HSLU Lucerne University of Applied Sciences and Arts

Python for Data Science: SW03

for-loop
while-loop
enumeration

Information Technology
March 6, 2025

FH Zentralschweiz

Content

- range() object
- List length: len()
- Sequence slicing
  - By index (particular element)
  - By slice object
- for loop
  - Header
  - else condition
- while (-True)
  - Header
  - else condition
- break/continue/pass
- Enumerate

range() object

Information Technology
March 6, 2025

range() object

In data science we encounter often numerical sequences with constant step size. Typical examples are:

- Time stamps or integer indices of data series
- Slicing sequences

The class **range()** allows creating sequence objects with constant step sizes.
Doc: https://docs.python.org/3/library/stdtypes.html#range

- Ranges implement all of the common sequence operations except concatenation and repetition
- it only stores the start, stop and step values
- stop element is not included!

range(start, stop[, step])

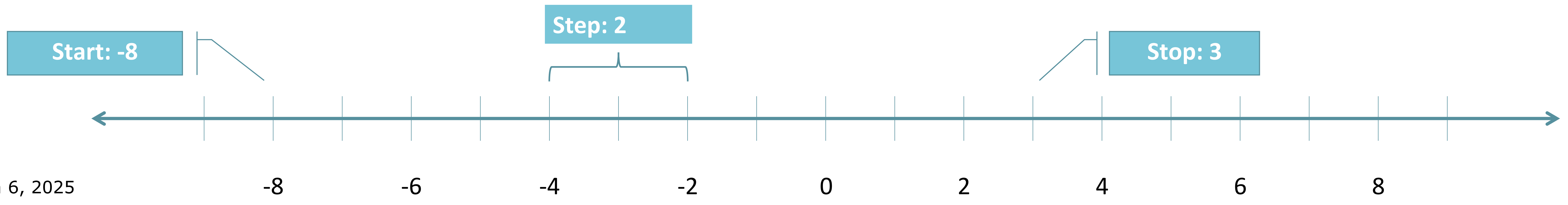
- start: starting index of sequence: default = 0
- stop: stopping index of sequence: stop > start **if** step > 0; stop < start **if** step < 0
- step: step size between elements: default 1

range() object: examples

In Python, **the last element is (almost) never included!**

Range examples:

list(range(7))	[0,1,2,3,4,5,6]
list(range(3,11))	[3,4,5,6,7,8,9,10]
list(range(3,-4))	[]
list(range(-8,3,2))	[-8,-6,-4,-2,0,2]
list(range(3,11,-2))	[]
list(range(3,-4,-2))	[3,1,-1,-3]



List length: len()

Information Technology
March 6, 2025

List length: len()

Sequence objects provide the function len() which returns the number of elements in a sequence.

- Considers only the first dimension of a sequence.
- Returns positive integer: 0 indicates an empty sequence
- Often used for pointing to the last element of a list

```
5 == len([2,5,7,23,44])
4 == len([[2,5,7], “hello”, 1, {‘a’:2, ‘b’:88})
```

2
-5
7
23
44

2	**5**	**7**		
'h'	**'e'**	**'l'**	**'l'**	**'o'**
1				
'a':2	**'b':88**			

Lucerne University of Applied Sciences and Arts

Sequence slicing

Information Technology
March 6, 2025

Sequence slicing

Extracting sub-sequences of larger data containers is an important and often used operation.

Slicing options are:

- By integer (particular element)
- By range or slice object

Examples:

- Calculate moving average of sub-sequence
- Calculate mean temperature for each day
- Get opening price of stock data

li :	2	11	16	19	48	40	30	32	9	39	40	2	14	10	18	9	10	7	43	22	19	36	35
nth:	**2**				**48**				**9**				**14**				**10**				**19**		
idx:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Sequence slicing: by index

Accessing elements by index **only** allows to access **one** particular element.

