ETH zürich

Departement Informatik

Programming and Problem-Solving

Control Structures

Dennis Komm

Spring 2021 – March 11, 2021

Caesar Encryption

Exercise – Caesar Encryption

Write a program that

- runs through a given string
- decrypts each letter with a key k
- tries out each key k
- uses the following formula

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

Decrypt the ciphertext `TYQZCXLETVTDEVCPLETGPLCMPTE`

Exercise – Caesar Encryption

```
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()
```

Changing the Step Size

Loops over Lists – Larger Steps

Traverse a list with steps of length 2

Loops over Lists – Larger Steps

Traverse a list with steps of length 2

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

Loops over Lists – Larger Steps

Traverse a list with steps of length 2

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

Output

All elements at even positions from 0 up to at most `len(data)` are output

⇨ 5,4

The Syntax of `range`

```
for i in range(start, end, step)
```

The Syntax of `range`

```
for i in range(start, end, step)
```

Iteration over all positions from `start` up to `end-1` with step length of `step`

The Syntax of `range`

```
for i in range(start, end, step)
```

Iteration over all positions from `start` up to `end-1` with step length of `step`

Shorthand notation

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

The Syntax of `range`

```
for i in range(start, end, step)
```

Iteration over all positions from `start` up to `end-1` with step length of `step`

Shorthand notation

`for i in range(start,end)` $\Longleftrightarrow$ `for i in range(start,end,1)`

Another shorthand notation

`for i in range(end)` $\Longleftrightarrow$ `for i in range(0, end)`

Improvement of Caesar Encryption

Use two keys alternatingly for even and odd positions

Improvement of Caesar Encryption

Use two keys alternatingly for even and odd positions

```
k = int(input("First key: "))
l = int(input("Second key: "))
x = input("Text (only uppercase, even length): ")
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()
```

Improvement of Caesar Encryption

Use two keys alternatingly for even and odd positions

```
k = int(input("First key: "))
l = int(input("Second key: "))
x = input("Text (only uppercase, even length): ")
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()
```

Still Caesar encryption remains insecure ⇨ **Project 1**

Logical Values

Boolean Values and Relational Operators

Boolean Values and Variables

Boolean expressions can take on one of two values **F** or **T**

Boolean Values and Variables

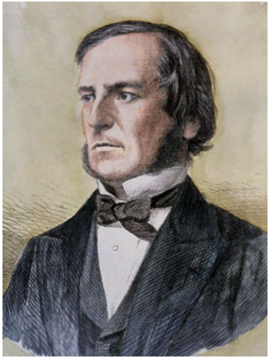
Boolean expressions can take on one of two values **F** or **T**

- **F** corresponds to “false”
- **T** corresponds to “true”

Boolean Values and Variables

Boolean expressions can take on one of two values **F** or **T**

- **F** corresponds to “false”
- **T** corresponds to “true”

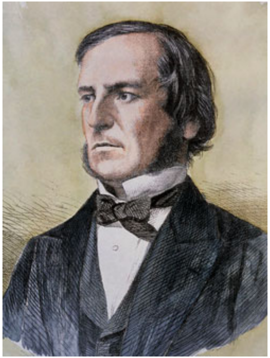


George Boole [Wikimedia]

Boolean Values and Variables

Boolean expressions can take on one of two values **F** or **T**

- **F** corresponds to “false”
- **T** corresponds to “true”



George Boole [Wikimedia]

Boolean variables in Python

- represent “logical values”
- Domain {`False`, `True`}

Boolean Values and Variables

Boolean expressions can take on one of two values **F** or **T**

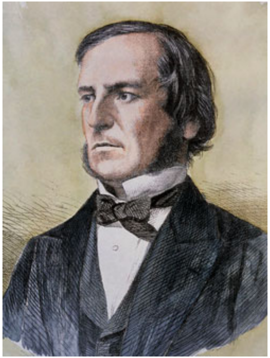
- **F** corresponds to “false”
- **T** corresponds to “true”

Boolean variables in Python

- represent “logical values”
- Domain {`False, True`}

Example

```
b = True   # Variable with value True
```



George Boole [Wikimedia]

