www.uni-frankfurt.de

Modul: B-PRG1
Grundlagen der Programmierung 1 und Einführung in die Programmierung EPR

V08 **Allererste Schritte im Softwareengineering („programmiernahe“ Anteile)**
mit Module - Kommentare - Docstring

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

Vorab:

Es sind noch Plätze im Förderkurs frei:

Di 10-12	Mi 10-12
Do 14-16	Fr 11.15 – 12.45

Bedingung: Teilnahme ist Pflicht!

Rückblick

Elementare Datentypen

Kontrollstrukturen: Verzweigungen, Schleifen, Prozeduren

➔ Das minimal Notwendige zum Programmieren ist geschafft

➔ … aber, es gibt noch Einiges mehr zu tun!

Unser heutiges Lernziele

Einige ***Strategien und Hilfsmittel*** *zum Programmieren kennenlernen.*

Hintergründe, das warum dazu kennenlernen.

Allererste Schritte im Softwareengineering gehen!

Übersicht

- Was ist Software Engineering (Software Technik)? – Warum?
  - Modulalisierung
  - Kommentare
  - Docstrings

Stichwort

"Software Engineering" oder "Software Technik" (dtsch.)

Teildisziplin der Informatik

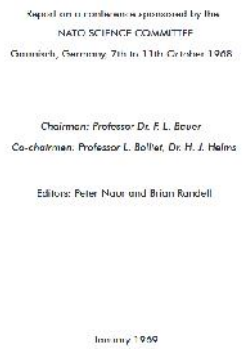
Nach Balzert:

> „Zielorientierte Bereitstellung und systematische Verwendung von **Prinzipien, Methoden und Werkzeugen** für die **arbeitsteilige, ingenieurmäßige Entwicklung** und Anwendung von umfangreichen Softwaresystemen."

Helmut Balzert: *Lehrbuch der Software-Technik.* Bd.1. Software-Entwicklung. Spektrum Akademischer Verlag, Heidelberg 1996, 1998, 2001.

Woher kommt dieser Begriff?

- Nato-Tagung in Deutschland … Reaktion auf die erste Software Krise
- 1968 (vor 50 jahren: da war nicht einmal der Begriff **Informatik** gebräuchlich)
- *The idea for the **first** NATO **Software Engineering Conference**, and in particular that of adopting the then **practically unknown term "software engineering"** as its **(deliberately provocative)** title, I believe came originally from Professor Fritz Bauer.*


Report on a conference sponsored by the
NATO SCIENCE COMMITTEE
Garmisch, Germany, 7th to 11th October 1968

Chairman: Professor Dr. F. L. Bauer
Co-chairmen: Professor L. Bolliet, Dr. H. J. Helms

Editors: Peter Naur and Brian Randell

January 1969

Brian Randell Dagstuhl Seminar **9635,** 1996http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html

Quelle: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

Das Problem (Die "1. Software-Krise")

- Man erkannte, dass "Software" eine geringe "Qualität" aufwies.
- Entwicklungsprojekte benötigten mehr Zeit als geplant und wurden teurer.
- **Also:** Software sollte systematischer, exakter; messbarer, fristgerecht, im Kostenrahmen und die Spezifikation erfüllend entwickelt werden.
- Engineering (die ingenierumäßige Vorgehensweise), allerdings in anderen Fächern konnte dies!
- **Idee: Übertragung auf Software**

Idee: Übertragung auf Software

Entwicklung einer "Best Practice"

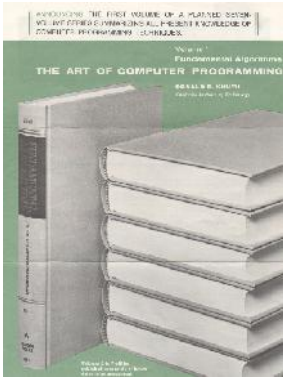
Aber: SWE war nicht die einzige Reaktion:

1968 erschien auch das erste Buch der Reihe:

"The Art of Computer Programming"

von **Donald E. Knuth**.

