ETH zürich

Departement Informatik

Programmieren und Problemlösen

Kontrollstrukturen

Dennis Komm

Frühling 2021 – 11. März 2021

Cäsar-Verschlüsselung

Übung – Cäsar-Verschlüsselung

Schreiben Sie ein Programm, das

- einen gegebenen String durchläuft
- jeden Buchstaben mit einem Schlüssel k entschlüsselt
- dabei jeden Schlüssel k ausprobiert
- dafür folgende Formel verwendet:

$$\mathsf{e} = (\mathsf{v} - 65 - k) \,\%\, 26 + 65$$

Entschlüsseln Sie `TYQZCXLETVTDEVCPLETGPLCMPTE`

Übung – Cäsar-Verschlüsselung

```
for k in range(0, 26):
    for item in ciphertext:
        print(chr((ord(item) - 65 - k) % 26 + 65), end="")
    print()
```

```
for k in range(0, 26):
    for i in range(0, len(ciphertext)):
        print(chr((ord(ciphertext[i]) - 65 - k) % 26 + 65), end="")
    print()
```

Änderung der Schrittweite

Schleifen über Listen – grössere Schrittweiten

Liste mit Schrittweite 2 durchlaufen

```
daten = [5, 1, 4, 3]
for i in range(0, len(daten), 2):
    print(daten[i])
```

Ausgabe

Alle Elemente an geraden Positionen von 0 bis maximal `len(daten)` werden ausgegeben

⇨ `5,4`

Die Syntax von `range`

```
for i in range(start, ende, schritt)
```

Iteration über alle Positionen von `start` bis `ende-1` mit Schrittweite `schritt`

Abkürzung

`for i in range(start,ende)` $\iff$ `for i in range(start,ende,1)`

Noch eine Abkürzung

`for i in range(ende)` $\iff$ `for i in range(0, ende)`

Verbesserung der Cäsar-Verschlüsselung

Verwende alternierend zwei Schlüssel für gerade und ungerade Positionen

```
k = int(input("Erster Schluessel: "))
l = int(input("Zweiter Schluessel: "))
x = input("Text (nur Grossbuchstaben, gerade Laenge): ")
for i in range(0, len(x), 2):
    print(chr((ord(text[i]) - 65 + k) % 26 + 65), end="")
    print(chr((ord(text[i+1]) - 65 + l) % 26 + 65), end="")
print()
```

Dennoch bleibt die Cäsar-Verschlüsselung unsicher ⇨ Projekt 1

Wahrheitswerte

Boolesche Werte und Relationale Operatoren

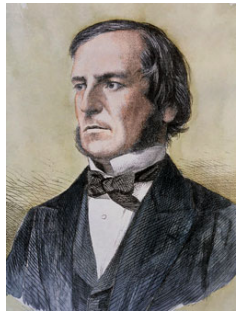
Boolesche Werte und Variablen

Boolesche Ausdrücke können mögliche Werte **F** oder **T** annehmen

- **F** entspricht „falsch“
- **T** entspricht „wahr“

Boolesche Variablen in Python

- repräsentieren „Wahrheitswerte“
- Wertebereich {`False, True`}



George Boole [Wikimedia]

Beispiel

```
b = True   # Variable mit Wert True
```


Relationale Operatoren

`x < y` (kleiner als)
`x >= y` (grösser gleich)
`x == y` (gleich)
`x != y` (ungleich)

Zahlentyp $\times$ Zahlentyp $\rightarrow$ {`False, True`}

Wahrheitswerte

Boolesche Funktionen und Logische Operatoren

Boolesche Funktionen in der Mathematik

Boolesche Funktion

$$f : \{\mathsf{F}, \mathsf{T}\}^2 \to \{\mathsf{F}, \mathsf{T}\}$$

- **F** entspricht „falsch“
- **T** entspricht „wahr“

$a \wedge b$

„Logisches Und“

$$f : \{\mathsf{F}, \mathsf{T}\}^2 \to \{\mathsf{F}, \mathsf{T}\}$$

- **F** entspricht „falsch“
- **T** entspricht „wahr“

a	b	$a \wedge b$
F	F	F
F	T	F
T	F	F
T	T	T

Logischer Operator `and`

`a and b` (logisches Und)

`{False, True}` $\times$ `{False, True}` $\to$ `{False, True}`

```
n = -1
p = 3
c = (n < 0) and (0 < p)  # c = True
```


$a \vee b$

„Logisches Oder“

$$f : \{\mathsf{F}, \mathsf{T}\}^2 \to \{\mathsf{F}, \mathsf{T}\}$$

- **F** entspricht „falsch“
- **T** entspricht „wahr“

a	b	$a \vee b$
F	F	F
F	T	T
T	F	T
T	T	T

Das **logische Oder** ist immer **einschliessend**: a oder b oder beide

Logischer Operator `or`

`a or b` (logisches Oder)

