Mehr als Rechnen

Mit Python Mathematik treiben

Prof Dr Tobias Kohn

1. Die Entstehung von Python

Ansatz: Reduziere die Syntax auf das Minimum

```
function ggT(a: Integer, b: Integer): Integer;
var
  t: Integer;
begin
  if a < b then
    begin
      t := a; a := b; b := t;
    end;
  while b > 0 do
    begin
      t := a % b;
      a := b;
      b := t;
```

```
function ggT(a: Integer, b: Integer): Integer
var
  t: Integer
begin
  if a < b then
    begin
      t := a; a := b; b := t
    end
  while b > 0 do
    begin
      t := a % b
      a := b
      b := t
```

```
function ggT(a, b)
begin
  if a < b then
    begin
      t := a; a := b; b := t
    end
  while b > 0 do
    begin
      t := a % b
      a := b
      b := t
    end
  ggT := a
```

```
function ggT(a, b):
  if a < b then
      t := a; a := b; b := t
  while b > 0 do
      t := a % b
      a := b
      b := t
  ggT := a
```

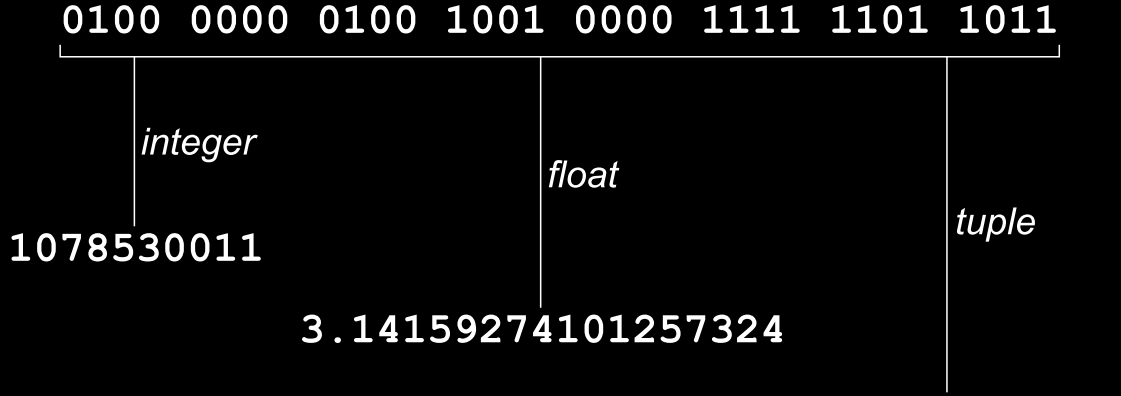
```
function ggT(a, b):
  if a < b then
      t = a; a = b; b = t
  while b > 0 do
      t = a % b
      a = b
      b = t
  return a
```

```
def ggT(a, b):
  if a < b:
      t = a; a = b; b = t
  while b > 0:
      t = a % b
      a = b
      b = t
  return a
```

```
def ggT(a, b):
    if a < b:
        a, b = b, a
    while b > 0:
        a, b = b, a % b
    return a
```

Was passiert, wenn die Typen (Deklarationen) wegfallen?

Wozu brauchen wir Typendeklarationen?


0100 0000 0100 1001 0000 1111 1101 1011
integer
1078530011
float
3.1415927410125732 4
tuple
(16457, 4059)

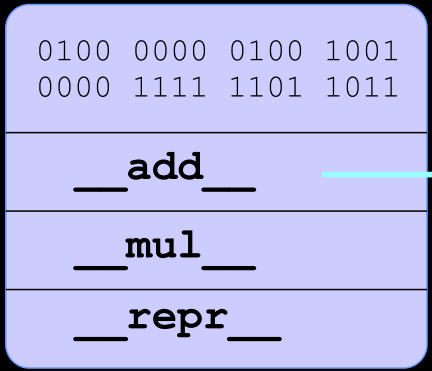
Was passiert, wenn die Typen (Deklarationen) wegfallen?

