Zahl« (engl. *integer).*
Würde die Ziffernfolge in Hochkommata oder Anführungsstrichen stehen, handelt es sich um eine Zeichenkette (engl. *string).* Die Zeichenkette '123' ist etwas anderes als die ganze Zahl 123. Der Typ eines Objektes ist für die Verarbeitung durch ein Programm wichtig. Mit Zahlen kann man arithmetische Operationen durchführen, mit Zeichenketten nicht.

Der Typ eines Objektes kann mit der Standardfunktion type() ermittelt werden, z.B.:

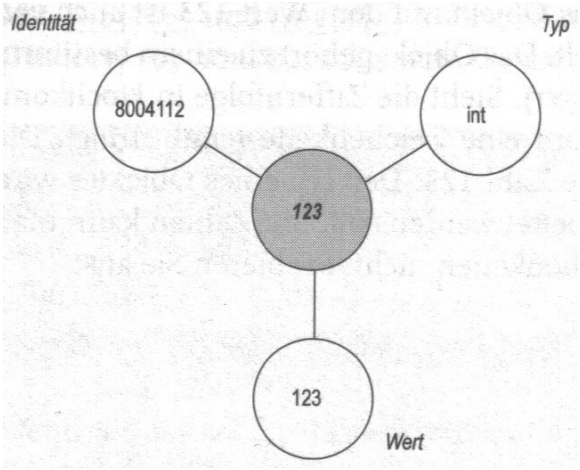
```
>>> type (123)
<class 'int'>
>>> type('123')
<class  'str'>
```

Ein drittes Merkmal aller Objekte ist, dass sie eine *Identität* besitzen. Bei Python wird die Identität eines Objekts durch eine (einmalige) ganze Zahl repräsentiert, die mit der Standardfunktion id() abgefragt werden kann, z.B.:

```
>>> id(123)
8004856
>>> id('123')
19427280
```

Die Identität dient der Identifizierung eines Objektes; sie ist jedoch mehr als nur eine verschlüsselte Form des Wertes. Es kann sein, dass zwei Objekte den gleichen Wert, aber unterschiedliche Identität besitzen. Diese Objekte sind dann *gleich,* aber nicht *identisch.*

Alle Objekte in Python besitzen einen Wert, einen Typ und eine Identität → siehe Bild unten.



Zeichenketten (string-Typ) und ganze Zahlen (int-Typ) sind Beispiele für Standard-Typen von Python, die in der Programmiersprache vorgegeben sind *(built-in types).*
Zusätzlich ist es auch möglich, eigene Datentypen zu definieren.

Namen und deren Bezeichner

Objekte können anonym sein oder einen Namen zugewiesen bekommen, über den ein Objekt angesprochen werden kann; man spricht auch vom *Namen* einer Variablen.

Bezeichner (*identifiers*) sind Zeichenketten, die für Namen verwendet werden dürfen. Ein Bezeichner beginnt mit einem Buchstaben oder Unterstrich, danach folgen beliebig viele Buchstaben, Unterstriche oder Dezimalziffern.

Schlüsselwörter (*keywords*) sind gültige Wörter, die für Python reserviert sind. Sie dürfen nicht für Namen, d.h. als Bezeichner für Variablen und/oder Funktionen verwendet werden, da ihnen bereits eine bestimmte Bedeutung zugeordnet ist:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Anweisungen

Anweisungen (*statements*) sind die Grundbausteine eines Programms ⇨ zum Überblick siehe Kapitel 2.8 - Einfache Anweisungen.

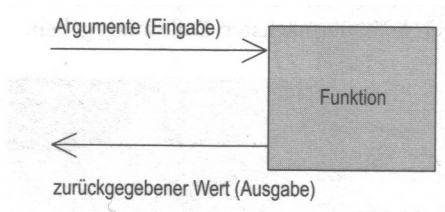
Ausdruck-Anweisungen sind mathematische Terme oder logische Beziehungen sowie Funktions- und Methodenaufrufe.

Hinweis: In Python-Skripten dürfen Ausdrücke und Funktionsaufrufe alleine stehen, sie müssen nicht - wie in anderen Programmiersprachen – Teil einer Zuweisungsoperation sein. Bei der Interpretation des Skripts wird in so einem Fall das Ergebnis der Auswertung nicht weiter verwendet.

Funktionsaufruf

In Python sind Funktionen aufrufbare Objekte *(callable objects),* die eine bestimmte *Teilaufgabe lösen* können. Wenn eine Funktion aufgerufen wird, übernimmt sie gewisse Werte als Eingabe, verarbeitet diese und liefert einen oder mehrere Wert/e als Ausgabe zurück (siehe Bild unten).

Die Werte, die man einer Funktion beim Aufruf übergibt, nennt man *Argumente* oder *aktuelle Parameter.*



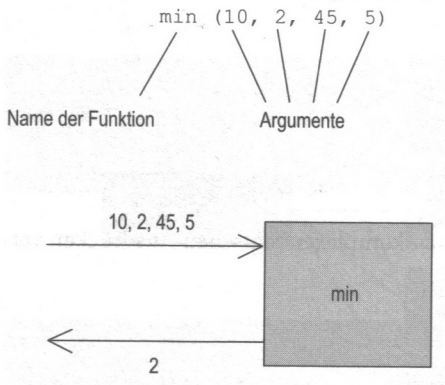
Im interaktiven Modus kann man Funktionen aufrufen und erhält dann den zurückgegebenen Wert in der nächsten Zeile angezeigt, z. B.:

>>> len(„Python“) → 6

‚len' ist der Name der Funktion und die Zeichenkette "Python" das Argument. Zurückgegeben wird die Länge der Zeichenkette, d.h. die Anzahl der Zeichen.

>>> min(10, 2, 45, 5) → 2

Die Funktion min() akzeptiert eine unterschiedliche Anzahl von Argumenten und gibt den kleinsten Wert der beim Aufruf übergebenen Argumente zurück.



Funktionen, die keinen Wert zurückgeben, nennt man (auch) *Prozeduren*;

z.B: >>> help()

Die Argumente eines Funktionsaufrufs müssen nicht unbedingt explizite Werte sein, es können auch Ausdrücke als Argumente verwendet werden. Sie werden dann vom Python-Interpreter zunächst ausgewertet und der resultierende Wert der Funktion übergeben.

In den Ausdrücken, die bei einem Funktionsaufruf als Argument eingesetzt werden, können auch wieder Funktionsaufrufe vorkommen. Derart verschachtelte Funktionsaufrufe werden von innen nach außen ausgewertet.

z.B.: >>> min(len(„süß"), len(„sauer")) → 3

Methodenaufrufe - Botschaften an Objekte

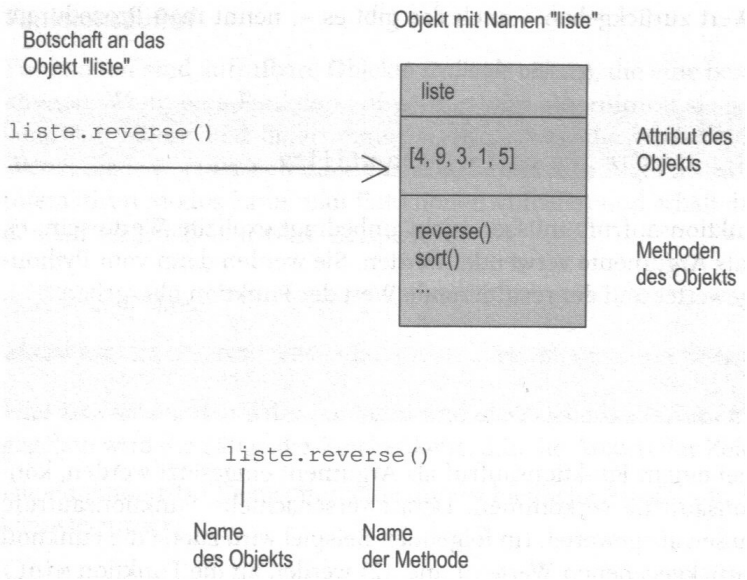
Objektorientierte Programme enthalten Botschaften an Objekte, in denen diese „aufgefordert" werden, „etwas zu tun". Eine solche Botschaft ist somit (auch) eine Anweisung; sie ist der Aufruf einer Methode des Objektes, an das die Botschaft gerichtet ist.

Methoden sind Funktionen, die an ein Objekt gekoppelt sind.

Ein Methodenaufruf beginnt mit dem Namen des Objektes, dahinter kommt ein Punkt, dann der Name der Methode und schließlich in Klammern die Argumente.

z.B.: >>> liste = [4, 9, 3, 1, 5]
>>> liste.reverse()
>>> liste → [5, 1, 3, 9, 4]

Mit der ersten Anweisung im obigen Beispiel wird ein Objekt mit dem Namen liste erzeugt. Es enthält als Wert eine Folge von fünf Zahlen. An dieses Objekt wird in der zweiten Anweisung eine Botschaft geschickt. Die Methode reverse() wird aufgerufen. Sie bewirkt, dass die Reihenfolge der Zahlen in der Liste umgekehrt wird; es wird jedoch kein Wert zurückgegeben. Das Objekt liste hat sich einfach nur verändert. Um den geänderten Wert von liste sichtbar zu machen, werden die Attribute (Eigenschaftswerte) des Objekts angezeigt.



2.10.5 (Weitere) Datentypen

Im Kapitel 2.5 sind die Datentypen von Python im Überblick schon dargestellt.
Die Datentypen für Zahlen (numerische Objekte) sind

-) int (ganze Zahlen) und -) float (Gleitkommazahlen) sowie

weiters der Datentyp für Wahrheitswerte -) bool .

Diese hier aufgezählten Typen nennt man einfache Datentypen, da solche Objekte nur einen (elementaren) Wert tragen.

Daneben gibt es auch Datenobjekte, die aus mehreren Einzelobjekten zusammengesetzt sind – solche „Dinge" nennt man *Kollektionen* oder Container-Objekte; sie bestehen aus Sequenzen, Mengen und Abbildungen.

Kollektionen bezeichnet man auch als ‚iterierbare' [‚Iteration' – lat. Wiederholung] Objekte, d.h., es können alle Elemente der „(An)Sammlung" nacheinander durchlaufen werden.

Objekte, die bei ihrer Erschaffung (→ „Instanzierung", „Ableitung") einen festen Wert erhalten, den sie bis zu ihrer Auflösung beibehalten, bezeichnet man als unveränderbar (immutable) – dazu gehören alle einfachen numerischen Typen, Zeichenketten und Tupel. Eine Liste ist ein änderbarer (mutable) Typ – d.h., diese Elemente sind modifizierbar (veränderbar).

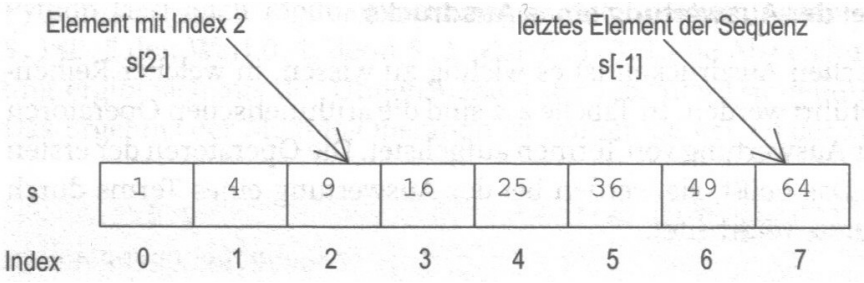
2.10.5.1 Sequenzen

Eine Sequenz ist eine geordnete Kollektion von Objekten, ihre Elemente (items) befinden sich in einer bestimmten Reihenfolge und sind durchnummeriert:
die Anzahl der Elemente ist die Länge einer Sequenz und wird mit n bezeichnet
die einzelnen Elemente werden durchnummeriert, dieser Zählwert heißt Index und beginnt bei Null – m.a.W.: die Indizes einer Sequenz laufen von 0 bis (n-1).
Über den Index kann man auf ein einzelnes Element einer Sequenz zugreifen; das i-te Element einer Sequenz s wird mit s[i] angesprochen, bzw. selektiert.
Das letzte Element einer Sequenz hat zusätzlich den Namen s[-1], das vorletzte s[-2] usw. (siehe Bild unten).

Zu den Sequenzen gehören Zeichenketten (Typ str), Tupel (Typ tuple) und Listen (Typ list). Tupel können beliebige Elementtypen enthalten, während Zeichenketten nur aus Schriftzeichen bestehen.

Sequenzen kennt man auch im Alltag:

- Ein Haus kann man als Sequenz von Stockwerken betrachten.
- Im Periodensystem der Elemente (PSE) sind chemische Elemente nach ihrer Ordnungszahl aufgeführt. Das erste Element im PSE ist Wasserstoff mit der Ordnungszahl I. Weiß man die Ordnungszahl, kann man im PSE eine Beschreibung des zugehörigen Elements finden.



2.10.5.2 Wiederholungsstruktur (Iteration) für Kollektionen – for-Schleife

Bei einer bedingten Schleife (while-Schleife) wird vor jedem Schleifendurchlauf eine Bedingung geprüft und jedes Mal neuerlich entschieden, ob die Anweisungsfolge innerhalb der Schleifenstruktur ein weiteres Mal ausgeführt wird oder ob die Schleife abgebrochen wird.

Für verschiedene Aufgaben muss man eine bestimmte Anweisungsfolge auf alle Elemente einer Kollektion anwenden; man spricht dann von einer ‚Iteration über eine Kollektion'.

Alltagsbeispiele dafür sind:

Iteration	Anweisung (wiederholt ausgeführt)	Kollektion
Zähle laut von eins bis zehn	Sage aktuelle Zahl	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Rufe drei Mal „Hallo!"	Ruf „Hallo!"	[1, 2, 3]
Gib jedem Gast der Party einen Begrüßungstrunk	Überreiche Glas mit Sekt	Alle Gäste des Festes
Gib jedem Schüler eine leistungsgerechte Note	Prüfe einen Schüler	Alle Schüler einer Klasse

Die Programmstruktur für diesen Zweck ist die for-Anweisung (bzw. for-Schleife); sie folgt folgender Syntax:

```
for element in kollektion:
    anweisungsblock
```

element ist dabei ein Name, der für das jeweils aktuelle Elementobjekt der Iterationskollektion steht, für das der Anweisungsblock ausgeführt wird.

Beispiele:

```
>>> for i in [1, 2, 3, 4, 5]:    ⇨  1  4  9  16  25
        print(i*i, end=" ")

>>> for farbe in ("rot", "grün", "blau"):    ⇨  rot  grün  blau
        print(farbe, end=" ")

>>> for z in "Schule":          ⇨   S   c   h   u   l   e
        print(z, end="   ")
```

2.10.5.3 Zählschleifen

Bei vielen Aufgabenstellungen muss der Index der Elemente einer Kollektion gezählt werden, d.h. es muss ganzzahlig von 0 bis n-1 gezählt werden. In Python gibt es dafür die range()-Funktion, die eine Folge ganzer Zahlen von 0 bis n-1 liefert.

Die Syntax für eine Zählschleife ist somit:

```
for i in range(n):        # n kann ein Wert, eine Variable oder ein Ausdruck sein
    anweisungsblock
```

Hier verhält sich die Variable i wie eine Zählvariable. Sie erhält zuerst den Wert 0 und nach jedem Schleifendurchlauf wird sie um eins erhöht (m.a.W.: i wird der nächste Rückgabewert der range()-Funktion zugewiesen), bis die Zahl n-1 erreicht ist.

[Anmerkung: Die Wirkung dieser Konstruktion ist gleich wie bei der bedingten Schleife, außer dass man die Zählvariable nicht selbst weiterzählen muss.]

```
>>> for i in range(6):        ⇨  0  1  4  9  16  25
        print(i*i, end=" ")


n = 2
>>> for i in range(n*4):        ⇨  2  3  4  5  6  7  8  9
        print(i+2, end=" ")
```

Der Funktion *range()* kann man auch Argumente für drei Parameter übergeben, mit denen der Zählbereich – start, stop - und die Schrittweite der Zählung – step - festgelegt werden können. Es wird dann ganzzahlig von *start* bis *stop-1* mit dem Abstand *step* gezählt.

Die folgende Zählschleife gibt alle geraden Zahlen von -6 bis 5 aus:

```
>>> for i in range(-6, 6, 2):        ⇨  -6  -4  -2  0  2  4
        print(i, end=" ")
```

Zum Hinunterzählen verwendet man eine negative Schrittweite:

```
>>> for i in range(10, 0, -1):       ⇨  10  9  8  7  6  5  4  3  2  1
        print(i, end=" ")
```

Die folgende Iteration gibt alle Zeichen des einfachen ASCII-Codes aus:

```
>>> for i in range(32, 128):
        print(chr(i), end=" ")      ⇨
```

```
 ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U
V W X Y Z [ \ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

2.10.5.4 Zeichenketten (Strings) → für Detailinformationen siehe die Datei: ‚Strings_Python.doc'

Zeichenketten sind Folgen von Zeichen aus einem Alphabet; sie werden auch als Strings bezeichnet. Beispiele für Zeichenketten-Literale sind „Python", ‚sehen und hören', „3456" .

Bei Python sind die Zeichen Unicode-Zeichen, das heißt, insbesondere Sonderzeichen des deutschen Alphabets wie ä, ö, ü, ß sind erlaubt.

[Jedem Unicode-Zeichen ist eine Nummer als vier- oder achtstellige Hexadezimalzahl (16 Bit bzw. 32 Bit) und ein Name eindeutig zugeordnet. Um für ein spezielles Zeichen die Unicode-Nummer zu finden, kann man im WWW in den offiziellen Code-Charts des Unicode-Konsortiums nachsehen - http://www.unicode.org/charts/ ; bzw.: http://www.unicode.org/ucd/ .]

Zeichenketten sind nicht änderbar. Das heißt, man kann bei einem String-Objekt keine Buchstaben einfügen, entfernen oder ändern.

Es gibt kurze und lange Zeichenketten.

Kurze Zeichenketten sind durch Hochkommata ' oder Anführungszeichen " eingerahmt. Sie müssen sich in *einer* (logischen) Programmzeile befinden. Eine kurze Zeichenkette enthält beliebig vielen Zeichen des ASCII-Zeichensatzes (Nummer 32 bis 127) mit Ausnahme des Backslash (\), des Newline-Zeichens und der Anführungszeichen oder Hochkommata, die zur Bildung des Strings verwendet wurden.
Das heißt: Wenn ein String in Hochkommata eingeschlossen ist, darf in der Zeichenkette selbst kein Hochkomma, wohl aber ein Anführungszeichen vorkommen; und umgekehrt ⇨
⇨ siehe auch: Escape-Sequenzen .

Lange Zeichenketten können über mehrere Zeilen gehen, sie werden durch drei hintereinander gestellte Anführungszeichen """ oder Hochkommata ''' eingeschlossen.

Python-intern werden lange Zeichenketten durch kurze Zeichenketten repräsentiert (Escape-Sequenz \n).

Escape-Sequenzen ermöglichen die Darstellung von Sonderzeichen und von Buchstaben, die man nicht auf der Tastatur findet. Escape-Sequenzen beginnen immer mit einem Backslash (\). Häufig braucht man die Unicode-Nummern (siehe oben) für die Codierung spezieller Zeichen.

Escape-Sequenz	Bedeutung	Beispiel
\\	um einen Backslash anzeigen zu können	"backslash\\" → backslash\
\'	um ein Hochkomma anzeigen zu können	"\'Python\' " → 'Python'
\"	um ein Anführungszeichen anzeigen zu können	"\"Zitat\"" → "Zitat"
\n	Zeilenumbruch (line feed) erzeugen	"eins\nzwei" → eins zwei
\t	Horizontaler Tabulator	"eins\tzwei" → eins zwei
\uxxxx	Zeichen, dessen 16-Bit-Unicode-Nummer als vierstellige Hexadezimalzahl angegeben wird	"\u0041" → A "\u00b5" → µ

2.10.5.5 Tupel

In einem Tupel sind mehrere Objekte eventuell unterschiedlicher Typen zu einem komplexen Objekt zusammengefasst.

Typische Tupel aus dem Alltag sind:

- Name, Geburtsdatum und Geschlecht einer Person
- Beschreibung eines Punktes in einem dreidimensionalen Koordinatensystem als Tripel aus x-, y- und z-Komponente
- Adresse als Tupel mit fünf Elementen (Name, Straße, Hausnummer, Postleitzahl, Ort)
- Beschreibung einer Farbe durch Angabe der Rot-, Grün- und Blaukomponente

Von der Idee her wird bei einem Tupel die Modellierung der Struktur *eines* komplexen Einzelobjektes betont, während man bei einer Liste (siehe unten) eher an die Aufzählung *vieler* Einzelobjekte denkt.

Formal besteht ein Tupel bei Python aus einer Folge von Objekten, die auch unterschiedliche Datentypen sein dürfen, und durch Komma getrennt angegeben werden; sie können (aber müssen nicht) in runden Klammern stehen.

Beispiele sind: 1, 2, 3 oder (1, 2, 3)
("Ampel", 3, ('rot', 'gelb', 'grün'))
(6,) ()

Das Tupel () ist ein leeres Tupel ohne ein einziges Element. Ein Tupel mit einem einzigen Element muss ein Komma enthalten, um von einem geklammerten Ausdruck unterschieden werden zu können.
Wenn man ein Tupel erzeugt, kann man die Klammern auch weglassen:
>>> t = 1, 2, 3 → (1, 2, 3)

Die Elemente eines Tupels können auch Variablen sein.

Tupel sind unveränderbar; d.h., dass die Elementwerte eines schon erzeugten Tupels nicht mehr verändert werden können (sie können nur gelesen werden).

2.10.5.6 Definition von Funktionen

Eine Funktion muss einen Namen haben, über den sie in anderen Teilen des Programms aufgerufen werden kann.
Eine Funktion muss mindestens eine Anweisung beinhalten (oder die Anweisung *pass*, die keine Aktion durchführt).
Eine Funktion muss eine Schnittstelle haben, über die Informationen vom aufrufenden Programmteil in den Kontext der Funktion übertragen werden können. Eine Schnittstelle kann aus beliebig vielen oder auch keinen Parametern bestehen.
Funktionsintern wird jedem dieser Parameter ein Name gegeben. Sie lassen sich dann wie Referenzen im Funktionskörper verwenden.

Eine Funktion muss einen Wert zurückgeben; jede Funktion gibt automatisch *None* zurück, wenn der Rückgabewert nicht ausdrücklich angegeben wird.

Vorteile von Funktionen:
Übersichtlichkeit, separate Lösung von Teilproblemen, Platzeinsparung, Wiederverwendbarkeit

Der Funktionskörper muss eingerückt sein.

Funktion ohne Parameter

Syntax:

```
def <funktionsname> ():
    #Anweisungen
```

Bsp:
```
def willkommen():
    print("hello_world")
```

Aufruf: mit dem Funktionsnamen

Bsp:
```
willkommen()          #Klammern müssen angegeben werden
```

Funktion mit Parameter

Über *Parameter* werden beim Aufruf einer Funktion Daten an sie übergeben, die von der Funktion für die durchzuführenden Operationen benötigt werden.

Parameter (formale): werden bei der Deklaration einer Funktion nach dem Funktionsnamen in runden Klammern angegeben. Parameter sind Variablen mit einer auf die aktuelle Funktion beschränkten Sichtbarkeit – also im Prinzip lokale Variable.
Parameter erhalten einen Namen, der Datentyp wird beim Aufruf der Funktion durch die Zuweisung der Argumente festgelegt.

Argumente (tatsächliche Parameterwerte): sind die konkreten Werte, die der Funktion beim Funktionsaufruf übergeben werden; d.h, den (lokalen) Variablen, die als Parameter in der Funktionsdefinition angegeben sind, zugewiesen werden.

- Die Reihenfolge der Argumente muss mit der Reihenfolge der Parameter übereinstimmen(➔ positional arguments)
- Als Argument kann jeder Datentyp übergeben werden.

Beispiel:

```
def fak(zahl):
    ergebnis = 1
    for i in range(2, zahl+1:
        ergebnis *= i
        print (ergebnis)
```

Aufruf:

```
eingabe = int(input("Geben Sie eine Zahl ein: "))
fak(eingabe)
```

Beispiel:

```
def formel(start,ende,schrittweite):
    for i in range(start,ende,schrittweite):
        print(i,end=",")
```

Aufruf:

```
formel(3,20,1)        #Ausgabe: 3,4,5,6,. . .,18,19,
formel(1,20,3)        #Ausgabe: 1,4,7,10,13,16,19,
formel(20,3,1)        #keine Ausgabe
```

Beispiel:

```
def punktkoordinaten(aktuellerPunkt):
    xKoordinate=aktuellerPunkt[0]
    yKoordinate=aktuellerPunkt[1]
    if xKoordinate > 0 and yKoordinate > 0:
        print ("Punkt liegt im 1.Quadranten")
    else :
        print ("Punkt liegt nicht im 1.Quadranten!")
```

Aufruf:

```
punkt=(10,20)
punktkoordinaten(punkt)
```

- Wird einer Variablen in einer Funktion ein Wert zugewiesen, dann handelt es sich um eine lokale Variable. Beim Zugriff auf eine Variable wird zuerst im aktuellen Bereich (local scope) nachgeschaut, dann im darüber liegenden usw. bis der Interpreter im Modul angekommen ist (global).

- Die Datentypen der Argumente müssen mit den Datentypen der Parameter übereinstimmen.

Beispiel:

```
def fu(a,b):
     c=a-b
     return c
```

Aufruf:

```
erg=fu(12,3)      #Ausgabe: 9
erg=fu("susi",3)  #dieser Aufruf ist falsch, weil von einem String kein
                   #Integerwert abgezogen werden kann ➔Fehlermeldung
print(erg)
```

Funktion mit Rückgabewert

Eine Funktion muss einen Wert zurückgeben. Jede Funktion gibt automatisch *None* zurück, wenn der Rückgabewert nicht ausdrücklich angegeben wurde.
Für die Rückgabe von Ergebnissen wird das Schlüsselwort `return` verwendet.
Eine Return-Anweisung führt zur unmittelbaren Beendigung der Funktion, wobei der angegebene Returnwert zurückgegeben wird.

Beispiel: eine Integer-Zahl wird zurückgeliefert

```
def summe():
    start=1; ende = 10; gesamtsumme=0;
    while (start < ende):
        gesamtsumme += start
        start+=1
    return Gesamtsumme
```

Aufruf:

```
print(summe())
erg=summe()*100
```

Beispiel: ein boolscher Wert wird zurückgeliefert

```
def vergleichen(a, b):
    if (a < b):
         return True
      else :
         return False
```

Aufruf:

```
x = 12; y = 23;     #Funktion gibt True zurück, weil 12 kleiner als 23 ist
if vergleichen(x, y):
    print (str(x) + " ist kleiner als " + str(y))
else:
    print (str(x) + " ist größer oder gleich " + str(y))
```

Beispiel: ein Tupel wird zurückgeliefert

```
def punktkoordinaten(x,y):
    x+=10
    y+=20
    return (x,y)

xPunkt = 0; yPunkt = 0;
(xNeu,yNeu)= punktkoordinaten(xPunkt,yPunkt)
print(xNeu,yNeu)
```

2.10.5.7 Liste

Die Liste ist ein Datentyp, den man sehr gut aus dem Alltag kennt:

- Bestsellerliste des Buchhandels
- Die Hörer-Hitparade von Ö3
- Abonnentenliste eines Zeitschriftenverlages

In einer Liste sind mehrere Objekte zusammengefasst. In der Praxis sind diese Objekte meistens vom gleichen Typ, sie müssen es aber nicht sein. (Die Abonnentenliste eines Zeitschriftenverlages kann z.B. Privatpersonen oder Firmen enthalten.)

Typisches Merkmal einer Liste ist, dass sie sich ständig ändert. (Ein Musiktitel kommt in die Hitparade, hält sich dort eine Weile und fliegt dann wieder hinaus.)

Eine Python-Liste entsteht, wenn man mehrere Literale durch Kommata getrennt in *eckige Klammern* schreibt; z.B.: [4, 5, 6, 7] oder [] oder tageszeit = [Morgen, Mittag, Abend]

Die Liste [] enthält kein einziges Element, sie ist leer. Wie Tupel können auch Listen Objekte völlig unterschiedlicher Typen und Variablen enthalten,
z.B.: liste01 = [69, „feine Sache", True, (4, 5, 6, 7)]

Eine Liste kann andere Listen als Elemente enthalten. Man spricht dann von einer Multilisten-Struktur oder von verschachtelten Listen,
z.B.: tabelle = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]

Das erste Element der Liste tabelle, das den Namen tabelle[0] trägt, ist nicht die Zahl 1, sondern die Liste [1, 2, 3].

Der Unterschied zwischen Listen und Tupeln ist, dass man Listen verändern kann, Tupel aber nicht; zum Beispiel kann man an eine Liste ein neues Element anhängen.

2.10.5.8 Einige Grundoperationen für Sequenzen

Zugriff auf Elemente

Man kann auf ein einzelnes Element zugreifen, indem man hinter dem Namen der Sequenz in eckigen Klammern den Index angibt;

z.B.: ampel = (‚rot', ‚gelb', ‚grün') ; ampel[1] → ‚gelb'

Verwendet man negative Indexe, so wird von hinten gezählt. Ist s der Name einer Sequenz, so bezeichnet s[-1] das letzte Element, s[-2] das vorletzte, usw.

Konkatenation

Sequenzen gleichen Datentyps können konkateniert, das heißt, aneinandergehängt werden - als Konkatenationsoperator wird das Pluszeichen + verwendet;

z.B.: [1, 2, 3] + [4] → [1, 2, 3, 4]

Sequenzen unterschiedlicher Datentypen können nicht konkateniert werden;
z.B. kann ein String nicht an eine Liste angehängt werden (…… um das zu erreichen, müsste man die Zeichenkette zuerst in eine Liste umwandeln, dann die beiden Listen zusammenhängen).

Vervielfältigung

Mit Hilfe des Multiplikationsoperators * werden Sequenzen vervielfältigt.
Seien s eine Sequenz beliebigen Typs und n eine nichtnegative ganze Zahl (Typ int).
Dann liefern die Ausdrücke s * n und n * s eine neue Sequenz des gleichen Typs wie s, die durch n-maliges Aneinanderhängen der Sequenz s gebildet wird;

z.B.: 4 * [4] → [4, 4, 4, 4] oder 2 * „Hört! „ → ‚Hört! Hört! ' oder 0 * (True) → ()

Bestimmung der Länge einer Sequenz

Im Unterschied zu Zahlen besitzen Sequenzen eine Länge. Die Länge ist die Anzahl der Elemente einer Sequenz; sie kann mit der Standardfunktion len() ermittelt werden;

z.B.: liste = [[1, 2, 3, 4]] ⇨ len(liste) → 1 ; len(liste[0]) → 4
len((5, 6, 7)) → 3 ; len('') → 0

Weitere Operationen für Sequenzen:

Operation	*Ergebnis*
x in s	1, wenn ein Element mit Wert x in der Sequenz s enthalten ist, sonst 0
x not in s	0, wenn ein Element mit Wert x in der Sequenz s enthalten ist, sonst 1
s[i]	das i+1-te Element der Sequenz s ; z.B.: liste = [[1, 2, 3, 4]] ⇨ liste[0][2] → 3
s[i:j]	ergibt einen Ausschnitt (slice) von s, der das i+1-te bis j-te Element umfasst (d.h., ein Ausschnitt enthält *nicht* das j+1-te Element)
s[:]	dupliziert die Sequenz s
min(s)	ergibt das kleinste Element in s
max(s)	ergibt das größte Element in s

Wichtige Operationen für Listen:

Listen sind veränderbare Sequenzen und können Objekte beliebigen Typs enthalten.
Die meisten damit verbundenen Operationen sind Methoden, deren Aufruf einer Botschaft an ein Objekt vom Typ ‚Liste' entspricht, etwas bestimmtes zu tun; sie hat folgendes Format:

objekt.methodenname(argumente)

Operation	*Ergebnis*
g[i] = x	Das Element von g mit Indexwert i wird durch x ersetzt.
g[i:j] = [a_1, …., a_k]	Die Elemente von g mit den Indexwerten von i bis j werden durch die Elemente der Liste [a_1, …., a_k] ersetzt.
g.append(s)	An die Liste g wird als neues Element s angehängt.
g.extend(s)	An die Liste g werden die Elemente der Sequenz s angehängt.
g.count(x)	Liefert die Anzahl der Listenelemente in g mit dem Wert x .
del g[i]	Das Element mit Index i in der Liste g wird gelöscht; die Länge von g wird um 1 weniger.
del g[i:j]	Die Elemente mit den Indexwerten i bis j in Liste g werden gelöscht.
g.index(x)	Es wird der kleinste Index i zurückgegeben, für den g[i] == x gilt.
g.insert(i, x)	Wenn i >= 0 , dann wird das Objekt x vor dem Element mit Indexwert i eingefügt.
g.pop()	Das letzte Listenelement in g wird weggenommen und als Wert zurückgegeben
g.remove(x)	Das erste Element in g mit dem Wert x wird aus der Liste gelöscht.
g.reverse()	Die Reihenfolge der Elemente in g wird umgedreht.
g.sort()	Die Elemente der Liste g werden im Wert aufsteigend sortiert.