Grundwissen:

Informatik

- **Strukturwissenschaft**
- **Ingenieurwissenschaft**
- **Bereiche:**
  - Theoretische Informatik
  - Praktische Informatik
  - Technische Informatik
  - Angewandte Informatik
- **4 Bereiche verstehen:**
  - Input/Output
  - Algorithmen
  - Programm
  - (Berechnungs-)Prozess

Algorithmen

- **Eigenschaften:**
  - Präzision:
    Die Bedeutung jedes Einzelschritts ist eindeutig festgelegt
  - Effektivität:
    Jeder Einzelschritt ist ausführbar
  - Finitheit (statisch):
    Die Vorschrift ist ein endlicher text
  - Finitheit (dynamisch):
    Für die Ausführung wird nur endlich viel Speicher benötigt
  - Terminierung:
    Die Berechnung endet nach endlich vielen Einzelschritten
- **wünschenswerte Eigenschaften:**
  - Determinismus:
    Die Folgeschritte sind immer eindeutig festgelegt
  - Determiniertheit:
    Bei gleicher Eingabe berechnet die Vorschrift die gleiche Ausgabe
  - Generaltität:
    Die Vorschrift kann eine ganze Klasse von Problemen lösen

Programm

- **Algorithmus aufgeschrieben in geeigneter Sprache**
- **Programmiersprachen:**
  - Systemprogrammiersprachen:
    Nahe an der Maschine
  - Höhere Programmiersprachen:
    Idealisiertes Berechnugsmodell
  - Deklarative Programmiersprachen
    Statt Berechnungsmodell Spezifikation der Aufgabe

Berechnungsprozess

- **Ablauf eines Programms zu einer bestimmten Zeit auf einem bestimmten Rechner**
- **Betriebssystem**

Python

- **Guido van Rossum**
- **Vorteile:**
  - Softwarequalität:
    lesbarkeit, ...
  - Programmierer Produktivität:
    Kürzer als C++, direkte Tests möglich
  - Portabilität
  - Support-Bibliotheken
  - Komponentenintegrierbarkeit:
    Java, .Net, COM...
- **Beispiele für Nutzung:**
  - Google:
    Web-search, App engine, YouTube
  - Dropbox
  - Intel, Cisco, HP, Seagate:
    Hardwaretesting
  - ...

- **Nachteile:**
  - LANGSAM
- **interpreter vs compiler Sprachen**
  - *sourcecode* ⇒ *interpreter* ⇒ *output*
  - *sourcecode* ⇒ *compiler* ⇒ *Objectcode* ⇒ *Executer* ⇒ *Output*

Zahlen

- **3 Typen:**
  - int: beliebig groß
  - float
  - complex: Imaginärteileinheit j
- **Präfixe:**
  - 0x—-Hexadezimal
  - 0o—-Oktal
  - 0b—-Binär
- **Rechenoperatoren:**
  - +,-,*,/
  - // Ganzzahlige Division, Es wird immer abgerundet!!! (20 // 3 = 6, -20 // 3 = 7)
  - % Modulo
  - ** Potenz
  - &, |, ˆ ∼ Bitweise Boolsche Operatoren
- **Gemischte Ausdrücke:**
  - ein Operand complex, alle complex
  - ein Operand float, kein complex, alle float
- **inf und nan**
  - inf —- infinity
  - nan —- not a number
  - mit beiden kann man weiter rechnen!!!

Werte und Typen

- **Variablen**
  - haben immer den Typ des zugewiesenen Literals
  - Sind dynamisch typisiert
- **Zuweisungen**
  - Immer erst Rechte Seite auswerten, dann Zuweisung
- **int**
- **float**
- **complex**
- **str**
  - String Konkatenation (`'monty ' + 'python'`)
  - Multiplikation mit Ganzzahlen (`int`)
- **bool**
  - liefern `True` oder `False` zurück und werden automatisch nach `int` konvertiert (`True` + `True` = 2)
  - Vergleichsoperatoren:
    - `==` gleich?
    - `!=` ungleich?
    - `<` kleiner ?
    - `>` größer?
    - `<=` kleiner gleich?
    - `>=` größer gleich?
    - Strings nach lexikographischer Ordnung (Unicode, siehe `ord()`) verglichen
    - Werte verschiedener Typen sind ungleich. Fehler bei Anordnungsrelationen
  - Logische Operatoren:
    - `or, and, not` (in aufsteigender Operatorpräzedenz)
    - Nullwerte (z.B. `None`) werden wie `False`, alle anderen Werte wie `True` behandelt
    - Verkettung möglich (`1 < x < 3`)
- `None`
  - Vergleiche mit Identirät oder Gleichheit möglich
  - Zuweisung mit Identität haben sich etabliert: `x is None` bzw. `x is not None`

Sequenztypen

- `str`
  - Operationen:
    - Verkettung: `"gambol" + "putty" == "gambolputty"`
    - Wiederholung: `2 * "spam" == "spamspam"`
    - Indizierung: `"Python"[1] == "y"`
    - Mitgliedschaftstest: `17 in [11, 13, 17, 19]`
    - Slicing: `"Monty Python's Flying circus"[6:12] == "python"`
    - slicing erklärungen:
      - `seq[i:j]`
        liefter den Bereich [i, j)
      - `seq[:j]`
        Bereich beginnt an Position 0
      - `seq[i:]`
        Bereich endet am Ende der Folge
      - `seq[:]`
        Kopie der gesamten Folge
      - Beim slicing gibt es keine Index Fehler, Bereich außerhalb der Folge sind einfach leer
      - Auch beim slicing kann man von hinten indizieren: `seq[-3:]` liefert die letzten 3 Elemente einer Folge
      - erweitertes slicing:
        Mit Schrittweite: `seq[i:j:k]` k = Schrittweite
    - Iteration: `for x in "egg"`
- `list`
  - veränderlich (mutable)
  - "Container"
  - tuple unpackiung:

    ```
    [a, (b, c), (d, e), f] = (42, (6, 9), "do", [1, 2, 3])
    ```

  - Operationen:
    - Verkettung: `"gambol" + "putty" == "gambolputty"`
    - Wiederholung: `2 * "spam" == "spamspam"`

  - Indizierung: `"Python"[1] == "y"`
  - Mitgliedschaftstest: `17 in [11, 13, 17, 19]`
  - Slicing: `"Monty Python's Flying circus"[6:12] == "python"`
  - slicing Zuweisungen:
    `seq[i, j] = ["bla"]` ersetzt die Elemente i bis j der Sequenz durch "bla"
  - bei slicing-Zuweisung mit Schrittweite müssen die Sequenzen gleichlang sein, sonst nicht
  - zum entfernen von Elementen kann `del` verwendet werden
  - Iteration: `for x in "egg"`

- `tuple`
  - unveränderlich (immutable)
  - "Container"
  - tuple unpackiung:

```
[a, (b, c), (d, e), f] = (42, (6, 9), "do", [1, 2, 3])
```

  - Operationen:
    - Verkettung: `"gambol" + "putty" == "gambolputty"`
    - Wiederholung: `2 * "spam" == "spamspam"`
    - Indizierung: `"Python"[1] == "y"`
    - Mitgliedschaftstest: `17 in [11, 13, 17, 19]`
    - Slicing: `"Monty Python's Flying circus"[6:12] == "python"`
    - Iteration: `for x in "egg"`

- **weitere Sequenzoperationen:**
  - `sum(seq)`
    Berechnet die Summe einer Zahlensequenz
  - `min(seq), min(x,y,....)`
    Berechnet das Minimum einer Sequenz (erste Form) bzw. der Argumente (zweite Form)
  - `max(seq), max(x,y,...)`
    analog zu min
  - any(seq)
    Äquivalent zu `elem1 or elem2 or elem3 or...`, wobei `elemi` die Elemente von `seq` sind und `True` oder `False` zurück geliefert wird

  - `all(seq)`
    analog zu `any`
  - `len(seq)`
    Berechnet die Länge einer Sequenz
  - `sorted(seq)`
    Liefert eine Liste, die dieselben Elemente hat wie `seq`, aber (stabil) sortiert ist

- **Iteration**
  - `for i in seq`
  - auch mehrfach Zuweisung möglich (vergleiche tuple unpacking) `for x, y in seq`
  - Schleifen Anweisungen:
    * `break`
      beendet eine Schleife vorzeitig wie bei `while`-Schleife
    * `continue`
      beendet die aktuelle Schleifeniteration vorzeitig, d.h. springt zum Schleifenkopf und setzt die Schleifenvariable(n) auf den nächsten Wert
    * Außerdem können Schleifen (so wie `if`-Abfragen) einen `else`-Zweig aufweisen. Dieser wird nach Beendigung der Schleife ausgeführt und zwar genau dann, wenn die Schleife nicht mit `break` verlassen wurde.
      textttbreak, continue, else funktionieren ebenso bei den bereits gesehenen `while`-Schleifen.

Funktionen/Methoden

- **Argumente vs Parameter**
  - Parameter—-Bei Funktionsdefinition eingeführt
  - Argumente—-Ersetzen Parameter bei Funktionsaufruf
- **Standardfunktionen**
  - `type()`
    bestimme Wert bzw. Typ eines Literal
  - `input()`
    Erwatret Benutzer Eingabe und gibt diese als String zurück
  - Typenkonversionen:
    * `int()`
    * `str()`

    * `float()`
    * `complex()`
  - Numerische Funktionen:
    * `abs()`:
      Berechnet Absolutwert (auch für complex)
    * `round()`:
      rundet
    * `pow()`:
      Berechnet Exponentation bei zwei Argumenten bzw. Exponentation der ersten beiden modulo dem dritten Argument
  - Zeichen-Konversion:
    * `chr()`:
      Zahlen in Unicode Zeichen—-in Python einbuchstabige Strings
    * `ord()`:
      Zeichen in Zahlen
- **Standardmethoden**
  - Methoden für sequentielle Objekte:
    * `s.index(value, start, stop)`
      `start` und `stop` sind optionale Parameter
      Sucht in der Sequenz nach einem Objekt mit dem Wert `value`
      Liefert index des ersten Treffers zurück
      Erzeugt eine Ausnahme falls kein passendes Element existiert
    * `s.count(value)`
      Liefert die Anzahl der Elemente in der Sequenz, die gleich `value` sind
  - Methoden für `list` und `bytearray`:
    * Einfügen
      · `s.append(element)`
      Hängt ein Element an die Liste an
      · `s.extend(seq)`
      Hängt die Elemente einer Sequenz an die Liste an
      · `s.insert(index, element)`
      Fügt `element` vor Position `index` in die Liste ein
    * Entfernen
      · `s.pop()`
      Entfernt das letzt Element und liefert es zurück

