Vorlesung

2

Inhalt

Vorlesung

2

Programmieren – Erste Schritte

Im ersten Teil (die ersten drei Wochen) dieser Veranstaltung sollen Sie die grundlegenden Mechanismen der ***imperativen Programmierung*** *(im Sinne der strukturierten, prozeduralen und modularen Programmierung) kennenlernen. Sie sollen dabei das Programmieren als Methode zur Problemlösung kennenlernen, das Programmieren als exakte und vom Computer interpretierbare Ausdrucksweise zur Formulierung von Bearbeitungsvorschriften , also Algorithmen, erlernen.*

1 Auswahl von Programmiersprachen

Je nachdem, was man als Unterscheidungsmerkmale für verschiedene Programmiersprachen wählt, sind heute bis zu 8500 Programmiersprachen dokumentiert. Für jedes Programmierprojekt, wie auch für diese Veranstaltung, muss eine Programmiersprache ausgewählt werden und das ist manchmal wirklich schwierig.

Dadurch, dass man in der Praxis in der Regel nicht auf der „grünen Wiese" beginnt, sind die Programmiersprache und die Programmierumgebung durch das Vorhandene oft schon festgelegt. Für kleine Programme (Programmieren als Werkzeug in der Hand der ProgrammiererIn) nehmen Sie immer die Sprache, die Sie am besten beherrschen. Wenn beides nicht gegeben ist, wird es schwierig – werden Diskussionen oft „religiös" geführt: **Es gibt keine optimale, universelle Sprache.**

Eine kleine Liste von Argumenten führte zu unserer Entscheidung in PRG 1 Python einzusetzen:

Wichtig für Sie:

- Sie müssen verschiedene Paradigmen kennenlernen und deren Stärken und Schwächen erfahren!
- Dynamisches versus statisches Typing erfahren.
- Interpreter- und Compiler-(Sprachen) kennenlernen.

- Lernen, wie man sich eine neue Programmiersprache selbständig erarbeitet, sie erlernt!
- Gefühl für einen guten Stil entwickeln.
- Programme so zu schreiben, dass sie von anderen verstanden werden können (➔ Teamarbeit).
- Denken in Strukturen und Algorithmen.
- Wissen erwerben: Wie lese ich größere Programme (was ist wichtig, was unwichtig).

Unwichtig:

- Eine bestimmte Sprache bis zur letzten Feinheit erlernen.
- (Hoch-)Effiziente Programme zu schreiben.
- Trickreiche Programme zu schreiben.

Dies alles ist nicht in einem Semester erreichbar. Wir haben uns für folgende Vorgehensweise und Sprachen entschieden:

PRG 1: **Python** (Interpreter, dynamisches Typing, imperativ (prozedural) und objektorientiert)
PRG 2: **SQL** (Abfragesprache) – **Haskell** (funktional)
PRG-Praktikum: **Java** (Umstieg in PRG 2 und einem speziellen Kurs, Nutzung einer größeren IDE)

Praktika im Vertiefungsstudium (je nachdem: C, C++, C#, Java)

In aller Regel liegen Programmiersprachen und die zugehörigen Werkzeuge (Compiler, Interpreter, Entwicklungswerkzeuge) in verschiedenen Versionen vor: Bei Python aktuell die **Version/Release 3.1.2** – die von uns benutzte Version. Auch dies ist „nur" eine Konvention: Die erste Ziffer (3) bezeichnet die Hauptversion, die zweite (1) eine Unterversion, usw. Dies ist zwar anfangs ärgerlich, aber doch ganz normal: Nicht nur Sie machen Programmierfehler, auch die „Profi-Programmierer", die den Interpreter und das Entwicklungswerkzeug IDLE oder gar die Sprachdefinition bauen: Ganz normal und dann wird dies eben in nachfolgenden Versionen korrigiert, auch während unseres laufenden Semesters:

- Python 3.4.0 alpha 3 (September 29, 2013)

