Lukas Müller

Programmieren für Studierende der Naturwissenschaften

V6 – Imports und NumPy

Modularisierung

- Ein Modul ist ein abgeschlossener Teil einer Software, bestehend aus mehreren Teilprogrammen (Prozeduren und Funktionen) und Datenstrukturen

- Module sind ein Mittel zur Kapselung (encapsualation) von Software, d.h. Trennung von „Schnittstelle“ und Implementierung und Schutz vor „unkontrollierter“ Fehlerausbreitung

- Programme oder Programmteile werden wiederverwendbar, ohne dass Code redundant erstellt und auch gepflegt werden muss

Modularisierung

- Größere, komplexe Programme können durch den Einsatz von Modulen gegliedert und strukturiert werden. Funktionalitäten können nach dem Baukastenprinzip eingebunden werden

- Mehrere Entwickler*innen(-gruppen) können nach erfolgter Schnittstellendefinition unabhängig voneinander einzelne Module bearbeiten und testen

Modularisierung

- Viele Programmiersprachen unterstützen das Modulkonzept durch integrierte Sprachmittel, so auch Python.

- Module können in vielen Programmiersprachen separat kompiliert und in Form von Programmbibliotheken bereitgestellt werden

Module in Python

- Ein Modul ist ein "Behälter", der Objekte enthält. Module definieren einen sogenannten Namensraum
- Ein Modul entsteht dadurch, dass Sie in Python ein Programm mit der Endung .py unter einem Namen abspeichern
- Mit der Anweisung import <Dateiname> werden Module für das aktuelle Programm verfügbar gemacht

Module in Python

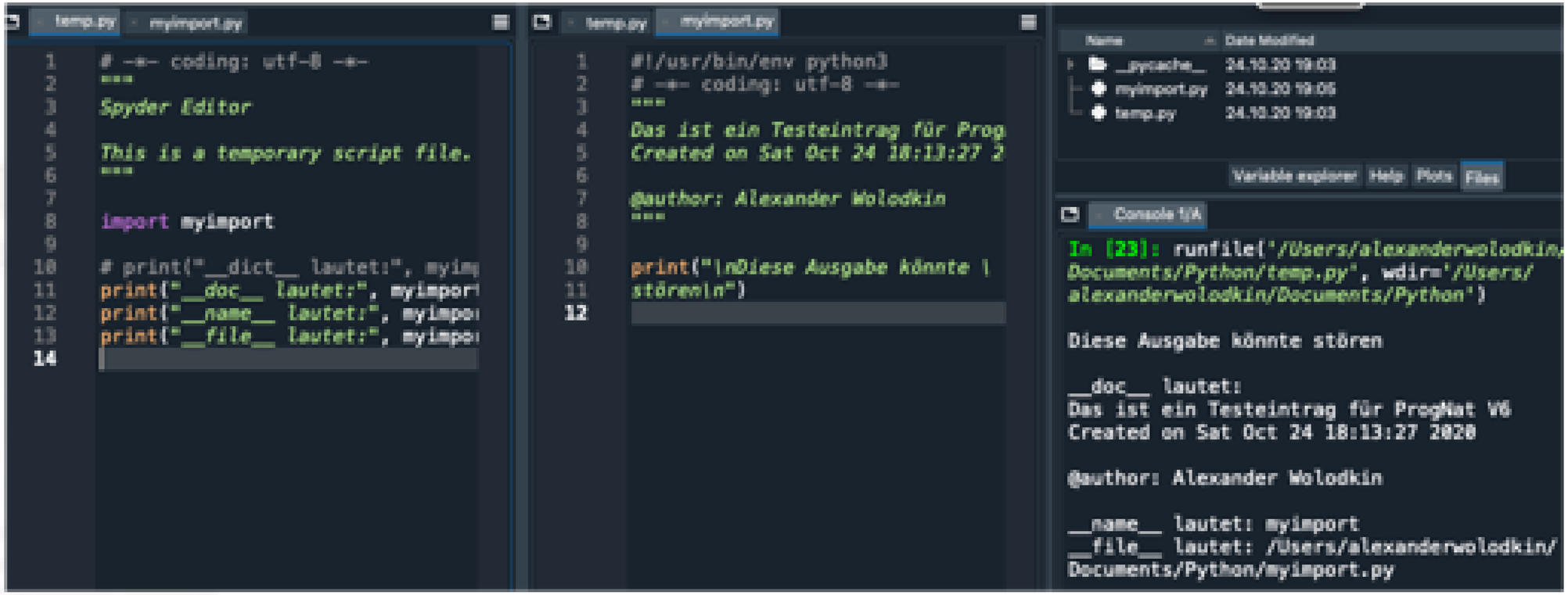
- Module sollen kurze Namen haben, alle in Kleinbuchstaben, keine Sonderzeichen (insbesondere keine Punkte!). Underscores nur dann, wenn es die Lesbarkeit erhöht

- Größere Python Programme werden oft als Paket von Modulen organisiert. Python Paketnamen nutzen nur Kleinbuchsstaben (kein Underscore)

Automatisch generierte Attribute eines Moduls m

Attribut	Beschreibung
m.__dict__	Das Dictionary, welches zum Modul gehört und insbesondere die Namensverwaltung unterstützt
m.__doc__	Doc-String des Moduls
m.__name__	Name des Moduls
m.__file__	Datei, aus der das Modul geladen wurde

Attribute des Moduls



Import eines Moduls

`import module`	Zugriff auf Funktionen des Moduls durch Qualifizierung: **`module.Funktionsname`**
`from module import name [,name]*`	Zugriff auf die **Funktion** `name` des Moduls einfach durch `name`
~~`from module import *`~~	~~Alle benutzten Namen werden importiert und können unqualifiziert benutzt werden~~

- **Wegen potentieller Namenskonflikte die letzte Variante nicht nutzen!**
- **Alternativ:** `import module as <Spitzname>`

Was passiert beim `import`?

- Die Moduldatei wird gesucht und hoffentlich gefunden
- Der Code wird in Byte Code übersetzt (wenn nötig)
- Das Modul wird einmal ausgeführt d.h. der Code auf „oberster Ebene“ des Moduls (aber nur einmal, beim ersten import !)
- Dabei werden die Namen des Moduls bekannt gemacht (wie ein „Run Module“ oder F5 in IDLE)

Die Sache mit dem `main()`

- Häufig findet man in Python Programmen/Modulen das Statement

```
if __name__ == "__main__":
    <Code>
```

- Was soll das? Wofür braucht man das?
- Jeder Python-Code kann zur Laufzeit erfragen, wie es heißt, indem es die Variable `__name__` ausliest
- Diese ist
  - `== '__main__'`, wenn es vom Betriebssystem, von der Console aus oder in IDLE mit Run Module (F5) gestartet wurde
  - `== <eigener Modulname>`, wenn es importiert wurde

main() - Beispiele

```
print('Hallo, hier ist das Modul testfile.')
print('Bei mir ist __name__ auf',__name__,'gesetzt.')

def sr_in_test ():
  print('Hier läuft sr_in_test .' )
  print('Bei mir ist __name__ auf',__name__,'gesetzt.')



def main():
  print('Hier könnte z.B.Initialisierungscode für das Programm testfile \
        stehen, wenn es als selbstständiges Programm aufgerufen wird.')


if __name__ =='__main__':
  main()
```

main() - Beispiele

```
>>>
========================= RESTART: /home/alina/testfile.py ===============
========
Hallo, hier ist das Modul testfile.
Bei mir ist __name__ auf __main__ gesetzt.
Hier könnte z.B.Initialisierungscode für das Programm testfile
stehen, wenn es als selbstständiges Programm aufgerufen wird.
>>> import testfile
Hallo, hier ist das Modul testfile.
Bei mir ist __name__ auf testfile gesetzt.
>>>
```

Warum schreiben wir das in eine separate Funktion statt in die if-Bedingung?

Antwort:

- Dies als Unterprogramm zu schreiben ist etwas sicherer, weil so keine „globalen" Variablen für testfile versehentlich erzeugt werden können

- Außerdem kann man so auch für den Fall dass das Programm von außen aufgerufen wurde, den Code noch einmal besuchen

Zusammenfassung

- Zusammengefasst: Mit einem .py - File kann man folgendes machen:
  - als Modul importieren und dann in einem anderen Programm nutzen
    - alle Statements (außer hinter def und class) werden (einmal) ausgeführt
  - als Skript (Programm) starten und nutzen:
    - Hier hat man ggf. Initialisierungscode, den man nur braucht, wenn es als Skript/Hauptprogramm ausgeführt wird
-
```
if __name__ == "__main__":
    main()
```

- Ist also eine Weiche, die erkennt, in welcher Umgebung der Code ausgeführt wird

Wo soll ich denn eigene Module speichern, damit der Interpreter sie findet?

Wenn man ein Modul z.B. module_1 importiert, sucht der Interpreter nach module_1.py auf dem sogenannten **Modulsuchpfad** in der folgenden Reihenfolge:

1. im aktuellen Verzeichnis des Interpreters (wo dieser das rufende Modul geladen hat)
2. in PYTHONPATH,
3. Im Default-Pfad „PATH“
    - Dies ist der Suchpfad für die Standardbibliotheken
    - Er ist installationsabhängig, hier speziell der Ordner für site packages.
4. In dem Inhalt von .pth - Dateien die im Default-Pfad liegen.

1. Suchpfad: Im aktuellen Verzeichnis des Interpreters

- Das ist praktisch, insbesondere während der Programmentwicklung
- Aufpassen: Es gilt das aktuelle Verzeichnis (current working directory) des Interpreters – nicht das von IDLE

```
>>> import os
>>> os.getcwd()
'/Users/alexanderwolodkin/Documents'
```

- Fertig entwickelte Bibliotheken speichert man aber woanders.

2. PYTHONPATH

- Wenn diese Umgebungsvariablen (environment Variable) vorhanden und mit systemspezifischen Dateipfaden belegt sind, dann schaut der Python Interpreter ob er den Modulnamen in diesen Ordnern findet.

- Nach der Standard-Installation ist PYTHONPATH nicht vorhanden.

- Diese Umgebungsvariablen müssen z.B. unter Windows allerdings vergleichsweise umständlich gesetzt werden (bitte nachschlagen für euer Betriebssystem, falls ihr es näher wissen wollt).

- Interessant ist, dass dieser Pfad dann beim booten in den Pfad (siehe 3.) übernommen wird.

3. Im Default-Pfad „PATH“

- Die Variable path im Modul sys, also sys.path, ist eine Liste von Strings, die den Default-Modul-Suchpfad angibt.
- Dieser Suchpfad ist system- und installationsabhängig - nachschauen!

```
>>> import sys
>>> sys.path
['', '/Users/alexanderwolodkin/Documents', '/Library/Frameworks/Python.framework
/Versions/3.9/lib/python39.zip', '/Library/Frameworks/Python.framework/Versions/
3.9/lib/python3.9', '/Library/Frameworks/Python.framework/Versions/3.9/lib/pytho
n3.9/lib-dynload', '/Library/Frameworks/Python.framework/Versions/3.9/lib/python
3.9/site-packages']
```

- Unter Windows gehören fertige Module in den Ordner [Installationspfad zu Python]\lib\site-packages (je nach Version)

Häufig genutzte Module aus den built-ins

- Modul `sys` Zugriff auf Umgebungskomponenten wie Kommandozeile etc.
- Modul `os` Werkzeuge der Betriebssystemumgebung: Prozesse, Dateien, Shell-Kommandos
- Modul `re` Mustererkennung, Reguläre Ausdrücke
- Modul `math` Grundlegende mathematische Funktionalitäten
- Modul `time` Messen von Zeit

Namensräume

- Bereits gesehen in den letzten Vorlesungen und Übungen!
- Ein Unterprogramm (eine Funktion in Python) eröffnet (beim Aufruf) seinen eigenen Namensraum zur Namenskapselung.
- Jedes Mal, wenn eine Funktion aufgerufen wird, entsteht deshalb ein neuer lokaler Namensraum.
- Dieser Namensraum enthält die Namen, die hier definiert werden: die (formalen) Funktionsparameter sowie die Namen von Variablen, denen im Rumpf der Funktion Werte zugewiesen werden.

Zusammenfassung: Strukturierte Programmierung

- Wir gliedern unsere Programme in Hauptprogramm (dem rufenden Programm, z.B. der interaktive Interpreter im IDLE), gegebenenfalls in Module und in Funktionen.
- Wenn wir „vorsichtig“, d.h. restriktiv, mit der `import`-Anweisung umgehen, können wir Namenskonflikte (gleicher Name für verschiedene Objekte, die zum Beispiel von verschiedenen Programmierern geschrieben wurden) weitgehend vermeiden.
- Der Schlüssel dazu ist der qualifizierte Name, der mithilfe der Punktnotation die verschiedenen Namensräume abtrennt: `import <module>` gibt Zugriff auf Funktionen des Moduls durch Qualifizierung: `module.Funktionsname`

Wichtige Pakete

- Pakete, die wir noch benutzen wollen:
  - NumPy
  - SciPy
  - Matplotlib

- Installation: https://www.scipy.org/install.html
- Dokumentation: https://docs.scipy.org/doc/
- Nachher ausprobieren!

Das Paket numpy

- Muss importiert werden, als

```
import numpy
```

- häufig auch

```
import numpy as np
```

Numpy Features - Array

- Das wichtigsten Feature, das numpy bereitstellt, sind die sogenannten multidimensionalen Arrays
- Diese funktionieren ähnlich wie die Listen in Python
- Integration von C/C++ (wird nicht näher betrachtet)
- Schnelle Erzeugung generischer Daten

Numpy Features - Array

- Wichtige Unterschiede zu Listen:
  - Die Einträge dürfen nur von einem (numerischen) Datentyp sein
  - Zugriff auf mehrdimensionale Arrays funktioniert, „wie man sich das wünscht“ (also wie über Indizes einer Matrix)
  - Eine nachträgliche Änderung der Anzahl der Einträge ist nicht möglich
  - Bei einer Konkatenation wird ein neues Objekt erstellt!

Beispiel

```
 9  import numpy as np
10
11  test = np.array([1, 15, 30])
12  print(test, end="\n\n")
13
14  print(type(test), end="\n\n")
15
16  print(test[0],
17        test[1],
18        test[:1],
19        test[:2],
20        test[-1],
21        end="\n\n")
22
23  print(test[0],
24        type(test[0]),
25        test[:1],
26        type(test[:2]),
27        end="\n\n")
```

```
In [8]: runfile('/Users/alexanderwolodkin/Documents/
Python/temp.py', wdir='/Users/alexanderwolodkin/
Documents/Python')
[ 1 15 30]

<class 'numpy.ndarray'>

1 15 [1] [ 1 15] 30

1 <class 'numpy.int64'> [1] <class 'numpy.ndarray'>


In [9]:
```

Beispiel 2

```
 9   import numpy as np
10
11   test = np.array([
12                    [1, 2, 3],
13                    [4, 5, 6],
14                   ],
15                   float)
16   print("Array, Datentyp float:",
17         test, end="\n\n")
18
19   print("Ein bestimmtes Element:",
20         test[0, 1], end="\n\n")
21
22   print("Und was passiert hier?",
23         test[:,2])
```

```
In [12]: runfile('/Users/alexanderwolodkin/Documents/
Python/temp.py', wdir='/Users/alexanderwolodkin/
Documents/Python')
Array, Datentyp float: [[1. 2. 3.]
 [4. 5. 6.]]

Ein bestimmtes Element: 2.0

Und was passiert hier? [3. 6.]

In [13]:
```

Beispiel 3

```
 9  import numpy as np
10
11  temp = np.array([
12                   [1, 2, 3],
13                   [4, 5, 6],
14                  ],
15                  float)
16  print("Wie lang ist unser temp:",
17        len(temp), end="\n\n")
18
19  print("Wie sieht temp aus",
20        temp, end="\n\n")
21
22  print("Welche Form hat temp?",
23        temp.shape, end="\n\n")
24
25  print("Dann ändern wir das:",
26        temp.reshape(3,2))
```

```
In [25]: runfile('/Users/alexanderwolodkin/Documents/
Python/temp.py', wdir='/Users/alexanderwolodkin/
Documents/Python')
Wie lang ist unser temp: 2

Wie sieht temp aus [[1. 2. 3.]
 [4. 5. 6.]]

Welche Form hat temp? (2, 3)

Dann ändern wir das: [[1. 2.]
 [3. 4.]
 [5. 6.]]

In [26]:
```