        - `s.pop(index)`
          Entfernt das Element an Position `index` und liefert es zurück
        - `s.remove(value)`
          Entfernt das erste Element aus der Liste das gleich `value` ist und liefert `None` zurück
      * Umdrehen
        - `s.reverse()`
          Dreht die Reihenfolge der Sequenz um
      * Methode von Liste: Sortieren
        - `l.sort(key=None, reverse=None` (nur auf Listen!!!)
          Sortiert die Liste `l`
          Der sortieralgorithmus ist stabil, dh. Elemente die den gleichen Wert haben werden in ihrer relativen Anordnung nicht geändert
          Damit kann man mehrstufig sortieren
          `reverse=True` $\longrightarrow$ absteigende Sortierung
          `key`: kann eine Funktion angeben, die für das jeweilige Element den Sortierschlüssel berechnet. Bsp: `str.lower`
  - Sortieren und umdrehen von unveränderlichen Sequenzen
    * da Tupel und Strings unveränderlich sind gibt es für sie auch keine mutierenden Methoden
    * es gibt aber 2 Funktionen:
      - `sorted(seq, key=None, reverse=None)`
        Liefert eine Liste die dieselben Elemente hat wie `seq`, aber stabil sortiert ist
      - `reversed(seq)`
        Generiert die Elemente von `seq` in umgekehrter Reihenfolge
        Liefert wie `enumerate` einen Iterator und sollte genauso verwendet werden

- **Strings**
  - `name.split()`:
    spaltet string an trennzeichen, default: Leerzeichen
  - Stringmethoden:
    * Suchen:
      - `start` und `stop` sind optionale positionale Parameter
      - `s.index(substring, start, stop)`
        Liefert analog zu `list.index` den Index des ersten Auftretens von `substring` in s

Im Gegensatz zu `list.index` kann ein Teilstring angegeben werden, nicht nur ein einzelnes Element

- `s.rindex(substring, start, stop)`
  Ähnlich wie `index`, aber von rechts suchend
- `s.find(substring, start, stop)`
  Wie `s.index`, erzeugt aber keine Ausnahme, falls `substring` nicht in s enthalten ist, sondern liefert dann -1 zurück
- `s.rfind(substring, start, stop)`
  Die Variante von rechts suchend

* Zählen und ersetzen
  - `s.count(substring, start, stop)`
    Berechnet wie oft `substring` als (nicht überlappender Teilstring in s enthalten ist
  - `s.replace(old, new, count)`
    `count` ist ein optionaler Parameter
    Ersetzt im Ergebnis überall den Teilstring `old` durch `new`
    maxímal `count` Ersetzungen
    Kein Fehler wenn `old` in s seltener oder gar nicht vorkommt
* Zusammenfügen
  - `s.join(seq)`
    `seq` muss eine Sequenz von Strings sein
    Berechnet `seq[0] + s + seq[1] + s + ...+ s + seq[-1]` nur effizenter
* In Worte aufteilen
  - `s.split()`
    Liefert eine Liste aller Wörter in s, wobei ein Wort ein Teilstring ist, der von Whitespace (Leerzeichen, Tabulator, Newlines, etc) umgeben ist
  - `s.split(seperator)`
    `seperator` muss ein String sein und s wird dann an den Stellen an denen sich `seperator` befindet zerteilt.
    Es wird die Liste der Teilstücke zurückgeliefert, wobei anders als bei der ersten Variante leere Teilstücke in die Liste aufgenommen werden
* In Zeilen aufteilen:
  - `s.splitlines(keepends=None)`
    Liefert eine Liste aller Zeilen in s, wobei eine Zeile ein Teilstring ist, der von Newlines umgeben ist
    Wird für `keepends True` angegeben , so werden die Newline-Zeichen erhalten

* Zerlegen
  - `s.partition(sep)`
    Zerlegt s in drei Teile
    Von links wird nach `sep` gesucht
    Wird `sep` in s gefunden, wird ein Tupel zurückgegeben, bei dem der erste Teil der Substring bis `sep` ist, dann kommt `sep` und dann der rechte Teilstring
  - `s.rpartition(sep)`
    Die Variante bee der von rechts gesucht wird
* Zeichen abtrennen
  - `s.strip(), s.lstrip(), s.rstrip()`
    Liefert s nach Entfernung von Whitespace an den beiden Enden (bzw. am linken bzw. rechten Rand)
  - `s.strip(chars), s.lstrip(chars), s.rstrip(chars)`
    Wie die erste Variante, trennt aber keine Whitespace-Zeichen ab, sondern alle Zeichen die in dem String `chars` auftauchen
* Groß- und Kleinscchreibung
  - `s.capitalize()`
    Erster Buchstabe des Strings wird Großbuchstabe, alle anderen Kleinbuchstaben
  - `s.lower()`
    Ersetzt im Ergebnis alle Groß- durch Kleinbuchstaben
  - `s.upper()`
    Ersetzt im Ergebnis alle Klein- durch Großbuchstaben
  - `s.casefold()`
    Transformiert wie `lower` alles in Kleinbuchstaben und macht noch weitere Ersetztungen wie "ß" in "ss"
    Speziell für Groß- und Kleinschreibungsfreie Vergleiche gedacht
  - `s.swapcase()`
    Großbuchstaben werden klein, Kleinbuchstaben groß
  - `s.title`
    Jedes einzelne Wort beginnt mit einem Großbuchstaben
* Eigenschaften
  - `s.isalnum()`
    True wenn alle Zeichen in s Ziffern oder Buchstaben sind
  - `s.isalpha()`
    True wenn alle Zeichen in s Buchstaben sind

- `s.isdigit()`
  True wenn alle Zeichen in l Ziffern sind
- `s.islower()`
  True wenn alle Buchstaben in s Kleinbuchstaben sind
- `s.isupper()`
  True wenn alle Buchstaben in s Großbuchstaben sind
- `s.isspace()`
  True wenn alle Zeichen in s Whitespace sind
- `s.istitle()`
  True wenn alle Worte in s groß geschrieben sind
- `s.startswith(prefix, start, stop))`
  True wenn s bzw. `s[start:stop]`) mit `prefix` beginnt
- `s.endswith(suffix, start, stop)`
  True wenn s (bzw. `s[start:stop]`) mit `suffix` endet

* Textformatierung
  - `fillchar` ist ein optionaler positionaler Parameter
  - `s.center(width, fillchar)`
    Zentriert s in einer Zeile der Breit `width`
  - `s.ljust(width, fillchar)`
    Richtet s in einer Zeile der Breite `width` linksbündig aus
  - `s.rjust(width, fillchar)`
    Richtet s in einer Zeile der Breite `width` rechtsbündig aus
  - `s.zfill(width)`
    Richtet s in einer Zeile der Breite `width` rechtsbündig aus, indem links mit Nullen aufgefüllt wird
* Decoding und Encoding
  - `s.encode(encoding)`
    Übersetzt einen String in eine Sequenz von Bytes (`bytes`) unter Benutzung des Encodings `encoding` (z.B. ascii, latin9, utf-8, cp1250)
  - `b.decode(encoding)`
    Übersetzt `bytes` in einen String unter Benutzung des angegebenen Encodings
* Alternative Stringformatierung, Tabs und Übersetzung
  - `s.format(*args, **kwargs)`
    Ermöglicht eine sehr komfortable, alternative Stringformatierung

· `s.expandtabs(tabsize)`
Expandiert Tabs, wobei der optionale Parameter `tabsize` den Defaultwert 8 hat.

· `s.translate(map)`
Transformiert einen String mit Hilfe eines Dictionarys `map`, dessen Schlüssel unicode-Werte sind und dessen Werte Unicode-Werte Strings, oder `None` sein können
Bei None wird das Zeichen gelöscht, sonst wird es ersetzt

- **nützliche Funktionen im Zusammenhang mit `for`-Schleifen**
  - `range()`:
    - `range(stop)`
      ergibt 0, 1 ....., stop-1
    - `range(start, stop`
      ergibt start, start + 1, ..., stop - 1
    - `range(start, stop, step)`
      ergibt start, start + step, start + 2 * step, ..., stop - 1
  - `enumerate(seq)`
    nimmt Sequenz als Argument und liefert einen Iterator mit Folgen von Paaren (`index, element`, zeigt also an welcher Position man sich gerade befindet
  - `zip()`
    - nimmt eine oder mehrere Sequenzen und liefert eine Liste von Tupeln mit korrespondierenden Elementen, erzeugt auch einen Iterator
    - besonders nützlich um mehrere Sequenzen parallel zu durhlaufen, das Ergebnis ist so lang wie die kürzeste Eingabe
  - `reversed`
    durchläuft Sequenz in umgekehrter Richtung, erzeugt einen Iterator
- **Identität**
  - `id(x)`
    liefert ein `int`, das eine Art "Sozialversicherungsnummer" für das durch `x` bezeichnete Objekt ist. ZU keinem Zeitpunkt während der Ausführung eines Programms haben zwei Objekte die gleiche `id`
  - Gleichheit vs Identität
    - Gleichheit testet ob `x` und `y` den gleichen Typ haben, gleich lang sind und korrespondierende Elemente gleich sind (die Definition ist rekursiv)
    - Bei Identität wird getestet ob `x` und `y` dasselbe Objekt bezeichnen
- **Dictionarys**
  -

bedingte Anweisungen

- **if-Anweisungen**
  - `if-else`
  - `if-elif-else`
  - `pass`
    Schlüsselwort für leeren Anweisungsblock
- **while-Schleifen**
  - `while` Bedingung:
    Anweisung
  - `break` kontrolliertes Verlassen der Schleife
  - `continue` Neuen Schleifendurchlauf starten

Module

- **import**
- **Punktnotation**
  - verhindert namenskollisionen
  - umständlich
- **Alternativ:**
  - `from module import name`
    Importiert name von modul
  - `from modul import *`:
    Importiert alle namen von modul
- **Math**
- **os**
  - enthält Funktionen, um den aktuellen Ordner festzustellen, zu ändern und den absoluten Pfadnamen zu bestimmen
  - `os.getcwd()`
    gibt aktuellen Ordner wieder
  - `os.path.abspath("../dateiname")`
    gibt absoluten Pfadnamen der Datei wieder
  - `os.chdir("../Ordnername")`
    ändert aktuellen Ordner

  - `os.path.exists(path)`
    testet ob unter dem Pfad `path` eine Datei oder ein Ordner existiert
  - `os.path.isdir(path)`
    testet ob es ein Ordner ist
  - `os.path.isfile(path)`
    testet ob es eine Datei ist
  - `os.listdir(path="."`
    zeigt den Inhalt des aktuellen Verzeichnisses
  - `os.path.join(dir, name)`
    Man kann Pfadbestandteile intelligent zusammensetzen:

```
def walk(dir):
    for name in os.listdir(dir):    # durchläuft jede Zeile in dir
        p = os.path.join(dir, name)    # weist Ordner und name P zu
        if os.path.isfile(p):    #wenn file, dann print
            print(p)
        else:
            walk(p)    #sonst weiter suchen
```