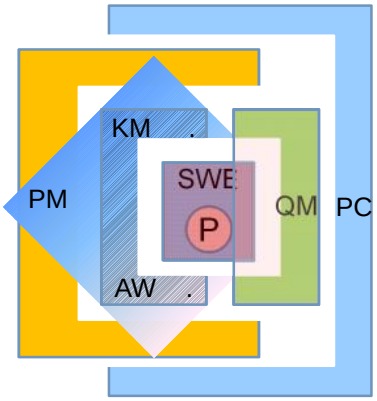
Obwohl bisher nur 3 der geplanten 7 Bände vollständig erschienen sind (von Volume 4 nur der erste Teil) ist diese Reihe die **kommerziell erfolgreichste der Informatik**



Wie groß ist das Problem eigentlich?

- Also ...ist SWE die Lösung aller Probleme?
- **Sicher nicht,** aber die Methoden des Software Engineering können helfen, komplexe Softwareanforderungen in den Griff zu bekommen, und bestimme Arten von Problemen bereits beim Entwurf zu vermeiden oder zu reduzieren.

Erinnern Sie sich an diese Folie (Vorlesung 0)
Programmieren als Teil einer größeren Aufgabe

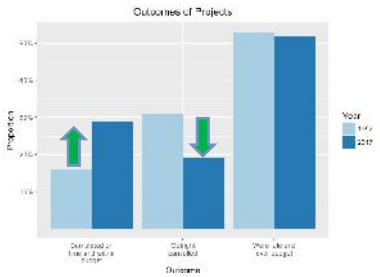


P	Programmieren
SWE	Softwareengineering
KM	Konfigurations-management
AW	Änderungswesen
PM	Projektmanagement
QM	Qualitätsmanagement
PC	Projektcontrolling

Immer noch **„scheitern“**
30 – 70 % aller Softwareprojekte!
Bessere Programmiersprachen?
Mehr Softwareengineering?
Mehr/besseres WAS?

Wie groß ist das Problem?

- Das ist schwer zu messen! Die bekannteste Langzeitstudie:
- Der CHAOS Report der **Standish Group** ("kleine Beratungsfirma aus Boston, U.S.A.) stellt entsprechende Statistiken **alle zwei Jahre** vor:



Aktuell seit 1994 ca. 50.000 Projekte in der Datenbank.

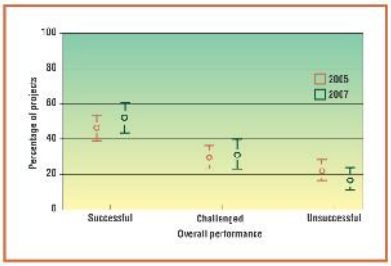
aber nur aus den U.S.A.

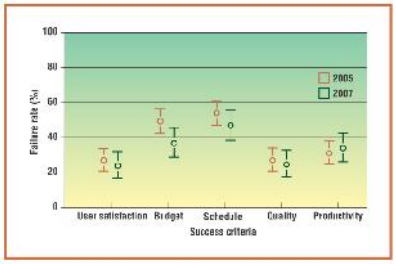
Interessengeleitet?

Schwierig, die Kriterien für Erfolg und Misserfolg festzulegen, Änderungen dabei erfolgt …

Ein wissenschaftlicher Ansatz:

Khaled El Emam, A. Günes Koru: A Replicated Survey of IT Software Project Failures, IEEE Software, 2008, siehe hier





Was gehört zum SWE? Kernprozesse

1. Planung
- Anforderungserhebung
- Lastenheft (Anforderungsdefinition)
- Pflichtenheft
- Aufwandsschätzung
- Vorgehensmodell

2. Analyse
- Auswertung
- Mock-up
- Prozessanalyse / Prozessmodell
- Systemanalyse
- Strukturierte Analyse (SA)
- Objektorientierte Analyse (OOA)

3. Entwurf
- Softwarearchitektur
- Strukturiertes Design (SD)
- Objektorientiertes Design (OOD)
- Fundamental Modeling Concepts (FMC)

4. Programmierung
- Normierte Programmierung
- Strukturierte Programmierung
- Objektorientierte Programmierung (OOP)
- Funktionale Programmierung

5. Validierung und Verifikation
- Modultests (Low-Level-Test)
- Integrationstests (Low-Level-Test)
- Systemtests (High-Level-Test)
- Akzeptanztests (High-Level-Test)

Was gehört zum SWE? Unterstützungsprozesse

6. Anforderungsmanagement

7. Projektmanagement
- Risikomanagement
- Projektplanung
- Projektverfolgung und -steuerung
- Management von Lieferantenvereinbarungen