- There is no way accessing multiple elements by index given in a list:

  ~~li[[7, 11, 20]]~~

  ~~li[7; 11; 20]~~

- Elements can only be accessed separately by **indexing**:

  a = li[7]

  b = li[11]

  c = li[20]

li :	2	11	16	19	48	40	30	32	9	39	40	2	14	10	18	9	10	7	43	22	19	36	35
idx:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Sequence slicing: by slice object

slice() returns a slice object representing the set of indices specified by range(start, stop, step). The start and step arguments default to None.

slice allows accessing multiple elements in forward or backward order either:

- with a constant step size li[slice(3,16,4)]
- in a row. li[slice(21,17,-1)]

Slice objects are also generated when extended indexing syntax is used. For example:

- [start:stop:step] or li[3:16:4]
- [start:stop, i] li[21:17,-1]

li :	**2**	**11**	**16**	**19**	**48**	**40**	**30**	**32**	**9**	**39**	**40**	**2**	**14**	**10**	**18**	**9**	**10**	**7**	**43**	**22**	**19**	**36**	**35**
idx:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Sequence slicing: by slice object

While indexing is allowed to start from beginning or end point, slices can NOT exceed the ends.

Indexes can be positive (forward) from starting point (0) or negative (backward) from end point (len() -1).

-> it holds: start < end and step > 0 or start > end and step < 0

- Positive (forward): **li[3:11] or li[-20:-12]**
- Negative (backward): **li[10:2:-1] or li[-13:-21:-1]**

Slices can NOT exceed the end, but must concatenated with +. Exceeding the ends results in empty slices.

- Irregular access: ~~**li[17:2]**~~ or ~~**li[-6:2]**~~ or ~~**li[1:-6:-1]**~~ or ~~**li[-22:-7:-1]**~~
- Concatenated access: **li[-6:] + li[:1]** or **li[1::-1] + li[:-7:-1]**

  -> open ends [-6:] for indexing last elements

li :	**2**	**11**	**16**	**19**	**48**	**40**	**30**	**32**	**9**	**39**	**40**	**2**	**14**	**10**	**18**	9	**10**	**7**	**43**	**22**	**19**	**36**	**35**
idx:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

for loop

Information Technology
March 6, 2025

for loop: Header

Loops are used to **iterate** over **sequence** objects by providing each element one after the other through a loop variable. The most important type of loops is called a for loop.

for loop:

- Assign each element of the sequence one after another to the **loop variable**
- Execute the body of the loop (indented block after ":")
- The loop ends when the sequence has no **next()** element anymore.

```
x = [3,11,-32,18-5]# assign sequence to variable
for e in x:                          # if has next(), assign next element of sequence x to e
    e = e + 10          # add 10 to current element
    print(e)                         # print current element (i.e. e + 10)

print(x)                             # print sequence [3,11,-32,18,5]
```

for loop: else condition

Similar to the if-else statement, the for loop also provides an **else** condition. This block