- Python 3.3.2 (May 15, 2013)
- Python 3.2.5 (May 15, 2013)
- Python 3.1.5 (April 10, 2012)
- Python 3.0.1 (February 13, 2009)
- Python 2.7.5 (May 15, 2013)
- Python 2.6.8 (April 10, 2012)
- Python 2.5.6 (May 26, 2011)
- Python 2.4.6 (December 19, 2008)
- Python 2.3.7 (March 11, 2008)
- Python 2.2.3 (May 30, 2003)
- Python 2.1.3 (April 8, 2002)
- Python 2.0.1 (June 2001)
- Python 1.6.1 (September 2000)
- Python 1.5.2 (April 1999)

Abb. 2.1: Release Schedule von Python-Versionen (von http://www.python.org/download/releases/)

Sie sehen: An Python wird aktuell intensiv gearbeitet

Vielleicht noch etwas zur **Versionsbezeichnung**, auch Sie sollten sich angewöhnen, damit zu arbeiten: Bevor eine Version das erste Mal freigegeben wird, ist die Hauptversion 0, also zum Beispiel 0.4.2. Die erste freigegebene Version trägt dann die Nummer 1. (z. B. Ihre Abgabe). Folgende Zusatzbezeichnungen sind noch üblich:

- Alpha: Die erste zum Test durch Fremde (also nicht den eigentlichen Entwicklern) bestimmte Version;
- Beta-Versionen sind öffentlich gemachte Versionen zum Review (meist für „Powerkunden");
- rc-Versionen sind „release candidates" (oder auch prerelease-Versionen) – Versionen, die kurz vor öffentlichen Freigabe einer Release veröffentlicht werden.

Grob kann man folgendes unterstellen: Eine wesentliche Änderung z.B. der Sprachdefinition findet seinen Niederschlag in der Hauptversionsnummer. Diese Änderungen bewirken oft auch **Inkompatibilitäten**, d.h. fast alle Programme sind in zwei verschiedenen Hauptversionen **nicht** lauffähig. Zumindest bei größeren Softwareprojekten werden in solchen Situationen in aller Regel Hilfswerkzeuge zur Verfügung gestellt, um eine Versionsänderung durchführen zu können. Bei Python gibt es ein Konvertierungsprogramm

2to3.py siehe http://docs.python.org/release/3.1.2/library/2to3.html#to3-fixers ,

das Version 2 – Programme in Version 3 – Programme übersetzt (dieses Programm (Skript) ist normalerweise mit der Installation installiert.

Wichtig: In unserer **LernBar** ist allerdings eine **Jython**-Implementierung und zwar Version 2.5.2.

Was ist Jython? - Jython (früher: JPython) ist eine reine Java-Implementierung der Programmiersprache Python und ermöglicht somit die Ausführung von Python-Programmen auf jeder Java-Plattform. Die Entwicklung hinkt der C-Version ein wenig hinterher, sie jetzt bei Version 2.5.

Ist das ein Problem: In aller Regel jetzt am Anfang des Programmierens nicht. Bei Unterschieden weisen wir darauf hin.

2 Regeln und Konventionen beim Programmieren

Der Quelltext eines Programms (*program*) in einer imperativen Programmiersprache besteht aus einer (wohlgeordneten) Folge von Anweisungen (*statements*). Die Anweisungen werden durch ein Trennsymbol, häufig einem Semikolon, voneinander getrennt. Diese Anweisungen werden bei der Programmausführung der Reihe nach, vom der ersten bis zur letzten Anweisung ausgeführt; danach hält der Interpreter an – das Programm ist beendet.

In Python wird als Trennsymbol zwischen zwei Anweisungen meist einfach das „Return-Zeichen" (↵) genommen; zulässig als Trennsymbol zwischen zwei Anweisungen ist aber auch das Semikolon. Gemäß der Syntaxregeln der Programmiersprache Python sind also zulässig:

```
Anweisung1      aber auch       Anweisung1; Anweisung2;
Anweisung2                      Anweisung3;
Anweisung3                      ...
...
```

Wir unterscheiden zwischen **Syntaxregeln,** oder kurz Regeln, und **Konventionen.** Unter Syntax versteht man versteht man ein System von Regeln, nach denen erlaubte Konstruktionen, hier Anweisungen, aufgebaut sein dürfen. Syntaxregeln dienen insbesondere der Eindeutigkeit der Interpretation der jeweiligen Anweisung. Falls Syntaxregeln verletzt werden, reagiert der Interpreter mit einer Fehlermeldung.