- **sys**
  - `sys.argv`-Liste
    * das erste Element ist der Name des aufgerufenen Skripts
    * danach folgen die auf der Kommandozeile angegebenen Elemente
- **shelve**
  - shelf zur Verwahrung persistenter Dateien
  - `shelve.open(filename, flag="c", writeback=false)`
    * `flag="c"`
      Lesen & schreiben, kreieren wenn nicht vorhanden
    * `flag="w"`
      Lesen & Schreiben
    * `flag="r"`
      Lesen
    * `flag="n"`
      Neues, leeres shelf
  - `writeback`
    * gibt an, ob jeder zugegriffene Wert zurückgeschrieben werden soll (wenn True) oder nur bei Zuweisungen an einen neuen Schlüssel.
- `re`

- `re.match(pattern, string, flags=0)`
  Prüft ob `pattern` auf ein Anfangsstück von `string` matcht.
  Dabei ist `flag` optional
  Ergebnis ist `None` wenn nicht erfolgreich, sonst ein Match-Objekt
- `re.search(pattern, string, flags=0)`
  Wie `match`, aber es wird innerhalb von `string` gesucht, statt nur den Anfang zu testen
- `re.findall(pattern, string, flags=0)`
  Wie `search`, aber liefert eine Liste mit allen nicht-überlappenden in String gematchten Teilstrings (oder Tupel mit Gruppenzugehörigkeit)
- `re.finditer(pattern, string, flags=0)`
  Wie `findall`, aber liefert einen Iterator, der alle Match-Objekte erzeugen kann
- `re.split(pattern, string, flags=0)`
  Zerlegt `string` an den Stellen, an denen es eine Übereinstimmung mit `pattern` gibt (u.U. noch mehr Resultate, wenn Gruppen vorhanden)
- `re.sub(pattern,repl, string, count=0, flags=0)`
  Ersetzt innerhalb von `string` alle Matches durch `repl`, wobei das ein String oder ein Funktionsobjekt sein kann, das das Match-Objekt als Parameter nehmen muss.
  Der optionale Parameter `count` begrenzt die Anzahl der Ersetzungen
- `re.subn(pattern, repl, string, count=0, flags=0)`
  Wie `sub`, aber es wird ein Tupel aus der Anzahl und neiem String zurück gegeben
- `re.compile(pattern, flags=0)`
  kompiliert einen regulären Ausdruck, falls dieser öfter verwendet werden soll
  Es entsteht ein Regular-Expression-Objekt
  Regular-Expression-Objekte besitzen Methoden entsprechend zu den `re`-Methoden
- Match-Objekte:
  - `m.group(*groups)`
    Gibt die gematchten Teilstrings für die angegebenen Gruppen zurück
    Gruppe 0 ist der gesamte gematchte String
  - `m.groups(default=None)`
    Gibt alle Teilstrings aller Gruppen zurück, wobei leere (nicht gematchte) Gruppen den Defaultwert erhalten
  - `m.groupdict(default=None)`
    Gibt ein dict mit sllen benannten Gruppen zurück
  - `m.start()`
    Anfangsindex des Matches `m.end()`
    Endindex des Matches

  * `m.span()`
    =(m.start(), m.end())

Automaten

- **Alphabet ($\sum$)**
  - endliche nicht-leere Menge (von Zeichen oder Symbolen)
- **Wort**
  - Folge von Zeichen aus einem Alphabet
- **(formale)Sprache**
  - beliebige endliche oder unendliche Menge von Wörtern
  - endliche Automaten kann man nutzen, um Sprachen zu akzeptieren
- **Deterministische endliche Automaten(DEA)**
  - Quintupel $A = \langle Q, \sum, \delta, q_0, F \rangle$
    * Q: endliche Zustandsmenge
    * $\sum$: Eingabealphabet
    * $\delta : Qx \sum \longrightarrow Q$: Übergangsfunktion
    * q_0: Anfangszustand
    * $F \subseteq Q$: Menge der (akzeptierenden) Endzustände
- **Transduktoren**
  - Automat der auch Ausgaben macht und nie stoppt
  - hier verzichtet man zumeist auf Endzustände
  - können gut das Verhalten eingebetteter Systeme beschreiben
- **Moore Automat**
  - nach Edward F. Moore
  - 6-Tupel $A = \langle Q, \sum, \wedge, \delta, \lambda, q_0 \rangle$
    * Q: endliche Zustandsmenge
    * $\sum$: Eingabealphabet
    * $\wedge$: Ausgabealphabet
    * $\delta : Qx \sum \longrightarrow Q$: Übergangsfunktion
    * $\lambda : Q \longrightarrow \wedge$: Ausgabefunktion
    * q_0: Startzustand
    * kommt der Automat in einen Zustand q, dann gibt er das Zeichen $\lambda(q)$ aus.

Spieltheorie

- **beschäftigt sich mit Entscheidungen von rationalen Agenten in Gruppensituationen**
- **von Neumann 1928, Nash 1950**
- **Heute:**
  - Multi-Agenten-Systeme (und KI allgemein)
  - Internet Routing
  - Internet Auktionen
- **Strategische Spiele**
  - Spiele bei denen alle Spieler gleichzeitig eine Entscheidung treffen und das Resultat von der getroffenen Entscheidung abhängt (Stein, Schere, Papier)
  - Fester Nutzen-Wert
  - Lösungen:
    - Maximin:
      Das Maximum über alle worst-case-Werte
    - Dominante Strategien:
      Ist eine Entscheidung immer besser, egal was der andere wählt, dann nimm diese
    - **Nash-Equilibrium(NE)**
  - Kombinationen von Aktionen werden als Lösung betrachtet bei denen sich kein Spieler durch eine Abweichung verbessern kann
  - Satz von Nash:
    - Erweitert man die wählbaren Aktionen auf Wahrscheinlichleitsverteilungen über den Aktionen, so gibt es (in endlichen strategischen Spielen) immer ein Nash-Equilibrium
- **wiederholte strategische Spiele**
  - das Spiel wird mit der Wahrscheinlichkeit p nach jeder Runde beendet
  - erwarteter Nutzen statt festem Nutzen
  - bringen dden Aspekt von Zeit und Erfahrungen in die Spieltheoretische Analyse
  - Erwartete Spiellänge:

$$\sum_{i=1}^{\infty} ip(1-p)^{i-1} = \frac{1}{p}$$

- erwarteter Nutzen:

$$\sum_{i=1}^{\infty} u(1-p)^i = \frac{u}{p}$$

- Strategien
  - müssen potentiell unendlich sein
  - endliche Automaten mit Ausgabe (=Moore Automaten) wären eine Lösung
  - Unkooperativ
  - Kooperativ
  - Grimmig
  - Tit-for-tat
  - Bipolar/Troll

Rekursion

- Fakultät:

```
def fak(n):
    if n <= 1:
        return 1
    else:
        return n*fak(n-1)
```

oder Iterativ mit while Schleife:

```
def ifak(n):
    result = 1
    while n >= 1:
        result = result * n
        n -= 1
    return result
```

- Fibonacci:

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

oder Iterativ:

```
def ifib(n):
    if n <=1:
        return n
    a = 0
    b = 1
    i = 2
    while i < n:
        new = a + b
        a = b
        b = new
        i += 1
    return a + b
```

Objekte

- **Identität**
  - `is` bzw. `is not` testet auf Identität
  - `x is y` ist `True`, wenn x und y das selbe Objekt bezeichnen und ansonsten `False` (`is not` umgekehrt
- **Veränderlich vs Unveränderlich**
  - Veränderlich (mutable):
    - können modifiziert werden.
    - Operationen auf `x` beeinflussen auch `y`
    - jedesmal neues Objekt erzeugt, wenn ein literal ausgewertet wird
  - Unveränderliche (immutable):
    - können nicht modifiziert werden
    - für Zweisungen völlig unkritisch
    - Bsp: Zahlen(`int, float, complex`), Strings (`str`), Tupel (`tuple`)
    - Python darf ein existierendes Objekt jederzeit "woederverwenden, um Speicherplatz zu sparen, muss aber nicht
    - wegen dieser Unsicherheit ist es meistens falsch unveränderliche Objekte mit `is` zu vergleichen

zyklische Datenstrukturen

- Datenstrukturen in denen ein Teil identisch mit sich selbst ist
- Fehler!
- vermeiden!!

Bäume

- **Syntaxbaum**
- **Ausdrucksbaum**
  - arithetische Ausdrücke darstellbar
    - Operatoren als Knoten, Zahlen als Blätter
- **Listen und Tupel als Bäume**
  - Der Typ ist Knotenmarkierung und die Elemente sind Teilbäume
  - Bsp: `[1, [2, 3, 4], 5]`
- **Binärbäume:**
  - Spezialfall eines Baumes, bei dem jeder Knoten zwei Teilbäume besitzt
  - Blätter sind dann die Knoten die zwei leere Teilbäume besitzen
  - Der leere Teilbaum wird durch `None` repräsentiert
  - Jeder Knoten wird durch eine Liste repräsentiert
  - Die Markierung ist das erste Element der Liste
  - Der linke Teilbaum ist das zweite Element
  - Der rechte Teilbaum ist das dritte Element
  - Bsp: Der Baum mit Wurzel "+", linkem Teilbaum mit Blatt 5, rechtem Teilbaum mit Blatt 6: `["+", [5, None, None], [6, None, None]]`
  - Tiefe eines Knotens k ist:
    - 0, falls k die Wurzel ist
    - i + 1, wenn i die Tiefe des Elternknotens ist
  - Höhe des Baumes ist:
    - 0 für den leeren Baum
    - m + 1, wenn m die maximale Tiefe über alle Knoten im Baum ist
  - Die Größe eines Baumes ist die Anzahl seiner Knoten

- **Traversierung von Bäumen:**
  - 3 vorgehensweisen:
  - pre-Order (Hauptreihenfolge)
    - zuerst der Knoten
    - linker Teilbaum
    - rechter Teilbaum
  - Post-order (Nebenreihenfolge)
    - linker Teilbaum
    - rechter Teilbaum
    - Knoten
    - Im Falle von arithmetischen Ausdrücken auch umgekehrt polnisch oder Postfix-Notation genannt
  - In-Order (symmetrische Reihenfolge)
    - linker Teilbaum
    - Knoten
    - rechter Teilbaum
  - Reverse-In-Order
    - rechter Teilbaum
    - Knoten
    - linker Teilbaum
  - (Level-Order)
    - Besuchen nach Tiefenlevel von links nach rechts
- **Suchbäume**
  - Binärbaum der die Suchbaumeigenschaften erfüllt:
    - alle Markierungen im linken Teilbaum sind kleiner als die aktuelle Knotenmarkierung
    - alle Markierungen im rechten Teilbaum sind größer
  - Suche nach einem item m:
    - wenn gleich stoppe und gebe m zurück
    - wenn m kleiner ist, gehe in linken Teilbaum
    - wenn m größer ist, in den rechten Teilbaum
  - Suchzeit proportional zur Höhe des Baumes, meist logarithmisch in der Größe des Baumes

