HSLU Lucerne University of Applied Sciences and Arts

Python for Data Science: SW10

Short Functions

Information Technology

May 1, 2025

Content

- Deep vs. Shallow Copy
- Lambda Function
  - Concept
  - Use in map(), sort() or Filter Functions
- List Comprehension
  - Concept
  - Conditional
  - Nested
  - Tuple Comprehension
  - Dictionary Comprehension
- Type Annotations
  - Concept

HSLU Lucerne University of Applied Sciences and Arts

Deep vs. Shallow Copy

Information Technology

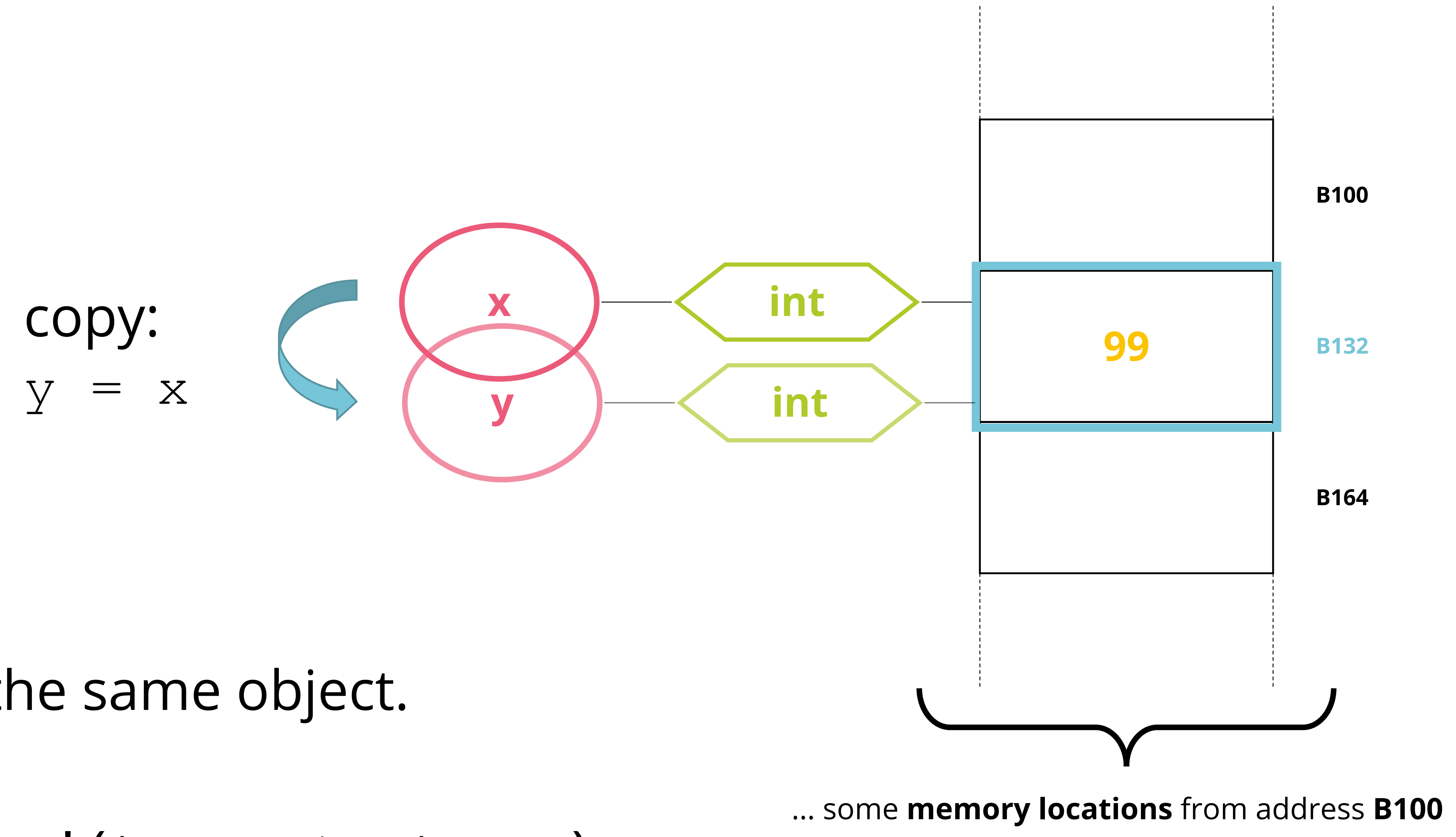
May 1, 2025

Deep vs. Shallow Copy

Python only has **objects** and **references** to them. So, variables are always references to objects stored in the memory in one or more storage cells.

