Empfohlene Unterrichtsinhalte als Vorbereitung für dieses Thema: 02-01/2/3/4/5 (Variablen, Strings, Listen, Integers und Floats, Bool)

Einführung in die computerlinguistische Programmierung mit Python

05-01: Funktionen in Python ⚙

Inzwischen haben wir schon eine Menge Programmiergrundlagen behandelt und sind bereit, komplexere Aufgaben zu bearbeiten. Funktionen bieten uns die Möglichkeit, Sinnabschnitte unserer Programme vom Rest abzukapseln und so den Überblick zu bewahren.

Stellen wir uns beispielsweise vor, dass wir Quadratzahlen berechnen wollen. Bisher haben wir das etwa so gemacht:

In [1]:
```
input_number = 6                                # Variable erzeugen und mit dem

result = input_number * input_number            # Quadratzahl berechnen
print(result)                                   # Ergebnis ausgeben
```

Wenn wir genau eine Quadratzahl berechnen wollen, ist gegen den Code oben nichts einzuwenden. Wenn wir die Berechnung aber mehrmals für unterschiedliche Werte durchführen wollen, lohnt es sich, den Code dafür allgemeiner zu schreiben. Dann können wir die Funktionalität "Quadratzahl berechnen" vom restlichen Code abkapseln und jederzeit aufrufen, wenn wir sie brauchen.

In [1]:
```
def calculate_square(input_number):          # Funktionsdefinition eröffnen
    result = input_number * input_number     # beliebig viele Befehle
    return result                            # Funktionsdefinition beenden
```

Die Einrückung zeigt, welche Zeilen zur Funktionsdefinition gehören. Der Kopf der Funktion enthält den Namen (hier: `calculate_square`) und eine Angabe aller Parameter, die wir in der Funktion brauchen. (Erinnert euch an die Parameter in der Sitzung am 09.11., z.B. `s.split(".", 2)`)

In der Beispielfunktion heißt der einzige Parameter `input_number`.

Im Körper der Funktion folgen beliebig viele Befehle, in denen das Argument (der konkrete Wert, der als Parameter verwendet wird) verarbeitet wird. Beispielsweise berechnen wir oben das Quadrat der übergebenen Zahl. Das Ergebnis wird hier in der Variable result gespeichert.

Schließlich gibt die Funktion mit `return` einen Wert zurück. Im Beispiel ist das der Wert, der in der Variable result gespeichert ist. Da return die Funktion sofort beendet, werden keine eingerückten Zeilen unterhalb von return mehr ausgeführt.

Wir können uns Funktionen wie eine Art Rezept vorstellen, nach dem bestimmte Aufgaben gelöst werden. Die Funktionsdefinition, die wir oben sehen, braucht noch einen konkreten Eingabewert (ein Argument für den Parameter $\text{input}_{\text{number}}$), damit ein Ergebnis berechnet werden kann.

Erst, wenn die Funktion im Code mit einem konkreten Argument aufgerufen wird, werden die

Befehle innerhalb des Funktionskörpers ausgeführt. Das Ergebnis - der Wert, der in der letzten Zeile der Funktion zurückgegeben wird - kann dann an der Stelle im Code, wo die Funktion aufgerufen wurde, verwendet werden.

Beachtet, dass der Interpreter eine Funktion gelesen haben muss, bevor sie ausgeführt werden kann. Eine Funktion muss also immer oberhalb ihres ersten Aufrufs im Hauptprogramm definiert werden.

Probiert es aus: Führt den Code in der nächsten Zelle aus und beachtet, in welcher Reihenfolge die print()-Befehle ausgeführt werden.

In [1]:
```
print("1")

def calculate_square(input_number):
    print("2")
    result = input_number * input_number
    return result

print("3")

my_square = calculate_square(4)     # das Ergebnis des Funktionsaufrufs calc
                                    # wird in der Variable my_square gespei

print("Ergebnis der Berechnung: " + str(my_square))

print("4")
```

Lokale Variablen in Funktionen

Im Gegensatz zu z.B. Schleifen sind Variablen, die in Funktionskörpern definiert werden, außerhalb der Funktion nicht mehr verfügbar. Probiert es aus: Ergänzt im Beispiel unterhalb des bisherigen Codes einen `print()` -Aufruf für `result` oder `input_number` . Das Programm stürzt ab.