Fehler

- **3 Arten von Fehlern:**
  - Syntaktische Fehler:
    - meist leicht zu finden
    - Interpreter bemerkt Fehler vor Ausführung
  - Laufzeit Fehler:
    - Während der Ausführung hängt das programm oder es gibt eine Fehlermeldung(Exception)
  - Semantische Fehler
    - Alles läuft, aber die Ausgabe und Aktionen des Programms sind anders als erwartet.
    - gefährlichste Fehler!!!
- **Debuggen**
  - Bugs $\longrightarrow$ Bugfixes
  - einfügen von print statements
    - Logging als generalisierte Form
  - Post-Mortem-Analyse-tools
    - Stack-Backtrace
    - Heute Variablenbelegungen global und lokal im Stapeldiagramm
  - Interaktive Debugger
    - Setzen von Breakpoints
    - Inspektion des Programmzustands
    - Änderung des Zustands
    - Einzelschrittausführung (Stepping / Tracíng)
      - Step in
        Mache einen Schritt , ggf. in eine Funktion hinein
      - Step over
        Mache einen Schritt, führe dabei ggf. eine Funktion aus
      - Step out
        Beende den aktuellen Funktionsaufruf
      - Go / Continue
        Starte Programmausführung bzw. setzt fort

    - Bsp: pdb, IDLE

- **Automatische Tests**
  - Testfälle erzeugen
    - Basisfälle und Grenzfälle
  - Testgetriebene Entwicklung
    - Regressionstest
      - Wiederholung von Tests um sicher zu stellen, dass nach der Änderung der Software keine neuen Fehler eingeschleppt wurden
    - erst Tests formulieren und diese dann Stück für Stück erfüllen
  - Modultests oder Unittests
    - `unittest`:
      - ein komfortables (aber aufwendig zu bedienendes) Modul für die Formulierung und Verwaltung von Unit-tests
    - `doctest`
      - ein einfaches Modul, das Testfälle aus den docstrings extrahiert und ggf. automatisch ausführt
    - Der main-Trick

      ```
      if __name == "__main__":
           doctest.testmod()
      ```

      - name-Attribut ist gleich main wenn das Modul mit dem Python-Interopreter gestartet wird
    - `pytest`-Modul:
      - `py.test` ist ein umfangreiches Framework um Tests zu schreiben
      - Idee:
        Funktionen werden getestet indem man Testfunktionen schreibt
      - Testfunktionen müssen immer den Prefix `test_` besitzen
      - für die zu testende Funktion werden die erwarteten Rückgabewerte als Assertions formuliert
      - `assert`-Anweisungen:
        `assert` Bedingung[, String]
      - `assert` sichert zu, dass die Befingung wahr ist.
      - Ist das nicht der Fall wird eine Exception ausgelöst und der String ausgegeben

Dictionarys

- **assoziative Arrays/Listen**
- **Speichern Paare von Schlüsseln (keys) und zugehörigen Werten (values)**
- **Im Gegensatz zu Sequenzen sind dicts ungeordnete Container**
- **Erzeugung von dicts:**
  - `{key1: value1, key2: value2, ....}`
  - `dict(key1=value1, key2=value2, ...)`
  - `dict(sequence of pairs):`
    `dict([(key1, value1), (key2, value2), ...])`
  - `dict.fromkeys(seq, value)`
    - Ist `seq` eine Sequenz mit Elementen `kex1, key2, ...`, erhalten wir `key1: value, key2: value, ...`
    - Wird value weggelassen wird `None` verwendet
- **Operationen auf dicts:**
  - `key in d`
    Falls das dict den key enthält
  - `bool(d)`
    True falls das dict nicht leer ist
  - `len(d)`
    Liefert die Zahl der Elemente (Paare) in d
  - `d.copy`
    Liefert eine flache Kopie von d
  - `d[key]`
    Liefert den Wert zum Schlüssel key.
    Fehler bei nicht vorhandenen Schlüssel
  - `d.get(key, default`
    Wie `d[key]`, aber es ist kein Fehler wenn key nicht vorhanden ist
    Stattdessen wird default zurückgeliefert(`None`, wenn kein Default angegeben wurde)
  - `d[key] = value`
    Weist dem Schlüssel key einen Wert zu
    Ersetzt bestehendes Paar
  - `d.setdefault(key, default`
    Von Rückgabe äquivalent zu `d.get(key, default)`
    Falls das dict den Schlüssel noch nicht enthält, wird zusätzlich `d[key] = default` ausgeführt

    - Alternativ:
      `import defaultdict` aus dem Modul `collections`
      `collections.defaultdict`(Defaultgenerator)
  - `d.update(another dict)`
    Führt `d[key] = value` für alle `(key, value)` Paare in `another dict` aus überschreibt bestehende Einträge
  - `d.update(sequence of pairs)`
    entspricht `d.update(dict(sequence of pairs))`
  - `d.update(key1=value1, key2=value2, ....)`
    Entspricht `d.update(dict(key1=value1, key2=value2))`
  - `del[key]`
    Entfernt das Paar mit dem Schlüssel key aus d.
    Fehler wenn kein solches Paar existiert
  - `d.pop(key, default)`
    Entfernt das Paar mit dem Schlüssel key aus d und liefert den zugehörigen Wert
    Existiert ken solches Paar wird default zurück geliefert, falls gegeben, sonst Fehler
  - `d.popitem()`
    Entfernt ein (willkürliches Paar (key, value) aus d ind liefert es zurück
    Fehler falls d leer ist
  - `d.clear`
    Entfernt alle Elemente aus d
  - `d.keys()`
    Liefert alle Schlüssel in d zurück
  - `d.values()`
    Liefert alle Werte in d zurück
  - `d.items()`
    Liefert alle Einträge, dh (key, value)-Paare in d zurück
- **hashable:**
  - Zugriff in konstanter Zeit
  - keys dürfen daher nicht veränderlich sein
  - Werte nicht eingeschränkt

Mengen

- **Enthalten Elemente nur einmal**

- **Mengenelemente müssen hashbar sein**
  - `set vs frozenset`
    - `sets` sind veränderlich
    - `frozensets` sind unveränderlich
- **Konstuktion von Mengen**
  - `{elem1, ....., elemN}`
  - `set()`
  - `set(iterable)`
  - `frozenset()`
  - `frozenset(iterable)`
  - Mengen können aus beliebigen Objekten `iterable` erstellt werden, also solchen die for-Schleifen unterstützen
  - allerdings dürfen innerhalb von `iterable` nur hashbare Objekte stehen(z.B. keine Listen), sonst TypeError
- **Operationen auf Mengen**
  - `element in s, element not in s`
    Test auf Mitgliedschaft (liefert True oder False)
  - `bool(s)`
    True falls die Menge s nicht leer ist
  - `len(s)`
    Liefert die Zahl der Elemente der Menge s
  - `for element in s`
    Über Mengen kann natürlich iteriert werden
  - `s.copy()`
    Liefert eine (flache) Kopie der Menge s
  - `s.add(element)`
    Fügt das Objekt `element` zur Menge s hinzu, falls noch nicht in s
  - `s.remove(element)`
    Entfernt `element` aus der Menge s, falls vorhanden, sonst: `keyError`
  - `s.discard(element`
    Wie `remove`, aber kein Fehler, wenn `element` nicht in der Menge enthalten ist
  - `s.pop()`
    Entfernt ein willkürliches Element aus s und liefert es zurück

  - `s.clear()`
    Entfernt alle Elemente aus der Menge s

- **Benannte Methoden vs Operatoren**
  - Operator: `s &= t`
  - Benannte Methoden: `s.intersection_update(t)`
  - `s.issubset(t)`, $s \leq t$
    Testet ob alle Elemente von s in t enthalten sind
  - `s < t`
    Wie s aber echter Teilmengentest
  - `s.issuperset(t)`
    Analog für Obermengentests bzw. echte Obermenge
  - `s == t`
    Gleichheitstest
  - `s != t`
    Äquivalent zu `not (s == t)`
- **Klassische Mengenoperationen**
  - `s.union, s | t`
  - `s.intersection(t), s & t`
  - `s.difference(t), s - t`
  - `s.symmetric_difference(t)`
  - `s.update(t), s |= t`
  - `s.intersection_update(t), s &= t`
  - `s.difference_update(t), s -= t`
  - `s.symmetric_difference_update(t)`

Funktionsaufrufe

- **benannte Argumente:**
  - `var=wert`
  - müssen am Ende der Argumentenliste stehen
  - Reihenfolge beliebig
  - Default-Argumente
    * können beim Aufruf wegelassen werden und bekommen dann einen bestimmten Wert zugewiesen

    - werden nur einmal ausgewertet, nicht bei jedem Aufruf
    - Achtung bei veränderlichen Default-Argumenten!!!

- **Variable Argumentenlisten**
  - Notation:
    - `def f(x, xy, *spam)`
      f benötigt mindestens zwei Argumente.
      Weitere positionale Argumente werden im Tupel `spam` übergeben
    - `def f(x, **egg`
      f benötigt mindestens ein Argument
      Weitere benannte Argumente werden im dictionary `egg` übergeben
  - Erweiterte Aufrufsyntax
    - `*argtuple` und `**argdict` können nicht nur in Funktionsdefinitionen verwendet werden, sondern auch in Funktionsaufrufen

Ausnahmebehandlung

- **Zur Ausnahmebehandlung dienen die Anwiesungen:**
  - `raise, try, except, finally, else`
- `try-except`**-Blöcke:**

```
try:
    call_critical_code
except NameError as e:
    print("sieh mal einer an:", e)
except KeyError:
    print("OOPS! Ein KeyError!")
except (IOError, OSError):
    print("Na sowas!)
except:
    print(Ich verschwinde lieber")
    raise
```

- **except-Spezifikationen:**
  - `except XYError as e`
    Wenn XYError auftritt wird er der Variablen e die Ausnahme zugeordnet
  - `except XYError`
    Wenn die Ausnahme nicht im Detail interessiert kann die Variable weggelassen werden

  - `except (XYError, YZError) as e`
    Behandlung mehrerer Ausnahmetypen gleichzeitig
  - `except`
    Behandelt beliebige Ausnahme, auch CTRL-C!!!! Vorsicht!!!
  - Blöcke werden der Reihe nach abgearbeitet, Reihenfolge beachten!!!
  - kann die Ausnahme im Block nocht behandelt werden, kann sie mit einer `raise`-Anweisung weitergereicht werden
  - `try-execpt-else`-Blöcke; `else` wird ausgeführt, wenn im `try`-Block keine Ausnahme ausgelöst wurde.
  - `try-finally`-Konstruktion:
    * `finally` wird in jedem Fall ausgeführt, wenn der `try`-Block betreten wurde, egal ob Ausnahmen aufgetreten sind oder nicht
    * auch bei einem `return` im `try`-Block wird der `finally`-Block ausgeführt
  - EAFP-Prinzip:
    * "it's easier to ask for forgiveness than permission"
    * ```
      try:
          x = my_dict["key"]
      except KeyError:
          # handle missing key
      ```
  - LBYL-Prinzip:
    * "look before you leap"
    * ```
      if "key" in my_dict:
          x = my_dict["key"]
      else:
          # handle missing key
      ```