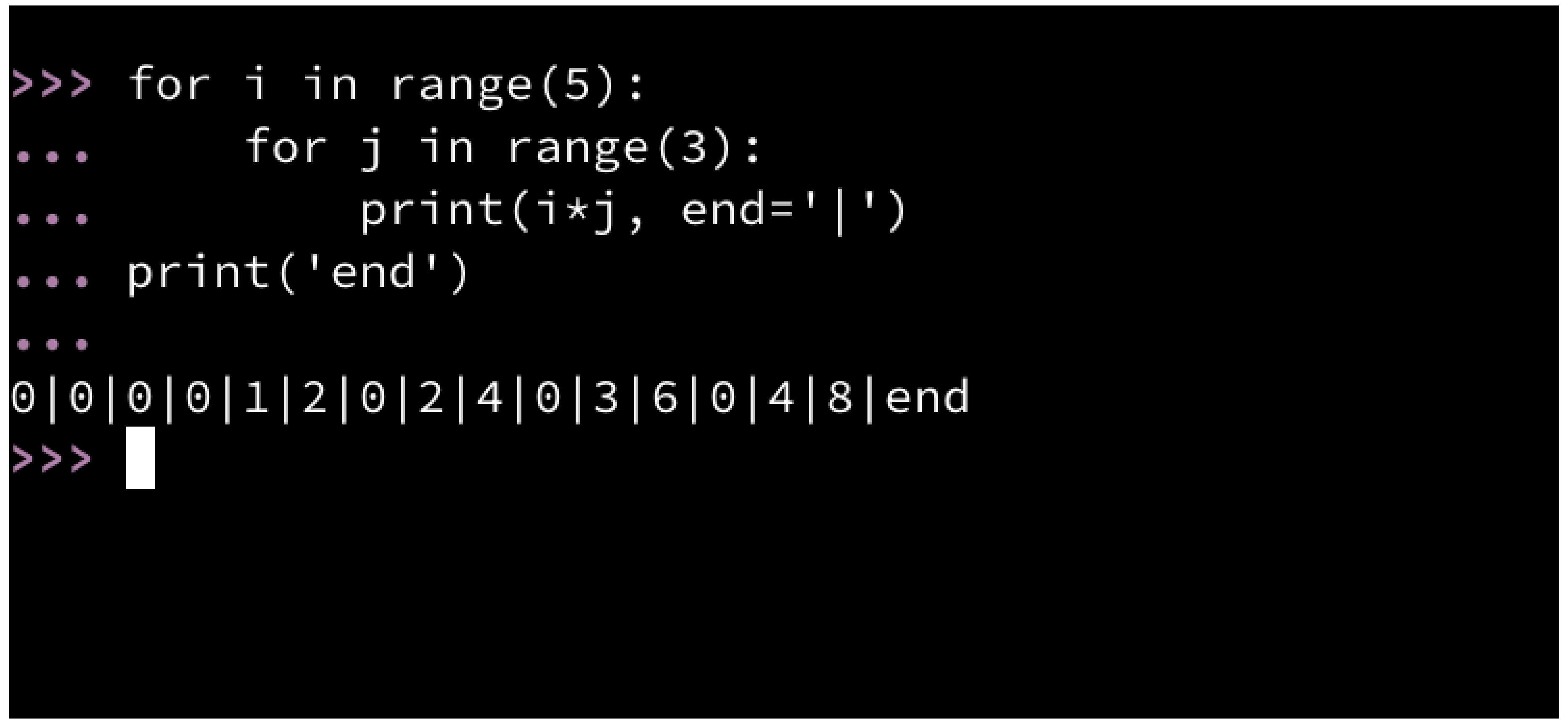
- is executed only if there is no more elements in the sequence (next() returns False), in particular, it is **not executed** if the loop does **not** reach the end:
  -> cause I: **break** key word
  -> cause II: error raised
- is **optional**

```
X = [3,11,-32,18-5]                 # assign sequence to variable
for e in x:                         # if has next(), assign next element of sequence x to e
    e = e + 10          # add 10 to current element
    print€                          # print current element (i.e. e + 10)
Else:                   # if has no more next() -> else is called
    print('end')        # print string: 'end'

print(x)                            # print sequence [3,11,-32,18,5]l
```

for loop: nested loop structure

```
x = [3,11,-32,18,-5]        # assign sequence (5 elements) to variable x
y = [1,2,3,4,5,6,7]         # assign sequence (7 elements) to variable y
for i in x:                 # if has next(), assign next element of sequence x to i
    for j in y:             # if has next(), assign next element of sequence y to j
        e = i*j        # multiply i and j
        print(e)                        # print product of i*j
    print('next')           # print marker for next element
else:                                   # if has no more next() in sequence x -> else is called
    print('end')                        # print string: 'end'
```



