Variablen

by Woche 2

In diesem Kapitel lernt man, wie man Variablen in Python definiert. Eine Variable ist ein Name, dem man einen Wert oder ein Objekt zuweist. Nach der Zuweisung einer Variable kann man auf den zugehörigen Wert oder das Objekt zugreifen, indem man den Namen der Variable verwendet. Variablen sind eine praktische Möglichkeit, Werte mit sinnvollen Namen zu speichern.

Variablen zuweisen

In Python weist man Variablen mit = zu:

```
a = 3
print(a)
```

```
3
```

```
b = 'Etwas Text'
print(b)
```

```
Etwas Text
```

Dabei wird stets erst die ggf. vorhandene Berechnung durchgeführt und nur das Ergebnise zugewiesen. Variablennamen in Python müssen nicht aus einzelnen Buchstaben bestehen. Es können beliebige Kombinationen von Buchstaben und Ziffern gewählt werden, solange sie nicht mit einer Ziffer beginnen. Das einzige erlaubte Sonderzeichen ist der Unterstrich _.

```
a = 3
c123 = a/2 + a
print(c123)
```

```
4.5
```

```
Ein_langer_name = 5**a
print(Ein_langer_name)
```

```
125
```

Es gibt Möglichkeiten eine Mehrfachzuweisung durchzuführen:

```
var1 = var2 = 5

print(var1)
print(var2)
```

```
5
5
```

```
var3, var4 = 10, 20

print(var3)
print(var4)
```

```
10
20
```

> **i** Anmerkung zum print() Befehl
>
> Im interaktiven Modus von Python, beispielsweise in Jupyter-Notebooks, reicht es, eine Variable einzugeben, um ihren Wert zu sehen (`x`). Bei normalen Python-Scripts, die man in einer Umgebung wie Visual Studio Code schreibt und ausführt, braucht man `print(x)`, um dies zu tun. Daher kann es Unterschiede im Umgang mit `print()` geben, je nachdem wo man arbeitet. In dieser Dokumentation wird `print()` ab hier nur wenn nötig verwendet, um den Code übersichtlich zu halten. In der Regel wird es dann verwendet, wenn mehrere Werte/Texte/etc. nach einem Code Chunk ausgegeben werden sollen, so wie oben mit `var3` und `var4`.

Methoden

Methoden sind einem Objekt zugeordnete Funktionen in Python, die spezifische Operationen mit oder auf diesem Objekt ausführen. Im Gegensatz zu eigenständigen Funktionen, die unabhängig existieren, sind Methoden eng mit den Objekten verbunden, zu denen sie gehören, und hängen vom Typ des Objekts ab. Um alle verfügbaren Methoden eines Objekts zu sehen, kann die Funktion `dir()` verwendet werden:

```
var = -5 # Typ: Integer
```

```
dir(var)
```

```
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 '__dir__',
 '__divmod__',
 '__doc__',
 '__eq__',
 '__float__',
 '__floor__',
 '__floordiv__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getnewargs__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__index__',
 '__init__',
 '__init_subclass__',
 '__int__',
 '__invert__',
 '__le__',
 '__lshift__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__neg__',
 '__new__',
 '__or__',
 '__pos__',
 '__pow__',
 '__radd__',
 '__rand__',
 '__rdivmod__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rfloordiv__',
 '__rlshift__',
```

```
 '__rmod__',
 '__rmul__',
 '__ror__',
 '__round__',
 '__rpow__',
 '__rrshift__',
 '__rshift__',
 '__rsub__',
 '__rtruediv__',
 '__rxor__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__sub__',
 '__subclasshook__',
 '__truediv__',
 '__trunc__',
 '__xor__',
 'as_integer_ratio',
 'bit_count',
 'bit_length',
 'conjugate',
 'denominator',
 'from_bytes',
 'imag',
 'is_integer',
 'numerator',
 'real',
 'to_bytes']
```