Konventionen hingegen sind Vereinbarungen unter Programmierern, um die Lesbarkeit und Übersichtlichkeit eines Programms zu verbessern: Sie dienen allein den Programmierern – insbesondere wenn diese im Team arbeiten müssen und sie auch Programme anderer zu lesen haben. In größeren Projekten werden diese Konventionen in sogenannten Programmierhandbüchern oder „Style Guides" festgehalten und ggf. durch sogenannte „Style Checker" (spezielle Programme zur Überprüfung der Einhaltung der Konventionen zusätzlich zum Interpreter oder Compiler) oder gar sogenannte „Buityfier", also Programme genutzt, die ein syntaktisch korrektes Programm in einen „guten Style" überführen.

Hier ein wichtiger Hinweis: Bitte gewöhnen Sie sich einen guten Programmierstil an und halten Sie sich an die Konventionen, nur so ist ihr Quelltext auch von anderen schnell lesbar. Dies nicht zu tun ist, wie in einem guten Restaurant mit dem Messer zu essen oder gar den Teller abzulecken. Wir werden die anzuwendenden Konventionen minimal halten aber dann auch in den Übungen darauf Wert legen, es gibt ggf. Punktabzug. Wir beziehen uns auf den „offiziellen" „Style Guide for Python Code" von Guido van Rossum, siehe http://www.python.org/dev/peps/pep-0008/ und die „Python Coding Guidelines 12/14/07" von Rob Knight, siehe http://jaynes.colorado.edu/PythonGuidelines.html .

2.1 Python (Syntax-)Regeln

Die Syntax-Regeln von Python lernen Sie im Zuge dieser Vorlesung, jeweils zusammen mit den entsprechenden Konzepten.

2.2 Python Konventionen

Begrenzen Sie eine Anweisungszeile auf maximal 79 Zeichen. Falls die Anweisung länger als 79 Zeichen ist, benutzen Sie den Backslash (\) als Zeilenfortsetzungszeichen (der Interpreter versteht die so physisch getrennten Zeilen dann logisch als eine Zeile).

Fast immer ist es übersichtlicher **nur eine** Anweisung pro Zeile zu haben, nur sehr eng logisch zusammengehörende Anweisungen sollten in eine Zeile durch Semikolon getrennt geschrieben werden. Verzichten Sie auf das Semikolon am Zeilenende. Auf keinen Fall möchten wir solche Schreibweisen sehen:

```
Anweisung1; Anweisung2;  ....  Anweisungn(alpha, \
                                    beta, gamma);
```

Strukturieren Sie Ihren Programmtext in „logische Blöcke", gewissermaßen „Absätze", durch die Verwendung von „Leerzeilen", aber angemessen, bitte nicht zu viele!

Programmier-Rabauken beachten Konventionen nicht – aber das sind wir nicht! Wir kennzeichnen Konventionen im Folgenden durch die hellblaue Hinterlegung.

3 Variable – Zuweisung – Literal

Ein zentraler Begriff in (fast) allen Programmiersprachen ist „Variable" (*variable*). Wir verstehen darunter folgendes Tripel, das fest miteinander verkoppelt ist:

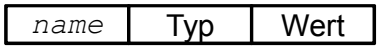

Abb. XX: Konzeptionelles Modell einer Variablen

Unter *name* oder Bezeichner (*identifier*) versteht man einen in einem Namensraum (hier zunächst das gesamte Programm) eindeutiges Wort (eindeutig = unterscheidbares), unter dem die Variable im Programmtext angesprochen werden kann. Ein *name* besteht aus einzelnen Zeichen, die meist aus einem eingeschränkten Zeichensatz entnommen sind, zum Beispiel keine Umlaute wie ö,ü,ä enthalten dürfen, usw.
Der Typ (*type*) oder Datentyp bezeichnet die Zusammenfassung konkreter Wertebereiche von Variablen und darauf definierten Operationen zu einer Einheit. Zum Beispiel eine Ganzzahl (*integer*) im Wertebereich (-128,

-127, ..., 0, 1, ...127) und die darauf definierten Operationen, wie +, -, *, usw. Der Typ bestimmt implizit auch, wie groß der Speicherbereich für den Wert sein muss und wie die Bitfolge „Wert“ zu interpretieren ist. Näheres dazu in der nächsten Vorlesung (Elementare Datentypen).