Relational Operators

`x < y` (smaller than)

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

Relational Operators

`x < y` (smaller than)

```
b = (1 < 3)          # b =
```

Relational Operators

`x < y` (smaller than)

```
b = (1 < 3)          # b = True
```

Relational Operators

`x >= y` (greater than)

```
x = 0
b = (x >= 3)        # b =
```

Relational Operators

`x >= y` (greater than)

```
x = 0
b = (x >= 3)        # b = False
```

Relational Operators

`x == y` (equals)

```
x = 4
b = (x % 3 == 1)  # b =
```

Relational Operators

`x == y` (equals)

```
x = 4
b = (x % 3 == 1)   # b = True
```

Relational Operators

`x != y` (unequal to)

```
x = 1
b = (x != 2 * x - 1)  # b =
```

Relational Operators

`x != y` (unequal to)

```
x = 1
b = (x != 2 * x - 1)  # b = False
```

Logical Values

Boolean Functions and Logical Operators

Boolean Functions in Mathematics

Boolean function

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

- **F** corresponds to "false"
- **T** corresponds to "true"

$a \wedge b$

“logical and”

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

- F corresponds to “false”
- T corresponds to “true”

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

Logical Operator `and`

`a and b` (logical and)

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

Logical Operator `and`

`a and b` (logical and)

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

Logical Operator `and`

`a and b` (logical and)

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

$a \vee b$

"logical or"

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

- F corresponds to "false"
- T corresponds to "true"

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

$a \lor b$

"logical or"

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

- **F** corresponds to "false"
- **T** corresponds to "true"

a	b	$a \lor b$
F	F	F
F	T	T
T	F	T
T	T	T

The **logical or** is always **inclusive**: a or b or both

Logical Operator `or`

`a or b` (logical or)

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

Logical Operator `or`

`a or b` (logical or)

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

Logical Operator `or`

`a or b` (logical or)

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

$\neg b$

"logical not"

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

- **F** corresponds to "false"
- **T** corresponds to "true"

b	$\neg b$
F	**T**
T	**F**

Logical Operator `not`

`not b` (logical not)

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

Logical Operator `not`

`not b` (logical not)

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

Logical Operator `not`

`not b` (logical not)

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

Logical Values

Precedences

Precedences

```
not b and a
```

Precedences

```
not b and a
```

$$\Updownarrow$$

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

Precedences

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

Precedences

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

$\Updownarrow$

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

Precedences

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

Precedences

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

$$\Updownarrow$$

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

Precedences

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

Precedences

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

- **Binary arithmetic operators** bind the strongest (multiplication and division first, then addition and subtraction)

Precedences

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

- **Binary arithmetic operators** bind the strongest (multiplication and division first, then addition and subtraction)
- These bind stronger than **relational operators** (and first, then or)

Precedences

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

- **Binary arithmetic operators** bind the strongest (multiplication and division first, then addition and subtraction)
- These bind stronger than **relational operators** (and first, then or)
- These bind stronger than the **unary logical operator** `not`

Precedences

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

- **Binary arithmetic operators** bind the strongest (multiplication and division first, then addition and subtraction)
- These bind stronger than **relational operators** (and first, then or)
- These bind stronger than the **unary logical operator** `not`
- These bind stronger than **binary logical** operators (`and` first, then `or`)

Precedences

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

- **Binary arithmetic operators** bind the strongest (multiplication and division first, then addition and subtraction)
- These bind stronger than **relational operators** (and first, then or)
- These bind stronger than the **unary logical operator** `not`
- These bind stronger than **binary logical** operators (`and` first, then `or`)
- These bind stronger than the assignment operator

Precedences

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

- **Binary arithmetic operators** bind the strongest (multiplication and division first, then addition and subtraction)
- These bind stronger than **relational operators** (and first, then or)
- These bind stronger than the **unary logical operator** `not`
- These bind stronger than **binary logical** operators (`and` first, then `or`)
- These bind stronger than the assignment operator
- It is often useful to use parentheses even if redundant

DeMorgan Rules

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

DeMorgan Rules

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

DeMorgan Rules

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

Examples

- (**not** black **and not** white) == **not** (black **or** white)
- **not** (rich **and** beautiful) == (poor **or** ugly)