Um darüber hinaus Informationen zu Methoden zu erhalten, kann die Funktion `help()` verwendet werden:

```
var = -5

help(var)
```

```
Help on int object:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
```

```
|
|  If x is not a number or if base is given, then x must be a string,
|  bytes, or bytearray instance representing an integer literal in the
|  given base.  The literal can be preceded by '+' or '-' and be surrounded
|  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
|  Base 0 means to interpret the base from the string as an integer literal.
|  >>> int('0b100', base=0)
|  4
|
|  Built-in subclasses:
|      bool
|
|  Methods defined here:
|
|  __abs__(self, /)
|      abs(self)
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __and__(self, value, /)
|      Return self&value.
|
|  __bool__(self, /)
|      True if self else False
|
|  __ceil__(...)
|      Ceiling of an Integral returns itself.
|
|  __divmod__(self, value, /)
|      Return divmod(self, value).
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __float__(self, /)
|      float(self)
|
|  __floor__(...)
|      Flooring an Integral returns itself.
|
|  __floordiv__(self, value, /)
|      Return self//value.
|
|  __format__(self, format_spec, /)
|      Convert to a string according to format_spec.
|
|  __ge__(self, value, /)
```

```
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getnewargs__(self, /)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as an
index into a list.
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __neg__(self, /)
 |      -self
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __pos__(self, /)
```

```
|      +self
|
|  __pow__(self, value, mod=None, /)
|      Return pow(self, value, mod).
|
|  __radd__(self, value, /)
|      Return value+self.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __rdivmod__(self, value, /)
|      Return divmod(value, self).
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rfloordiv__(self, value, /)
|      Return value//self.
|
|  __rlshift__(self, value, /)
|      Return value<<self.
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __round__(...)
|      Rounding an Integral returns itself.
|
|      Rounding with an ndigits argument also returns an integer.
|
|  __rpow__(self, value, mod=None, /)
|      Return pow(value, self, mod).
|
|  __rrshift__(self, value, /)
|      Return value>>self.
|
|  __rshift__(self, value, /)
|      Return self>>value.
|
|  __rsub__(self, value, /)
|      Return value-self.
```

```
 |
 |  __rtruediv__(self, value, /)
 |      Return value/self.
 |
 |  __rxor__(self, value, /)
 |      Return value^self.
 |
 |  __sizeof__(self, /)
 |      Returns size in memory, in bytes.
 |
 |  __sub__(self, value, /)
 |      Return self-value.
 |
 |  __truediv__(self, value, /)
 |      Return self/value.
 |
 |  __trunc__(...)
 |      Truncating an Integral returns itself.
 |
 |  __xor__(self, value, /)
 |      Return self^value.
 |
 |  as_integer_ratio(self, /)
 |      Return a pair of integers, whose ratio is equal to the original int.
 |
 |      The ratio is in lowest terms and has a positive denominator.
 |
 |      >>> (10).as_integer_ratio()
 |      (10, 1)
 |      >>> (-10).as_integer_ratio()
 |      (-10, 1)
 |      >>> (0).as_integer_ratio()
 |      (0, 1)
 |
 |  bit_count(self, /)
 |      Number of ones in the binary representation of the absolute value of
self.
 |
 |      Also known as the population count.
 |
 |      >>> bin(13)
 |      '0b1101'
 |      >>> (13).bit_count()
 |      3
 |
 |  bit_length(self, /)
 |      Number of bits necessary to represent self in binary.
 |
```

```
 |      >>> bin(37)
 |      '0b100101'
 |      >>> (37).bit_length()
 |      6
 |
 |  conjugate(...)
 |      Returns self, the complex conjugate of any int.
 |
 |  is_integer(self, /)
 |      Returns True. Exists for duck type compatibility with
float.is_integer.
 |
 |  to_bytes(self, /, length=1, byteorder='big', *, signed=False)
 |      Return an array of bytes representing an integer.
 |
 |      length
 |        Length of bytes object to use.  An OverflowError is raised if the
 |        integer is not representable with the given number of bytes.
Default
 |        is length 1.
 |      byteorder
 |        The byte order used to represent the integer.  If byteorder is
'big',
 |        the most significant byte is at the beginning of the byte array.  If
 |        byteorder is 'little', the most significant byte is at the end of
the
 |        byte array.  To request the native byte order of the host system,
use
 |        `sys.byteorder' as the byte order value.  Default is to use 'big'.
 |      signed
 |        Determines whether two's complement is used to represent the
integer.
 |        If signed is False and a negative integer is given, an OverflowError
 |        is raised.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  from_bytes(bytes, byteorder='big', *, signed=False)
 |      Return the integer represented by the given array of bytes.
 |
 |      bytes
 |        Holds the array of bytes to convert.  The argument must either
 |        support the buffer protocol or be an iterable object producing
bytes.
 |        Bytes and bytearray are examples of built-in objects that support
the
 |        buffer protocol.
```

```
 |      byteorder
 |        The byte order used to represent the integer.  If byteorder is
'big',
 |        the most significant byte is at the beginning of the byte array.  If
 |        byteorder is 'little', the most significant byte is at the end of
the
 |        byte array.  To request the native byte order of the host system,
use
 |        `sys.byteorder' as the byte order value.  Default is to use 'big'.
 |      signed
 |        Indicates whether two's complement is used to represent the integer.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs)
 |      Create and return a new object.  See help(type) for accurate
signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number
```

Magic Methods

Wie man sieht, gibt es selbst für ein einfaches Integer-Objekt viele Methoden. Außerdem fällt auf, dass viele aber nicht alle Methoden mit Unterstrichen beginnen und enden. Diese Methoden werden auch als “Magic Methods” oder “Dunder Methods” (Double Underscore Methods) bezeichnet. Man kann sie direkt nutzen, indem man den Namen des Objekts gefolgt von einem Punkt und dem Namen der Methode eingibt:

```
# Absoluter Wert via Methode
var1.__abs__()
```

```
5
```

Obwohl es möglich ist, Magic Methods direkt aufzurufen, empfiehlt es sich in der Regel, die entsprechenden Python-Funktionen zu verwenden, die eine höhere Abstraktionsebene und oft eine klarere, intentionale Syntax bieten. Für den absoluten Wert eines Objekts wäre dies zum Beispiel die `abs()`-Funktion, anstatt der Magic Method `.__abs__()`:

```
# Absoluter Wert via Funktion
abs(var1)
```

```
5
```

Es ist wichtig zu verstehen, dass “normale” Methoden, die nicht mit doppelten Unterstrichen beginnen und enden, Teil der täglichen Arbeit mit Python sind und häufig verwendet werden, um Operationen mit Objekten durchzuführen. Die Zurückhaltung beim direkten Aufruf von Magic Methods dient dazu, die Lesbarkeit und Wartbarkeit des Codes zu verbessern und sich an etablierte Python-Konventionen zu halten.