Eine Variable wird häufig auch als „Container“ beschrieben, entspricht also einen bestimmten Bereich des „benamten“ Speichers, in dem Daten für die Laufzeit des Programms gehalten werden können. Ein solcher Speicherbereich kann sowohl gelesen, als auch beschrieben werden. Das Schreiben einer Variable erfolgt in höheren Programmiersprachen durch den **Zuweisungsoperator,** bei dem die Variable auf der linken Seite steht, in Python einfach dadurch, dass man schreibt:

```
foo = 42
```

foo ist dabei der Name der Variablen die geschrieben werden soll, = ist der Zuweisungsoperator (*assignment*) und 42 ist ein sogenanntes **Literal,** d.h. eine Zeichenfolge, die zur Darstellung des Wertes und des Typs von Daten (hier z. B. eine Ganzzahl) definiert bzw. zulässig sind. Durch das Literal wird auch der Datentyp auf der rechten Seite des Gleichheitszeichens festgelegt und speziell in Python damit auch der Datentyp von foo , in diesem Fall als Ganzzahl.

Für jeden Datentyp müssen wir genauer betrachten:

- den zulässigen Wertebereich
- die spezifisch definierten Literale
- die für diesen Typ definierten Operatoren
- die spezifischen Funktionen auf diesem Datentyp
- die spezifischen Eigenarten

Neben den eingebauten (vordefinierten) **Operatoren** (einstellige oder zweistellige) bieten sogenannte **Funktionen** ggf. komplexere Berechnungen an: (mehr als zwei Eingabeparameter, mehr als ein Ausgabewert, usw.). Funktionen können selbst definiert werden (das betrachten wir später), hier beleuchten wir nur die sogenannten builtins, das heißt die Funktionen, die Python bereithält.
Funktionen werden wie in der Mathematik üblich z.B. mit

```
Funktionsname(x,y)
```

aufgerufen (für eine zweistellige Funktion mit den Parametern x und y) und liefern einen Funktionswert. Eine Funktion kann überall dort stehen, wo auch eine Variable stehen kann.

Konventionen für Funktionsaufrufe: Nach dem Funktionsnamen steht k**ein** Leerzeichen und nach der öffnenden Klammer „(„ sowie vor der schleißenden Klammer „)“ stehen **keine** Leerzeichen (*blanks*).

4 Der Zuweisungsoperator

Durch den Zuweisungsoperator = wird der Typ und der Wert einer Variablen (konzeptionell) verändert, der alte Wert ist danach unter diesem Namen nicht mehr verfügbar. Mit der Ausführung des Zuweisungsoperators wird in Python auch der benötigte Speicherplatz für diesen Datentyp reserviert und bereitgestellt, dies nennt man dynamische Speicherverwaltung. – Sie als Programmiererin müssen sich hierbei um nichts kümmern, die Speicherverwaltung übernimmt für Sie der Python Interpreter.

Auf der rechten Seite eines Zuweisungsoperators dürfen nicht nur Variablennamen stehen, sondern sogenannte Ausdrücke (Kombinationen aus Namen, Operatoren und Funktionen), die für numerische Datentypen einem algebraischen Term entsprechen. Der Datentyp bestimmt, welche Operatoren erlaubt sind. Für numerische Typen sind insbesondere + , -, * (als Multiplikation), / (als Division) definiert, wir können also schreiben:

```
a = 5
b = 2
c = a + b
d = a * b
```

Achtung: Aus der Mathematik ist uns der Begriff *„Variable"* bereits bekannt, allerdings in einer deutlich anderen Bedeutung. Dort kennen wir Variable im Sinne eines Platzhalters für einen unbekannten Wert, den es zu errechnen gilt. Anstelle eines konkreten Zahlenwertes werden dafür Symbole, meist Buchstaben, wie x und y, genutzt. Durch die Verwendung von Buchstaben als Platzhalter können wir beispielsweise ein Gleichungssystem aufstellen. Lösen wir dieses Gleichungssystem, so erhalten wir einen konkreten Wert für unsere Variable. Bis zum Lösen des Gleichungssystems haben wir kein Wissen über den Wert der Variablen, daher der Name Variable.