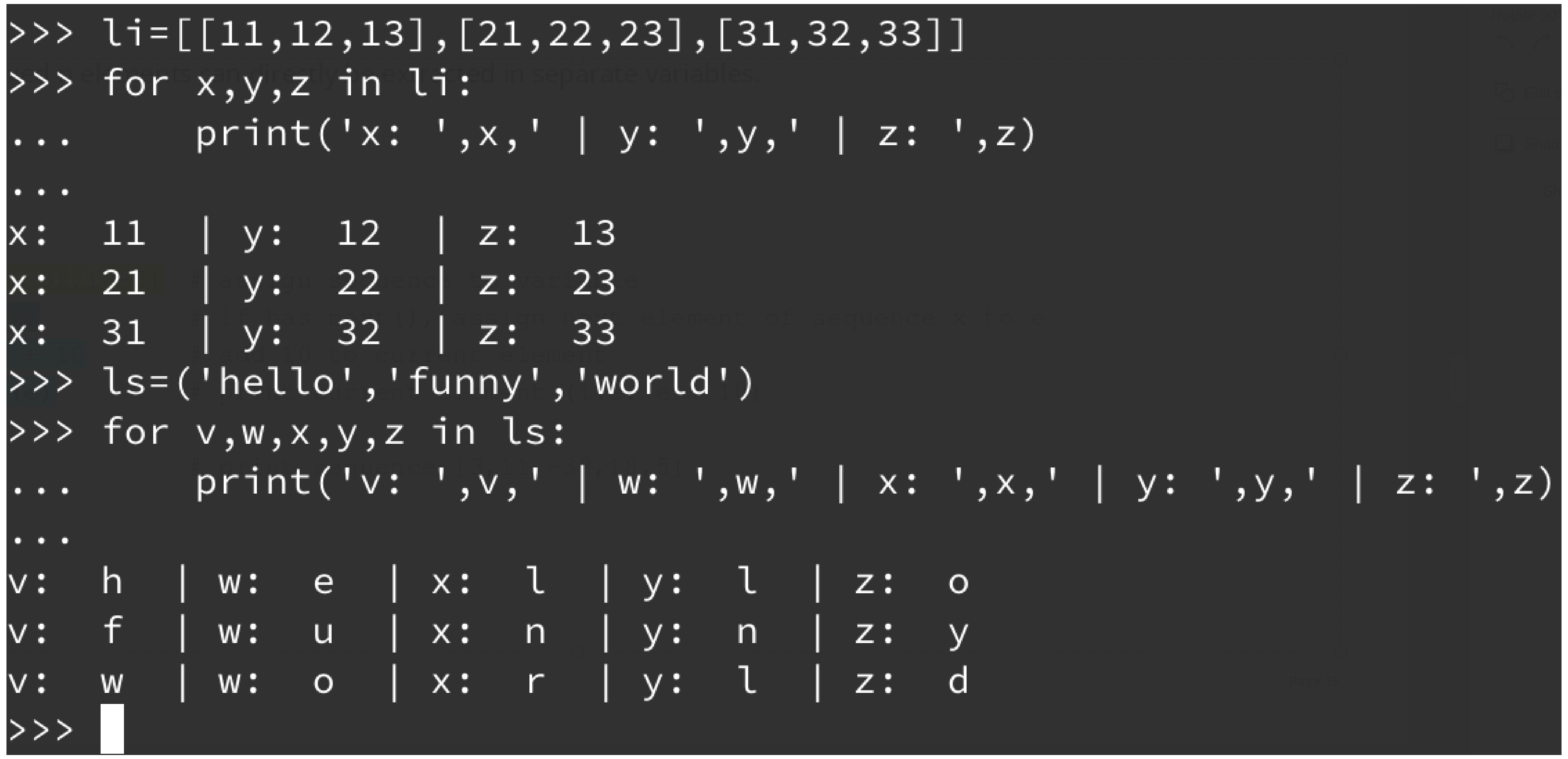
```
>>> for i in range(5):
...     for j in range(3):
...         print(i*j, end='|')
... print('end')
...
0|0|0|0|1|2|0|2|4|0|3|6|0|4|8|end
>>> 
```

for loop: Header shortcut for nested structures

Nested sequences with a constant “number of elements”-structure can be extracted directly in separate variables.