Grundsätzlich kann während der Funktionsausführung auf Variablen zugegriffen werden, die außerhalb der Funktion definiert wurden, wenn keine Variable mit dem gleichen Namen in der Funktion definiert wird. Aus Gründen der Lesbarkeit und Wiederverwendbarkeit der Funktion sollten wir aber alle Parameter einer Funktion explizit definieren. Im Beispiel unten könnten wir den Parameter `power` ergänzen und zur Berechnung der Potenz nutzen.

In [1]:
```
##########################
# Achtung Negativbeispiel #
##########################

def calculate_power(input_number):
    # n = 4                                     # Nanu? Was ist das für ein
    nth_power_of_input = input_number ** n      # Ah, das soll wohl der Expo
    # n = 5                                     # Das ist nicht der richtige
    return nth_power_of_input

n = 3                                           # Nanu? Was ist das für ein

nth_power = calculate_power(4)
print(nth_power)
```

Mehrere `return` -Statements

Wir können Funktionen mit mehreren `return` -Statements versehen um beispielsweise abhängig vom Input den Rechenaufwand für unser Programm zu verringern. Unsere Funktion von oben können wir verbessern, indem wir eine Fallunterscheidung einführen in der geprüft wird, ob der Exponent `0` ist. In diesem Fall muss nicht gerechnet werden, da der Rückgabewert immer `1` ist. Weitere Fälle, die wir unterscheiden können, sind diese in denen die Basis `0` ist oder der Exponent `1` . Nur falls keiner der genannten Fälle zutrifft, wird tatsächlich ein Wert berechnet und zurückgegeben.

In [1]:
```
def calculate_power(input_number, exponent):
    if exponent == 0:
        return 1
    elif input_number == 0:
        return 0
    elif exponent == 1:
        return input_number
    else:
        nth_power_of_input = input_number ** exponent
        return nth_power_of_input
```

Es ist möglich Funktionen ohne `return` -Statement zu definieren. Der Rückgabewert ist dann immer `None` . **Wir wollen grundsätzlich Funktionen mit `return` -Statement schreiben.** Das ist eine gute Konvention und zeigt, dass die Funktion tatsächlich vollständig definiert ist. Falls wir eine Funktion definieren wollen, die tatsächlich keine Rückgabewert benötigt, dann können wir dies mit `return None` ebenfalls tun.

Don’t Repeat Yourself

Mit Funktionen können wir Redundanz (unnötige Wiederholungen) in unserem Code vermeiden. Jedesmal, wenn wir später im Programm eine Quadratzahl berechnen wollen, können wir die eben definierte Funktion verwenden, statt immer wieder die gleichen Codezeilen an verschiedenen Stellen einzufügen. Die Beispielfunktion ist sehr übersichtlich, aber wenn eine Funktion eine komplexere Aufgabe erfüllt - z.B. Daten nach bestimmten Kriterien zu filtern - , wollen wir vermeiden, Codezeilen mehrfach zu schreiben.

Codewiederholungen sind eine typische Fehlerquelle. Außerdem machen sie unser Programm länger und dadurch unübersichtlicher. Mit Funktionen vermeiden wir solche Wiederholungen.

Mithilfe von Funktionen können wir Programmcode, der vorher unübersichtlich war, auslagern. Wir geben Funktionen dafür sprechende Namen und lassen jede Funktion genau eine Aufgabe erfüllen.

Zusammenfassung:

- Jede Funktion soll genau eine Aufgabe erfüllen.
- Funktionsdefinitionen müssen im Code immer oberhalb der dazugehörigen Funktionsaufrufe stehen.

- Eine Funktion kann mehrere `return` -Statements enthalten, die in unterschiedlichen Fällen greifen. **Achtung: Die Funktion soll in jeder möglichen Verwendung eine**

Rückgabe liefern.

Weitere Themen dieser Woche

- None
- Module
- Namenskonflikte