- **`raise`-Anweisung**
  - Ausnahmen werfen
  - `raise KeyError("Fehlerbeschreibung"`
  - `raise KeyError()`
  - `raise KeyError`
  - `raise` alleine verwendet man, wenn man in einer Ausnahme "weiter reichen" möchte

- **`assert`-Anweisung**
  - Mit der `assert`-Anweisung macht man eine Zusicherung:
    `assert` test[, data]

- dies ist nichts anderes als eine konditionale `raise`-Anweisung:

```
if __debug__:
    if not test:
        raise AssertionError(data)
```

  * debug ist eine globale Variable, die normalerweise `True` ist
  * wird Python mit der Option `-O` gestartet, wird debug auf `False` gesetzt

Kapitel brainfuck

- **Software Designkriterien**
  - (Praktische) Effizienz:
    Wie schnell läuft das Programm und wie viel Speicher benötigt es?
  - Skalierbarkeit:
    Wie stark wächst die Laufzeit und der Speicherbedarf mit der Größe der Eingabe
  - Eleganz: Wie "schön" sieht das Programm aus?
  - Lesbarkeit:
    Wie einfach ist das Programm zu verstehen?
  - Wartbarkeit:
    Wie einfach ist es, die Fehler zu finden oder neue Funktionen zu integrieren?
- Einlesen der Daten:

```
def read():
    prog, inp, readprog = "", "", True
    Try:
        while True:
            nextline = input()
            if readprog:
                if nextline == "!":
                    readprog = False
                else:
                    prog += nextine + "\n"
            else:
                inp += nextline + "\n"
    except EOFError:
        return (prog, inp)
```

- **Datengetriebene Programmierung:**
  - Daten werden nicht sequentiell abgearbeitet, sondern der Datenstrom determiniert die Operationen

Strings

- **generelles:**
  - Zeilenumbruch als `backslash n` (Newline)
  - Backslashes schützen sonderzeichen
  - Präfix r kennzeichnet den rohen String
    - in rohen Strings findet keine backslash Ersetzung statt
- **String-Interpolation**
  - Alternativ `format`-Methode von Strings
  - String-Interpolation wird vorgenommen, wenn der %-Operator auf einen String angewandt wird
  - Ersetzung: `string % ersetzung`
    `ersetzung` muss ein Tupel sein, mit genauso vielen Lücken, wie Lücken im String enthalten sind
  - soll ein wörtliches Prozentzeichen enthalten sein im String, notiert man %%
  - Zwischen Lückenzeichen "%" und Formatierungscode (z.B. `s` oder `r`) kann man eine Feldbreite angeben
    - Bei positiven Feldbreiten wird rechtsbündig, bei negativen linksbündig ausgerichtet
    - Bei der Angabe * wird die Felsbreite dem Ersetzungstupel entnommen
    - wichtigste Lückentypen:
      - `%s`
        Das ersetzte Element wird so formatiert, wie wenn es mit `print` ausgegeben würde
      - `%r`
        Das ersetzte Element wird so formatiert, wie wenn es als nackter Ausdruck im Interpreter wingegeben würde
- `str vs repr`
  - `str`
    Lesbare Darstellung
  - `repr`
    eindeutig und von Python evaluierbare darstellung
- **Quine**
  - Programm, dass sich selbst repliziert
- **Lückentypen**

- %d
  Funktioniert für `ints`.
  Bei vorangestellter 0 wird mit Nullen aufgefüllt
- statt 0 kann auch mit "+", "", "-" oder "#" aufgefüllt werden
- %f
  Funktioniert für beliebige (nicht-komplexe) Zahlen
  Zahl der Nachkommastellen kann mit .i oder .* angegeben werden
- %c
  Character/Zeichen (aus String oder `int`
- %i
  Integer (wie d)
- %o
  Oktaldarstellung
- %x
  Hexadezimal
- %X
  Hexadezimal (mit Großbuchstaben)
- %e
  Exponentenschreibweise
- %E
  Exponentenscchreibweise (Großbuchstaben)
- %g
  e oder f
- %G
  E oder f
-

Dateien

- **Datei öffnen:**
  - `open(filename, mode, bufsize)`
    - `mode`:
      Bestimmt, ob die Datei gelesen oder geschrieben (oder beides) werden soll
      weggelassen $\longrightarrow$ lesend geöffnet
      - "r"
        Lesend

· "w"
schreibend
Achtung!!! Existiert die Datei bereits wird sie überschrieben

· "r+"
Lesend und schreibend

· "a"
Schreibt an das Ende einer bestehenden Datei
Legt neue Datei an falls erforderlich

· `bytearray` (mutable) und `bytes` (immutable) neue Datentypen zum Umgang mit Binärdateien

* `bufsize`:
Gibt an, ob und wie Zugriffe auf diese Datei gepuffert werden sollen

– Alternatives öffnen

```
with open(...) as myfile:
    ...# process myfile
```

- **Datei schließen**
  - `f.close()`
- **Operationen auf Dateien**
  - `f.read(n)`
    Lese n Zeichen oder alle Zeichen bis zum Ende der Datei, wenn der Parameter nicht angegeben wurde.
  - `f.readline(limit)`
    Lese eine Zeile, aber höchstens `limit` Zeichen, wobei das Zeilenendezeichen erhalten bleibt
    Letzte Zeile ist leer!!!
  - `f.readlines(hint)`
    Liest alle Zeilen in eine Liste, wobei aber nur so viele Zeilen gelesen werden, dass `hint` Zeichen nicht überschritten werden, falls angegeben
  - `f.write(string)`
    Hängt einen String an die Datei an (oder überschreibt)
- **Dateien: Iterationen**
  - `for line in f`
  - über Dateien kann ebenso wie über Sequenzen iteriert werden
  - Bei jedem Schleifendurchlauf wird eine Zeile aus der Datei gelesen und der Schleifenvariablen (hier `line` zugewiesen (inkl. Newline-Zeichen am Ende auch unter Windows!!!)

  - `list(f)`
    erstellt eine Liste mit allen Zeilen einer Datei
  - `max(f)`
    Bestimmt die lexikographisch größte Zeile

- **Dateien Ausgabe**
  - `print(..., sep=x, file=f, end=" ")`
    * `file`
      zum schreiben in eine Datei f
    * `sep`
      Bestimmt den seperator zwischen einzelnen Ausdrücken
    * `end`
      Es wird kein Zeilenende erzeugt
      stattdessen wird die Ausgabe von nachfolgenden Ausgaben durch das Zeichen nach `end` getrennt

- **Dateinamen und Ordner**
  - absoluter Pfadname
  - relativer Pfadausdruck:
    * ein Programm läuft immer in einem aktuellen Ordner
    * relativ dazu kann man einen Pfad angeben (kein / am Anfang)
    * `..`
      Gehe eine Ordnerebene hoch
    * Initial ist der aktuelle Ordner in dem das Skript gestartet wurde
      IDLE hat immer einen festen Ordner

- **Persistente Dateien**
  - Dateien die über das Programm hinaus aufbewahrt werden sollen
  - siehe Modul `shelve`

- **Pipes**
  - in Unix "—"
  - ```
    >>> p = os.popen("date")
    >>> print(p.read())
    Mon Nov 25 21:45:44 CET 2013

    >>> print(p.close())
    None
    ```
    ```

OOP

- **Beschreibung eines Systems anhand kooperierender Objekte**
- **Klasse**
  - "Bauplan" für bestimmte Objekte
  - enthält die Definition der Attribute und Methoden
  - macht alleine praktisch gar nichts
- Klassenhierarchie
  - Superklasse, Oberklasse, Elternklasse oder Basisklasse (für die obere Klasse)
  - Subklasse, Unterklasse, Kindklasse bzw. abgeleitete Klasse (für die unteren Klassen)
  - Unterklassen erben Attribute und Methoden von Oberklassen
  - Unterklassen können Attribute und Methoden einführen oder bestehende überschreiben
- **Objekte**
  - in der realen Welt: Zustand und Verhalten
  - in OOP modelliert durch: Attributwerte bzw. Methoden
  - wird dem "Bauplan" entsprechend erzeugt
  - ist dann eine Instanz der Klasse
- **Datenkapselung**
  - interner Zustand wird versteckt
  - Python ist liberal
  - Attribute die nicht mit Unterstrich beginnen, sind für alle sichtbar
  - Attribute die mit einem Unterstrich beginnen sind intern und sollten außerhalb nicht benutzt werden
  - Attribute die mit zwei Unterstrichen beginnen sind nicht direkt sichtbar, da der Klassenname intern mit eingefügt wird (Namen-Massage)
- **Vorteile OOP**
  - Abstraktion
  - Vererbung
  - Datenkapselung
  - Wiederverwendbarkeit
- **Nachteile von OOP**

  - Formulierung
  - Klassenhierarchie
  - Transparenz
  - Ausführungseffizienz
  - Programmiereffizienz

- **Erzeugung von Instanzen**
  - Klasse aufrufen wie eine Funktion:

```
>>>class MyClass:
...    pass
...
>>> instance1 = MyClass()
>>> instance1
<__main__.MyClass object at .......>
>>> instance2 = MyClass()
>>> instance2 is instance1
False
>>> instance1 == instance2
False
```

  - Instanzen kann man dynamisch neue Attribute zuordnen
  - Instanzen haben einen eigenen Namensraum

- **Methoden**
  - werden als Funktionen innerhalb von Klassen definiert
  - Den ersten Parameter einer Methode nennt man per Konvention self
    Dies ist die Instanz / das Objekt
  - können über den Klassennamen (dann muss das `self`-Argument angegeben werden
  - Normal über den Instanznamen (dann wird die Instanz implizit übergeben)

- **Konstruktion**
  - `__init__`
    wird aufgerufen wenn die Instanz erzeugt wird
  - In dieser Methode erzeugt man die Attribute durch Zuweisungen
  - Beachte: Alle Attribute sind öffentlich zugreifbar
  - Beachte: Auch bei Methoden-Definitionen sind benannte und Default-Parameter möglich
  - Beachte: Attributnamen und Parameternamen von Methoden gehören zu verschiedenen Namensräumen