For this case, the number of loop variables **must** meet the number of elements in the nested sequence:

- 2 variables (x,y) for: ([‘hello’, 1], [66, 99], [2, ‘world’])
- 3 variables (x,y,z) for: [[1,2,3], [4,5,6], [7,8,9], [10,11,12]]
- 5 variables (v,w,x,y,z) for: [‘hello’, ‘funny’, ‘world’]



```
>>> li=[[11,12,13],[21,22,23],[31,32,33]]
>>> for x,y,z in li:
...     print('x: ',x,' | y: ',y,' | z: ',z)
...
x:  11  | y:  12  | z:  13
x:  21  | y:  22  | z:  23
x:  31  | y:  32  | z:  33
>>> ls=('hello','funny','world')
>>> for v,w,x,y,z in ls:
...     print('v: ',v,' | w: ',w,' | x: ',x,' | y: ',y,' | z: ',z)
...
v:  h  | w:  e  | x:  l  | y:  l  | z:  o
v:  f  | w:  u  | x:  n  | y:  n  | z:  y
v:  w  | w:  o  | x:  r  | y:  l  | z:  d
>>> 
```

while (¬True)

Information Technology
March 6, 2025

FH Zentralschweiz

while (-True)

In contrast to a for loop, a while loop executes the loop body **as long as** the condition equals **True**.

The while loop:

- Checks the condition each time before re-executing the loop body
- Terminates as soon as the condition is False

```
x = 1              # assign start value to variable
while x < 10:      # check condition: x < 10
    print(x)                  # print current value of x
    x = x * 2      # multiply x by 2
print(x)                      # print last value of x breaking condition (i.e. x=16)
```

while (-True): else condition

Similar to the if-else statement, the while loop also provides an **else** condition. This part

- is **only called** if the checked condition (i.e. x < 10) equals False,
  in particular, it is **not executed** if
  -> cause I: break in loop
  -> cause II: error raised
- is **optional**

```
x = 1             # assign start value to variable
while x < 10:     # check condition: x < 10
    print(x)                  # print current value of x
    x = x * 2     # multiply x by 2
else:             # while condition (i.e. x < 10) equals False
    print('end')  # print string: 'end'

print(x)                      # print last value of x: either x=16 or the value when loop was breaked
```

