Closures und Decorators

Closures

MicroPython kennt wie LUA sogenannte **Closures**. Grundlage dazu ist eine Funktion (**g**), die innerhalb einer umgebenden Funktion (**f**) deklariert ist und weitere Bedingungen erfüllen muss.

Beispiel 1:

```
t=8
def f(x):
    a=7
    def g(n):
        b=3
        y=5*b
        return y
```

Ausgabe:

```
>>> f(3)
>>> g(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g' isn't defined
```

Alle Objekte, die innerhalb einer Funktion deklariert sind, sind zu dieser Funktion **lokal** und von außerhalb der Funktion nicht referenzierbar. Das heißt, man kann lokale Variablen außerhalb der Funktion nicht in Ausdrücken verwenden und Funktionen, die innerhalb definiert sind, nicht außerhalb aufrufen.

In Beispiel1 sind **b** und **y** weder in **f** noch außerhalb von **f** bekannt und nur in **g** verfügbar. Nach der Beendigung von **g** werden beide Variablen eingestampft. Die Funktion **g** ist nur von **f** aus aufrufbar aber nicht von außerhalb **f**. Weil **g** keine freien Variablen von außerhalb oder von der Parameterliste nutzt, heißt **g abgeschlossen (closed term**).

Die Variable **t** wird als **global** angenommen und wird in Bezug auf **f** und **g** als freie Variable bezeichnet, weil sie außerhalb der Funktionen frei belegt werden kann. Ähnlich verhält es sich mit der Variable **a** in Bezug auf **g**. Die Parameter **x** und **n** gelten auch als freie Variablen.

Beispiel 2:

```
t=8
def f(x):
    a=7
    def g(n):
        b=3
        y=a*b+n
        return y
    return g
```

In MicroPython wird im Namen einer Funktion, allgemein eines Objekts, stets die Referenz auf den Funktionskörper (oder die Speicherstelle des Objekts) verwaltet. Daher kann eine Funktion auch die Referenz auf eine Funktion als Rückgabewert haben. Das ist in Beispiel 2 der Fall, wo die Funktion **f** die Referenz auf **g** zurückgibt. Die Schreibweisen **g** und **g(3)** darf man also nicht verwechseln. **g** enthält die Referenz auf den Speicherort der Funktion **g**, während **g(3)** den in **g** berechneten **Wert** von **y** zurückgibt, nämlich 24.

Im Beispiel 2 wird **g** als **offen** (**open term**) bezeichnet, weil es neben seinen eigenen, lokalen Variablen **b** und **y** auch die für **g** freien Variablen **a** und **n** referenziert. Mit dem Aufruf von **f** werden die Referenzen auf **a** und **n** in den **Scope** (aka **Namensraum**) von **g** mit einbezogen. Aus der offenen Funktion wird damit eine geschlossene Funktion, eine **Closure**. Und weil **f** die Referenz auf die Funktion **g** zurückgibt, ist die Funktion **g** jetzt auch nach dem Beenden von **f** verfügbar und kann aus der globalen Umgebung aufgerufen werden. Ein Nebeneffekt ist, dass auch die Variablen **a** und **n** gespeichert werden und nach dem Beenden von **f** im Scope von **g** verfügbar bleiben. Wir speichern die Rückgabe von **f** in der Variablen **h**. Geben wir **h** ein, sagt uns MicroPython, dass es sich bei **h** um eine **Closure** handelt. Als Referenz auf die Funktion **g** ist **h** als Funktion aufrufbar und liefert den Rückgabewert von **g(10), also 31** (7*3+10). Der Wert **7** für **a** ist fest gespeichert.

```
>>> h=f(3)
>>> h
<closure>
>>> h(10)
31
```

Es ist nicht einmal nötig, der Rückgabe von **f** einen Namen zuzuweisen.

```
>>> f(3)
<closure>
>>> f(3)(10)
31
(f(3))(10)
31
```

Gehen wir noch einen Schritt weiter und schließen wir den Parameter **x** der Funktion **f** in den Scope von **g** mit ein. Damit wird auch dieser Wert dauerhaft gespeichert, wenn die Funktion **f** verlassen wird.

Beispiel 3:

```
t=8
def f(x):
    a=7
    def g(n):
        y=a*x+n
        return y
    return g

>>> f(3)
<closure>
>>> f(3)(10)
31
>>> h=f(3)
31
>>> k=f(5)
>>> k(10)
45
```

In der Referenz von **h** wird für **x** der Wert 3 gespeichert, in der Closure **k** ist es der Wert **5**.

Einen hab' ich noch, einen hab' ich noch! Wir unterscheiden zwischen **globalen** Variablen wie **t**, und **lokalen** Variablen wie **a** für **f** oder, wie oben, **b** für **g**. Wollte man den Wert von **t** aus **f** heraus verändern, müsste man **t** in **f** als **global** deklarieren. Um **a** von **g** aus als nicht lokal zu kennzeichnen, verwendet man das Schlüsselwort **nonlocal**. Damit erreicht man, dass der Wert von **a** innerhalb der Funktion **g** verändert werden kann. Das funktioniert auch zusammen mit der Closure. Wir erhalten damit eine Art **automatischen Zähler**, um dessen Erhöhung wir uns nicht kümmern müssen.

Beispiel 4:

```
t=8
def f(x):
    global t
    a=7
    z=20
    def g(n):
        nonlocal a
        y=a*x+n
        a+=1
        return y
    return g
```

Eingaben und Ergebnisse:

```
>>> h=f(3)
>>> h(10)
31
>>> h(10)
34
>>> h(10)
37
>>> h(10)
40
```

Der Wert der Variablen **a** wird bei jedem Aufruf um 1 erhöht.

Fazit:

Eine Funktion **g**, die innerhalb des Funktionskörpers einer umgebenden Funktion **f** deklariert wird, ist ein **geschlossener Ausdruck** (**closed term)**, wenn innerhalb von **g** nur **lokale** Variablen (**bound variables**) referenziert werden.

Freie Variablen (**free variables**) sind in Bezug auf **g** alle, die außerhalb von **g** festgelegt werden. Dazu zählen auch die Parameter der Parameterliste von **g** sowie die Parameter der umschließenden Funktionen.

Die innere Funktion g ist ein **offener Ausdruck** (**open term**), so lange darin freie Variablen referenziert werden, die noch keinen Wert erhalten haben.

Beim Aufruf von **f** werden die freien, nicht lokalen Variablen, die innerhalb von **g** referenziert werden, in den Scope von **g** eingebunden und die Funktion **g** somit zu einem **geschlossenen Ausdruck** (**closed term**), einer **Closure**.

Wird eine, für die Funktion **g**, freie Variable (hier **a**) innerhalb **g** als nonlocal deklariert, dann kann der Wert dieser Variable von **g** aus nachhaltig verändert und im folgenden Durchgang erneut abgerufen werden.

Referenziert eine innere Funktion **g** keine der freien Variablen der umgebenden Funktionen dann handelt es sich von vorne herein um einen **closed term. g** stellt dann keine Closure dar und fängt damit auch nicht die Werte der freien Variablen. Eine Closure entsteht dann, wenn die umgebende Funktion aufgerufen und mindestens eine freie Variable in **g** referenziert wird.

Decoraters

Funktionsnamen enthalten, wie oben bereits erwähnt, nur eine Referenz auf den Funktionskörper im Speicher. Weisen wie dem Namen einer Funktion die Referenz auf eine andere Funktion zu, dann wird beim Aufruf dieses Namens statt der alten, die neue Funktion ausgeführt.

Definieren wir eine Funktion **toString** und zwei Funktionen **mal** und **plus** wie folgt.

Beispiel 5:

```
def toString(f):
    def g(x,y):
        r=str(f(x,y))
        print("Ergebnis",r)
        return r
    return g

def mal(x,y):
    return x*y

def plus(x,y):
    return x+y
```

Die innere Funktion **g** ist eine Closure, weil sie die Funktion **f** sowie die Parameter **x** und **y** der Parameterliste beim internen Aufruf die Funktion **f** mit den Argumenten **x** und **y** referenziert. Die umgebende Funktion **toString** gibt die Referenz auf **g** zurück. Die Funktionen **mal** und **plus** liefern die erwarteten Ergebnisse als Zahlenwert.

```
print(mal(2,3))
print(plus(2,3))
print(mal(2,3)+plus(2,3))
```

Ausgabe:

```
6
5
11
```

Jetzt übergeben wir mal und plus der Funktion **toString** als Argumente und weisen den gleichen Funktionsnamen die Ausgabe der Funktion **toString** zu. Das ist die Referenz auf die Closure **g**, welche die übergebene Funktion aufruft.

```
mal=toString(mal)
plus=toString(plus)

mal(2,3)
plus(2,3)
print(mal(2,3)+plus(2,3))
```

Ausgabe:
Ergebnis 6
Ergebnis 5
Ergebnis 6
Ergebnis 5
65

Die Berechnungen wurden wie erwartet durchgeführt, nur die Addition am Schluss ist statt des Summenwerts eine Verkettung der Strings "6" und "5". **toString** nimmt die Funktionsreferenzen auf **mal** beziehungsweise **plus** als Argument und weist den ursprünglichen Namen das Ergebnis der Closure zu. Der vorige Wert wird dadurch überschrieben. Wir haben die ursprünglichen Funktionen durch Anwendung von **toString** dekoriert und **toString** ist ein sogenannter **Decorator** für **mal** und **plus**.

