# BYTE

# LANGUAGE

Version 5.0

*Offizielle Dokumentation*

by ByteLaunch
https://bytelaunch.de

*Eine Programmiersprache, die sich liest wie natürliches Englisch*

# Schnellstart

Byte ist eine fertige .exe-Datei. Einfach herunterladen, kompilieren und loslegen. Keine Installation, keine Laufzeit, keine Abhängigkeiten.

## Kompilieren

### Windows

```
gcc -O2 -o byte.exe main.c lexer.c parser.c eval.c -lm -lwininet
```

### Linux / Mac

```
gcc -O2 -o byte main.c lexer.c parser.c eval.c -lm
```

## Benutzen

```
byte.exe                      -- Interaktiven REPL starten
byte.exe meinprogramm.byte    -- Byte-Datei ausführen
byte.exe --version            -- Version anzeigen
byte.exe --help               -- Hilfe anzeigen
```

## Systemweit verfügbar machen (optional)

Windows: byte.exe in C:\Windows\System32\ kopieren.

Linux/Mac:

```
sudo cp byte /usr/local/bin/
```

## Erstes Programm

```
-- hallo.byte
let name = input("Wie heisst du? ")
say f"Hallo, {name}! Willkommen bei Byte v5.0"
```

## Interaktiver REPL

```
byte> let x = 42
byte> say f"Das Doppelte von {x} ist {x * 2}"
Das Doppelte von 42 ist 84
byte> help        -- Beispiele anzeigen
byte> exit        -- Beenden
```

## Prinzipien der Sprache

- Lesbar wie Englisch: if x is greater than 10: statt if (x > 10)
- Klares Scoping: let erstellt, set aktualisiert
- Kein Semikolon, keine geschweiften Klammern
- Einrückung definiert Blöcke (wie Python)
- Umlaute unterstützt: let größe = 180, say f"Größe: {größe}"
- Einzige .exe-Datei - keine weitere Installation nötig

# Variablen

Byte trennt strikt zwischen Erstellen und Aktualisieren von Variablen.

## let — Variable erstellen

let erstellt eine neue Variable im aktuellen Block. Die Variable ist nur in diesem Block und seinen Kinderblöcken sichtbar.

```
let x = 42
let name = "Welt"
let aktiv = true
let liste = [1, 2, 3]
let straße = "Hauptstraße"
let größe = 180
```

## set — Variable aktualisieren

set sucht die Variable in der Scope-Kette nach oben und aktualisiert sie. Wenn die Variable nicht mit let erstellt wurde, gibt es einen Fehler.

```
let zaehler = 0
set zaehler = zaehler + 1    -- zaehler ist jetzt 1

set unbekannt = 5            -- FEHLER: unbekannt wurde nie mit let erstellt
```

## Scoping-Regel

let in einem inneren Block erstellt immer eine neue lokale Variable. Die äußere Variable bleibt unverändert. Für äußere Variablen muss set verwendet werden.

```
let x = 0

repeat 5 times:
    let x = 99    -- lokale x, ändert die äußere NICHT

say x             -- gibt 0 aus

-- Äußere Variable ändern: set verwenden
repeat 5 times:
    set x = x + 1
say x             -- gibt 5 aus
```

# Datentypen

| Typ | Beispiel |
|---|---|
| `number` | 42 \| 3.14 \| 0-7 (negativ) |
| `string` | "Hallo" \| "Byte ist toll" |
| `bool` | true \| false |
| `none` | Kein Wert (wie null/nil) |
| `list` | [1, 2, 3] \| ["a", "b", "c"] |
| `dict` | {"name": "Byte", "version": 5} |
| `tuple` | (1, 2, 3) -- unveränderlich |
| `function` | define ... -- Funktion als Wert |
| `class` | class ... -- Klasse als Wert |

## Typ-Konvertierung

```
say string(42)          -- "42"
say number("3.14")      -- 3.14
say bool(0)             -- false
say bool(1)             -- true
say type(42)            -- "number"
say type([1,2])         -- "list"
say f"Typ von 42: {type(42)}"
```

## f-Strings — Ausdrücke direkt im Text

Mit f"..." können Variablen und Ausdrücke direkt in einen String eingebettet werden. Alles zwischen { und } wird als Byte-Ausdruck ausgewertet.

```
let name = "Max"
let alter = 25
say f"Hallo {name}, du bist {alter} Jahre alt!"
-- Ausgabe: Hallo Max, du bist 25 Jahre alt!

-- Rechnung direkt im String:
let preis = 9.99
let menge = 3
say f"Gesamt: {preis * menge} Euro"
-- Ausgabe: Gesamt: 29.97 Euro

-- Methodenaufruf im String:
let wörter = ["Apfel", "Birne"]
say f"Liste hat {wörter.length()} Einträge"

-- none wird als "none" ausgegeben:
let x = none
say f"Wert: {x}"   -- Wert: none
```