Lucerne University
of Applied Sciences
and Arts

break/continue/pass

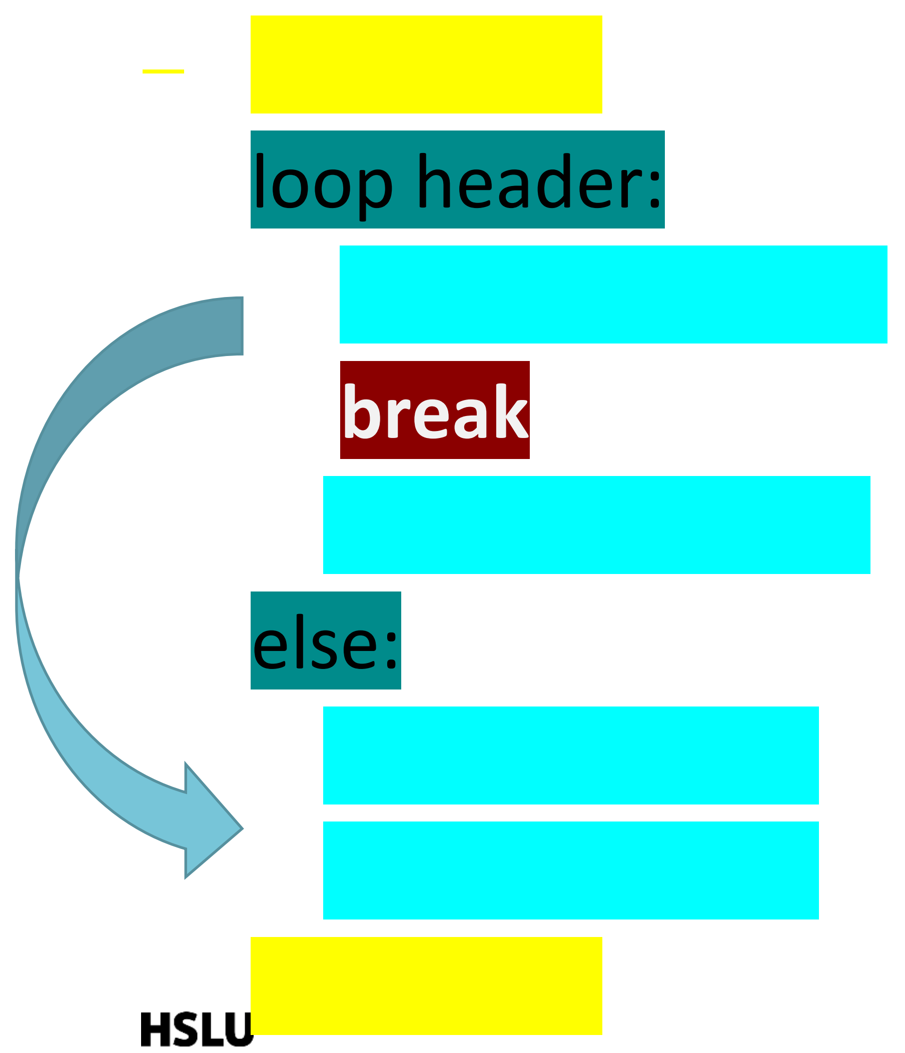
Information Technology
March 6, 2025

break/continue/pass

Python has three keywords to control the loop (and function) process flow:

