LU10b - Python-Module

Es gibt eigentlich drei verschiedene Möglichkeiten, ein Modul in Python zu definieren:

- Ein Modul kann in Python selbst geschrieben werden.
- Ein Modul kann in C geschrieben und zur Laufzeit dynamisch geladen werden, wie das `re` (regular expression) Modul.
- Ein eingebautes Modul ist von Haus aus im Interpreter enthalten, wie das Modul `itertools`.

Der Zugriff auf den Inhalt eines Moduls erfolgt in allen drei Fällen auf die gleiche Weise: mit der `import`-Anweisung.

Hier geht es hauptsächlich um Module, die in Python geschrieben sind. Das Tolle an in Python geschriebenen Modulen ist, dass sie äußerst einfach zu erstellen sind. Alles, was Sie tun müssen, ist, eine Datei zu erstellen, die legitimen Python-Code enthält, und der Datei einen Namen mit der Erweiterung `.py` zu geben. Das war's! Es ist keine spezielle Syntax oder Voodoo nötig.

Nehmen wir zum Beispiel an, Sie haben eine Datei namens `mod.py` erstellt, die Folgendes enthält:

mod.py

```
""" example of a module with various
stuff """
s = "If Comrade Napoleon says it, it
must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass
```

Mehrere Objekte sind in mod.py definiert:

- `s` (eine Zeichenkette)
- `a` (eine Liste)
- `foo()` (eine Funktion)
- `Foo` (eine Klasse)

Vorausgesetzt, `mod.py` befindet sich an einem geeigneten Ort, über den Sie gleich mehr erfahren werden, kann auf diese Objekte zugegriffen werden, indem das Modul wie folgt importiert wird:

```
""" example for using a module """
```

```
import mod

print(mod.s)
#If Comrade Napoleon says it, it must be
right.

print(mod.a)
#[100, 200, 300]

mod.foo(['quux', 'corge', 'grault'])
#arg = ['quux', 'corge', 'grault']

x = mod.Foo()
print(x)
#<mod.Foo object at 0x03C181F0>
```

Der Modulsuchpfad

Fahren wir mit dem obigen Beispiel fort und schauen wir uns an, was passiert, wenn Python die Anweisung ausführt:

`import mod` Wenn der Interpreter die obige Import-Anweisung ausführt, sucht er nach `mod.py` in einer Liste von Verzeichnissen, die aus den folgenden Quellen zusammengestellt wurde:

- Das Verzeichnis, aus dem das Eingabeskript ausgeführt wurde
- Die Liste der Verzeichnisse, die in der Umgebungsvariablen PYTHONPATH enthalten sind, falls diese gesetzt ist. (Das Format für PYTHONPATH ist betriebssystemabhängig, sollte aber die PATH-Umgebungsvariable nachahmen).
- Eine installationsabhängige Liste von Verzeichnissen, die zum Zeitpunkt der Installation von Python konfiguriert wurde

Der resultierende Suchpfad ist in der Python-Variablen `sys.path` zugänglich, die von einem Modul namens `sys` bezogen wird:

```
import sys
print(sys.path)
#['',
'C:\\Users\\peter\\Documents\\Python\\doc',
'C:\\Python36\\Lib\\idlelib','C:\\Python310\\
python310.zip', 'C:\\Python310\\DLLs',
'C:\\Python310\\lib','C:\\Python310',
'C:\\Python310\\lib\\site-packages']
```

Der genaue Inhalt von `sys.path` ist installationsabhängig.

Die obigen Angaben werden auf Ihrem Computer höchstwahrscheinlich etwas anders aussehen.

Um also sicherzustellen, dass Ihr Modul gefunden wird, müssen Sie eine der folgenden Maßnahmen ergreifen:

- Legen Sie `mod.py` in das Verzeichnis, in dem sich das aufrufende Skript befindet.
- Ändern Sie die Umgebungsvariable `PYTHONPATH` so, dass sie das Verzeichnis enthält, in dem sich `mod.py` befindet, bevor Sie den Interpreter starten.
  - **Oder:** Legen Sie `mod.py` in einem der Verzeichnisse ab, die bereits in der `PYTHONPATH`-Variable enthalten sind
- Legen Sie `mod.py` in eines der installationsabhängigen Verzeichnisse, auf die Sie je nach Betriebssystem Schreibzugriff haben oder nicht

Zur Laufzeit, manuell

Sie können die Moduldatei in ein beliebiges Verzeichnis Ihrer Wahl ablegen und dann `sys.path` zur Laufzeit so ändern, dass es dieses Verzeichnis enthält. In diesem Fall könnten Sie z.B. `mod.py` in das Verzeichnis `C:\Users\peter` ablegen und dann die folgenden Anweisungen eingeben:

```
sys.path.append(r'C:\Users\peter')
print(sys.path)
#['', 'C:\\Users\\peter\\Documents\\Python\\doc',
'C:\\Python36\\Lib\\idlelib', 'C:\\Python36\\python36.zip',
'C:\\Python36\\DLLs', 'C:\\Python36\\lib', 'C:\\Python36',
'C:\\Python36\\lib\\site-packages', 'C:\\Users\\peter']

import mod
```