8. Qualitätsmanagement
- Capability Maturity Model
- SPICE (Software Process Improvement and Capability Determination)
- Incident Management
- Problem-Management
- Softwaremetrik (Messung von Softwareeigenschaften)
- Programmierstil

8. Qualitätsmanagement (Fortsetzung)
- statische Analyse (Berechnung von Schwachstellen)
- Software-Ergonomie

9. Konfigurationsmanagement
- Versionsverwaltung
- Änderungsmanagement / Veränderungsmanagement
- Releasemanagement
- Application-Management (ITIL)

10. Softwareeinführung

11. Dokumentation
- Technische Dokumentation
- Softwaredokumentation
- Systemdokumentation
- Betriebsdokumentation (Betreiber/Service)
- Bedienungsanleitung (Anwender)
- Verfahrensdokumentation (Beschreibung rechtlich relevanter Softwareprozesse)

Was gehört zum SWE? … und was behandeln wir? Kernprozesse und Unterstützungsprozessse

1. Planung
- Vorgehensmodell

2. Analyse
- Strukturierte Analyse (SA)
- Objektorientierte Analyse (OOA)

3. Entwurf
- Strukturiertes Design (SD)
- Objektorientiertes Design (OOD)

4. Programmierung
- Strukturierte Programmierung
- Objektorientierte Programmierung (OOP)

5. Validierung und Verifikation
- Modultests (Low-Level-Test)
- Integrationstests (Low-Level-Test)

8. Qualitätsmanagement
- Programmierstil
- Software-Ergonomie

11. Dokumentation
- Technische Dokumentation
- Softwaredokumentation

Also: Nur ein "kleiner" Teil und im wesentlichen "nur" Python-spezifisch.

Empfehlung für ein Reading:
de Vries: Software Engineering

… und das sind die Themen die zu PS2 gehören grün markierte

1. Planung
- Vorgehensmodell

2. Analyse
- Strukturierte Analyse (SA)
- Objektorientierte Analyse (OOA)

3. Entwurf
- Strukturiertes Design (SD)
- Objektorientiertes Design (OOD)

4. Programmierung
- Strukturierte Programmierung
- Objektorientierte Programmierung (OOP)

5. Validierung und Verifikation
- Modultests (Low-Level-Test)
- Integrationstests (Low-Level-Test)

8. Qualitätsmanagement
- Programmierstil
- Software-Ergonomie

11. Dokumentation
- Technische Dokumentation
- Softwaredokumentation

Und was behandeln wir heute? Kernprozesse und Unterstützungsprozessse

1. Planung
- Vorgehensmodell

2. Analyse
- Strukturierte Analyse (SA)
- Objektorientierte Analyse (OOA)

3. Entwurf
- Strukturiertes Design (SD)
- Objektorientiertes Design (OOD)

4. Programmierung
- Strukturierte Programmierung
- Objektorientierte Programmierung (OOP)

5. Validierung und Verifikation
- Modultests (Low-Level-Test)
- Integrationstests (Low-Level-Test)

8. Qualitätsmanagement
- Programmierstil
- Software-Ergonomie

11. Dokumentation
- Technische Dokumentation
- Softwaredokumentation

Übersicht

- Was ist Software Engineering (Software Technik)? – Warum?
- Modularisierung
- Kommentare
- Docstrings
- Softwaredokumentation

Modularisierung

- war eine der **ersten Maßnahmen in/nach der ersten Softwarekrise** ab ca. 1968 (➔ Beginn des Softwareengineerings - *David Parnas*)
- Ein **Modul** ist ein **abgeschlossener** Teil einer Software, bestehend aus mehreren Teilprogrammen (Prozeduren und Funktionen) **und** den zugehörigen Datenstrukturen.
- Module sind ein Mittel zur **Kapselung** (*encapsualation*) von Software, d.h. Trennung von „Schnittstelle“ und Implementierung und damit Schutz vor „unkontrollierter“ Fehlerausbreitung (*Parnas*).
- Die Schnittstelle eines Moduls definiert die Daten, die als Eingabe erlaubt sind und die Ergebnisse der Verarbeitung.

Bedeutung der Modularisierung (= Strukturierung)

- Programme oder -teile werden wiederverwendbar, ohne dass Code redundant erstellt und redundant **gepflegt** (!!!) werden muss.
- Größere, komplexe Programme können durch den Einsatz von Modulen *gegliedert und strukturiert* werden. Funktionalitäten können nach dem Baukastenprinzip eingebunden werden.
- Entwurf und Definition von Modulen und Schnittstellen ist Teil der Designphase in der Softwareentwicklung.
- Mehrere Entwickler(-gruppen) können nach erfolgter Schnittstellen-festlegung unabhängig voneinander einzelne Module bearbeiten **und auch testen**.