Beispiel:
x = 1
x = 4 * x - 2

Fasst man diese zwei Zeilen als Anweisungsfolge auf, so ist nach Ausführung der Wert der Variablen x = 2. Fasst man diese Zeilen als Gleichungssystem auf, so ist es die Lösungsmenge $= \left\{\frac{3}{4}\right\}$.

Um diese verschiedene Bedeutungen des Gleiheitszeichens deutlich zu machen und Verwechselungen zu vermeiden, werden in manchen Programmiersprachen als **Zuweisungsoperator** andere Symbole benutzt, z.B.

= in **Python**, Java, C, C++
=: in Pascal, Ada
← in APL

In Python gibt es neben der einfachen Zuordnung

```
ziel = ausdruck
```

zwei weitere Zuordnungsvarianten, nämlich

```
ziel1 = ziel2 ...  = ausdruck                und
ziel1, ziel2, ...  = ausdruck1,  ausdruck2, ...
```

Die erste Form weißt jedem Ziel ziel1, ziel2, ... denselben Wert zu. Die zweite Form weist paarweise zu von links nach rechts, also ziel1 = ausdruck1 und ziel2 = ausdruck2, usw. Dabei werden zunächst die Werte beider Ausdrücke berechnet und dann den Zielen zugewiesen. Bitte entscheiden Sie selbst, ob dies übersichtlicher ist, als die zeilenweise Schreibweise. Die zweite Form ist allerdings hilfreich, wenn Sie die Werte zweier Variablen vertauschen wollen:

```
x = y            und        x, y = y, x
y = x
```

liefern verschiedene Ergebnisse: Die linke Anweisungsfolge hat zum Ergebnis, dass sowohl x als auch y denselben Wert, nämlich y haben, während durch den Ausdruck rechts die Werte von x und y ausgetauscht werden. In zeilenweiser Schreibweise wäre dies nur über eine Hilfsvariable möglich, also

```
z = y
y = x
x = y
```

Python Konventionen: Leerzeichen haben für den Interpreter **keine** syntaktische Bedeutung (außer am Anfang einer Zeile, aber das betrachten wir später) sie werden einfach „überlesen". Für den Programmierer schaffen sie Übersichtlichkeit. Beachten Sie, dass man üblicherweise vor und nach jedem Operatorzeichen, vor und nach dem Gleichheitszeichen als Zuordnungsoperator und nach jedem Komma ein Leerzeichen (*blank*) lässt.

Python (wie viele andere Programmiersprachen) verfügt über weitere Varianten der Zuweisung, nämlich die sogenannten „erweiterte Zuweisungen" (*augmented assignments oder in-place assignments*). Vor dem Gleichheitszeichen steht dabei ein zweistelliger (binärer, *binary*) Operator (siehe unten). Folgende Anweisungen liefern dasselbe Ergebnis:

`x += y` entspricht `x = x + y`

Die erste Form ist etwas effizienter, weil der Interpreter den Namen x nur einmal „auflösen" muss.

5 Ausdrücke und Operatoren

Der Begriff „Ausdruck" ist im Wesentlichen synonym zu dem in der Mathematik gebräuchlichen Begriff „Term". Der Unterschied ist „feinsinnig". Ein Ausdruck in einer formalen Sprache (also einer Programmiersprache) ist immer eindeutig interpretierbar. Da die Symbolik der Mathematik nicht fix definiert, sondern beliebig erweiterbar ist, ist damit auch Term ein erweiterbarer Begriff, nicht fix.