`{False, True}` $\times$ `{False, True}` $\rightarrow$ `{False, True}`

```
n = 1
p = 0
c = (n < 0) or (0 < p)     # c = False
```


$\neg b$

„Logisches Nicht“

$$f : \{\mathsf{F}, \mathsf{T}\} \rightarrow \{\mathsf{F}, \mathsf{T}\}$$

b	$\neg b$
F	T
T	F

- F entspricht „falsch“
- T entspricht „wahr“

Logischer Operator `not`

`not b` (logisches Nicht)

`{False, True}` $\rightarrow$ `{False, True}`

```
n = 1
a = not (n < 0)     # a = True
```


Wahrheitswerte

Präzedenzen

Präzedenzen

```
not b and a
```

⇕

```
(not b) and a
```

```
a and b or c and d
```

⇕

```
(a and b) or (c and d)
```

```
a or b and c or d
```

⇕

```
a or (b and c) or d
```


Präzedenzen

```
b = ((((7 + x) < y) and (y != (3 * z))) or (not b))
```

- Am stärksten binden **binäre arithmetische Operatoren** (Punkt vor Strich)
- Diese binden stärker als **relationale Operatoren**
- Diese binden stärker als der **unäre logische Operator** `not`
- Diese binden stärker als **binäre logische** Operatoren (`and` vor `or`)
- Diese binden stärker als der Zuweisungsoperator
- Oftmals ist es sinnvoll, Klammern zu setzen, die nicht unbedingt nötig sind

DeMorgansche Regeln

- `not (a and b) == (not a or not b)`
- `not (a or b) == (not a and not b)`

Beispiele

- (**nicht** schwarz **und nicht** weiss) == **nicht** (schwarz **oder** weiss)
- **nicht** (reich **und** schön) == (arm **oder** hässlich)

Anwendung – Entweder . . . Oder (XOR)

`(a or b) and not (a and b)`	a oder b, und nicht beide
`(a or b) and (not a or not b)`	a oder b, und eines nicht
`not (not a and not b) and not (a and b)`	nicht keines, und nicht beide
`not ((not a and not b) or (a and b))`	nicht: keines oder beide

Kontrollstrukturen

Kontrollfluss

Bisher...

- **linear** (von oben nach unten)
- `for`-Schleife für **Wiederholung von Blöcken**

```
x = int(input("Eingabe: "))

for i in range(1, x+1):
    print(i*i)
```


Kontrollstrukturen

Auswahlanweisungen

Auswahlanweisungen

Realisieren Verzweigungen

- `if`-Anweisung
- `if-else`-Anweisung
- `if-elif-else`-Anweisung (später)

if-Anweisung

```
if condition:
    statement
```

```
x = int(input("Eingabe: "))
if x % 2 == 0:
    print("gerade")
```

Ist *condition* wahr,
dann wird *statement* ausgeführt

- *statement:*
  - beliebige Anweisung
  - **Rumpf** der `if`-Anweisung
- *condition:* Boolescher Ausdruck

if-else-Anweisung

```
if condition:
    statement1
else:
    statement2
```

```
x = int(input("Eingabe: "))
if x % 2 == 0:
    print("gerade")
else:
    print("ungerade")
```

Ist *condition* wahr,
so wird *statement1* ausgeführt,
andernfalls wird *statement2* ausgeführt

- *condition:* Boolescher Ausdruck
- *statement1:*
  Rumpf des `if`-Zweiges
- *statement2:*
  Rumpf des `else`-Zweiges

Layout

```
x = int(input("Eingabe: "))

if x % 2 == 0:
    print("gerade")      <-- Einrückung
else:
    print("ungerade")    <-- Einrückung
```


if-else-Anweisung

Vorsicht bei == und =

An attempt to make a change in this way is suspicious, to say the least, so there was a lot of interest in what the attempted change was. The actual patch confirmed all suspicious; the relevant code was:

```
+       if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
+                       retval = -EINVAL;
```

It looks much like a standard error check, until you notice that the code is not testing `current->uid` - it is, instead setting it to zero. A program which called `wait4()` with the given flags set would, thereafter, be running as root. This is, in other words, a classic back door.

The resulting vulnerability, had it ever made it to a deployed system, would have been a locally-exploitable hole. Some sites have said that the hole would have been susceptible to remote exploits, but that is not the case. An attacker would need to be able to run a program on the target system first.