Consequently, when copying a variable, we **copy** the **reference** to a particular object – not the object itself.

```
>>> x = 98765.4321
>>> y=x
>>> id(x)
140434081955632
>>> id(y)
140434081955632
```



After copying, `x` and `y` share the same reference to one and the same object.

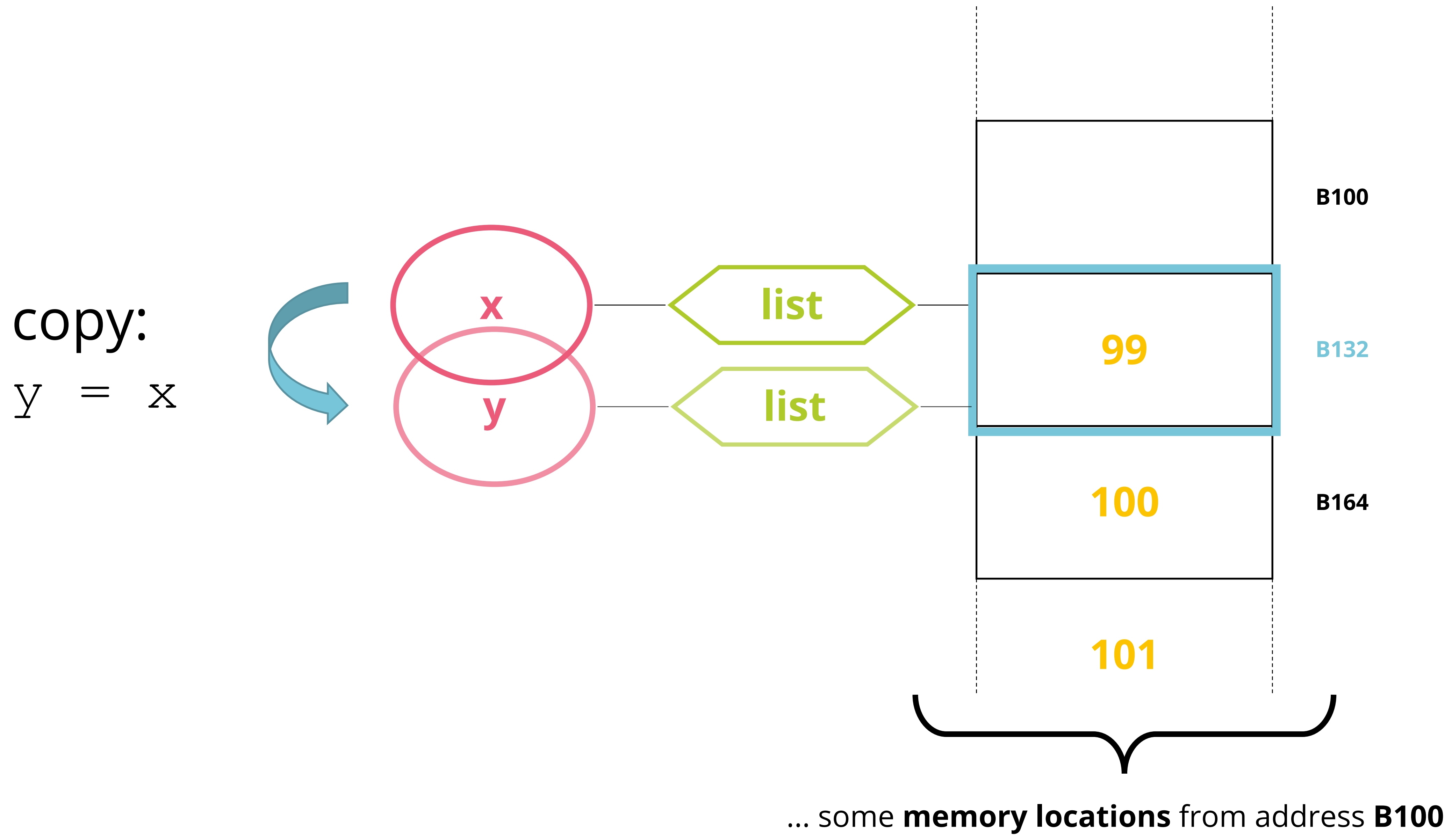
- `id(x) == id(y)`
- Assigning a new value to x means, x gets a new reference! (`id(x) != id(y)`)

Deep vs. Shallow Copy

As for integer values, this holds for all object including sequences such as `list, tuple, set, dictionary` and sub-sequences.

- Object mutation affects all copies, namely all variables referring to the same object.

```
>>> x = [99,100,101]
>>> y=x
>>> id(x)
140434082548160
>>> id(y)
140434082548160
>>> y[1]=200
>>> x
[99, 200, 101]
```



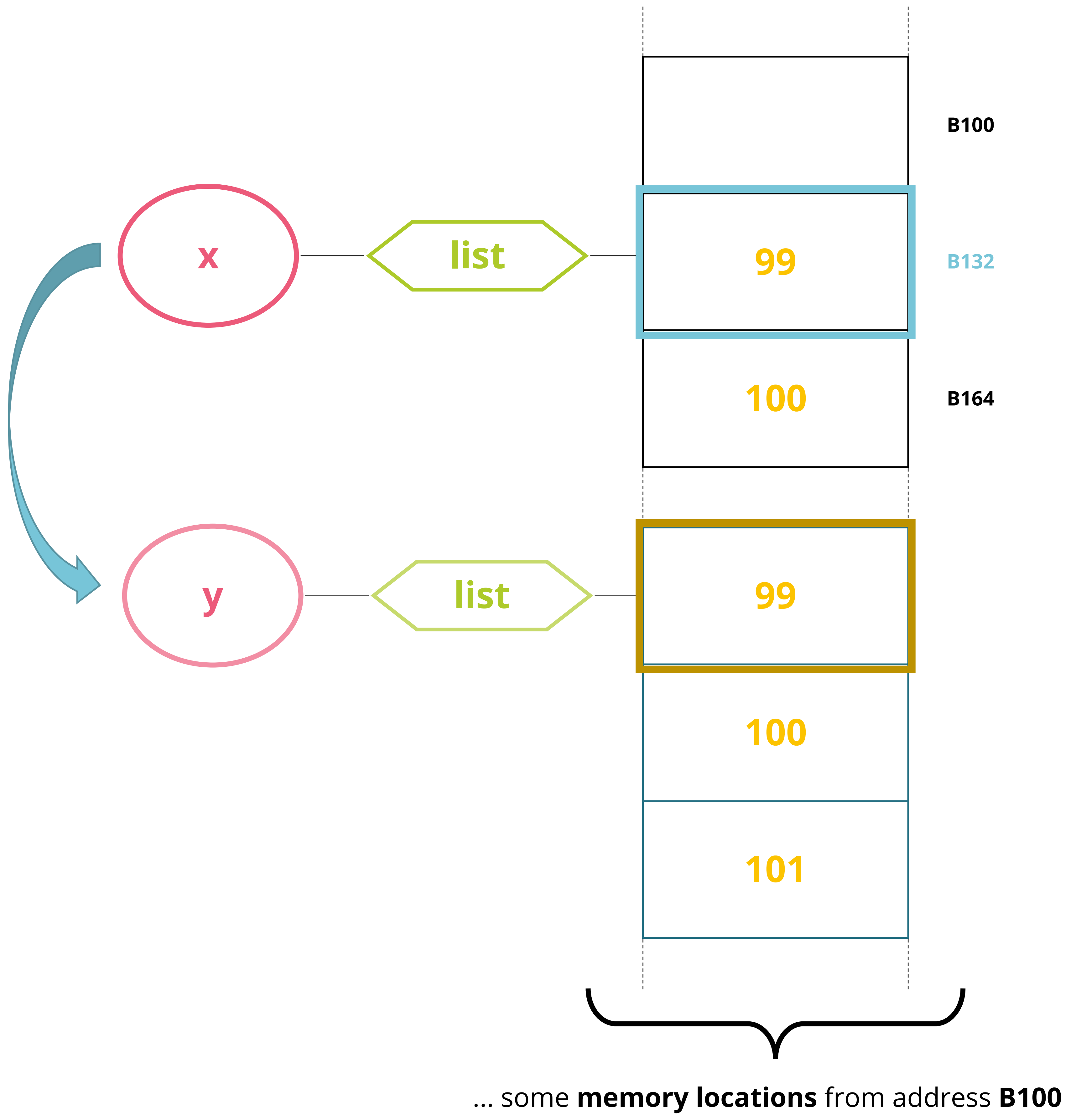
Deep vs. Shallow Copy

To create a copy of a sequence object with a new reference (ie. id), sequence objects implement the `copy()` method.

- The method `copy()` is **only** available for **mutable** objects.
  -> immutable types like `int` or `str` provide no `copy()` method.

```
>>> x = [99, [1,2], 101]
>>> y = x.copy()
>>> id(x)
140434079124544
>>> id(y)
140434078906688
```

copy:
`y = x.copy()`



Copying a sequence results two different objects.

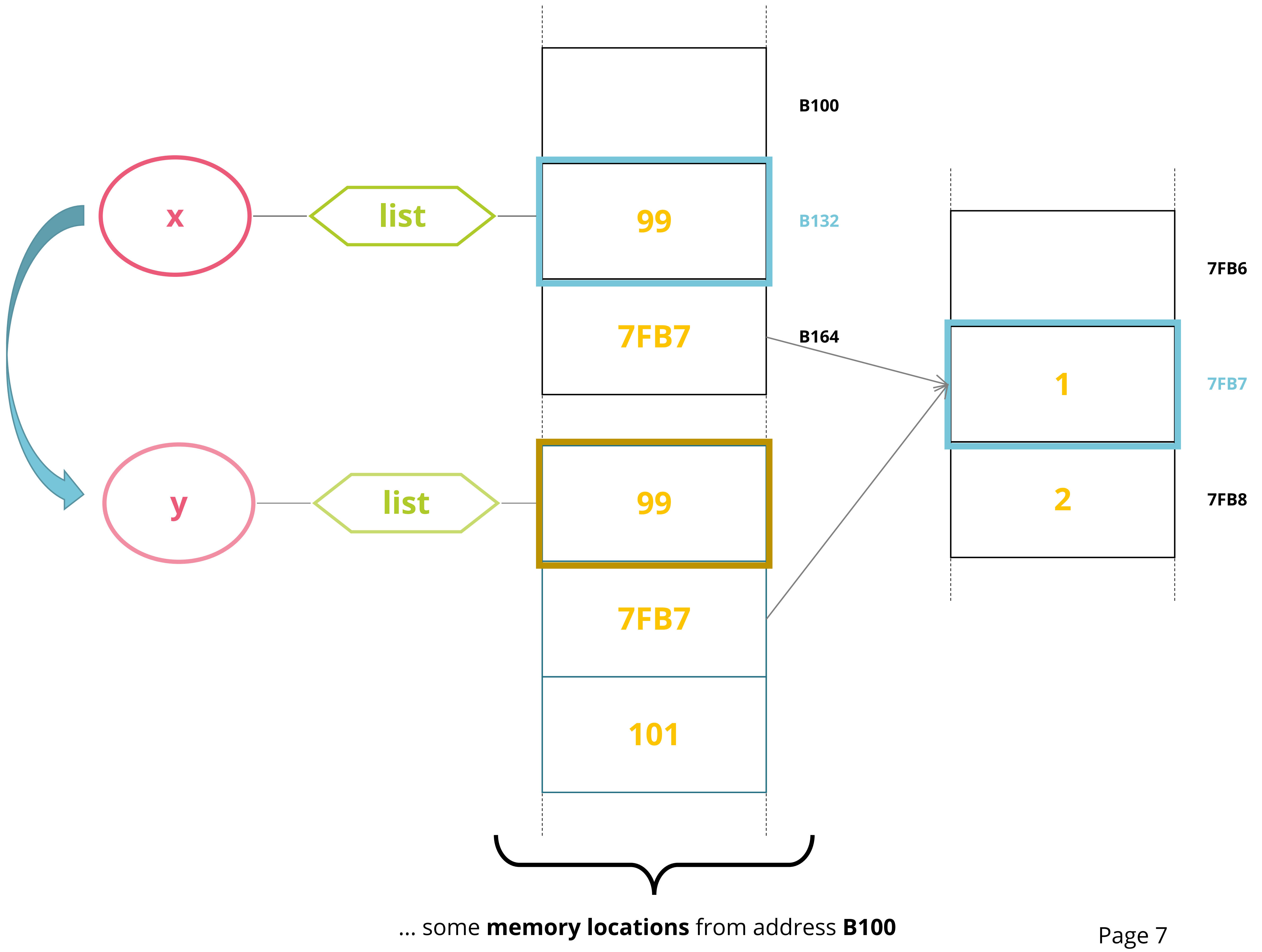
- `id(x) != id(y)`

Deep vs. Shallow Copy

Yet, copying a sequence returns a **shallow** copy of the object **only**!

References to **sub-sequences** still remain the **same** and are affected by the same side-effect as when duplicating the reference by re-assignment: `y = x`

copy:
`y = x.copy()`




```
>>> help(list)
```

```
|
|  copy(self, /)
|      Return a shallow copy of the list.
|
```

Lucerne University of Applied Sciences and Arts

Lambda Function

Information Technology

May 1, 2025

Lambda Function: Concept

Usually, a function consist of a **header** that can be referred to and a **body**.

```
function_header():
    block instruction
    block instruction
    block instruction
```

Function references:

- can be returned for another function:

```python
def my_fun():
    def inner_fun(x):
        return x
    return inner_fun

my_if = my_fun()
print(my_if('hello world'))
```

- can be associated to a variable:

```python
def my_fun():
    def inner_fun(x):
        return x
    return inner_fun

my_if = my_fun
print(my_if()('hello world'))
```

Lambda Function: Concept

When associating a function to a variable, this can be done directly without function declaration first.

Functions without explicit declarations are:

- called **`labmda`** functions,
- comprise **one** expression only, and
- always return the result of the expression.

Lambda functions are declared with the `lambda` **keyword**.

```
lambda parameters: expression
```

Associated to a variable, they can be called as regular functions by the variable as reference instead of the function name.

```
>>> x = lambda x,y: x+y
>>> x(3,6)
9
```

Lambda Function: Use in map(), sort() or Filter Functions

Lambda functions are usually used as **anonymous** functions to **pass expressions** to other functions. They are typically used for expressions in combination with filter functions such as for the `map()` or `sort()` functions:

`map()` function:

- applies a function to every single item of an iterable.
- returns an iterator.

```
>>> import math
>>> x = [0,15,30,45,60,75,90]
>>> list(map(lambda z: round(math.cos(z),3), x))
[1.0, -0.76, 0.154, 0.525, -0.952, 0.922, -0.448]
```

`sort()` function:

- sorts an iterable based on a given comparator.

```
>>> x = [6,1,3,7,9,5,2,4,8,0]
>>> x.sort(key=lambda x: x%3)
>>> x
[6, 3, 9, 0, 1, 7, 4, 5, 2, 8]
```

Docu: https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions

Lucerne University of Applied Sciences and Arts

List Comprehension

Information Technology

May 1, 2025

List Comprehension: Concept

Extending combined `lambda` and `map` functions, list iterations can be done using comprehensions.

Instead iterating a list `x = [1,2,3,4,5,]` with a for loop for applying a **single expression** like `i**2` on **each** element:

```
x = [1,2,3,4,5], x2 = []
for i in x:
    x2.append(i**2)
```

a single expression can be applied in a simpler way on one line using a **list comprehension**:

Same result!

```
x = [1,2,3,4,5]
x2 = [i**2 for i in x]
```

In contrast, comprehensions can be applied **in-place**:

```
x = [i**2 for i in x]
```

Docu: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

List Comprehension: Conditional

Comprehensions **only** apply a **single expression** on each list element. However, it provides the option defining conditions for the elements on that it should be applied.

Expression: `i**2`
Condition: `i == odd number`

```
x = [1,2,3,4,5]
x2 = [i**2 for i in x if i%2]
```

This applies the expression on the elements **only** for those the condition is **true**. There is **NO** else part.

- Otherwise the element is skipped.

[1,~~2~~,3,~~4~~,5] -> [1, 9, 25]

When an alternative expression should be applied, the expression must become a **shorthand if-else** clause:

Expression: `i**2 if i%2 else i-10`
Condition: `None`

```
x = [1,2,3,4,5]
x2 = [i**2 if i%2 else i-10 for i in x ]
```

Result: `[1,2,3,4,5] -> [1, -8, 9, -6, 25]`

List Comprehension: Conditional

For enhanced complexity, the single expression can also be a regular **function** `f(x)`.