Ein Ausdruck (in Programmiersprachen) kombiniert Operatoren, Operanden und Funktionen. Operanden sind hier Variablen (-namen) oder Literale. Ein Operator ist eine mathematische Vorschrift, durch die man aus mathematischen Objekten neue Objekte erzeugt. Operator verknüpfen Operanden und sind entweder ein- oder zweistellige Verknüpfung. Operatoren werden durch spezielle, kennzeichnende mathematische Symbole (meist ein spezielles Schriftzeichen der Formelschreibweise) dargestellt.
Beispiele für Ausdrücke:

`-x`	einstelliger Operator der Negation, anwendbar für $x \in \mathbb{Q} \vee \mathbb{R}$
`x+y`	zweistelliger Operator für die Addition, anwendbar für $x, y \in \mathbb{N} \vee \mathbb{Q} \vee \mathbb{R}\ x \in \mathbb{Q} \vee \mathbb{R}$

In Programmiersprachen besteht zusätzlich die Anforderung, dass Operatoren einfach durch die Tastatur einzugeben sind. Viele in der Mathematik üblichen Symbole für Operatoren wie

$$\neq \leq \in \otimes \frac{x}{y} \; x^y \; \int x dx \; \sqrt[x]{y} \; x^y \text{ usw.}$$

scheiden damit aus. In Programmiersprachen benutzt man übelicherweise nur die im ASCII-Alphabet (englisches Standardalphabet) vorhandenen Sonderzeichen, manchmal auch zwei hintereinander geschriebene Zeichen, wie „!=", anstelle von „≠" oder durch reservierte Schlüsselwörter (siehe unten unter Variablennamen) wie **not**, **and**, **or**, usw..

In Python werden die in Tabelle 2.1 angegebenen Operatoren unterstützt. Wichtig ist noch die Auswertereihenfolge:

1. Bei verschiedenen Operatoren muss geregelt sein, welcher zuerst ausgewertet wird: Zum Beispiel wie in der Mathematik üblich: „Punkt vor Strichrechnung", d.h. die Punktoperatoren (· und :) **binden stärker** (werden zuerst ausgeführt) als die Strichoperatoren (+ und -).
2. Bei gleichstark bindenden Operatoren muss geregelt werden, in welcher Reihenfolge
   x OP y OP z

ausgewertet wird. Bei echt assoziativen Operatoren wie + oder · ist dies gleichgültig, weil
a+b+c = (a+b)+c = a+(b+c) gilt. Aber schon bei der Subtraktion ist dies relevant:
4-3-2 ist nicht eindeutig: (4-3)-2 = -1 während 4-(3-2) = 3.

In der Mathematik wird in der Regel Links-Assoziativität angenommen, dass heißt der links stehende Operator wird zuerst ausgewertet, aber bei x^{y^z} nimmt man aber meist Rechts-Assoziativität an, d.h. y^z wird zuerst ausgewertet.

3. Durch Klammern kann die Auswertereihenfolge geändert werden. Der Ausdruck in der Klammer wird zuerst ausgewertet. Anmerkung: Klammern muss man beim Programmieren „zählen“, eckige oder geschweifte Klammern haben eine andere Bedeutung als runde Klammern!

In Python werden die in Tabelle 1 genannten Operatoren, deren Auswertungsreihenfolge (Vorrangregeln) und deren Verwendbarkeit in der erweiterten Zuweisung angegeben. Man sieht, dass die in der Mathematik übliche Priorität genutzt wird – man kann also Ausdrücke wie üblich formulieren. Alle Operatoren außer der Exponentenbildung (x ** y ** z) werden von **links nach rechts ausgewertet**.

Die Tabelle ist nach Priorität sortiert aufgeführt (oben höchste, unten geringste Priorität/Bindung). Das heißt Operatoren, die weiter oben in der Tabelle aufgeführt sind, werden vor solchen ausgewertet, die weiter unten aufgeführt sind. (Man beachte, dass einige Operatoren, wie x * y, x / y, x//y und x % y) gleiche Priorität haben und deshalb von links nach rechts ausgewertet werden.