# Operatoren & Vergleiche

## Mathematische Operatoren

| Operator | Bedeutung & Beispiel |
|---|---|
| `+` | Addition oder String-Verkettung: 3+4 = 7 \| "a"+"b" = "ab" |
| `-` | Subtraktion: 10-3 = 7 |
| `*` | Multiplikation: 4*5 = 20 |
| `/` | Division: 10/4 = 2.5 (ergibt immer Dezimalzahl) |
| `mod` | Modulo (Rest): 17 mod 5 = 2 |
| `%` | Modulo (alternativ): 17 % 5 = 2 |

## Vergleichsoperatoren

| Operator | Bedeutung & Beispiel |
|---|---|
| `is` | Gleichheit: x is 5 |
| `is not` | Ungleichheit: x is not 5 |
| `is greater than` | Größer als: x is greater than 3 |
| `is less than` | Kleiner als: x is less than 10 |
| `is greater than or equal to` | Größer oder gleich: x is greater than or equal to 5 |
| `is less than or equal to` | Kleiner oder gleich: x is less than or equal to 5 |

## Logische Operatoren & sonstige

| Operator | Bedeutung & Beispiel |
|---|---|
| `and` | Logisches UND: true and true |
| `or` | Logisches ODER: false or true |
| `not` | Logisches NICHT: not false |
| `in` | Element-Prüfung: 3 in [1,2,3] \| "ll" in "hello" |
| `->` | Pfeil für map/filter (alte Syntax): map l with n -> n*2 |

## Beispiele

```
say 5 is 5                          -- true
say 5 is not 6                      -- true
say 10 is greater than 5            -- true
say true and false                  -- false
say true or false                   -- true
say not false                       -- true
say 3 in [1, 2, 3]                  -- true
say 9 in [1, 2, 3]                  -- false
say "ll" in "hello"                 -- true
```

# Kontrollfluss

## Bedingungen: if / elif / else

```
let punkte = 85

if punkte is greater than 90:
    say f"Note A ({punkte} Punkte)"
elif punkte is greater than 70:
    say f"Note B ({punkte} Punkte)"
elif punkte is greater than 50:
    say f"Note C ({punkte} Punkte)"
else:
    say f"Durchgefallen ({punkte} Punkte)"

-- Inline-Syntax (Einzeiler):
if punkte is greater than 90: say "Sehr gut!"

-- Kombinierte Bedingungen:
if alter is greater than or equal to 18 and land is "DE":
    say "Volljährig in Deutschland"
```

## Schleifen

### while — solange Bedingung true

```
let i = 0
while i is less than 5:
    say f"Schritt {i}"
    set i = i + 1

-- Inline-Syntax:
while i is less than 10: set i = i + 1
```

### repeat — genau N mal

```
repeat 5 times:
    say "Hallo!"

-- _i enthält den aktuellen Index (0-basiert):
repeat 3 times:
    say f"Durchlauf {_i}"  -- 0, 1, 2
```

# For-Schleifen

## for — über Liste, Bereich, String

```
-- Über Liste:
let früchte = ["Apfel", "Birne", "Pflaume"]
for frucht in früchte:
    say f"Ich esse {frucht}"

-- Über Bereich mit range():
for i in range(5):          -- 0, 1, 2, 3, 4
    say f"{i} * {i} = {i * i}"

for i in range(1, 6):       -- 1, 2, 3, 4, 5
    say i

for i in range(0, 10, 2):   -- 0, 2, 4, 6, 8 (Schrittweite 2)
    say i

for i in range(5, 0, -1):   -- 5, 4, 3, 2, 1 (rückwärts)
    say i

-- Zeichenweise über String:
for zeichen in "Hallo":
    say zeichen

-- Inline-Syntax:
let s = 0
for i in range(5): set s = s + i
say s   -- 10
```

## break und continue

```
for i in range(10):
    if i is 5: break           -- Schleife sofort beenden
    if i mod 2 is 0: continue  -- Rest überspringen
    say f"Ungerade: {i}"       -- gibt 1, 3 aus

-- break/continue funktionieren auch in while und repeat:
let i = 0
while true:
    if i is greater than or equal to 3: break
    set i = i + 1
say i   -- 3
```