– Typinformation ist an Wert gebunden, nicht mehr an Variable

– Nur noch ein Integer-Typ und ein Float-Typ

– Auflösung der Operationen zur Laufzeit

Werte werden zu Objekten mit Operatoren




```
PyObj float_add(PyObj a, PyObj b) {
  if (type(a) != PyFloat)
    raise_error("wrong type");
  if (type(b) != PyFloat)
    b = b.__to_float__();
  double res = a._value + b._value;
  return new_float(res);
}
```

Der Code bleibt fast unverändert

```
x = x + 1
y = 3 * x - 4
```

```
x = x.__add__(1)
y = ( 3.__mul__(x) ).__subtract__( 4 )
```

Was passiert, wenn die Typen (Deklarationen) wegfallen?

– Programmausführung wird (viel) langsamer

– Umsetzung der Operatoren wird sehr flexibel

2. Eigene Typen und Operationen

Übersetzung der Operatoren

```
x + y      ⇒  x.__add__(y)
x - y      ⇒  x.__subtract__(y)
x * y      ⇒  x.__mul__(y)
x / y      ⇒  x.__truediv__(y)
str(a)     ⇒  a.__str__()
len(a)     ⇒  a.__len__()
a[i]       ⇒  a.__getitem__(i)
a[i] = b   ⇒  a.__setitem__(i, b)
```

```
class Fraction:
    def __init__(frac, zaehler, nenner):
        frac.z = zaehler
        frac.n = nenner

    def __repr__(frac):
        return f"{frac.z} / {frac.n}"

two_thirds = Fraction(2, 3)
print( two_thirds )
```

```
class Fraction:
    def __init__(frac, zaehler, nenner):
        frac.z = zaehler
        frac.n = nenner

    def __repr__(frac):
        return f"{frac.z} / {frac.n}"

    def __mul__(frac, other):
        z = frac.z * other.z
        n = frac.n * other.n
        return Fraction(z, n)
```

```
class Fraction:
    def __mul__(frac, other):
        z = frac.z * other.z
        n = frac.n * other.n
        t = ggT(z, n)
        if t == n:
            return z
        else:
            return Fraction(z // t, n // t)
```

```
class Fraction:
    def __mul__(frac, other):
        z = frac.z * other.z
        n = frac.n * other.n
        return Fraction(z, n)

two_thirds = Fraction(2, 3)
x = two_thirds * 5
```

```
class Fraction:
    def __mul__(frac, other):
        if isinstance(other, Fraction):
            z = frac.z * other.z
            n = frac.n * other.n
            return Fraction(z, n)
        elif isinstance(other, int):
            z = frac.z * other
            return Fraction(z, frac.n)
        else:
            return NotImplemented
```

```
class Fraction:
    def __mul__(frac, other):
        if isinstance(other, Fraction):
            z = frac.z * other.z
            n = frac.n * other.n
            return Fraction(z, n)
        elif isinstance(other, int):
            z = frac.z * other
            return Fraction(z, frac.n)
        elif isinstance(other, float):
            return (frac.z / frac.n) * other
        else:
            return NotImplemented
```

```
class Fraction:
    def __mul__(frac, other):
        if isinstance(other, Fraction):
            ...
        elif isinstance(other, int):
            z = frac.z * other
            return Fraction(z, frac.n)
        else:
            return NotImplemented

two_thirds = Fraction(2, 3)
x = two_thirds * 5
```

```
class Fraction:
    def __mul__(frac, other):
        if isinstance(other, Fraction):
            ...
        elif isinstance(other, int):
            z = frac.z * other
            return Fraction(z, frac.n)
        else:
            return NotImplemented

two_thirds = Fraction(2, 3)
x = 5 * two_thirds
```

```
class Fraction:
    def __mul__(frac, other):
        ...

    def __rmul__(frac, other):
        if isinstance(other, int):
            return frac * other
        else:
            return NotImplemented
```

```
class Fraction:
    def __mul__(frac, other):
        ...

    def __rmul__(frac, other):
        if isinstance(other, (float, int)):
            return frac * other
        else:
            return NotImplemented
```