```
>>>class Circle:
...     def __init__(self, radius=1):
...         self.radius = radius
...
>>> circle = Circle(5)
```

- **Vererbung**
  - Aufruf von Methoden der Elternklasse:
    - `Elternklasse__Methode__(self, Parameter)`
    - `super().__Methode__(Parameter)`
    - Beachte: Die Parameter müssen bekannt sein oder man muss mit `**kwlist` arbeiten
- **Klassenattribute (oder statische Attribute)**
  - Variablen die innerhalb des Klassenkörpers eingeführt werden
  - sind (auch) in allen Instanzen (zum Lesen) sichtbar
  - Zum Schreiben müssen sie über den Klassennamen angesprochen werden
- `tkinter`
  - ```
    import tkinter as tk
    import sys


    root = tk.Tk()
    lab = tk.Label(root, text="Hello World")
    lab.pack()

    if "idlelib" not in sys.modules:
        root.mainloop()
    ```
  - `root` wird das Wurzelobjekt, in das alle anderen Objekte hineinkommen
  - `lab` wird ein Label-Widget innerhalb des `root`-Objekts erzeugt
  - Dann wird `lab` in seinem Elternfenster positioniert
  - Schließlich wird die Eventschleife aufgerufen
  - IDLE selbst ist mit Hilfe von `tkinter` implementiert worden
  - man muss daher etwas vorsichtig sein, wenn man `tkinter` in IDLE entwickelt
  - Man sollte nicht nochmal `mainloop` aufrufen (dafr sorgt das if-statement
  - man sollte das Programm nicht beenden, da sonst `tkinter` mit beendet wird
  - `tk.Canvas(root, height=x, width=y)`
    ```

    * Ein Canvas ist eine Leinwand, auf der man "malen" kann
    * Darauf kann man dann verschiedene geometrische Figuren erzeugen
    * Nullpunkt immer oben links
    * Canvas-Methoden:
        · `canvas.create_line(x1, y1, x2, y2, **kw)`
        zeichnet Linie von (x1, y1) nach (x2, y2)
        · `canvas.create_rectangle(x1, y1, x2, y2, **kw)`
        zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2)
        · `canvas.create_oval(x1, y1, x2, y2, **kw)`
        zeichnet ein Oval innerhaln des Rechtecks geformt durch obere linke Ecke (x1, y1) und untere rechte Ecke (x2, y2)
        · Alle `create`-Methoden liefern den Index des erzeugten Objekts
        · `canvas.delete(i)`
        löscht Objekt mit dem Index i
        · `canvas.move(i, xdelta, ydelta)`
        bewegt Objekt um xdelta und ydelta
        · `canvas.update()`
        erneuert die Darstellung auf dem Bildschirm (auch für andere Fenster möglich)

- **Aggregierung**
    - Objekte sind oft aus anderen Objekten zusammengesetzt
    - Methodenaufrufe an einem Objekt führen dann zu Methoden-Aufrufen der eingebetteten Objekte
- **Properties**
    - Setter und Getter:

```
class Name:
    def __init__(self, x=0, y=0)
        self.x = x
        self.y = y
        Name.counter += 1

    ...
    def setX(self, x):
        print("setX Name: ", x) #Debug Ausgabe
        self._x = x
```

```
def getX(self)
    return self._x

x = property(getX, setX)
... # und für y
```

  - getX und setX sind zwei völlig normale Methoden
  - Die Zuweisung `x = property(getX, setX)` bewirkt dass `x` ein Attribut wird, wobei bei lesendem Zugriff getX und bei schreibendem Zugriff setX aufgerufen wird
  - Bei der Angabe von `None` ist entsprechend der Zugriff nicht möglich

- **Operator-Überladung**
  - wenn ein Operator je nach Kontext etwas anderes bedeutet oder macht
  - Bsp:
    arithmetische Operatoren
    "+" und "*" für Strings ist überladen

- **Mehrfachvererbung**
  - `super()` ist problematisch
    - Bei Erweiterungen von Methoden mit Hilfe von `super()` muss man mit einbeziehen, dass die Signatur (die Parameterstruktur) u.U. unbekannt ist: Man verwende eine kooperative Weise der Bearbeitung der Parameter mit Hilfe von positionalen und Schlüsselwortlisten (*list, **kwlist)
    - Dies betrifft in den meisten Fällen die `__init__`-Methode
    - Es muss immer eine oberste Klasse geben, die den Schluss der `super()`-Aufrufe bildet (bei `__init__` ist das inplizit `object`
    - Achtung: Mit MRO ist es möglich das mit `super()` micht eine Superklasse sondern eine Geschwisterklasse als nächstes aufgerufen wird
  - MRO
    - Im Normalfall kann man die MRO mit der Regel links vor rechts, wobei immer Unterklasse vor Oberklasse kommen muss, einfach selbst bestimmen
    - Die Standard-Klassenmethode `mro()` gibt die Liste der Oberklassen entspechend der MRO aus
  - der C3-Algorithmus
    - Die Linearisierung einer Klasse C mit den (geordneten) Superklassen S1, ...Sn, symbolisch L(C), ist eine Liste von KLassen, die rekursiv wie folgt gebildet wird:

$$L(C) = [C] + merge(L(S_1), ...L(S_n), [S_1, ..., S_n])$$

    - Die Funktion merge selektiert dabei nacheinander Elemente aus den Listen und fügt diese der Linearisierung hinzu
    - merge funktion
        1. Es wird das erste Element (der head) der ersten Liste betrachtet
        2. Taucht dieses nicht als zweites oder späteres Element in einer der späteren Listen auf (im tail), dann wird es zur Linearisierung hinzugenommen und aus allen Listen gestrichen
        3. Ansonsten lässt man die erste Liste so und probiert das erste Element der nächsten Liste usw.
        4. Nachdem ein Element entfernt wurde, fängt man wieder mit der ersten Liste an
        5. Können so alle Listen geleert werden, ist das Ergebnis die Linearisierung von C
        6. Ansonsten gibt es keine Linearisierung!
    - Ablauf:
        - beginne bei der höchsten Klasse
        - stelle die Linearisierung L(Klasse) für jede Klasse als Liste auf

- **Deligieren**
    - die Superklasse führt eine Methode ein, überlässt die exakte Implementierung aber den Subklassen

Magische Methoden

- **Methoden die mit zwei Unterstrichen beginnen und enden bezeichnet man als magisch**
- **Magische Methoden sind nicht prinzipiell anders als andere Methoden**
- **Python versucht jedoch intern bspw. bei der Addition die Methode `__add__` aufzurufen**
- **3 Arten von magischen Methoden:**
    - Allgemeine Methoden
      Verantwortlich für Objekterzeugung, Ausgabe und ähnliche grundlegende Dinge
    - Numerische Methoden
      Verantwortlich für Addtion, Bitshift und ähnliches
    - Container-Methoden
      Verantwortlich für Indexzugriff, Sicing und ähnliches

- **Allgemeine magische Methoden**
  - Konstruktion und Destruktion:
    - `__init__`
    - `__new__`
      ist im Wesentlichen für fortgeschrittene Anwendungen mit nicht-Python-Klassen interessant
    - `__del__`
      wird aufgerufen, wenn das Objekt aus dem Speicher gelöscht wird, weil es über keinen Namen mehr erreichbar ist: Destruktor
      Sollte nicht verwendet werden um ein Objekt auf der Programmierungsebene "abzumelden", da nicht direkt vorhersehbar ist, wann `__del__` aufgerufen wird
  - Vergleich und hashing:
    - `obj.__eq__(other)`
      Wird aufgerufen bei Test `obj == other`
    - `obj.__ne__(other)`
      Wird bei Test `obj != other)` aufgerufen
    - definiert man diese Methoden nicht, werden Objekte nur auf Identität verglichen
    - `obj.__ge__(other)`
      Wird bei Test `obj >= other` aufgerufen
      Wird auch bei Aufruf <= verwendet, falls `other` über keine `__le__`- Methode besitzt
    - `obj.__gt__(other), obj.__le__(other), obj.__lt__(other)`
      Wird analog für die Vergleiche `obj > other` aufgerufen
    - `obj__hash__(self)`
      Liefert einen hashwert für `obj` bei Verwendung in einem Dictionary
      Wird von der Builtin-Funktion `hash` verwendet
    - `obj.__bool__(self)`
      Wird von `bool(obj)` aufgerufen und damit auch bei `if obj:` und `while obj:`
      Sollte True zurückliefern, wenn das Objekt als wahr einzustufen ist, sonst False
  - Sting Konversion:
    - `__repr__`
      Wird aufgerufen, um eine String-Darstellung von `obj` zu bekommen
      `obj.__repr__(self)` sollte eine möglichst exakte (für Computer geeignete) Darstellung erzeugen, idealerweise eine die korrekte Python-Syntax wäre , um dieses Objekt zu erzeugen

    - `obj.__str__(self)`
      Wird aufgerufen, um eine String-Darstellung von `obj` zu bekommen
      `__str__` sollte eine menschenlesbare Darstellung erzeugen

  - Attributzugriff

    - `obj.__getattr__(name)`
      Wird aufgerufen, wenn für `obj.name` kein Attribut gefunden wird.
      Soll enteweder einen Wert liefern oder einen `AttributeError` erzeugen
    - `obj.__getattribute__(name)`
      Wird bei jedem lesenden Zugriff auf `obj.name` aufgerufen
      Wichtig: Um innerhalb der Methode auf den Wert zuzugreifen muss man die `__getattribute__`-Methode der Superklasse aufrufen
    - `__setattr__(name, value)`
      Wird bei jedem schreibenden Zugriff auf `obj.name` aufgerufen.
      Das bei `__getatrribute__` gesagte gilt entsprechend

- **Numerische magische Methoden**

  - Bei Operatoren wie $+, *, -$*oder*$/$ verhält sich Python wie folgt (am Bespiel $+$)

    - Zunächst wird versucht die Methode `__add__` des linken operanden mit dem rechten Operanden als Argument aufzurufen
    - Wenn `__add__` mit dem Typ des rechten Operanden nichts anfangen kann, kann sie die spezielle Konstante `NotImplemented` zurückliefern
    - dann wird versucht die Methode `__radd__` des rechten Operanden mit dem linken Operanden als Argument auzurufen
    - Wenn das auch nicht funktioniert, schlägt die Operation fehl

  - Grundrechenarten

    - `+`: `__add__` und `__radd__`
    - `-`: `__sub__` und `__rsub__`
    - `*`: `__mul__` und `__rmul__`
    - `/`: `__truediv__` und `__rtruediv__`
    - `//`: `__floordiv__` und `__rfloordiv__`
    - `%`: `__mod__` und `__rmod__`
    - unäres `-`: `__neg__`