Application – either . . . or (XOR)

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

Application – either . . . or (XOR)

`(a or b) and not (a and b)` a or b, and not both

Application – either . . . or (XOR)

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

a or b, and not both

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

Application – either . . . or (XOR)

`(a or b) and not (a and b)` a or b, and not both

`(a or b) and (not a or not b)` a or b, and one of them not

Application – either . . . or (XOR)

`(a or b) and not (a and b)` a or b, and not both

`(a or b) and (not a or not b)` a or b, and one of them not

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

Application – either … or (XOR)

`(a or b) and not (a and b)`	a or b, and not both
`(a or b) and (not a or not b)`	a or b, and one of them not
`not (not a and not b) and not (a and b)`	not none and not both

Application – either . . . or (XOR)

`(a or b) and not (a and b)`	a or b, and not both
`(a or b) and (not a or not b)`	a or b, and one of them not
`not (not a and not b) and not (a and b)`	not none and not both
`not ((not a and not b) or (a and b))`	

Application – either . . . or (XOR)

`(a or b) and not (a and b)`	a or b, and not both
`(a or b) and (not a or not b)`	a or b, and one of them not
`not (not a and not b) and not (a and b)`	not none and not both
`not ((not a and not b) or (a and b))`	not: both or none

Control Structures

Control Flow

So far...

- Up to now **linear** (from top to bottom)

Control Flow

So far...

- Up to now **linear** (from top to bottom)
- `for` loop to **repeat blocks**

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

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

Control Structures

Selection Statements

Selection Statements

Implement branches

- `if` statement
- `if-else` statement
- `if-elif-else` statement (later)

`if` Statement

```
if condition:
    statement
```

`if` Statement

```
if condition:
    statement
```

```python
x = int(input("Input: "))
if x % 2 == 0:
    print("even")
```

`if` Statement

```
if condition:
    statement
```

If *condition* is true,
then *statement* is executed

```
x = int(input("Input: "))
if x % 2 == 0:
    print("even")
```

`if` Statement

```
if condition:
    statement
```

```python
x = int(input("Input: "))
if x % 2 == 0:
    print("even")
```

If *condition* is true,
then *statement* is executed

- *statement:*
  - arbitrary statement
  - **body** of the `if`-Statement
- *condition:* Boolean expression

`if-else` Statement

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

`if-else` Statement

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

```python
x = int(input("Input: "))
if x % 2 == 0:
    print("even")
else:
    print("odd")
```

`if-else` Statement

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

If *condition* is true,
then *statement1* is executed,
otherwise *statement2* is executed

```python
x = int(input("Input: "))
if x % 2 == 0:
    print("even")
else:
    print("odd")
```

`if-else` Statement

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

```
x = int(input("Input: "))
if x % 2 == 0:
    print("even")
else:
    print("odd")
```

If *condition* is true,
then *statement1* is executed,
otherwise *statement2* is executed

- *condition:* Boolean expression
- *statement1:*
  body of the `if`-branch
- *statement2:*
  body of the `else`-branch

Layout

```python
x = int(input("Input: "))

if x % 2 == 0:
    print("even")
else:
    print("odd")
```

Layout

```python
x = int(input("Input: "))

if x % 2 == 0:
    print("even")        ← Indentation
else:
    print("odd")         ← Indentation
```

`if-else` Statement

Attention when using == or =

`if-else` Statement

Attention when using == or =

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.

Control Structures

`while` Loops

`while` Loops

```
while condition:
    statement
```

- *statement:*
  - arbitrary statement
  - body of the `while` loop
- *condition:* Boolean expression

`while` Loops

```
while condition:
    statement
```

Indentation

- *statement:*
  - arbitrary statement
  - body of the `while` loop
- *condition:* Boolean expression

`while` Loops

```
while condition:
    statement
```

- *condition* is evaluated
  - `True`: iteration starts
    statement is executed
  - `False`: `while` loop ends

`while` Loops

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

`i`	*condition*	`s`

`while` Loops

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

`i`	*condition*	`s`
`i = 1`		

`while` Loops

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

`i`	*condition*	`s`
`i = 1`	`i <= 2?`	

`while` Loops

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

`i`	*condition*	`s`
`i = 1`	true	

`while` Loops

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

`i`	*condition*	`s`
`i = 1`	true	`s = 1`

`while` Loops

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

`i`	*condition*	`s`
`i = 1`	true	`s = 1`
`i = 2`		

`while` Loops

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

`i`	*condition*	`s`
`i = 1`	true	`s = 1`
`i = 2`	`i <= 2`?	