- In this case, each element of the list is passed as a parameter to the function `f(x)`.

```
x = range(1,21)

# prime factorization
def prim(x):
    p = [2,3,5,7,11,13,17,19,23,29]
    for i in p:
        if i >= x: break
        while x % i == 0:
            x //= i
    return x

prim_fac = [prim(i) for i in x]
```

List Comprehension: Nested

Similar to for-loops, comprehension allow to apply expressions to multi-dimensional lists – **nested** list comprehension
Docu: https://docs.python.org/3/tutorial/datastructures.html#nested-list-comprehensions

Assume, all elements of the given list have to be divided by 10 (`i//10`):

```
ll = [[1000, 200, 30], [22, 33], [555, 333, 222, 111]]
```

- for-loop:
  (simple list return)

```
res = []
for x in ll:
    for y in x:
        res.append(y//10)
```

- Comprehension:
  (simple list return)

```
res = [y//10 for x in ll for y in x]
```

- Comprehension:
  (**nested** list return)

```
res = [[y//10 for y in x] for x in ll]
```

List Comprehension: on Tuple

Despite the similarity between tuple and list in Python, there exist **no** tuple comprehension.

- applying a comprehension on a tuple returns a **generator**: see generator input in the next few weeks.

```
>>> t = (10,20,33,55,99)
>>> (i//10 for i in t)
<generator object <genexpr> at 0x7fb95b913850>
```

Generating a tuple with a comprehension requires an explicit typecast `tuple()`.

```
x = tuple([i//10 for i in t])
```

Or a tweak using unpacking operator for variable assignment:

```
x = *[i//10 for i in t],
```

List Comprehension: on Dictionary

In contrast to tuples, Python supports **dictionary** comprehension.

Dictionary comprehension requires the format of **dictionary elements** as expression results:

- column separated key-value pairs: **`key:value`**
- encapsulated in curly brackets: `{ k1:v1, k2:v2, k3:v3, ... }`

That requires a separate `key` **`k`** and `value` **`v`** declaration. The basic syntax is:

{key:value for item in iterable}

Following examples show three alternatives for dictionary comprehension:

```
x = [1, 2, 3, 4, 5]

d1 = {i:i for i in x}
d2 = {k:v for (k,v) in enumerate(x)}
d3 = {k:v for (k,v) in zip(['a', 'b', 'c', 'd', 'e'], x)}
```

HSLU Lucerne University of Applied Sciences and Arts

Type Annotation

Information Technology

May 1, 2025

Type Annotations: Concept

“The Python runtime does not enforce function and variable type annotations. They can be used by third party tools such as type checkers, IDEs, linters, etc.”

Cheatsheet: https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html
Docu: https://docs.python.org/3/library/typing.html#typing-support-for-type-hints

In particular, we only define the **type** (any Python class) of **references** separated by **colon** (:) and **arrows** (->). This includes:

- Variables: `x: type = value` `x: int = 3`
- function parameters: `def fnc(a: type, b: type)` `def fnc(a: int, b: str)`
- function returns: `def fnc() -> type:` `def fnc() -> None:`

Despite reference type declaration, Python still **allows** type **conflicting** value assignments.

- Type annotation is very helpful for programmer working on complex scripts.

Assert Function

Information Technology

May 1, 2025

Function Input Assert for Libraries

Sharing functions in libraries requires proper interfaces with adequate error handling.

Despite Python does not support typing, data types assert the correct data format only in a limited way. Consequently, input must be asserted and meaningful error messages returned before executing algorithms.

Assume, you provide a function in a library, that returns the n^{th} root of the absolute difference between subsequent data points in a data series.

$$x = [x_1, x_2, x_3, \ldots, x_k]$$

$$c_i = (|x_{(i+1)} - x_i|)^{1/n} \qquad \forall\ i \in \{1, 2, 3, \ldots, (k-1)\}$$

In this case, the function provides two parameters that must comply following conditions:

- x: list with length >=2
- n: integer value

Function Input Assert for Libraries (Exercise)

Provide a function **fuzz_sum** in a new module called: **exerc_lib.py**

The function calculates a proportional sum (fuzzy hat-function) under a moving window along a passed data series. In fact, the function takes three parameters:

- `Data: Dict[str, str, List[float | int]]` e.g. `{‘name’:’test’, ‘loc’:’north’, ‘x’:[2, 3.9, 8, 5, 1.2]}`
- `Hat_coeff: List[float]` e.g. `coeff=[0.1, 0.55, 1, 0.55, 0.1]`
- `W_length: int` e.g. `w_length=5`

Thereby, the hat-coefficients and the window length should be optional.

How do you have to test the parameters passed to the function?

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