  - Boolsche Operatoren

    - `&`: `__and__` und `__rand__`
    - `|`: `__or__` und `__ror__`

    * `Dach: __xor__ und __rxor__`
    * `<<: __lshift__ und __rlshift__`
    * `>>: __rshift__ und __rrshift__`
    * `Welle (unär): __invert__`
  - Bei Klassen, deren Instanzen veränderlich sein sollen, wird man in der Regel zusätzlich zu den Operatoren wie + auch Operatoren wie += unterstützen wollen
  - Dazu gibt es zu jeder magischen Methode für binäre Operatoren wie `__add__` auch eine magische Methode wie `__iadd__`, die das Objekt selbst modifiziert und `self` zurückliefern sollte
  - Implementiert man `__add__`, aber nicht `__iadd__`, dann ist x += y äquivalent zu x = x + y
- **Container-Methoden**
  - `obj.__len__(self)`
    Wird von `len(obj)` aufgerufen
  - `obj.__contains__(item)`
    Wird von `item in obj` aufgerufen
  - `obj.__iter__(self)`
    Wird von `for x in obj` aufgerufen
    Ferner für Zugriffe mit der eckigen Klammer
  - `obj.__getitem__(key)`, `obj.__setitem__(key, value)`
    Wird aufgerufen wenn mit `obj[key]` zugegriffen wird
  - `obj.__delitem__(key)`
    Wird beim löschen eines Items aufgerufen

Methoden für Klassen

- **Ohne Instanz als erstes Argument aufrufbar**
- **statische Methoden**
  - kein Instanzen-Argument wird übergeben
  - innerhalb der Klasse wie normale Funktion definiert (ohne `self`-Parameter), dann folgt die Zeile
    `methodname = staticmethod(methodname)`
- **Klassenmethoden**
  - nützlich, damit nicht in jeder Subklasse eine Klassenvariable eingeführt werden muss

  - als erster Parameter wird ein Klassenobject übergeben
  - ähnlich wie statische Methoden, aber mit folgender Zeile deklariert
    `methodname = classmethod(methodname)`

- **Übersicht**
  - Instanzmethoden
    (normale) Methoden die auf einer Instanz agieren und als ersten Parameter immer die Instanz `self` erwarten
  - Statische Methoden
    Methoden die keine Referenz auf das Klassenobject haben.
    Diese sollten am besten dann benutzt werden, wenn nur auf lokale Klassenvariablen zugegriffen werden soll
  - Klassenmethoden
    Methoden die als erswten Parameter ein Klassenobjekt erwarten
    Gut zu beniútzen , wenn es Attribute gibt die in mehreren Klassen mit gleichem Namen eingeführt werden
  - Metaklassenmethoden
    ...

Ethik

- **GI**
  - 4 Bereiche:
    * Das Mitglied:
      · Fachkompetenz
      · Sachkompetenz
      · kommunikative Kompetenz
      · juristische Kompetenz und Urteilsfähigkeit
    * Das Mitglied in Führungspositionen
      · Arbeitsbedingungen
      · Organisationsstruktur und Beteiligung
    * Das Mitglied in Lehre und Forschung
      · Lehre/Forschung
    * Sie Gesellschaft für Informatik
      · Zivilcourage

        - soziale Verantwortung
        - Mediation
        - interdisziplinäre Diskurse

- **ACM**
  - 8 Bereiche
    - Public
      - act consistently with the public interest
    - Client and Employer
      - act in a manner that is in the best interest of their client and employer
    - Product
      - ensure that their products meet the highest professional standards possible
    - Judgement
      - maintain integry and independence in their professional judgement
    - Management
      - promote an ethical approach to the management of software development and maintenance
    - Profession
      - advance the integry and reputation of the profession consistent with the public interest
    - Colleagues
      - be fair to and supportive of their colleagues
    - self
      - participate in lifelong learning and shall promote an ethical approach to the practice of the proffession

Regular Expressions

- **Syntax regExp**
  - rohen String verwenden, da "\" häufig vorkommt
  - .
    matcht ein beliebiges Zeichen

- \.
  Wenn man wirklich . meint
- Sonderzeichen in Zeichenklassen:
  * "."
    ist kein Sonderzeichen in Zeichenklassen
  * "\"
    ist das Quotierungszeichen, mit dem man andere Sonderzeichen präfigiert, um diese in die Zeichenklasse aufzunehmen
    z.B. `[.\- \\]` bezeichnet Zeichenklasse aus Punkt, Minus, Rückstrich
  * "-"
    Bereichssonderzeichen
  * "]"
    Abschlusszeichen
  * "^"
    Komplementzeichen, wenn es das erste Zeichen ist
- vordefinierte Zeichenklassen:
  * \d
    matcht alle Ziffern, dh im ascii 0-9, normalerweise aber auch alle Ziffern in anderen Schriftsystemen
  * \D
    matcht alles, was keine Ziffer ist
  * \s
    matcht alle Weißtraumzeichen
  * \S
    matcht Komplement zu \s
  * \w
    ist im ascii Fall `[a-zA-Z0-9]`
    Im Unicode Fall kommen alle Buchstaben und Ziffern aus anderen Schriftsystemen dazu
  * \W
    matcht Komplement zu \w
- manchmal ist es sinnvoll den leeren String zu matchen:
  * \A
    passt nur am Anfang des Strings, dh. `re.match(regex, string)` ist äquivalent zu `re.search(r"\A" + regex, string)`
  * \b
    passt nur vor und nach jedem Wort (bestehend aus \w-Zeichen)

    * \B
      matcht nur dann, wenn \b nicht matcht
    * \Z
      passt nur am Ende des Strings
    * ^
      passt wie \A nur am Anfang eines Strings
      Wenn `MULTILINE`-flag gesetzt, passt es an jedem Zeilenanfang
- Gruppenbildung und Alternativen
    * "|"
      Alternativ-Operator
      bindet weniger stark als die Konkatenation
    * "()"
      Runde Klammern können zur Gruppenbildung genutzt werden
- Einfache Quantoren:
    * oft soll ein Zeichen wiederholt werden
    * "?"
      Das vorangegangene Zeichen oder die Gruppe ist optional
    * "*"
      Das Zeichen oder die Gruppe kann beliebig oft (auch 0-mal) wiederholt werden
    * "+"
      Das Zeichen oder die Gruppe kann beliebig oft (aber mindestens einmal) wiederholt werden
- Zählquantoren
    * {i}
      spezifiziert, dass die vorangegangene Gruppe (bzw. Zeichen) genau i-mal wiederholt werden soll
    * {min, max}
      gibt an, dass die Gruppe oder das Zeichen `min` und `max` mal wiederholt werden soll
    * { ,max}
      gibt Obergrenze an
    * {min, }
      gibt untere Grenze an
- Gruppen und Rückbeziehung
    * Jedesmal wenn eine gruppe mit runden Klammern gebildet wird, wird der damit gematchte Teilstring referenzierbar gemacht

    * \n
      zugreifen aif den Teilstring, wobei n eine Zahl zwischen 1 und 99 entsprechend der Stellung der Gruppe im regulären Ausdruck ist (Position der öffnenden Klammer zählt)
    * Gibt es Gruppen im regulären Ausdruck, dann gibt `findall` statt dem gematchten Text die von der Gruppe gematchten Teilstring in Tupeln zurück
- Gieriges (greedy) und genügsames (non-greedy) Matchen
    * Der Matcher versucht immer, möglichst viel im String zu überdecken
    * Nachgestelltes "?" führt zu genügsamem Matchen, versucht also einen möglichst kurzen Teilstring zu matchen
- Erweiterungen:
    * `Flags`
        · `(?aiLmsux)`
        · erlaubt es, eine oder mehr Flags für den gesamten Ausdruck zu setzen (alternativ können die beim Aufruf der Match-Funktion angegeben werden)
        · a `re.a`
          ASCII Matching
        · i `re.I`
          ignoriere Groß- und Kleinschreibung
        · L `re.L`
          Locale, vordefinierte Zeichenklassen werden von aktueller Lokalisierung abhängig gemacht - vom Gebrauch wird abgeraten
        · m `re.M`
          Multi-Zeilen Matching (betrifft $ und ^)
        · s `re.S`
          Der Punkt matcht auch \n
        · u `re.U`
          nur für Rückwärtskompatibilität, da auf allen Unicode-Strings per Default mit Unicode-matching gearbeitet wird
        · x `re.x`
          Weißraumzeichen im regulären Ausdruck werden ignoriert, wenn sie nicht mit einem \eingeleitet werden
          Nach # kann man kommentieren
    * Anonyme Gruppen und Gruppennamen

  - `(?:...)`
    Hiermit wird eine "anonyme" Gruppe gebildet, auf die kein Bezug genommen werden kann
  - `(?P<name>...)`
    Hiermit wird eine benannte Gruppe erzeugt, auf die man sich mit dem Namen `name` beziehen kann
  - `(?P=name)`
    Ist das Gegenstück, mit dem man sich auf die benannte Gruppe beziehen kann

* Positive Rück- und Vorschau
  - `(?=...)`
    matcht, falls der Ausdruck ... als nächstes matcht!
    Dabei wird dieser aber nicht konsumiert, sondern kann wieder verwendet werden
    Dh. wir schauen voraus
    Man redet von positiver Vorschau (positive lookahead)
  - `(?<=...)`
    matcht falls der Ausdruck ... den String vor der aktuellen Position matcht!
    Dabei wird dieser aber nicht konsumiert, sondern kann wieder verwendet werden
    Dh. wir schauen zurück
    Man redet von positiver Rückschau (positive lookahead)
    Achtung: Der Ausdruck muss Strings fester Länge beschreiben!
* Negative Rück- und Vorschau
  - `(?!...)`
    matcht falls der Ausdruck ... als nächstes nicht matcht!
    Dabei wird dieser aber nicht konsumiert
  - `(?<!...)`
    matcht falls der Ausdruck ... den String vor der aktuellen Position nicht matcht!
    Dabei wird dieser aber nicht konsumiert
    Auch hier muss der Ausdruck Strings einer festen Länge beschreiben!
* Konditionales Matching
  - `(?(id/name)yes-pattern|no-pattern)`
    Falls die Gruppe mit dem angegebenen Index oder Namen einen wert erhalten hat, wird der erste Teil zum matchen benutzt, sonst der zweite Teil (der optinal ist)
  - Generell werden konditionale Pattern aber als schwierig zu lesen und eher überflüssig angesehen

Das WWW befragen

- **Das `urllib`-Paket**
  - Komfortable Schnittstelle, um auf Ressourcen im WWW zuzugreifen
  - enthält mehrere Module:
    - `urllib.request`
      Enthält Funktionen und Klassen zum Zugriff auf Ressourcen im Internet
      - `urlopen(url, data=None, timeout, *, cafile=None, capath=None, cadefault=False)`
        Wichtigste Funktion aus `urllib.request`
        Stellt ein Datei-ähnliches Objekt zur Verfügung
        `url` ist die URL auf die zugegriffen werden soll
        `data` sind zusätzliche Daten, die bei einer Anfrage geschickt werden
        `timeout` ist ein optionaler Parameter für eine obere Zeitschranke
        Die anderen Parameter sind für Zertifikate (bei HTTPS)
      - Nach `urlopen` kann man auf dem resultierenden Objekt `read`-Methoden anwenden und erhält `bytes` zurück
      - Das funktionierte bisher, aber:
      - Webseitenbetreiber mögen keine Zugriffe über Skripte
      - daher vortäuschen eines firefox-Browsers:
        ```
        ...
        from urllib.request import Request
        req = Request(url="http://www.wetteronline.de/", data=b"None"
        headers="User-Agent":"Mozilla/5.0 (Windows NT 6.1; WOW64; rv:12.0)
        Gecko/20100101 Firefox/12.0")
        ```
    - `urllib.parse`
      Unterstützt das parsen von URLs (Universal Resource Locators)
- **Web-Scraping**
  - Informationen von Webseiten auslesen
  - im privaten ok, sonst fast immer illegal
  - Achtung! Webserver Belastung
  - Alternativ: Webservices nutzen!!!