`while` Loops

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

`i`	*condition*	`s`
`i = 1`	true	`s = 1`
`i = 2`	true	

`while` Loops

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

i	*condition*	s
`i = 1`	true	`s = 1`
`i = 2`	true	`s = 3`

`while` Loops

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

`i`	*condition*	`s`
`i = 1`	true	`s = 1`
`i = 2`	true	`s = 3`
`i = 3`		

`while` Loops

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

`i`	*condition*	`s`
`i = 1`	true	`s = 1`
`i = 2`	true	`s = 3`
`i = 3`	`i <= 2`?	

`while` Loops

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

`i`	*condition*	`s`
`i = 1`	true	`s = 1`
`i = 2`	true	`s = 3`
`i = 3`	false	

`while` Loops

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

i	*condition*	s
i = 1	true	s = 1
i = 2	true	s = 3
i = 3	false	s = 3

Incrementation of Variables

Use simplified syntax for changing values of variables

Incrementation of Variables

Use simplified syntax for changing values of variables

- `n = n + 1` is written as `n += 1`

Incrementation of Variables

Use simplified syntax for changing values of variables

- `n = n + 1` is written as `n += 1`
- `n = n + i` is written as `n += i`

Incrementation of Variables

Use simplified syntax for changing values of variables

- `n = n + 1` is written as `n += 1`
- `n = n + i` is written as `n += i`
- `n = n - 15` is written as `n -= 15`

Incrementation of Variables

Use simplified syntax for changing values of variables

- `n = n + 1` is written as `n += 1`
- `n = n + i` is written as `n += i`
- `n = n - 15` is written as `n -= 15`
- `n = n * j` is written as `n *= j`

Incrementation of Variables

Use simplified syntax for changing values of variables

- `n = n + 1` is written as `n += 1`
- `n = n + i` is written as `n += i`
- `n = n - 15` is written as `n -= 15`
- `n = n * j` is written as `n *= j`
- `n = n ** 4` is written as `n **= 4`

Incrementation of Variables

Use simplified syntax for changing values of variables

- `n = n + 1` is written as `n += 1`
- `n = n + i` is written as `n += i`
- `n = n - 15` is written as `n -= 15`
- `n = n * j` is written as `n *= j`
- `n = n ** 4` is written as `n **= 4`
- ...

The Jump Statements `break`

`break`

- Immediately leave the enclosing loop
- Useful in order to be able to break a loop “in the middle”

The Jump Statements `break`

`break`

- Immediately leave the enclosing loop
- Useful in order to be able to break a loop “in the middle”

```
s = 0

while True:
    x = int(input("Enter a positive number, abort with 0: "))
    if x == 0:
        break
    s += x

print(s)
```

Control Structures

Termination

Termination

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

Termination

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

Here and commonly

- *statement* changes its value that appears in *condition*

Termination

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

Here and commonly

- *statement* changes its value that appears in *condition*
- After a finite number of iterations *condition* becomes false

Termination

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

Here and commonly

- *statement* changes its value that appears in *condition*
- After a finite number of iterations *condition* becomes false

⇨ **Termination**

Infinite Loops

- Infinite loops are easy to generate

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

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

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

Infinite Loops

- Infinite loops are easy to generate

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

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

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

- . . . but can in general not be automatically detected

Halting Problem

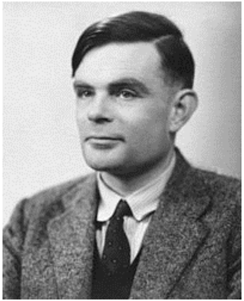
Undecidability of the Halting Problem [Alan Turing, 1936]