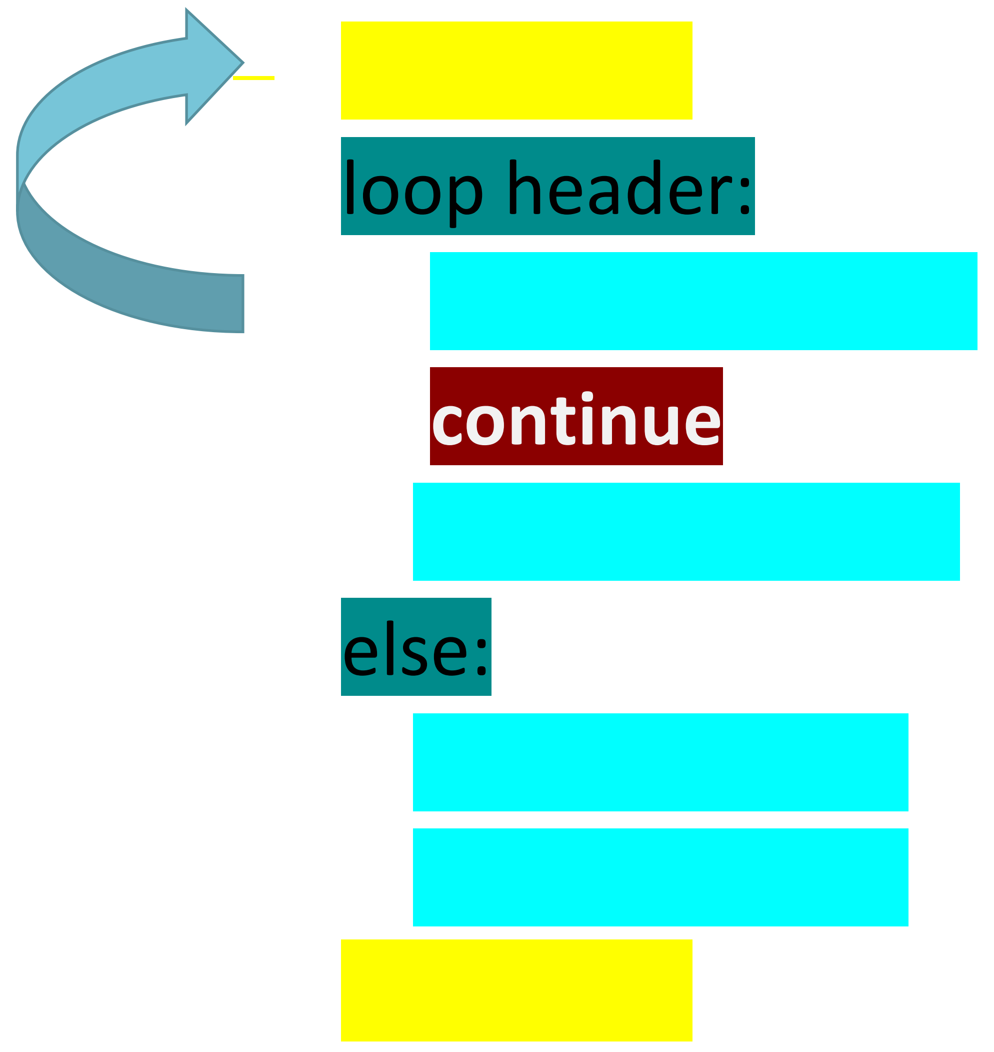
break

Immediately break the current loop.



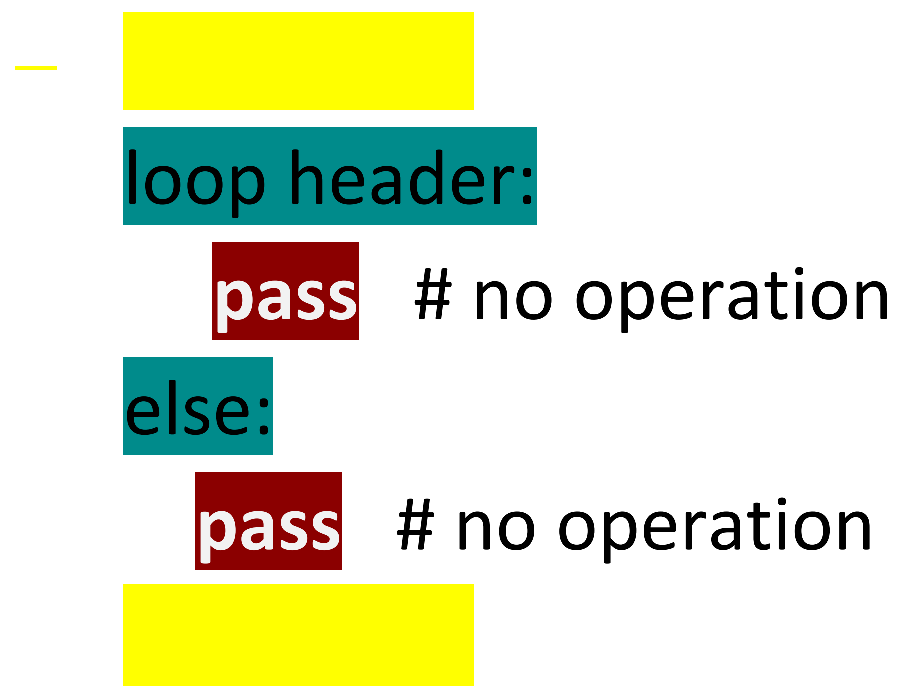
continue

Ignoring the rest of the loop body and jumping back to the header.



pass

No operation; regular iteration with no execution.



HSLU Lucerne University of Applied Sciences and Arts

Enumerate

Information Technology
March 6, 2025

FH Zentralschweiz

Enumerate

The enumerate returns an iterator that returns a tuple with an incrementing number for each element of the sequence.
Docu: https://docs.python.org/3/library/functions.html#enumerate

- In brief: it assigns a incrementing number to each element of the sequence.

Advantage:

- For each element we know the exact index without any supplementary counting variable.
- Index can be set with an offset.

```
enumerate(iterable, start=0)
```

li
2
-5
7
23
44

idx	li
(0,	2)
(1,	-5)
(2,	7)
(3,	23)
(4,	44)

HSLU Lucerne University of Applied Sciences and Arts

School of Computer Science and Information Technology
Research
Ramón Christen
Research Associate Doctoral Student

Phone direct +41 41 757 68 96
ramon.christen@hslu.ch

HSLU T&A, Competence Center Thermal Energy Storage
Research
Andreas Melillo
Lecturer

Phone direct +41 41 349 35 91
andreas.melillo@hslu.ch