Effiziente Programme

- **Grundregeln**

1. Schreibe lesbaren Code
2. Überprüfe Korrektheit (schreibe Tests)
3. Optimiere die Implementierung dort so es sich lohnt
   - Softwarewerkzeuge können bei der Analyse hilfreich sein
   - Profiler:
     Tool mit dem sich Laufzeitverhalten von Programm-Code analysieren lässt
     erlaubt es verwschiedene Implementierungen zu vergleichen
   - Messen der Laufzeit
     Wie oft wird eine Funktion aufgerufen?
     Wie lange dauert das Ausführen der Funktion?
   - Speichernutzung:
     Wie viel Arbeitsspeicher wird benötigt?
     Wird nicht benötigter Speicher wieder freigegebe?
     IN Python: Garbage-Collection

- **Daumenregeln für Python**
  - Benutze Pathon Tuples anstelle von Listen, sofern nur eine immutable Sequenz benötigt wird
  - Benutze Iteratoren anstelle von großen Tuples oder Listen (sofern die Sequenz nicht wiederholt gebraucht wird)
  - Benutze (wo möglich und sinnvoll) Python's built-in Funktionen und Datenstrukturen
  - Benutze iterative anstelle rekursiver Lösungen, sofern möglich
  - Solche Daumenregeln sind aber im Einzelfall zu überprüfen!!!
- **Zeitmessung mit dem Modul `time`**
  - Welche Zeiten kann man messen?
    * process time:
      Die Zeit in der die CPU für den Prozess aktiv war
    * Kernel time:
      Die Zeit der Process time, die der Kernel-(System-)Routinen verbracht wurde
    * User space time:
      Process time - Kernel time
    * Wall clock time:
      Die tatsächlich vergangene Zeit vom Start des Prozesses bis zum Ende (inkl. Warten auf I/O und Schlafen)

  - Welche Zeiten sind relevant?
    * `time.process_time()`:
      Will man einen CPU-intensiven Algorithmus evaluieren, dann misst man normalerweise die Prozesszeit
    * Das herausrechnen der Kernelzeit geht in Python nicht
    * `time.time()`:
      Ist man an antwortzeiten interessiert, dann ist die Wanduhr-Zeit entscheidend. Auf einer Einbenutzermaschine mit einem CPU-intensiven Prozess sollte es praktisch keinen Unterschied zur Prozesszeit geben.

- **Das Modul `timeit`:**
  - Das Modul `timeit` erlaubt es kleine (und auch größere) Programm-Teile auf ihre Laufzeit zu untersuchen
  - Wie das Doctest-Modul hat das Modul `timeit` auch ein Command-line Interface
  - Neben der Klasse `Timer`, mit dem sich spezielle Timer-Objekte erzeugen lassen, die folgenden beiden Funktionen zur Verfügung:
    * `timeit.timeit(stmt='pass', setup='pass', timer=<default timer>, number=1000000)`:
      Erzeugt eine Timer Instanz mit dem gegebenen Python-Snippet `stmt` (quotiert) und einem Python-Snippet `setup`, der initial ausgeführt wird. `timer` ist per Default `time.perf_counter()`. Anschließend wird die `timeit()`-Methode des Timers `numbers`-oft aufgerufen.
    * `timeit.repeat(stmt='pass', setup='pass', timer=<default timer>, repeat=3, number=1000000)`:
      Wie die letzte Funktion mit dem Unterschied, dass die `repeat`-Methode des Timers aufgerufen wird (und zwar `repeat`-mal

- **Funktionsaufrufe zählen und Memoisierung**
  - Man kann Zwischenergebnisse speichern (z.B. in einem Dict) und somit Funktionen weniger häufig aufrufen
  - Die Implementierung ist dann bei weitem effektiver
  - Bsp: Fibonacci Zahlen

- **Das Modul `cProfile`**
  - Bei großen Programmen muss man zunächst feststellen wo sich das Optimieren lohnt
  - dafür gibt es Profiler
  - `cProfile.run(command, sort=1`

    * `command` ist das Python-Kommando (als String); das aufgerufen werden soll, `sort` spezifiziert, nach welcher Spalte sortiert werden soll.
    * Es wird dann eine Tabelle ausgegeben, in der in jeder Zeile eine Funktion beschrieben wird. Es gibt folgende Spalten:
        - `ncalls`:
          Anzahl der Aufrufe
          Im Fall von zwei Zahlen beschreibt die erste Zahl die totale Anzahl von Aufrufen, die zweite die Anzahl der primitiven (nicht rekursiven) Aufrufe
        - `tottime`:
          CPU Sekunden, die die Funktion verbraucht hat ohne Zeit für aufgerufene Funktionen zu berücksichtigen
        - `percall=totttime/ncalls`
        - `cumtime`:
          CPU Sekunden, die die Funktion inklusive der Zeit für aufgerufene Funktionen verbraucht hat
        - `percall = cumtime/ncalls`

- **Tracing mit dem Modul `trace`**
    - Der Profiler arbeitet nur auf Ebene von Funktionen, nicht auf Ebene von Zeilen
    - Manchmal möchte man wissen welche Zeilen nicht ausgeführt wurden, das ist wichtig wenn man alle Zeilen mindestens einmal getestet haben möchte
      $\longrightarrow$ Überdeckungsanalyse mit dem Modul `trace`
    - hier wird gezählt wie oft eine Zeile ausgeführt wird

Laufzeitanalyse von Algorithmen

- **In der Informatik untersucht man Algorithmen meist darauf, wie gut sie skalieren:**
  Wie stark wächst die Laufzeit (oder der Speicherplatzbedarf) mit der Größe der Eingabe?
- **Wie misst man die Laufzeit von Algorithmen?**
    - Identifizierung der (etwa gleich teuren Grundoperationen (z.B. Vergleiche, arithmetische Operationen Zuweisungen usw.) und bestimmt, wie häufig die bei der Ausführung des Algorithmus A bei einer bestimmten Eingabe x ausgeführt werden
    - Dies sei die abstrakte Laufzeit von A auf x: $T_A(x)$

  - Darauf basierend kann man über alle Eingaben der Größe n gehen und die Laufzeit für die Größe n im besten, im schlechtesten und im mittleren Fall bestimmen:
    * Bester Fall: $T_A^b(n) = minT_A(x) \mid x \mid= n$, x Eingabe für A
    * Schlechtester Fall: $T_A^w(n) = maxT_A(x) \mid x \mid= n$, x Eingabe für A
    * Mittlerer Fall: $T_{A,q_n}^a(n) = \sum_{|x|\text{n, x Eingabe für A}} T_A(x)q_n(x)$
  - Für das Laufzeitwachstum (man spricht auch vom asymptotischen Laufzeitverhalten) sind i.W. die Anzahl der Schleifendurchläufe entscheidend.
  - Man betrachtet dabei meist den schlechtesten Fall, da er einfach zu bestimmen ist und eine Garantie gibt

- **Suche in sortierten Listen**
  - Nehmen wir an, dass die Liste sortiert ist, so gibt es einen effizienteren Suchalgorithmus:
    * Die binäre Suche
    * Wir gehen ähnnlich ewie bei einer Suche im Telefonbuch vor:
      1. Wir betrachten das ganze Buch als interessant
      2. Wir wählen im interessanten Bereich die mittlere Seite und schauen, ob der gesuchte Name da steht. Falls ja sind wir fertig
      3. Falls der Name später in der Lexikon-Ordnung kommt, dann konzentrieren wir uns auf die hintere Hälfte
      4. Ansonsten auf die vordere
      5. Die neue ausgewählte Hälfte ist unser neuer interessanter Bereich und wir machen mit Schritt 2 weiter

- **Die O-Notation (Landausche o-Notation)**
  - Mit O(g) bezeichnet man die Menge von Funktionen f, für die gilt:

$$\exists\, c \in R^+, x_0 \in R^+, f.a.\ x > x_0 : f(x) \leqslant cg(x)$$

  - Dh. O(g) umfasst alle Funktionen f, die nicht schneller wachsen als g (wenn man konstante Faktoren ignoriert und endliche Anfangsstücke vernachlässigt)
  - Notation: $f \in O(g)$
  - einfache Regeln:
    * f = O(f) (= bedeutet $\in$)
    * O(O(f)) = O(f) (= bedeutet hier und im weiteren $\subseteq$)
    * O(kf) = O(f) für eine Konstante $k \geq 0$

    * O(k+f) = O(f) für eine Konstante k
    * Additionsregel: O(f) + O(g) = O(max{f, g})
      Relevant für Hintereinanderausführung von Anweisungen in Programmen
    * Multiplikationsregel: $O(f) \cdot O(g) = O(f \cdot g)$
      Relevant für die ineinander Schachtelung von Schleifen in Programmen
  - Hierarchie und Größenordnung
    * O(1): Konstante Funktionen
    * $\mathcal{O}$(log n): logarithmische Funktionen
    * O(n): lineare Funktionen
    * O(n log n): log-lineare Funktionen
    * O(n^2): quadratische Funktionen
    * O(n^k): für beliebiges festes $k \in N$: polynomielle Funktionen
    * O(k^n)): für beliebiges festes $k \in N$: exponentielle Funktionen
  - Bestimmen der Größenordnung der Laufzeit für Programmstück A
    * A ist einfache Zuweisung oder I/O-Anweisung:
      O(1)
    * A ist eine Folge von Anweisung oder Folge von Operationen:
      Additionsregel anwenden
    * A ist eine `if`-Anweisung:
      · `if cond B`: Additionsregel für Laufzeit von `cond` und Laufzeit von B
      · `if cond: B else: C`: Maximum der Laufzeit von B und C. Dann Additionsregel für `cond` und das Maximum
    * A ist eine Schleife `"while cond"`: Bestimme Maximum der Laufzeit von `cond` und B innerhalb der Schleifenausführung. Multipliziere mit Anzahl der Schleifenausführung
    * Wenn A `for`-Schleife ist, entsprechend
    * A ist Funktionsaufruf: Bestimme den Laufzeitaufwand für die aufgerufene Funktion

- **Skalierbarkeit**
- **Weiter Ressourcenmessungen und asymptotisch Notationen**
  - