EINFÜHRUNG IN DIE PROGRAMMIERUNG

QUICK OVERVIEW: REFERENCES

DHBW MANNHEIM
WIRTSCHAFTSINFORMATIK (DATA SCIENCE)

Markus Menth
Martin Gropp

REFERENCES

What is this about?

```
a = [3, 17, 9, -5]
b = a

b[0] = 4

print(a[0])
```

Output:

REFERENCES

What is this about?

```
a = [3, 17, 9, -5]
b = a

b[0] = 4

print(a[0])
```

Output:

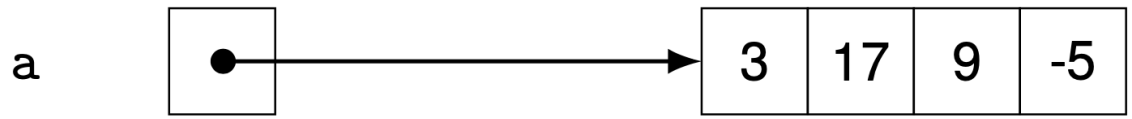
4

…?!

REFERENCES

The list `[3, 17, 9, –5]` is (typically) stored in a separate memory area called the "heap".

The variable `a` contains only a **reference** to that data ("the address").

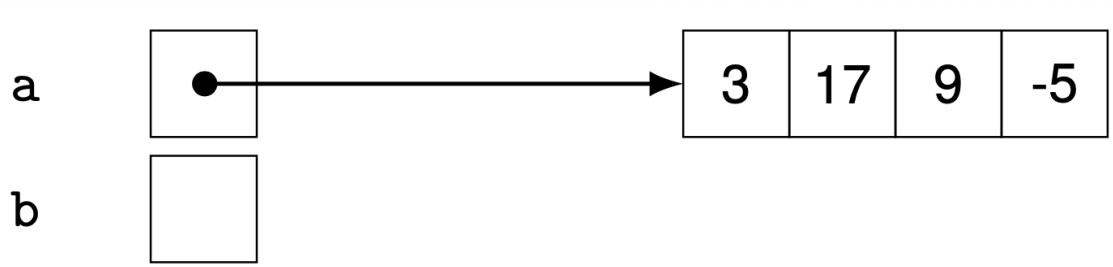


ASSIGNMENTS

When assigning to a new variable, only the reference is copied.

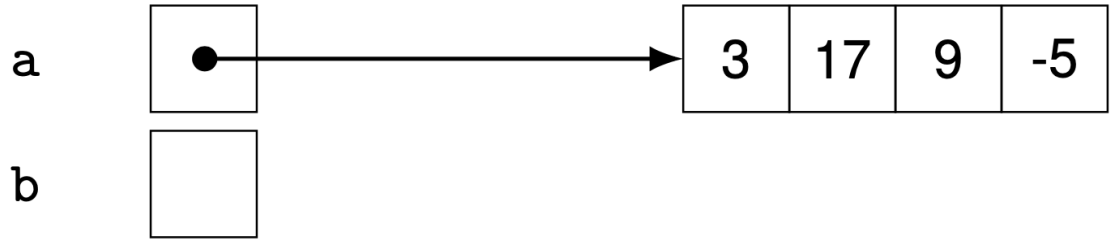
ASSIGNMENTS

When assigning to a new variable, only the reference is copied.



ASSIGNMENTS

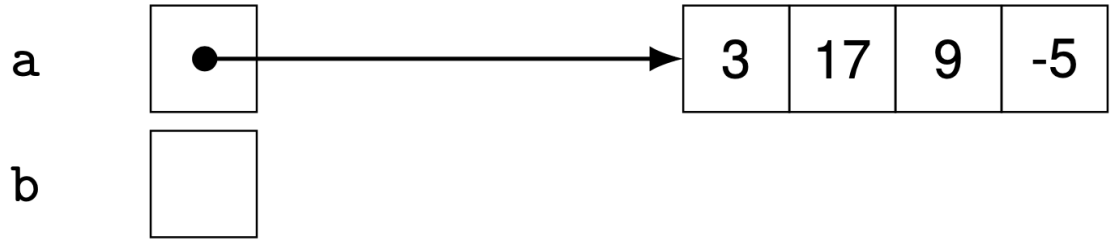
When assigning to a new variable, only the reference is copied.



