Potenzen und Logarithmen Lösung Theorie

Potenzieren

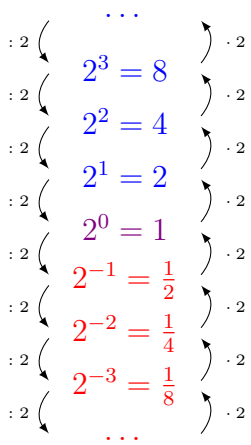
Das Multiplizieren ist eine Abkürzung für wiederholtes Addieren desselben Summanden:

$3 \cdot 2 = 2 + 2 + 2 = 6$ (3: Faktor, 2: Faktor, 6: Produkt)

Das Potenzieren ist eine Abkürzung für wiederholtes Multiplizieren desselben Faktors:

$2^3 = 2 \cdot 2 \cdot 2 = 8$ (2: Basis, 3: Exponent, 8: Potenz)

Zweierpotenzen (sind wichtig in der Informatik)



Liest man die Grafik von oben nach unten, erkennt man, dass jede Verkleinerung des Exponenten um 1 die Potenz halbiert. Damit werden die Resultate $2^0 = 1$, $2^{-1} = \frac{1}{2}$, ... verständlich. Die Potenzen 2^0, 2^1, ..., 2^{10} musst du auswendig lernen.

Logarithmieren

Beim Potenzieren sind Basis und Exponent gegeben und die Potenz wird gesucht.

$2^3 = x \quad \Rightarrow \quad x = 8$

Beim Logarithmieren sind Basis und Potenz gegeben und der Exponent wird gesucht.

$2^x = 8 \quad \Rightarrow \quad x = 3$

Für das Logarithmieren verwendet man folgende Schreibweise:

$x = \log_2 8 = 3$ (2: Basis, 8: Numerus, 3: Logarithmus)

[sprich: Der Logarithmus von 8 zur Basis 2 ist 3]

Bemerkung: Es gibt noch eine zweite Umkehrung des Potenzierens, bei der die Basis unbekannt ist ($x^3 = 8$, $x = ?$). Ihr kennt sie vermutlich aus dem Mathematikunterricht aber sie spielt in unserem Informatikunterricht kaum eine Rolle. Wie heisst diese Operation?

$x^3 = 8 \quad \Rightarrow \quad x = \sqrt[3]{8} = 2$ Radizieren (Wurzelziehen)

Aufgabe 1

(a) $2^4 = 16$

(b) $2^{-3} = \frac{1}{8}$

(c) $2^8 = 256$

(d) $2^{-1} = \frac{1}{2}$

(e) $2^2 = 4$

(f) $2^7 = 128$

(g) $2^{10} = 1024$

(h) $2^{-5} = \frac{1}{32}$

(i) $2^9 = 512$

(j) $2^6 = 64$

(k) $\log_2(32) = 5$

(l) $\log_2(128) = 7$

(m) $\log_2(2) = 1$

(n) $\log_2(1024) = 10$

(o) $\log_2(1) = 0$

(p) $\log_2(256) = 8$

(q) $\log_2(64) = 6$

(r) $\log_{10}(10\,000) = 4$

(s) $\log_{10}(1) = 0$

Rechnen mit dem Python-Interpreter

Python ist eine universelle, interpretierte, höhere Programmiersprache.

- Eine *universelle* Programmiersprache hat keinen besonderen Anwendungszweck.
- Bei einer *interpretierten* Programmiersprache wird der Quellcode zeilenweise übersetzt und dann vom Prozessor ausgeführt. Im Gegensatz dazu übersetzt eine *compilierte* Programmiersprache den gesamte Quelltext und speichert ihn als ausführbare Datei ab, die vom Prozessor direkt (und damit oft schneller) ausgeführt wird.
- *Höhere Programmiersprachen* zeichnen sich dadurch aus, dass die Programme in einer menschenähnlichen Sprache formuliert werden können. Der Quellcode dieser Sprachen ist meist prozessorunabhängig und es können Variablen und Funktionen definiert werden. Letztere erlauben es, bereits geschriebenen Code in anderen Programmen wiederzuverwenden.