Zur Laufzeit, automatisch via Skript

Sie können das Verzeichnis des aktuell ausgeführten Skripts direkt in `sys.path` einfügen. Dies ist besonders praktisch, wenn Sie möchten, dass Python automatisch alle Module im gleichen Verzeichnis wie das ausgeführte Skript findet. Dazu können Sie das Modul `os` verwenden, um den Pfad des aktuellen Skripts zu ermitteln und diesen dann `sys.path` hinzuzufügen. Hier ein Beispiel:

```
import sys
import os

# Fügt den Pfad des aktuellen Skripts zu
sys.path hinzu
script_directory =
os.path.dirname(os.path.abspath(__file__))
```

```
sys.path.append(script_directory)

print(sys.path)
# Die Ausgabe enthält nun auch das
Verzeichnis des aktuellen Skripts

import mod
```

Durch das Hinzufügen des Skriptverzeichnisses zu `sys.path` können Sie sicherstellen, dass alle Module, die sich im gleichen Verzeichnis wie Ihr Skript befinden, von Python gefunden und importiert werden können.

Die import-Anweisung

Modulinhalte werden dem Aufrufer mit der `import`-Anweisung zur Verfügung gestellt. Die `import`-Anweisung kann viele verschiedene Formen annehmen, wie unten gezeigt.

import <Modul_Name>

Die einfachste Form ist die bereits oben gezeigte:

```
import <modul_name>
```

Beachten Sie, dass dies den Inhalt des Moduls für den Aufrufer nicht direkt zugänglich macht. Jedes Modul hat einen eigenen Namensraum. Die Anweisung `import <Modulname>` platziert nur `<Modulname>` im Namensraum des Aufrufers. Die Objekte, die im Modul definiert sind, bleiben im privaten Namensraum des Moduls. Für den Aufrufer sind die Objekte des Moduls nur dann zugänglich, wenn mit `<modul_name>.<object>` darauf zugegriffen wird:

Trotz des Imports bleiben `s` und `foo` bleiben im privaten Namensraum des Moduls und sind im lokalen Kontext nicht von Bedeutung:

```
import mod

print(s)
# NameError: name 's' is not defined
foo('quux')
#NameError: name 'foo' is not defined
```

Um im lokalen Kontext zugänglich zu sein, muss den Namen

der im Modul definierten Objekte das Kürzel mod vorangestellt werden:

```
""" example for using the module "mod" """
import mod

print(mod.s)
#If Comrade Napoleon says it, it must be
right.
mod.foo('quux')
#arg = quux
```

import <module_name> as <alt_name>

Sie können auch ein ganzes Modul unter einem alternativen Namen importieren:

```
import <module_name> as <alt_name>

import mod as my_module

print(my_module.a)
#[100, 200, 300]
my_module.foo('qux')
#arg = qux
```

from <module_name> import <name(s)>

Eine alternative Form der `Import`-Anweisung erlaubt es, einzelne Objekte aus dem Modul direkt in den Namensraum des Aufrufers zu importieren:

```
from <module_name> import <name(s)>
```

Nach Ausführung der obigen Anweisung kann in der Umgebung des Aufrufers auf <name(s)> ohne den Präfix <module_name> verwiesen werden:

```
from mod import s, foo

print(s)
#If Comrade Napoleon says it, it must be
```

```
right.

foo('quux')
#arg = quux
```

Da bei dieser Form des Imports die Objektnamen direkt in den Namensraum des Aufrufers eingetragen werden, werden bereits vorhandene Objekte mit demselben Namen überschrieben:

```
a = ['foo', 'bar', 'baz']
print(a)
#['foo', 'bar', 'baz']

from mod import a
print(a)
#[100, 200, 300]
```

from <module_name> import <name> as <alt_name>

Es ist auch möglich, einzelne Objekte zu importieren, diese aber mit alternativen Namen im lokalen Namensraum einzutragen:

```
from <module_name> import <name> as
<alt_name>[, <name> as <alt_name> …]
```

Dadurch ist es möglich, Namen direkt in den lokalen Namensraum einzugeben, ohne dass es zu Konflikten mit bereits vorhandenen Namen kommt:

```
s = 'foo'
a = ['foo', 'bar', 'baz']

from mod import s as string, a as alist

print(s)
#'foo'
print(string)
#'If Comrade Napoleon says it, it must be
right.'
print(a)
#['foo', 'bar', 'baz']
print(alist)
```

```
#[100, 200, 300]
```

Inhalt von RealPython