Michael Inden

Einfach Python

Gleich richtig programmieren lernen

dpunkt.verlag

`if` und `else` in Kombination

Oftmals möchte man nicht nur eine Bedingung mit `if` prüfen, sondern auch auf deren Nichterfüllung reagieren können. Das wird bereits an den vorherigen Beispielen deutlich. Modifizieren wir die Prüfung und nutzen dort eine Wertzuweisung statt einer Konsolenausgabe:

```
>>> user = "Michael"
>>> if user == "Michael":
...     print("Access granted")
... else:
...     print("You have invalid rights!")
...
Access granted
```

Wir sehen, dass diese Darstellung etwas lang wird. Für derart einfache Fallunterscheidungen existiert eine Kurzschreibweise, die folgende Struktur besitzt:

```
positiveCase if condition else negativeCase
```

Das kann man nutzen, um eine Zuweisung oder Aktion wie zuvor etwas kürzer wie folgt zu schreiben:

```
>>> result = "allowed" if age > 18 else "NOT ALLOWED"
>>> result
'allowed'
```

Allerdings kann dies auch recht schnell zu schlechterer Lesbarkeit führen. Die Thematik behandle ich in Abschnitt 6.4 genauer.

`if`, `elif` und `else` in Kombination

Oftmals sollen neben einer Bedingung mit `if` noch weitere Prüfungen erfolgen können. Im nachfolgenden Beispiel wollen wir abhängig von einer Uhrzeit einen entsprechenden Gruß aufbereiten, also etwa »Good morning«, wenn es noch nicht 12 Uhr ist, oder aber »Good evening« ab 18 Uhr, aber vor 22 Uhr. Das können wir wie folgt mit `if`, `elif` und `else` in Kombination umsetzen – `elif` steht für `else if` in anderen Sprachen.

Wir wechseln hier in die PyCharm-IDE, weil diese das Editieren von mehrzeiligen Anweisungen komfortabel erlaubt, was eine Schwäche vom Python-Kommandozeileninterpreter ist:

```
time = 15
if time < 12:
   print("Good morning")
elif time < 18:
   print("Good afternoon")
elif time < 22:
   print("Good evening")
else:
   print("Good night")
Good afternoon
```

Im Beispiel ist die Uhrzeit 15 nicht kleiner als 12, sodass die erste Bedingung `False` ist. Dagegen ist die zweite Bedingung `True`, sodass »`Good afternoon`« ausgegeben wird. Wäre die Uhrzeit nach 18 Uhr und vor 22 Uhr, erhielten wir »`Good evening`« und ansonsten resultiert es in »`Good night`« als Ausgabe.

Rekapitulation

Insgesamt bietet Python die folgenden bedingten Anweisungen:

- Verwenden Sie `if`, um einen Codeblock anzugeben, der ausgeführt werden soll, wenn eine bestimmte Bedingung wahr ist.
- Mit `else` geben Sie einen Codeblock an, der ausgeführt werden soll, wenn die gleiche Bedingung falsch ist.
- Verwenden Sie `elif`, um eine weitere alternative Bedingung anzugeben, die getestet wird, wenn die erste Bedingung falsch ist.

Tipp: Kein `switch` in Python

Während diverse andere Sprachen, wie Java oder C++, noch das Schlüsselwort `switch` bieten, um mehrere alternative Codeblöcke anzugeben, die bedingt ausgeführt werden sollen, existiert dieses in Python leider nicht. Es lässt sich aber durch die zuvor gezeigten Konstrukte nachahmen. Mit Python 3.10 kommt das Schlüsselwort `match`, das deutlich mächtiger ist als `switch` und kurz in Anhang C beschrieben wird.

2.5 Funktionen

Wir haben schon einiges Basiswissen zusammengetragen. Nehmen wir einmal an, wir wollten eine Art Taschenrechner programmieren bzw. in unserem Programm immer wieder ähnliche Berechnungen anstellen. Obwohl es etwa bei einfachen Additionen noch problemlos möglich ist, wiederholt die gleichen Zeilen zu schreiben, ändert sich die Situation bereits bei einer Funktionalität, die ein paar Vergleiche oder andere Logik enthält. Um Funktionalität unter einem Namen zu bündeln, existieren Funktionen.

Funktionen definieren

Eine Funktion ist ein benanntes Stück Sourcecode, also eine Folge von Anweisungen, die nur dann ausgeführt werden, wenn die Funktion aufgerufen wird. An diese können Sie Eingabedaten, sogenannte Parameter, übergeben. Funktionen werden verwendet, um bestimmte Aktionen auszuführen und dies auf wiederverwendbare Art und Weise. Das Ganze besitzt folgende Syntax:

```
def funktion_name(Parameter1, Parameter2, ...):
    Anweisung1
    Anweisung2
```

Definieren Sie eine Funktion also einmal und verwenden Sie die Aktionen viele Male überall dort im Python-Programm, wo dies benötigt wird.

Funktionen aufrufen

Um eine Funktion und damit die dort enthaltenen Anweisungen auszuführen, nutzen wir deren Namen, eine öffnende runde Klammer, optional eine kommaseparierte Liste von Parametern und zum Abschluss eine schließende runde Klammer. Für zwei Parameter sieht das wie folgt aus:

```
funktion_name(parameterWert1, parameterWert2)
```

Klingt noch etwas abstrakt, im Anschluss machen wir das Ganze konkreter mit der Standardfunktionalität `print()` und mit drei Parametern:

```
print("Michael", "likes", "Python")
```

2.5.1 Eigene Funktionen definieren

Greifen wir das Beispiel des einfachen Taschenrechners wieder auf. Schauen wir uns als Beispiele die Funktionen zur Addition und Multiplikation an:

```
>>> def add(value1, value2):
...     return value1 + value2
...
>>> def mult(value1, value2):
...     return value1 * value2
...
>>> result = add(7, 2)
>>> mult(result, 3)
27
```

Genauso wäre es möglich, die in den vorher gezeigten Altersprüfungen genutzte Abfrage als verständliche Funktion bereitzustellen:

```
>>> def is_adult(age):
...     return age >= 18
...
```

Wir erkennen Folgendes: Funktionen bestehen aus einem Block mit einer oder mehreren Anweisungen. Um beim Aufruf Informationen übergeben zu können, dienen Parameter, die wiederum Typen besitzen. Insgesamt erlauben es uns Funktionen, wiederkehrende Funktionalität zu bündeln und über den Namen anzusprechen, statt die Sourcecode-Zeilen im Programm wiederholen zu müssen.

Spezialfall: Keine Rückgabe

Aufgerufene Funktionen dienen oftmals dazu, eine Aufgabe auszuführen, die ein Ergebnis berechnet und dieses dem Aufrufer zurückliefert, etwa für eine Addition. Dazu wird das Schlüsselwort `return` genutzt. Manchmal, etwa wie bei der Funktion `print()`, erhalten wir keine Rückgabe. In derartigen Fällen gibt es dann allerdings kein `return` mit Wertangabe, sondern nur eine oder mehrere Anweisungen:

```
>>> def greet(name):
...     print("Hello", name)
...
>>> greet("Michael")
Hello Michael
```

Kombination mit Bedingungen

Im nachfolgenden Beispiel wollen wir als wiederverwendbare Funktionalität das Minimum von drei Ganzzahlen ermitteln, zu deren Repräsentation wir `x`, `y` und `z` als Parameternamen verwenden.[3]

```
def min_of_3(x, y, z):
    if x < y:
        if x < z:
            return x
        else:
            return z
    else:
        if y < z:
            return y
        else:
            return z
```

Mal ganz ehrlich: Sind Sie allein beim Betrachten sicher, dass sich dort kein Flüchtigkeitsfehler eingeschlichen hat? Zumindest der Kontrollfluss ist doch ein wenig unübersichtlich. Einfacher geht das Ganze, wenn man vordefinierte Bausteine verwendet, was wir uns im Anschluss anschauen. Zuvor probieren wir die obige Implementierung der Funktionalität einmal für verschiedene Wertekombinationen aus:

```
min_of_3(1,2,3)
1
min_of_3(11,2,3)
2
min_of_3(11,22,3)
3
```

[3] Alternativ hätte man auch die etwas besser lesbaren Namen `value1`, `value2` und `value3` verwenden können. Da wir uns hier im mathematischen Kontext bewegen, scheinen mir die Einbuchstabenvariablen gleichfalls adäquat.

Funktionen als wiederverwendbare Bausteine

Wir könnten beispielsweise mit folgender Implementierung der Funktion `min_of()` zur Berechnung des Minimums für zwei Zahlen starten:

```
def min_of(x, y):
    if x < y:
        return x
    else:
        return y
```

Tatsächlich ließe sich das mit dem ternären Operator noch kürzer schreiben:

```
def min_of_shorter(x, y):
    return x if x < y else y
```

Wenn man diese Bausteine besitzt, kann man die ursprüngliche Funktion `min_of_3()` und das komplizierte `if`-Gebilde folgendermaßen vereinfachen:

```
def min_of_3(x, y, z):
    return min_of(x, min_of(y, z))
```

Überprüfen wir kurz, ob die gleichen Werte wie zuvor geliefert werden:

```
min_of_3(1,2,3)
1
min_of_3(11,2,3)
2
min_of_3(11,22,3)
3
```

Perfekt, funktional ist das gleichwertig, aber es war durch den Einsatz der Funktionen viel weniger Aufwand und ist deutlich kürzer. Auch die Verständlichkeit ist viel besser, da wir nicht mehrfach verschachtelte `if`-Gebilde nachvollziehen müssen.

Nebenbei ist es immer eine gute Idee, sich ein wenig in den Python-Basisbausteinen umzuschauen, weil es dort eine Vielzahl an nützlichen Funktionalitäten bereits vordefiniert gibt. In unserem Fall könnten wir `min()` zur Berechnung des Minimums nutzen.

2.5.2 Nützliche Beispiele aus Python

Wir haben bereits die praktische Funktion `print()` kennengelernt. Betrachten wir, was man beispielsweise für mathematische Funktionalitäten sonst noch so in Python findet:

- `max(x,y)` – Die Funktion `max(x,y)` liefert das Maximum, also den höchsten Wert von `x` und `y`.
- `min(x,y)` – Die Funktion `min(x,y)` ermittelt das Minimum, also den kleinsten Wert von `x` und `y`.
- `abs(x)` – Die Funktion `abs(x)` gibt den Absolutwert von `x` zurück: Falls `x` negativ ist, dann ist das der positive Wert.