Decorators sind Funktionen, die es erlauben das Erscheinungsbild der dekorierten Funktionen zu verändern, ohne diese Funktionen selbst zu verändern. Der Zugriff auf die dekorierten Funktionen ist dadurch nicht mehr direkt möglich. Natürlich wird beim Aufruf des Decorators **toString** mit dem Argument **f** eine Referenz auf die ursprüngliche Funktion gespeichert. Diese ist daher innerhalb von **g** aufrufbar. Um **mal** aufzurufen, rufen wir die zugehörige Closure, die ihrerseits dann die ursprüngliche Funktion ruft. Danach wird das Ergebnis ausgegeben und die Ziffernfolge des Ergebnisses als String zurückgegeben.

Dieses Verfahren lässt sich gut einsetzen, um zum Beispiel Funktionen oder die Methoden einer Klasse nachträglich zu dekorieren, ohne in die Funktions- oder Klassendefinition eingreifen zu müssen. Dadurch ist es möglich vor oder nach dem Aufruf der eigentlichen Funktion weitere Anweisungen ausführen zu lassen. Im folgenden Beispiel erweitern wir die Parameterliste der Funktionen **mal** und **plus** durch einen weiteren führenden Parameter, der einen Ausgabekanal angibt.

Beispiel 6:
```
class A:
    def __init__(self,N=0):
        self.n=N

    def mal(self, x,y=5):
        return y*x

def plus(x,y=5):
    return y+x


def deco(f):
    def g(k,*args,**kwargs):
        n=k
        print("Kanal:",n, "rufe",f.__name__)
        res=f(*args,**kwargs)
        return res
    return g
```

```
a=A(10)
a.mal=deco(a.mal)
plus=deco(plus)
print(a.mal(1,3))
print(a.mal(2,3,6))
print(plus(0,4,5))
```

Ausgabe:
```
Kanal: 1
15
Kanal: 2
18
Kanal: 0
9
```

Die Liste ***args** und das Dictionary ****kwargs** nehmen die Argumente auf, die an die durch **deco** dekorierte Funktion **f** übergeben werden sollen. Das ermöglicht die Verarbeitung von beliebig vielen Positions- und optionalen Parametern. Davor wurde **k** als Parameter für die Kanalnummer platziert. Innerhalb von **g** wird der Kanal registriert und ausgegeben und sodann die dekorierte Funktion aufgerufen und schließlich das Ergebnis der Berechnung zurückgegeben. Das Ganze funktioniert sowohl für die Methode **mal** als auch für die Funktion **plus**.

Syntaktischer Zucker

Als syntaktischen Zucker bezeichnet man Schreibweisen, die Anweisungen verkürzen oder besser lesbar machen. Dazu zählen zum Beispiel die verkürzten Schreibweisen für arithmetische Operationen

a=a+1 wird zu
a+=1

Für das Dekorieren von Funktionen gibt es die **pie**-Schreibweise. Statt plus = deco(plus) schreiben wir **@deco** direkt über die zu dekorierende Funktion. Natürlich muss **deco** zuvor deklariert werden.

```
def deco(f):
    def g(k,*args,**kwargs):
        n=k
        print("Kanal:",n, "rufe",f.__name__)
        res=f(*args,**kwargs)
        return res
    return g

@deco
def plus(x,y=5):
    return y+x
```

Das funktioniert aber nur für Funktionen auf derselben Ebene. Um die Methode **mal** aus der Klasse **A** zu dekorieren, bedarf es eines Tricks. Wir verstecken den Aufruf der Methode in einer globalen Funktion.

```
a=A(10)

@deco
def mal(x,y=6):
    return a.mal(x,y)

print(plus(1,2,3))
print(mal(0,2,3))
```

Ausgabe:

```
Kanal: 1 rufe plus
5
Kanal: 0 rufe mal
6
```

Decorators können auch kaskadiert werden.

```
@deco2
@deco
def plus(x,y=5):
    return y+x
```

steht für plus = deco2(deco(plus))