- There is no Python program that can determine, for each Python program P and each input I, whether P terminates with the input I
- This means that the termination of programs can in general **not** be automatically checked

Halting Problem

Undecidability of the Halting Problem [Alan Turing, 1936]

- There is no Python program that can determine, for each Python program P and each input I, whether P terminates with the input I
- This means that the termination of programs can in general **not** be automatically checked



Alan Turing [Wikimedia]

Theoretical questions of this kind were the main motivation for Turing to design his computing machine

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

Example for $n = 5$

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

Example for $n = 5$

5

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

Example for $n = 5$

5, 16

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

Example for $n = 5$

5, 16, 8

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

Example for $n = 5$

5, 16, 8, 4

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

Example for $n = 5$

5, 16, 8, 4, 2

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

Example for $n = 5$

5, 16, 8, 4, 2, 1

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

Example for $n = 5$

5, 16, 8, 4, 2, 1, 4

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

Example for $n = 5$

5, 16, 8, 4, 2, 1, 4, 2

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

Example for $n = 5$

5, 16, 8, 4, 2, 1, 4, 2, 1

The Collatz Sequence

Sequence of natural numbers $n_0, n_1, n_2, n_3, n_4, n_5, \ldots$

- $n_0 = n$
- for every $i \geq 1$, $n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$

Example for $n = 5$

5, 16, 8, 4, 2, 1, 4, 2, 1, . . . (repetition at 1)

Exercise – The Collatz Sequence

Write a program that

- takes an integer n as input
- outputs the Collatz sequence using

  $n_0 = n$ and

$$n_i = \begin{cases} n_{i-1}/2 & \text{if } n_{i-1} \text{ even} \\ 3 \cdot n_{i-1} + 1 & \text{if } n_{i-1} \text{ odd} \end{cases}$$

Exercise – The Collatz Sequence

```
n = int(input("Compute the Collatz sequence for n = "))

while n > 1:            # stop when 1 is reached
    if n % 2 == 0:      # n is even
        n //= 2
    else:               # n is odd
        n = 3 * n + 1
    print(n, end=" ")
```

The Collatz Sequence

Example for $n = 27$

```
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
```

The Collatz Sequence

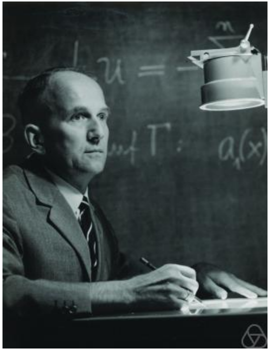
The Collatz Concecture [Lothar Collatz, 1937]

For every $n \geq 1$, 1 will occur in the sequence

The Collatz Sequence

The Collatz Concecture [Lothar Collatz, 1937]

For every $n \geq 1$, 1 will occur in the sequence



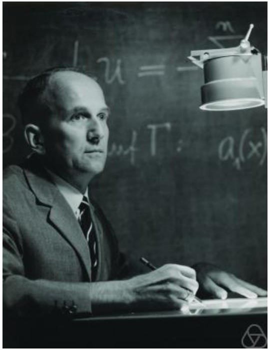
Lothar Collatz [Wikimedia]

The Collatz Sequence

The Collatz Concecture [Lothar Collatz, 1937]

For every $n \geq 1$, 1 will occur in the sequence

- Nobody could prove the conjecture so far

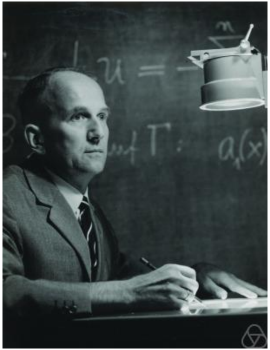


Lothar Collatz [Wikimedia]

The Collatz Sequence

The Collatz Concecture	[Lothar Collatz, 1937]
For every $n \geq 1$, 1 will occur in the sequence	

- Nobody could prove the conjecture so far
- If it is wrong, then the `while` loop for computing the Collatz sequence can be an endless loop for some n as input



Lothar Collatz [Wikimedia]

Thanks for your attention

ETH zürich