Modularisierung in Programmiersprachen

- 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.
- Ggf. können Module auch in anderen Programmiersprachen implementiert sein, z.B. wie in Python viele builtin Module in C/C++.

Module in Python

- Der Typ des ***Moduls*** **ist** ein "**Behälter"**, der Objekte enthält, die **mit der Anweisung** `import` **für das aktuelle Programm verfügbar werden**.
- **Module definieren einen Namensraum**.
- Ein Modul entsteht dadurch, dass Sie in Python ein **Programm mit der Endung .py** unter einem Namen **abspeichern**.
- Module sollen kurze Namen haben, alle in Kleinbuchstaben. 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)

(Attribute des Moduls m)

Attribut	Beschreibung
m.__dict__	Das Dictionary, das zum Modul gehört und insbesondere die Namensverwaltung unterstützt
m.__doc__	Dokumentations-String des Moduls
m.__name__	Name des Moduls
m.__file__	Datei, aus der das Modul geladen wurde, wenn vorhanden
m.__path__	Vollständig qualifizierter Paketname; definiert, wenn das Modulobjekt sich auf ein Paket bezieht

(Bevor Sie diese Attribute des Moduls abfragen können, müssen Sie dieses erst durch `import` bekannt machen.)

Import eines Moduls namens `module`

`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 besser **nicht nutzen!**

Was passiert beim Import?

- Die Moduldatei suchen und **finden**!
- 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)

module_1.py

```
import os
x = 42

def f1():
    print(x)

print(os.getcwd())
f1()
```

```
>>> ================== RESTART =======
>>>
C:\Users\kroemker\Python-P\EPR
42
>>> import module_1 #nur einmal ausgef
>>>
```

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.

Also, zusammengefasst:

Mit einem .py-File kann man folgendes machen:

1. als Modul importieren und dann in einem anderen Programm nutzen
   ➔ alle Statements außer `def` (und `class`) werden (einmal) ausgeführt.
2. als Skript (Programm) starten und nutzen
   ➔ Hier hab ich ggf. Initialisierungscode, den man nur braucht, wenn es als Skript/Hauptprogramm ausgeführt wird.

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

Ist also eine **Weiche**, die entscheidet, **in welcher Umgebung** der Code ausgeführt wird. Wir werden diesen "Mechanismus" übrigens beim Testen nutzen (nächste Woche schon!).

Varianten des imports

- Es gibt diverse Varianten und Erweiterungen zum import (z.B. import as - das führt hier aber zu weit).
- Das Python-Tutorial liefert die passenden Ergänzungen:

Englisch: https://docs.python.org/3/tutorial/modules.html
oder
Deutsch:http://py-tutorial-de.readthedocs.org/de/latest/modules.html

Aber Vorsicht: Dies ist ein advanced topic … kostet ggf. viel Zeit!

Pakete

- sind (meist thematisch orientierte) **Sammlungen von Modulen**
- sind in (Paket-)Ordnern gespeichert, die eine Anzahl von Modulen (= Dateien) oder anderen Paketen enthalten (Pakete von Paketen sind erlaubt.)
- Diese **Paket-Ordner müssen** eine Datei **`__init__.py`** als Kennzeichen enthalten! Diese Datei **kann leer sein** oder Initialisierungscode für das Paket enthalten!
- Benutzer eines Pakets können individuelle Module aus dem Paket importieren, indem die Punktschreibweise benutzt wird:
  - `Paket-name.modul-name` oder
  - `paket-name.paket-name.modul-name` u.s.w.
- und noch einige weitere Details, siehe z.B. Python-Tutorial.