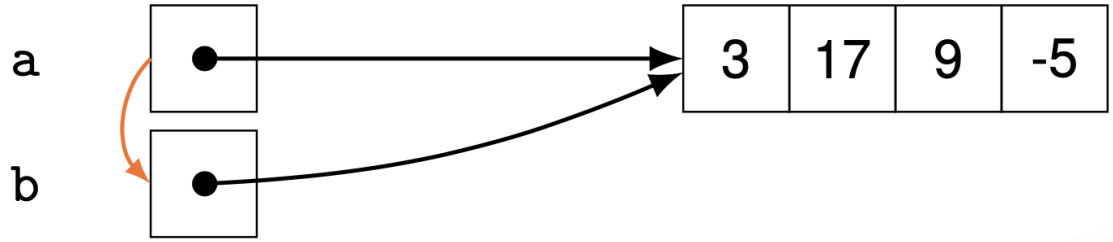

```
b = a
```

ASSIGNMENTS

When assigning to a new variable, only the reference is copied.

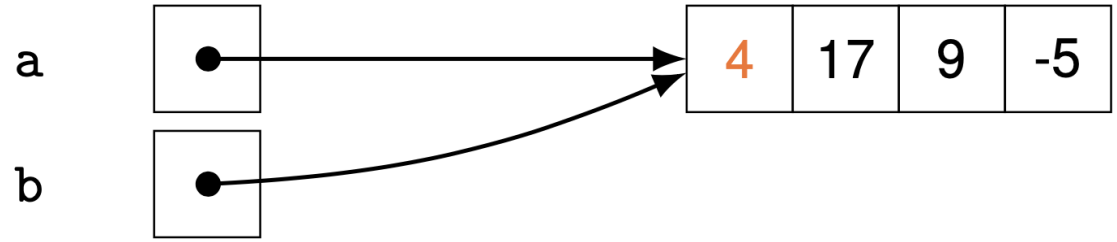



```
b = a
```



ASSIGNMENTS

```
b[0] = 4
```



Operators like

- `.`: `a.append(42)`
- `[…]`: `a[0]`

"follow" the reference and access its target.

WHY DOES THIS MAKE SENSE?

During the execution of a program, data is typically passed around between many functions.

Creating copies of large data structures every time would be very inefficient (→ *Call by Value*!).

Passing only a reference to the data solves this problem.

WARNING 🤯

Python types can modify the behavior of the *augmented arithmetic assignment* operators (+=, ∗=, etc.)

This means: `a += b` is not always the same as `a = a + b`.

```
def f(x, y):
    x += y
```

```
a = 0
f(a, 1)
print(a)
```

```
0
```

→ no change

```
a = []
f(a, [1])
print(a)
```

```
[1]
```

→ changed

WARNING 🤯

Why?!

Under the hood, `x += y` is actually equivalent to `x.__iadd__(y)` (not to `x = x + y`). The `__iadd__` method works in just the same way as `append`, for example, and modifies the list referenced by `x` instead of assigning a new reference to the variable `x`.

```
def f(x, y):
    x = x + y

a = []
f(a, [1])
print(a)
```

```
[]
```

```
def f(x, y):
    x += y

a = []
f(a, [1])
print(a)
```

```
[1]
```

(To be exact, `x += y` translates to `x = x.__iadd__(y)`, where `x.__iadd__(...)` commonly returns `x`.)

THE IS OPERATOR

Unlike `a == b`,
`a is b` will only be true
if `a` and `b` are the same object.

```
>>> a = [1]
>>> b = [1]

>>> a == b
True
>>> a is b
False

>>> a = b
>>> a is b
True
```

THE IS OPERATOR

Careful, for some (immutable) types, this might not always work as expected:

```
>>> a = 1
>>> b = 1

>>> a is b
True
```

```
>>> a = 257
>>> b = 257

>>> a is b
False
```

...but we're not going into even more details here.

EXERCISE

What's the output of the following program?

```
a = [
    [1, 2],
    [3, 4],
]
```

EXERCISE

What's the output of the following program?

```
a = [
    [1, 2],
    [3, 4],
]
```

```
def f1(l):
    l[0] = 10

f1(a)
print(a)
```

EXERCISE

What's the output of the following program?

```
a = [
    [1, 2],
    [3, 4],
]
```

```
def f1(l):
    l[0] = 10

f1(a)
print(a)
```

```
def f2(l):
    f1(l[1])

f2(a)
print(a)
```

EXERCISE

```
def f3(l):
    l = 20

f3(a)
print(a)
```

EXERCISE

```
def f3(l):
    l = 20

f3(a)
print(a)
```

```
def f4(l):
    l.append(10)

f4(a)
print(a)
```

EXERCISE

```
def f3(l):
    l = 20

f3(a)
print(a)
```

```
def f4(l):
    l.append(10)

f4(a)
print(a)
```

```
f4(a[1])
print(a)
```