Übersetzung der Operatoren

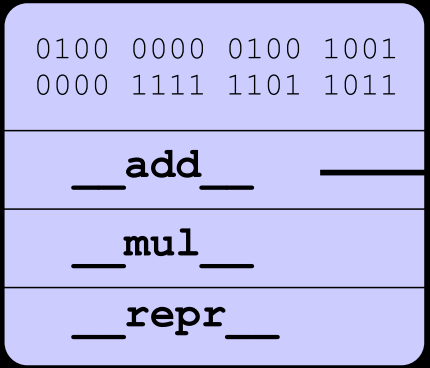
```
x + y  ⇒  x.__add__(y)
x - y  ⇒  x.__subtract__(y)
x * y  ⇒  x.__mul__(y)
x / y  ⇒  x.__truediv__(y)
```

Übersetzung der Operatoren

```
x + y  ⇒  x.__add__(y)
       ⇒  y.__radd__(x)
x - y  ⇒  x.__subtract__(y)
       ⇒  y.__rsubtract__(x)
x * y  ⇒  x.__mul__(y)
       ⇒  y.__rmul__(x)
x / y  ⇒  x.__truediv__(y)
       ⇒  y.__rtruediv__(x)
```

3. High-Performance Python

Python greift nie direkt auf die Daten zu, sondern nur auf Operatoren



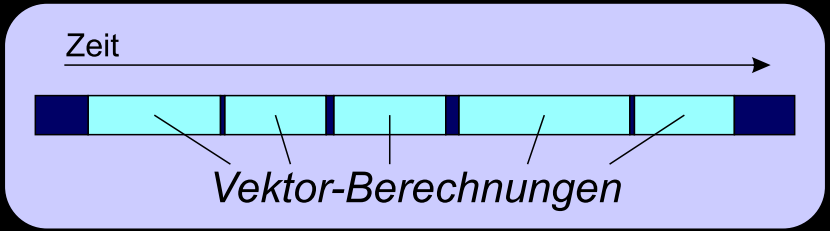

```
PyObj float_add(PyObj a, PyObj b) {
  if (type(a) != PyFloat)
    raise_error("wrong type");
  if (type(b) != PyFloat)
    b = b.__to_float__();
  double res = a._value + b._value;
  return new_float(res);
}
```

Python greift nie direkt auf die Daten zu, sondern nur auf Operatoren

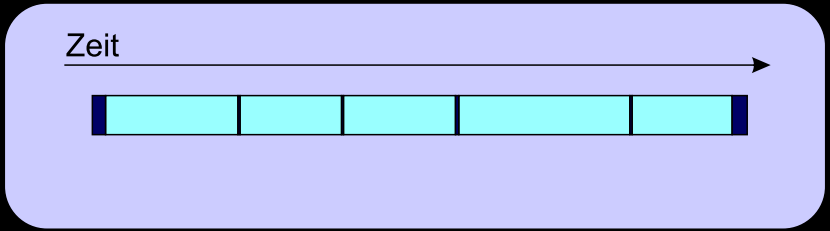
Die Operatoren können auch in C implementiert sein

NumPy: «Parallele» Datenverarbeitung (SIMD)

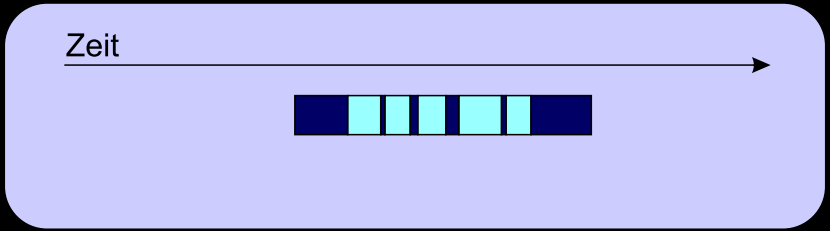
An der richtigen Stelle sparen



An der richtigen Stelle sparen



An der richtigen Stelle sparen



Mehr als Rechnen

– Operatoren in Python sind flexibel

– «Parallele» Datenverarbeitung

– Syntaktische Reduktion aus Wesentliche