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

Wenn man einen 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** (dem Startverzeichnis - von dort, 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 sehr praktisch, gerade bei der Programmentwicklung.
- **Aufpassen:** Es gilt das aktuelle Verzeichnis (**c**urrent **w**orking **d**irectory) **des Interpreters – nicht das von IDLE**
- Fertig entwickelte Bibliotheken speichert man aber woanders.

module1.py – C:\Users\kroemker\Python-P\EPR

```
import os
x = 42

def f1():
    print(x)

print(os.getcwd())
f1()
```

module2.py – C:\Users\kroemker\Python-P\EPR

```
import module_1
module_1.f1()
```

```
>>> ================== RESTART ==
>>>
C:\Users\kroemker\Python-P\EPR
 42
42
>>>
```

Zusammenfassung Module und Modul-Suchpfade

- Wenn es so viele unterschiedliche Mechanismen gibt, scheinen **Module** etwas **sehr Wichtig**es zu sein – sind sie auch!
- Für die **Programmentwicklung** ist es am einfachsten, beide, das zu entwickelnde Modul und das „rufende“ Programm in einem Ordner zu halten!
- Anders als beim `#include` in C ist ein `import` in Python mehr als eine "einfache Texteinfügung". `import` ist eine Operation zur Laufzeit, in der Dreierlei ausgeführt wird:
  - Die Moduldatei suchen und finden!
  - Der Code wird in Byte-Code übersetzt (wenn nötig).
  - **Das Modul wird einmal ausgeführt,** d.h. der Code auf „oberster Ebene“ des Moduls, **aber dies nur beim ersten import!**

Arten von Modulen

- In Python geschriebene Programme (.py-Dateien) – ein **Modul** „entsteht“ durch abspeichern eines Python Programms in einer **Datei mit der Endung .py**
- (oder Erweiterungen in C oder C++, die als dynamische Bibliotheken oder DLLs (Dynamic Link Libraries) übersetzt wurden) – Advanced!.
- Eingebaute (builtin) Module, in der Regel in C implementiert, die in den Interpreter gebunden sind, davon gibt es etwa zweihundert!

Anmerkungen

- Bei .py-Dateien gilt, dass sie beim **ersten import in Byte-Code übersetzt,** falls nicht schon vorhanden ein Unterverzeichnis `__pycache__` `erzeugt` und dortdrin eine neue Datei mit Erweiterung **.pyc** geschrieben wird.
- Bei einem späteren **import**, lädt der Interpreter diese **vorübersetzte** Datei, es sei denn, der Zeitstempel der letzten Änderung der .py-Datei ist jünger (dann wird die .pyc-Datei neu erstellt).
- Die Übersetzung von Dateien in solche mit Erweiterung .pyc erfolgt **nur in Zusammenhang mit der import-Anweisung, also bei Modulen.**
- Programme, die auf der Kommandozeile oder in der Standardeingabe definiert werden, erzeugen solche Dateien nicht.

Python: *„Batteries included“*

Drei „eingebaute“ Ebenen von „Erweiterungen“:

- **builtins‘** (in einem Modul `builtins` – **kein `import` nötig**)
- mehr als **200 Module** in der **Standard-Bibliothek** (werden bei der Installation mitgeliefert und sollte man grob kennen)
  - **Windows**: hier wird normal die gesamte **Standard-Bibliothek** installiert plus einiges mehr.
  - Unix-like systems: typisch erfolgt die Installation als “collection of packages – dann müssen Sie ggf. die “packaging” tools nutzen.
- Im Netz frei verfügbar, z.B. auf dem **PyPI – (Python Package Index)** http://pypi.python.org/pypi
  2017: **109107 –** 2015: **66604 Pakete –** 2012: **37504 Pakete**

Häufig genutzte Python Module aus den builtins (1)

Modul sys	Zugriff auf einige Umgebungskomponenten, wie Kommandozeile, Standardströme, etc.
Modul string	Konstante und Variable zur Bearbeitung von String-Objekten
Modul os	Werkzeuge der Betriebssystemumgebung: Prozesse Dateien, Shell-Kommandos, etc.
Modul re	Mustererkennung und Reguläre Ausdrücke
Module anydbm, pickle, shelve	Module zur Objekt-Persistenz

Häufig genutzte Python Module aus den builtins (2)

- **GUI-Modul Tkinter** (hiermit werden wir noch arbeiten)
- Internet Module
- Modul math
- Modul time
- Modul datetime
- Module zum Threading
- ...

Eingebaute Funktionen in Python

- befinden sich im Modul `__builtin__`
- Alphabetische Liste siehe
  http://docs.python.org/py3k/library/functions.html
- Unbedingt mal durchschauen!
- **sind ohne import verfügbar, … und einige kennen wir ja schon**
- Beispiele: ... abs(N), cmp (X,Y), print () , …
  alle casting-Funktionen: int(), bin(), …

Übersicht

- Module
- Kommentare
- Docstrings
- Softwaredokumentation
- Ein-/Ausgabe

Übersicht

- Module
- Namensräume, Programmierkonventionen
- Kommentare
- Docstrings
- Softwaredokumentation
- Ein- / Ausgabe

Kommentare

- **Kommentare** sind Annotationen in einer Programmiersprache.
- **You have to use (simple) English.**
  Sorry, aber Englisch ist die lingua franca der Informatik. Überlegen Sie, wie weit Ihnen Kommentare auf Malaiisch (*Bahasa Melayu* – (بهاس ملايو) helfen würden!
- Wird ein Quelltext weiterverarbeitet (kompiliert, interpretiert, etc.), dann werden Kommentare ignoriert und haben keinen Einfluss auf das Ergebnis.
- Kommentare **in Python** beginnen mit einem Doppelkreuz **#** („Lattenzaun“) und reichen bis zum Zeilenende, sofern sie nicht in einem string stehen (in ‘‘jjh#jk‘‘)

Verwendung von Kommentaren (Konventionen)

- **Informationen über den gesamten Quelltext (Programm, Modul, Funktion, ...)**
  Zu Beginn eines Quelltextes, z.B. Name des Autors, der Lizenz, des Erstellungsdatums, Kontaktadresse bei Fragen, Liste anderer benötigter Dateien, etc.
  ggf. auch Input und Output (hier **Doc-string** verwenden, siehe unten)

- **Gliederung des Quelltextes** Überschriften und Abschnitte können als solche gekennzeichnet werden. Dabei werden häufig nicht nur sprachliche Mittel verwendet
  # Hier beginnt der spannende Teil *(Ist der Text sinnvoll?)*
  sondern auch grafische Mittel, die sich durch Text umsetzen lassen
  # ****=- Spannender Teil -=****)

Verwendung von Kommentaren (Konventionen)

- **Erläuterung einzelner Zeilen**
  `Anweisung  #Kommentar`

- **Hinweise auf zu erledigende Arbeit (für Zukunft, Merkposten)**
  Kommentare können unzureichende Codestücke kennzeichnen (# Hier muss noch die Unterstützung von Umlauten verbessert werden) oder Platzhalter für komplett fehlende Codestücke sein
  `#implement table here`

- **Auskommentierung**
  Soll ein Teil des Codes vorübergehend gelöscht, jedoch eventuell später wieder eingesetzt werden, so wird er auskommentiert, d.h ein # in die erste Spalte der betreffenden Zeilen gesetzt Das Codestück ist dann für den Interpreter nicht mehr vorhanden.

Dokumentations-Strings (Doc string)

Falls die **erste Anweisung** eines **Moduls**, (einer Klasse) oder einer **Funktion** ein String ist, so wird aus diesem String ein Dokumentations-String des entsprechenden Objektes, wie im folgenden Beispiel:

```
def fact(n):
    """Computes the factorial of n.

    """
    if (n <= 1):
        return 1
    else:
        return n*fact(n-1)
```

Dies ist sehr hilfreich zur Programmdokumentation und für den interaktiven Einsatz!

apropos "docstring"

(so geschrieben in der Python Documentation)

manchmal auch "doc string" geschrieben (bei Google)

Erlaubt die **Dokumentation des Programms** durch Kommentare im Code (und docstrings sind als Konvention bei uns verbindlich).

Ja: **"A Foolish Consistency is the Hobgoblin of Little Minds."** **(siehe PEP 8)**
Aber, beachte auch den Wert der Konsistenz!

Ein docstring ist ein Stringliteral am Anfang einiger Objekte
- eines Moduls,
- einer Funktion,

Für später auch dort
- einer Klasse oder
- einer Methode

Schreibweisen des Docstrings (1)

Die **bevorzugte** Schreibweise ist der "Mehrzeiler"

```
""" Docstring: A very short sentence explaining the \
function.  < 79 characters.

additional informaion if required
and more infos

"""
```

Wichtig sind:

- Drei (doppelte) Anführungszeichen, blank_line nach der ersten und der letzten Zeile.
- Der erste Satz soll ein vollständiger kurzer Satz sein (mit Punkt am Ende). Den Objektnamen nicht nennen, sondern die Funktion.

Schreibweisen des Docstrings (2)

Insbesondere bei kleinen Funktionen ist auch ein Einzeiler möglich:

```
def yourfunction_2():
    """Docstring: one liner possible for short functions.

    """
    print ("yourfunction_2 was called--not yet written \
    or incomplete")
```

Auch hier: Ganzer Satz (englisch), etwa "Do this ..." oder "Return that ..." Immer kleinergleich 79 Zeichen, auch wenn ich 99 als Standard habe.

Beachte:

- **Der Docstring ist ein spezieller "Kommentar".**
- Der Teil bis zum 1. Linefeed wird beim Import des Moduls m in dessen Attribut `m.__doc__` gespeichert.

"additional information if required and more infos" Was soll dort rein?

- Der docstring soll genug Informationen geben, um das Objekt aufrufen zu können, ohne den Quellcode zu lesen oder weitere Dokumentationen zu benutzen.
  Hierzu gehört die sogenannte Calling Syntax und Semantik (aber nicht die Implementierung). – sind wir uns da einig?

  Es gibt diverse Ansätze, siehe
  https://wiki.python.org/moin/DocumentationTools
  - Google (informell, aber sehr detailliert)
  - DocUtils, http://docutils.sourceforge.net/ eine reStructuredText processing engine
  - Sphinx, http://sphinx.pocoo.org/ - konvertiert reStructuredText documentation into various formats

Unsere Vereinbarung

- Wir verzichten auf die zwangsweise informale oder formale Parameterbeschreibung bei Funktionen
- TIPP: Schauen Sie sich die Dokumentation der Python-Standardbibliotheken an und wähle Sie das für Sie "vernünftige" Maß.
- Schauen Sie sich vielleicht auch einmal die Vorgehensweise bei Google an, den **Google Python Style Guide**:

  https://google.github.io/styleguide/pyguide.html

Was kann man mit Docstrings machen?

Docstrings können auf unterschiedliche Arten gelesen/genutzt werden. Sei m das dokumentierte Modul (oder auch Funktion, Klasse, Methode auf das sich der Docstring bezieht.

1. über ein spezielles Attribut (dass es zu jedem Objekt gibt)
   `m.__doc__` (auch zur Laufzeit)
2. über Aufruf von `help(m)`
3. über mit externem Werkzeug generierte Dokumentation:
   - pydoc (ist ein Python-Programm, dass man auch aus der Betriebssystem-Shell aufrufen kann), aber (nach import) natürlich auch im Interpreter
   - Epydoc (wird zur Zeit nicht weiterentwickelt).

Am Beispiel (unser header = `module_head()`) (1)

```
>>> import module_head
>>> help(module_head)
Help on module module_head:

NAME
    module_head - Docstring: A very short sentence explaining the function.
< 79 characters.

DESCRIPTION
    additional informaion if required
    and more infos

FUNCTIONS
    log(...)
        log(x[, base])

        Return the logarithm of x to the given base.
        If the base not specified, returns the natural logarithm (base e)
of x.
```

Am Beispiel (unser header = module_head()) (2)

```
 sqrt(...)
        sqrt(x)

        Return the square root of x.

    yourfunction_2()
        Docstring: one liner is possible for short functions

    yourroutine_1()
        Docstring: Every subroutine and function has a docstring.

DATA
    __copyright__ = 'Copyright 2015/2016 – EPR-Goethe-Uni'
    __email__ = 'your email address'
    e = 2.718281828459045
```


Am Beispiel (unser header = module_head()) (3)

```
AUTHOR
    123456: John Cleese, 654321: Terry Gilliam

CREDITS
    If you would like to thank somebody i.e. an other student for
her/his code

FILE
    c:\users\kroemker\desktop\prg1-epr-2015\15-skript-folien-
orga\übungen\module_head.py


>>>
```

… übrigens, das `help()` nutzt auch das Programm pydoc.

Zusammenfassung

Wieder diverse Details gelernt. Machen Sie die Quizzes! – Wenn Sie das bisher vorgestellte beherrschen, können Sie wirklich schon interessante Programme schreiben.

*Achten Sie auf die Vorgaben im **Programmierhandbuch (Style Guide).***

Sie brauchen dies für die Programmieraufgabe … schon für EPR_3.

Wir haben diese Konzepte eingeführt ...

die Praxis, das Programmieren müssen Sie jetzt üben!

Ausblick

... und, danke für Ihre Aufmerksamkeit!