## Sondervariable _i in repeat

```
-- _i ist automatisch verfügbar in repeat-Schleifen:
repeat 4 times:
    say f"Index: {_i}"   -- 0, 1, 2, 3

-- Nützlich für Animationen und Canvas:
canvas_new(20, 1)
canvas_fill(".")
repeat 20 times:
    canvas_set(_i, 0, "#")
canvas_draw()
```

# Funktionen

## Definieren und aufrufen

```
define begrüßen with name, uhrzeit:
    if uhrzeit is less than 12:
        return f"Guten Morgen, {name}!"
    elif uhrzeit is less than 18:
        return f"Guten Tag, {name}!"
    else:
        return f"Guten Abend, {name}!"

say begrüßen("Max", 9)
say begrüßen("Anna", 14)
```

## Ohne Parameter

```
define zufallszahl:
    return random(1, 100)

say zufallszahl()
```

## Rekursion

```
define fakultaet with n:
    if n is less than or equal to 1:
        return 1
    return n * fakultaet(n - 1)

for i in range(1, 8):
    say f"{i}! = {fakultaet(i)}"
-- 1! = 1
-- 2! = 2
-- ...
-- 7! = 5040

define fibonacci with n:
    if n is less than or equal to 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

say fibonacci(10)   -- 55
```

# Closures & Höherwertige Funktionen

Funktionen in Byte können andere Funktionen zurückgeben und haben Zugriff auf den Scope, in dem sie definiert wurden.

## Closure-Beispiel

```
define erstelle_multiplikator with faktor:
    define multipliziere with x:
        return x * faktor       -- faktor aus äußerem Scope
    return multipliziere

let verdoppeln  = erstelle_multiplikator(2)
let verdreifachen = erstelle_multiplikator(3)

for i in [1, 2, 3, 4, 5]:
    say f"{i} x2={verdoppeln(i)}  x3={verdreifachen(i)}"

-- Direkt verkettet aufrufen:
say erstelle_multiplikator(5)(7)   -- 35
```

## Zähler mit Closure

```
define erstelle_zaehler with start:
    let n = start
    define nächster:
        set n = n + 1
        return n
    return nächster

let zaehler = erstelle_zaehler(0)
say zaehler()   -- 1
say zaehler()   -- 2
say zaehler()   -- 3
```

## Funktionen als Parameter

```
define anwenden with funktion, wert:
    return funktion(wert)

define verdoppeln with x:
    return x * 2

say anwenden(verdoppeln, 7)    -- 14

-- Mit map kombinieren:
let zahlen = [1, 2, 3, 4, 5]
let doppelt = map zahlen where n becomes verdoppeln(n)
say doppelt.join(", ")         -- 2, 4, 6, 8, 10
```

# Listen

## Erstellen & Zugreifen

```
let zahlen = [1, 2, 3, 4, 5]
let städte = ["Berlin", "Hamburg", "München"]

say f"Erste Stadt: {städte[0]}"
say f"Letzte Stadt: {städte[-1]}"
say f"Anzahl: {städte.length()}"

-- Verschachtelte Listen:
let matrix = [[1,2,3],[4,5,6],[7,8,9]]
say matrix[1][2]   -- 6
```

## Alle Listen-Methoden

| Methode | Beschreibung | Beispiel |
|---|---|---|
| `push(x)` | Element hinzufügen | l.push(4) |
| `pop()` | Letztes Element entfernen & zurückgeben | let x = l.pop() |
| `first()` | Erstes Element (ohne entfernen) | l.first() |
| `last()` | Letztes Element (ohne entfernen) | l.last() |
| `sort()` | Sortierte Kopie | l.sort() |
| `reverse()` | Umgekehrte Kopie | l.reverse() |
| `slice(a, b)` | Teilliste von a bis b | l.slice(1, 3) |
| `extend(andere)` | Andere Liste anhängen | l.extend([4,5]) |
| `join(trenner)` | Liste zu String verbinden | l.join(", ") |
| `length()` | Anzahl Elemente | l.length() |
| `contains(x)` | Enthält x? true/false | l.contains(3) |
| `index_of(x)` | Position von x (-1 wenn nicht da) | l.index_of(3) |
| `count(x)` | Wie oft kommt x vor? | l.count(1) |
| `remove(x)` | Element x entfernen | l.remove(2) |
| `clear()` | Liste leeren | l.clear() |

# Strings

## Grundlagen

```
let s = "Hallo, Welt!"
say f"Erstes Zeichen: {s[0]}"     -- H
say f"Letztes Zeichen: {s[-1]}"   -- !
say f"Länge: {s.length()}"        -- 13

-- Verkettung:
let vorname = "Max"
let nachname = "Mustermann"
say vorname + " " + nachname

-- Umlaute funktionieren vollständig:
let straße = "Schöne Straße"
say straße.upper()
```

## Alle String-Methoden

| Methode | Beschreibung | Beispiel |
|---|---|---|
| `upper()` | Großbuchstaben | "hello".upper() -> "HELLO" |
| `lower()` | Kleinbuchstaben | "HELLO".lower() -> "hello" |
| `length()` | Anzahl Zeichen | "Hallo".length() -> 5 |
| `trim()` | Leerzeichen am Rand entfernen | "  hi  ".trim() -> "hi" |
| `contains(x)` | Enthält Teilstring x? | "hello".contains("ll") |
| `replace(a, b)` | Alle a durch b ersetzen | "hello".replace("l","r") |
| `split(trenner)` | An Trennzeichen aufteilen | "a,b".split(",") |
| `starts_with(x)` | Beginnt mit x? | "hello".starts_with("hel") |
| `ends_with(x)` | Endet mit x? | "hello".ends_with("llo") |
| `repeat(n)` | String n-mal wiederholen | "ab".repeat(3) -> "ababab" |
| `slice(a, b)` | Teilstring von Index a bis b | "hello".slice(1,4) -> "ell" |
| `index_of(x)` | Position von x (-1 wenn nicht da) | "hello".index_of("ll") -> 2 |
| `count(x)` | Wie oft kommt x vor? | "hello".count("l") -> 2 |

# Dictionaries

## Erstellen & Zugreifen

```
let person = {"name": "Alice", "alter": 30, "stadt": "Berlin"}

say f"{person["name"]} ist {person["alter"]} Jahre alt"
say f"Wohnt in: {person["stadt"]}"

-- Dict aktualisieren:
person.put("beruf", "Entwicklerin")
say person["beruf"]   -- Entwicklerin
```

## Alle Dict-Methoden

| Methode | Beschreibung | Beispiel |
|---|---|---|
| `put(k, v)` | Schlüssel k auf Wert v setzen | d.put("b", 2) |
| `get(k)` | Wert für k (none wenn nicht da) | d.get("b") |
| `get(k, standard)` | Wert für k oder Standard-Wert | d.get("z", "def") |
| `has(k)` | Enthält Schlüssel k? | d.has("a") |
| `remove(k)` | Schlüssel löschen | d.remove("a") |
| `keys()` | Liste aller Schlüssel | d.keys() |
| `values()` | Liste aller Werte | d.values() |
| `items()` | Liste von [schlüssel, wert]-Paaren | d.items() |
| `length()` | Anzahl Einträge | d.length() |
| `clear()` | Alle Einträge löschen | d.clear() |

## Praktisches Beispiel

```
let produkt = {"name": "Laptop", "preis": 999.99, "lager": 15}
produkt.put("rabatt", 0.1)

let endpreis = produkt["preis"] * (1 - produkt["rabatt"])
say f"{produkt["name"]}: {endpreis} Euro"
say f"Schlüssel: {produkt.keys().join(", ")}"
say f"Lager: {produkt.get("lager", 0)} Stück"
```

# Map, Filter & Tupel

## map — Liste transformieren

Byte bietet eine besonders lesbare Syntax für Listenverarbeitung.

```
let zahlen = [1, 2, 3, 4, 5]

-- Neue Syntax (empfohlen):
let quadrate = map zahlen where n becomes n * n
say quadrate.join(", ")   -- 1, 4, 9, 16, 25

-- Strings transformieren:
let namen = ["alice", "bob", "charlie"]
let gross = map namen where n becomes n.upper()
say gross.join(", ")      -- ALICE, BOB, CHARLIE

-- Alte Syntax (funktioniert noch):
let doppelt = map zahlen with n -> n * 2
```

## filter — Liste filtern

```
let zahlen = range(1, 21)   -- 1 bis 20

let gerade = filter zahlen where n becomes n mod 2 is 0
say gerade.join(", ")       -- 2, 4, 6, 8, 10, 12, 14, 16, 18, 20

let gross = filter zahlen where n becomes n is greater than 10
say gross.join(", ")        -- 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

-- Kombinieren (filter dann map):
let gerade_quadrate = map (filter zahlen where n becomes n mod 2 is 0)
                          where n becomes n * n
say gerade_quadrate.join(", ")   -- 4, 16, 36, 64, 100, ...
```