Kontrollstrukturen

`while`-Schleifen

`while`-Schleifen

```
while condition:
    statement
```

Einrückung

- *statement:*
  - beliebige Anweisung
  - Rumpf der `while`-Schleife
- *condition:* Boolescher Ausdruck

`while`-Schleifen

```
while condition:
    statement
```

- *condition* wird ausgewertet
  - `True`: Iteration beginnt
    statement wird ausgeführt
  - `False`: `while`-Schleife wird beendet

`while`-Schleifen

```
s = 0
i = 1
while i <= 2:
    s = s + i
    i = i + 1
```

`i`	*condition*	`s`
`i = 1`	wahr	`s = 1`
`i = 2`	wahr	`s = 3`
`i = 3`	falsch	`s = 3`

Inkrementierung von Variablen

Zum Ändern von Werten von Variablen wird vereinfachte Syntax verwendet

- `n = n + 1` wird geschrieben als `n += 1`
- `n = n + i` wird geschrieben als `n += i`
- `n = n - 15` wird geschrieben als `n -= 15`
- `n = n * j` wird geschrieben als `n *= j`
- `n = n ** 4` wird geschrieben als `n **= 4`
- `...`

Die Sprunganweisung `break`

`break`

- Umschliessende Schleife wird sofort beendet
- Nützlich, um Schleife „in der Mitte“ abbrechen zu können

```
s = 0

while True:
    x = int(input("Geben Sie eine positive Zahl ein, Abbruch mit 0: "))
    if x == 0:
        break
    s += x

print(s)
```


Kontrollstrukturen

Terminierung

Terminierung

```
i = 1
while i <= n:
    s += i
    i += 1
```

Hier und meistens

- *statement* ändert einen Wert, der in *condition* vorkommt
- Nach endlich vielen Iterationen wird *condition* falsch

⇨ **Terminierung**

Endlosschleifen

- Endlosschleifen sind leicht zu produzieren

```
while True:
    print("0")
```

```
while not False:
    print("1")
```

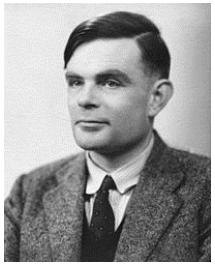
```
while 2 > 1:
    print("2")
```

- . . . aber nicht automatisch zu erkennen

Halteproblem

Unentscheidbarkeit des Halteproblems [Alan Turing, 1936]

- Es gibt kein Python-Programm, das für jedes Python-Programm P und jede Eingabe I korrekt feststellen kann, ob P bei Eingabe I terminiert
- Das heisst, die Terminierung von Programmen kann **nicht** automatisch überprüft werden



Alan Turing [Wikimedia]

Theoretische Fragestellungen dieser Art waren für Turing die Hauptmotivation für die Konstruktion seiner Rechenmaschine

Die Collatz-Folge

Folge von natürlichen Zahlen $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- für jedes $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2, & \text{falls } n_{i-1} \text{ gerade} \\ 3 \cdot n_{i-1} + 1, & \text{falls } n_{i-1} \text{ ungerade} \end{cases}$

Beispiel für $n = 5$

5, 16, 8, 4, 2, 1, 4, 2, 1, . . . (Repetition bei 1)

Übung – Die Collatz-Folge

Schreiben Sie ein Programm, das

- eine ganze Zahl n als Eingabe erhält
- die Collatz-Folge ausgibt mit der Formel $n_0 = n$ und

$$n_i = \begin{cases} n_{i-1}/2, & \text{falls } n_{i-1} \text{ gerade} \\ 3 \cdot n_{i-1} + 1, & \text{falls } n_{i-1} \text{ ungerade} \end{cases}$$

Übung – Die Collatz-Folge

```
n = int(input("Berechne die Collatz-Folge fuer n = "))

while n > 1:          # stopp, wenn 1 erreicht ist
    if n % 2 == 0:    # n ist gerade
        n //= 2
    else:             # n ist ungerade
        n = 3 * n + 1
    print(n, end=" ")
```


Die Collatz-Folge

Beispiel für $n = 27$

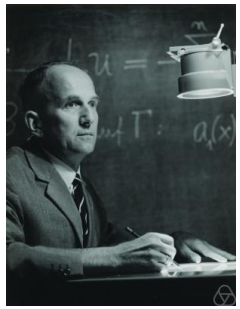
27 82 41 124 62 31 94 47 142 71 214 107 322 161 484 242 121
364 182 91 274 137 412 206 103 310 155 466 233 700 350 175
526 263 790 395 1186 593 1780 890 445 1336 668 334 167 502
251 754 377 1132 566 283 850 425 1276 638 319 958 479 1438
719 2158 1079 3238 1619 4858 2429 7288 3644 1822 911 2734
1367 4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866 433
1300 650 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160
80 40 20 10 5 16 8 4 2 1

Die Collatz-Folge

Die Collatz-Vermutung [Lothar Collatz, 1937]

Für jedes $n \geq 1$ erscheint die 1 in der Folge

- Niemand konnte die Vermutung bislang beweisen
- Falls sie nicht stimmt, ist die `while`-Schleife zur Berechnung der Collatz-Folge für einige n eine Endlosschleife



Lothar Collatz [Wikimedia]