Programmierhandzettel 1: Operatoren und Auswertereihenfolge in Python

Operatoren	**Kurzbeschreibung**	**String / Unicode**	**Float**	**Integer**	**Boolean**	**In erweiterter Zuweisung anwendbar**
`(...)`	(Vorrang) Klammerung	x	x	x	x	
`s[i]`	Indizierung (bei Sequenztypen)	x				
`s[i:j]`	Teilbereiche (bei Sequenztypen)	x				
`f(...)`	Funktionsaufruf	x	x	x	x	
`+x, -x,`	Einstellige Operatoren		x	x		
`~x`	Invertiere x			x		
`x ** y`	Exponential-Bildung x^y (Achtung: rechts-assoziativ)		x	x		x
`x * y`	Multiplikation (Wiederholung)	x	x	x		x
`x / y`	Division		x	x		x
`x % y`	Modulo (-Division) = (Ganzzahliger) Rest		x	x		x
`x // y`	Restlose Division [2])		x	x		x
`x + y`	Addition (Konkatenation)	x	x	x		x
`x - y`	Subtraktion		x	x		
`x << y, x >> y`	Bitweises Schieben (nur bei Integer)			x		x
`x & y`	Bitweises Und (nur bei Integer)			x		x
`x ^ y`	Bitweises exklusives Oder (nur bei Integer)			x		x
`x | y`	Bitweises Oder (nur bei Integer)			x		x
`x >= y, x == y, x != y`	Vergleichsoperatoren liefern als	x	x	x	x	
`[x <> y]`	Ergebnis **`True`** oder **`False`** [1])					
	Test auf Identität					
`x is y, x is not y`	Tests auf Enthaltensein in	x	x	x	x	
`x in s, x not in s`	Sequenzen	x				
`not x`	Logische Negation				x	
`x and y`	Logisches Und				x	
`x or y`	Logisches Oder				x	

Anmerkungen: Oben stehen die Operatoren mit höchster Priorität.

[1]) Vergleichsoperatoren dürfen verkettet werden: `x < y < z` ist im Ergebnis identisch mit `(x < y) and (y < z)`, nur in der Ausführung etwas schneller.

[2]) `x // y` schneidet den Rest bei einer Division ab (in Python bis Version 2.6 ist dies bei einer Ganzzahldivision identisch zu `x / y`. Ab Python 3 wird im Falle eines Restes ≠ 0 automatisch in eine Gleitpunktzahl konvertiert.

Ausdrücke sind sehr mächtige Programmiermittel. Beachten Sie, dass einige Operatoren nur für bestimmte Datentypen definiert sind, wendet man sie auf andere an, so folgt eine Fehlermeldung und damit der Abbruch des Programms. Die konkrete Semantik (Bedeutung) eines Operators wird nur im Zusammenhang mit einem Datentyp festgelegt. Beispiel: + ist für numerische Datentypen die Addition, beim String (Zeichenkette) die Konkatenation (Aneinanderfügung). Wir müssen also die Semantik der Operatoren im Zusammenhang mit den Datentypen noch einmal genauer betrachten. Und, man braucht etwas Übung um diesen Teil der Programmierung zu beherrschen, siehe hierzu den *eKurs2 – Erste Python Programme*.

Sehr wichtig sind auch die sogenannten Vergleichsoperatoren. Vergleiche geben als Ergebnis Wahr oder Falsch (`True oder False`). Details betrachten wir in der nächsten Vorlesung: Boolescher Datentyp.

6 Variablennamen

In Python gelten folgende Namensregeln: Von der Programmiererin definierte Namen beginnen mit einem Buchstaben (`a, ..., z, A, ... Z`) oder einem Unterstrich (*underscore*) (_). Namen können beliebig lang sein und ab 2. Zeichen zusätzlich auch Ziffern (0,...,9) enthalten.

Kommentar hierzu: Groß- und Kleinschreibung ist **immer** relevant, also `spam` und `Spam` sind verschiedene Namen. Umlaute wie ä,ö,ü oder Ä,Ö,Ü oder das ß oder andere Sonderzeichen wie !,§,$, ... sind als Zeichen in Namen nicht erlaubt.

In Python (3.x) sind die folgenden Wörter (für die Sprache selbst) reserviert und als Variablennamen verboten, weil sie eine spezielle Bedeutung in der Sprachsyntax haben. Wir nennen sie „Schlüsselworte der Sprache und diese werden in der IDLE (Python Shell) farblich markiert.

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

Namens-Konventionen für Variablen:

`nomen_mit_unterstrich` Beispiel: `current_index`

In Python gibt es keine Möglichkeit Konstanten zu deklarieren. Konstanten sind Variablen, die zur Laufzeit des Programmes nicht geändert werden sollen/dürfen (Dafür müssen Sie in Python allerdings selbst sorgen!).