## Tupel — unveränderliche Listen

Tupel sind unveränderliche Listen. Einmal erstellt, können keine Elemente hinzugefügt oder entfernt werden.

```
let punkt = (10, 20)
let rgb   = (255, 128, 0)

say punkt[0]       -- 10
say punkt[1]       -- 20
say len(punkt)     -- 2

-- Tupel als Rückgabewert von enumerate und zip:
for paar in enumerate(["a", "b", "c"]):
    say f"Index {paar[0]}: {paar[1]}"
```

# Klassen & Objekte

## Klasse definieren

```
class Person with name, alter:
    define vorstellen:
        return f"Ich bin {self.name}, {self.alter} Jahre alt"
    define geburtstag:
        set self.alter = self.alter + 1
        return f"{self.name} ist jetzt {self.alter}!"

let p = new Person("Max", 30)
say p.vorstellen()
say p.geburtstag()
say p.vorstellen()
```

## Vererbung mit extends

```
class Fahrzeug with marke, km:
    define info:
        return f"{self.marke} ({self.km} km)"
    define fahre with strecke:
        set self.km = self.km + strecke
        return f"{self.marke} fährt {strecke} km"

class Elektroauto extends Fahrzeug:
    define info:
        return f"E-Auto: {self.marke} ({self.km} km)"

let e = new Elektroauto("Tesla", 0)
say e.info()
say e.fahre(250)
say e.info()
```

Hinweis: In Methoden immer set self.feld = wert verwenden (nicht let) um Felder des Objekts zu setzen.

# Fehlerbehandlung

## try / catch / throw

```
define dividieren with a, b:
    if b is 0:
        throw f"Division durch Null: {a} / {b}"
    return a / b

let paare = [[10, 2], [15, 3], [8, 0], [20, 4]]

for paar in paare:
    try:
        let ergebnis = dividieren(paar[0], paar[1])
        say f"{paar[0]} / {paar[1]} = {ergebnis}"
    catch fehler:
        say f"Fehler: {fehler}"
```

## assert — Bedingungen prüfen

```
assert(x is greater than 0, "x muss positiv sein")
assert(liste.length() is greater than 0, "Liste ist leer")

-- Bei Fehler wird eine Exception geworfen, die catchbar ist:
try:
    assert(false, "Dieser Fehler wird gefangen")
catch e:
    say f"Gefangen: {e}"
```

## Alle catchbaren Fehler

| Fehlerquelle | Beispiel |
|---|---|
| `Division durch Null` | 10 / 0 |
| `Modulo durch Null` | 10 mod 0 |
| `Undefinierte Variable` | set unbekannt = 5 |
| `Falsche Argumente` | f(1) wenn f zwei Parameter erwartet |
| `Index außerhalb` | liste[99] bei 3 Elementen |
| `None in Arithmetik` | 10 / none |
| `throw` | throw "Eigene Fehlermeldung" |
| `assert` | assert(false, "Nachricht") |
| `Stack Overflow` | Unendliche Rekursion (> 400 Ebenen) |
| `Listen zu gross` | Liste mit mehr als 1.000.000 Elementen |
| `HTTP-Fehler` | Verbindung fehlgeschlagen, DNS-Fehler |

# Eingebaute Funktionen

## Mathematik

| Funktion | Beschreibung | Beispiel |
|---|---|---|
| `abs(n)` | Absolutwert | abs(-5) -> 5 |
| `round(n, p)` | Runden auf p Stellen | round(3.14159, 2) -> 3.14 |
| `floor(n)` | Abrunden | floor(3.9) -> 3 |
| `ceil(n)` | Aufrunden | ceil(3.1) -> 4 |
| `sqrt(n)` | Quadratwurzel | sqrt(16) -> 4 |
| `pow(b, e)` | Potenz | pow(2, 8) -> 256 |

## Listen & Sequenzen

| Funktion | Beschreibung | Beispiel |
|---|---|---|
| `len(x)` | Länge von Liste, String, Dict | len([1,2,3]) -> 3 |
| `sum(liste)` | Summe aller Zahlen | sum([1,2,3,4,5]) -> 15 |
| `max(liste)` | Größtes Element | max([3,1,9]) -> 9 |
| `min(liste)` | Kleinstes Element | min([3,1,9]) -> 1 |
| `range(n)` | [0, 1, ..., n-1] | range(5) -> [0,1,2,3,4] |
| `range(a, b)` | [a, a+1, ..., b-1] | range(2,5) -> [2,3,4] |
| `range(a, b, step)` | Mit Schrittweite | range(0,9,3) -> [0,3,6] |
| `sorted(liste)` | Sortierte Kopie | sorted([3,1,2]) -> [1,2,3] |
| `reversed(liste)` | Umgekehrte Kopie | reversed([1,2,3]) -> [3,2,1] |
| `any(liste)` | true wenn mind. ein Element truthy | any([false,true]) -> true |
| `all(liste)` | true wenn alle Elemente truthy | all([true,true]) -> true |
| `enumerate(liste)` | [(0,a), (1,b), ...] Paare | enumerate(["a","b"]) |
| `zip(a, b)` | [(a0,b0), (a1,b1), ...] Paare | zip([1,2],["x","y"]) |
| `reduce(l, fn, start)` | Liste zu Einzelwert falten | reduce([1,2,3], add, 0) |

# Eingebaute Funktionen (Fortsetzung)

## Typ & Konvertierung

| Funktion | Beschreibung | Beispiel |
|---|---|---|
| `string(x)` | Zu String konvertieren | string(42) -> "42" |
| `number(x)` | Zu Zahl konvertieren | number("3.14") -> 3.14 |
| `bool(x)` | Zu bool konvertieren | bool(0) -> false |
| `type(x)` | Typ als String | type(42) -> "number" |
| `chr(n)` | ASCII -> Zeichen | chr(65) -> "A" |
| `ord(z)` | Zeichen -> ASCII | ord("A") -> 65 |

## Zufall & Auswahl

| Funktion | Beschreibung | Beispiel |
|---|---|---|
| `random()` | Zufallszahl 0.0 bis 1.0 | random() |
| `random(n)` | Zufallszahl 0 bis n-1 | random(10) |
| `random(a, b)` | Zufallszahl a bis b | random(1, 6)  -- Würfel |
| `choice(liste)` | Zufälliges Element | choice(["rock","paper","scissors"]) |

## Ein- & Ausgabe, Sonstiges

| Funktion | Beschreibung | Beispiel |
|---|---|---|
| `say ausdruck` | Ausgabe mit Zeilenumbruch | say f"Hallo {name}!" |
| `print(a, b, ...)` | Mehrere Werte ausgeben | print("x =", x, "y =", y) |
| `input("Prompt")` | Benutzereingabe lesen | input("Name: ") |
| `sleep(ms)` | Pause in Millisekunden | sleep(1000)  -- 1 Sekunde |
| `exit(n)` | Programm beenden | exit(0) |
| `assert(b, msg)` | Fehler wenn Bedingung false | assert(x > 0, "positiv") |

# Datei-Operationen

| Funktion | Beschreibung | Beispiel |
|---|---|---|
| `file_write(pfad, text)` | Datei schreiben (überschreibt) | file_write("log.txt", inhalt) |
| `file_read(pfad)` | Datei als String lesen | let s = file_read("log.txt") |
| `file_lines(pfad)` | Datei als Zeilen-Liste | let z = file_lines("log.txt") |
| `file_append(pfad, text)` | Text an Datei anhängen | file_append("log.txt", "mehr") |
| `file_exists(pfad)` | Datei vorhanden? true/false | file_exists("config.txt") |
| `file_delete(pfad)` | Datei löschen | file_delete("temp.txt") |
| `file_copy(quelle, ziel)` | Datei kopieren | file_copy("a.txt","b.txt") |
| `file_size(pfad)` | Größe in Bytes | file_size("bild.png") |
| `path_join("a","b","c")` | Pfade verbinden | path_join("C:\\","Max","f.txt") |

## Praxisbeispiel: Protokoll-Datei

```
let logdatei = "protokoll.txt"

define log with nachricht:
    file_append(logdatei, f"{nachricht}\n")

file_write(logdatei, "=== Programm gestartet ===\n")
log("Verarbeitung läuft...")
log("Fertig!")

let zeilen = file_lines(logdatei)
say f"Protokoll hat {zeilen.length()} Zeilen:"
for zeile in zeilen:
    say f"  {zeile}"

say f"Dateigröße: {file_size(logdatei)} Bytes"
file_delete(logdatei)
say f"Datei gelöscht: {not file_exists(logdatei)}"
```

# HTTP & Netzwerk

Byte kann HTTP-Anfragen senden ohne externe Bibliotheken. Windows nutzt WinInet (eingebaut), Linux/Mac nutzt POSIX-Sockets.

| Funktion | Beschreibung |
|---|---|
| `http_get(url)` | GET-Anfrage -> Antworttext als String |
| `http_post(url, body)` | POST-Anfrage -> Antworttext als String |
| `http_status(url)` | Nur den HTTP-Statuscode zurückgeben (200, 404 ...) |
| `http_request(methode, url, body)` | Gibt Dictionary {status, body} zurück |

## Beispiele

```
-- Einfacher GET-Request:
try:
    let html = http_get("http://example.com")
    say f"Länge: {html.length()} Zeichen"
    say html.contains("Example")
catch fehler:
    say f"Netzwerkfehler: {fehler}"

-- HTTP-Statuscode prüfen:
try:
    let code = http_status("http://example.com")
    say f"Status: {code}"   -- 200
catch fehler:
    say f"Fehler: {fehler}"

-- Vollständige Anfrage mit Status und Body:
try:
    let resp = http_request("GET", "http://example.com", "")
    say f"Status: {resp[\"status\"]}"
    say f"Body-Länge: {resp[\"body\"].length()}"
catch fehler:
    say f"Fehler: {fehler}"

-- POST-Request (z.B. JSON senden):
try:
    let antwort = http_post("http://api.example.com/daten",
        '{"name": "Byte", "version": 5}')
    say antwort
catch fehler:
    say f"POST-Fehler: {fehler}"
```

Wichtig: Immer try/catch verwenden! Netzwerkfehler (kein Internet, DNS, Timeout) werden als catchbare Fehler geworfen.

# Module & Imports

Byte-Dateien können als Module importiert werden, um Code wiederzuverwenden.

## Modul erstellen (mathe.byte)

```
-- mathe.byte
define quadrat with x:
    return x * x

define kreis_flaeche with r:
    return 3.14159 * r * r

let PI = 3.14159
let VERSION = "1.0"
```

## Modul importieren

```
-- Namen direkt in aktuellen Scope einbinden:
import "mathe.byte"
say f"Quadrat von 5: {quadrat(5)}"
say f"PI = {PI}"

-- Mit Namensraum (empfohlen für größere Projekte):
import "mathe.byte" as mathe
say f"Quadrat von 4: {mathe.quadrat(4)}"
say f"Kreisfläche r=5: {mathe.kreis_flaeche(5)}"
say f"Modul Version: {mathe.VERSION}"
```

## Tipps für Module

- Namensräume (as name) verwenden um Konflikte zu vermeiden
- Module können selbst andere Module importieren
- Pfad kann relativ oder absolut sein: import "../utils/tools.byte"
- Bis zu 64 Module gleichzeitig ladbar

# Canvas & Terminal-Grafik

Byte hat ein eingebautes ASCII-Canvas-System für Terminal-Grafiken.

## Canvas-Funktionen

| Funktion | Beschreibung | Beispiel |
|---|---|---|
| `canvas_new(b, h)` | Neues Canvas erstellen | canvas_new(40, 15) |
| `canvas_fill(zeichen)` | Canvas komplett füllen | canvas_fill(" ") |
| `canvas_set(x, y, zeichen)` | Einzelne Zelle setzen | canvas_set(5, 3, "#") |
| `canvas_text(x, y, text)` | Text an Position schreiben | canvas_text(2, 1, "Hi!") |
| `canvas_line(x1,y1,x2,y2,z)` | Linie zeichnen (Bresenham) | canvas_line(0,0,10,5,"*") |
| `canvas_rect(x,y,b,h,z)` | Rechteck zeichnen | canvas_rect(0,0,20,10,"#") |
| `canvas_draw()` | Canvas ausgeben (normal) | canvas_draw() |
| `canvas_draw(true)` | Canvas ausgeben (ANSI-positioniert) | canvas_draw(true) |

## Terminal-Steuerung

| Funktion | Beschreibung | Beispiel |
|---|---|---|
| `screen_clear()` | Terminal leeren | screen_clear() |
| `screen_color(n)` | Farbe setzen (0-7) | screen_color(2)  -- grün |
| `screen_reset()` | Farbe zurücksetzen | screen_reset() |
| `screen_print_at(x,y,txt)` | Text an Terminal-Position | screen_print_at(5, 3, "Hi") |
| `screen_width()` | Terminalbreite in Zeichen | screen_width() |
| `screen_height()` | Terminalhöhe in Zeilen | screen_height() |
| `sleep(ms)` | Pause in Millisekunden | sleep(16)  -- ~60fps |
| `input_key()` | Einzelnen Tastendruck lesen | let k = input_key() |

## Farbnummern für screen_color()

| Nummer | Farbe |
|---|---|
| 0 | Schwarz |
| 1 | Rot |
| 2 | Grün |
| 3 | Gelb |
| 4 | Blau |
| 5 | Lila / Magenta |
| 6 | Cyan |
| 7 | Weiß |

# Sicherheit & Schutz

Byte v5 ist mit mehreren Schutzmechanismen ausgestattet. Alle Fehler sind catchbar.