IDLE

Mit der Installation der offiziellen Python-Distribution wird auch das Programm `IDLE` installiert, das eine einfache Schaltzentrale für das Arbeiten mit Python ist. Ein solches Programm wird auch IDE (Integrated Development Environment = Integrierte Entwicklungsumgebung) genannt. IDLE umfasst ...

- eine *Shell*, mit der Code zeilenweise eingegeben und ausgeführt werden kann. Die Shell ist nur für kleine Codeschnipsel geeignet.
- einen *Editor* (ein einfaches Schreibprogramm) mit dem auch längere Programme geschrieben und danach von Python ausgeführt werden können sowie

- einen *Debugger*, mit dem sich Fehler in einem Programm analysieren lassen.

Es gibt auch weit umfangreichere IDEs wie z.B. *Visual Studio Code* von Microsoft, *NetBeans* oder *PyCharm*.

Die Shell

Nach dem Start von IDLE befindet man sich in der *Shell*. Die Symbole `>>>` zeigen den nächsten *Prompt* (Eingabeaufforderung) an. Dort gibt man einen Anweisung ein und schliesst sie mit ENTER ab. IDLE sendet sie dann an den Python-Interpreter, der prüft, ob sie syntaktisch korrekt ist. Wenn ja, wird sie vom Interpreter in Maschinensprache übersetzt und ausgeführt[1]. Andernfalls, erhält man eine Fehlermeldung auf der Shell .

In längeren Python-Programmen, die man mit einem Editor schreibt, muss man die `print(...)`-Anweisung verwenden, um einen Output auf der Shell auszugegeben. Wenn man einen Python-Ausdruck direkt in der Shell eingibt und mit ENTER abschliesst, erhält man meist auch ohne `print(...)` eine Ausgabe.

Da sich einmal eingegebene Befehle nach der Eingabe nicht mehr verändern lassen, kann man mit `alt-P` (*Previous line*) und `alt-N` (für *Next line*) frühere oder nachfolgende Eingaben in den Prompt zurückholen und verändern.

Grundoperationen

Addition

```
>>> 3 + 5
    8
```

Subtraktion

```
>>> 3 - 5
    -2
```

Multiplikation

```
>>> (-3) * (-5)
    15
```

Division

```
>>> 10 / 2
    5.0
```

[1] In Wirklichkeit ist es etwas komplizierter aber für uns genügt dies vorerst.

Beachte

- Addition, Subtraktion und Multiplikation ganzer Zahlen ergeben wieder eine ganze Zahl (*integer*).
- Weil bei der Division (`/`) (Dezimal)Brüche entstehen können, ist das Resultat eine Gleitkommazahl (*float*). Beim Rechnen mit Gleitkommazahlen können Rundungsfehler auftreten, wie das folgende Beispiel zeigt.

```
>>> 0.1 + 0.1 + 0.1
    0.30000000000000004
```

- Zusätzliche Leerzeichen (*blanks*) zwischen Operanden und Operatoren können die Lesbarkeit erhöhen und führen im Allgemeinen zu keinen Fehlermeldungen

```
>>> 3+5
    8

>>> 3 +    5
    8
```

Höhere Rechenoperationen

Potenzieren

```
>>> 2**3
    8
```

Radizieren

```
>>> 9**(1/2)
    3.0

>>> 8**(1/3)
    2.0
```

Logarithmieren

Die Logarithmusfunktion muss einmal vor dem ersten Gebrauch aus dem `math`-Modul *importiert* werden. Danach steht sie bis zum Beenden von IDLE zur Verfügung.

```
>>> from math import log
>>> log(8, 2)
    3.0
```

Rechenreihenfolge und Vorzeichen

Python wertet arithmetische Ausdrücke in der üblichen Reihenfolge aus: 1. Klammern; 2. Potenzen, Wurzeln, Logarithmen 3. Multiplikation, Division, 4. Addition, Subtraktion.

Klammern werden von innen nach aussen und Operationen gleicher Stufe von links nach rechts ausgewertet. Ein negatives Vorzeichen wird als Multiplikation mit -1 interpretiert.

`5 + - 3` $\Rightarrow$ `5 + (-1)*3` $\Rightarrow$ `2` `besser/klarer: 5 + (-3)`

Aber Vorsicht: `-3**2 = (-1)*3**2 ergibt -9`
`(-3)**2 = (-3)*(-3) ergibt 9`