Namens-Konventionen für Konstanten:

NOMEN_IN_GROSSBUCHSTABEN Beispiel: `MAX_LENGTH`

Zur Namensgebung selbst: Gewöhnen Sie sich gleich an, englische Namen zu benutzen. Stellen Sie sich vor, Sie sollen ein Programm abändern und der Autor hat Variablennamen in Jukagir gewählt (eine sehr seltene Sprache, die von ca. 170 Menschen im Nordosten Russlands oberhalb des Polarkreises gesprochen wird). Englisch ist nun einmal die "Lingua Franca" (Umgangssprache) der Informatik.

Noch ein paar **Hinweise/Tipps**: Wählen Sie Namen, aus denen das Bezeichnete wahrscheinlich richtig erraten wird: current_line ist viel besser als c oder cl. Dies ist sicher nicht immer leicht. Wer weiß, in ein paar Monaten sind Sie es vielleicht selbst, die/der die Bedeutung dieses Namens erraten muss.

Benutzen Sie den Singular für individuelle Objekte und den Plural für Kollektionen. Ich nehme an, wenn Sie name lesen erwarten Sie einen Namen, wenn Sie names lesen mehrere.

Wählen Sie die Namen so präzis wie möglich: Wenn eine Variable eine eingelesene Zahl ist, nennen Sie diese readin_value oder in_value und nicht input oder value.

Ein Buchstaben Bezeichner sollten nur in sehr überschaubaren Zusammenhängen vorkommen (wo die Variable i oder k) nur eine oder zwei Zeilen „überlebt“, das heißt Bedeutung hat und dann nicht mehr benutzt wird, zum Beispiel als einfacher Index. Strenger: Benutzen Sie nie „l“ (kleines L) oder I (großes i) oder O (großes o) als „Ein Buchstaben Bezeichner“. In einigen Fonts sind diese Zeichen nicht oder kaum unterscheidbar von den Ziffern „1“ (Eins) oder „0“ (Null).

Machen Sie den Datentyp nicht zum Teil des Namens (dieses sieht man manchmal als „falsch verstandene“ „Ungarische Notation“, also benutzen Sie nicht int_number.

Benutzen Sie so wenig wie möglich Abkürzungen, insbesondere keine selbst erfundenen wie zum Beispiel: finutowo für „final number to work on“. Aber allgemein übliche Abkürzungen sind erlaubt, wie

`curr` für current,
`eof` für end of file,
`eol` für end of line,
`in` für input ,
`max` für maximum,
`min` für minimum,
`out` für output,
`prev` für previous,
`ref` für reference,
`str` für string,
`struct` für structure,
`temp` oder `tmp` für temporary.

Jetzt ist es aber erst mal genug mit diesen Formalien. Vieles ist hier auch Geschmacksache. **Es geht allerdings um etwas sehr Wichtiges**, möglichst gut verständlichen Quelltext schreiben, nicht nur für Sie, sondern auch für andere. Jedes gut geführte Softwareentwicklungsprojekt hat Programmierrichtlinien und dieses gilt eben auch für unser Projekt „Python erlenen in PRG1“. Etwas wichtiges noch: Verschiedene Programmiersprachen haben zum Teil auch verschiedene Konventionen: Z.B. sollen Java-Bezeichner (Namen) mit Binnenmajuskeln geschrieben werden (auch CamelCase genannt) und keine Unterstriche („ _ “) enthalten, mit Ausnahme von Konstanten. Tut mir leid.

„A Foolish Consistency is the Hobgoblin (*Kobold*) of Little Minds (*Kleingeister*)", ja, aber es ist trotzdem wichtig: Es geht um gute Lesbarkeit eines Programms. Es kommt einigen von Ihnen vielleicht lächerlich oder nervig vor, dass wir auf Namen so großen Wert legen. Tatsächlich ist aber eine gute Benennung von Variablen (später dann auch für Funktionen, Methoden und Klassen) das **allerwichtigste** Kriterium für eine gute ProgrammiererIn (und ein gutes Design)! Besonders beim objekt-orientierten Programmieren ist das fast noch wichtiger als beim prozeduralen Programmieren. Eine schlechte Benennung kann eine Klasse fast unbrauchbar machen.

Und dabei sagte Goethe: „***Name ist Schall und Rauch***". (Marthens Garten), aber **nicht** in der Programmierung.