## Übersicht aller Schutzmechanismen

| Schutzmechanismus | Was passiert |
|---|---|
| Buffer Overflow | safe_malloc / safe_realloc prüfen auf NULL. Absturz wird zur Fehlermeldung. |
| Stack Overflow | Rekursionstiefe begrenzt auf 400 Ebenen. Darüberhinaus wird ein catchbarer Fehler geworfen. |
| Null-Safety | none in Arithmetik wirft Fehler. Methodenaufruf auf none wirft Fehler. Beide catchbar. |
| Memory Limits | Listen: max. 1.000.000 Elemente. range() mit zu grossem Argument wirft Fehler. |
| Division durch Null | 10 / 0 und 10 mod 0 werfen catchbare Fehler (kein Absturz). |
| Index außerhalb | liste[99] bei 3 Elementen wirft catchbaren Fehler. |
| HTTP-Fehler | Netzwerkfehler, DNS-Fehler, Verbindungsabbruch: alle catchbar. |
| String.repeat riesig | str.repeat(10000000) wirft Fehler statt OOM-Absturz. |
| Undefinierte Variablen | Lesen oder set auf nicht-definierte Variable wirft catchbaren Fehler. |

## Empfehlungen für sicheren Code

```
-- 1. HTTP immer mit try/catch:
try:
    let html = http_get("http://api.example.com")
catch e:
    say f"Netzwerkfehler: {e}"

-- 2. Dateien prüfen bevor lesen:
if file_exists("config.txt"):
    let config = file_read("config.txt")
else:
    say "config.txt nicht gefunden"

-- 3. Rekursion mit sinnvoller Basis:
define fakultaet with n:
    if n is less than or equal to 1:
        return 1
    return n * fakultaet(n - 1)

-- 4. Divison absichern:
define sicher_div with a, b:
    if b is 0:
        throw f"Division durch 0 nicht erlaubt"
    return a / b
```

# Komplettes Beispielprogramm

Dieses Programm zeigt alle wichtigen Features von Byte v5 auf einen Blick.

```
-- Byte v5.0 -- Vollständiges Beispiel
-- by ByteLaunch -- https://bytelaunch.de

-- 1. Variablen und f-strings
let sprache = "Byte"
let version = 5
say f"Willkommen bei {sprache} v{version}!"

-- 2. Funktion mit Rekursion
define fakultaet with n:
    if n is less than or equal to 1:
        return 1
    return n * fakultaet(n - 1)

for i in range(1, 8):
    say f"{i}! = {fakultaet(i)}"

-- 3. Map & Filter
let zahlen = range(1, 11)
let gerade = filter zahlen where n becomes n mod 2 is 0
let quadrate = map gerade where n becomes n * n
say f"Gerade Quadrate: {quadrate.join(", ")}"

-- 4. Klasse mit Vererbung
class Auto with marke, km:
    define info:
        return f"{self.marke}: {self.km} km gefahren"
    define fahre with strecke:
        set self.km = self.km + strecke
        return f"{self.marke} fährt {strecke} km"

let auto = new Auto("BMW", 0)
say auto.fahre(150)
say auto.fahre(200)
say auto.info()

-- 5. Fehlerbehandlung
define sicher_dividieren with a, b:
    if b is 0:
        throw f"Kann nicht {a} durch 0 teilen!"
    return a / b

for paar in [[10, 2], [15, 0], [8, 4]]:
    try:
        let r = sicher_dividieren(paar[0], paar[1])
        say f"{paar[0]} / {paar[1]} = {r}"
    catch fehler:
        say f"Fehler: {fehler}"

-- 6. Datei schreiben und lesen
file_write("bericht.txt", f"Byte v{version} Test\n")
file_append("bericht.txt", "Alles funktioniert!\n")
let zeilen = file_lines("bericht.txt")
say f"Bericht: {zeilen.length()} Zeilen"
file_delete("bericht.txt")

-- 7. Canvas
canvas_new(30, 3)
canvas_fill("-")
canvas_text(2, 1, "Byte Language v5.0")
canvas_draw()
```

```
say ""
say f"Fertig! Besuche https://bytelaunch.de"
```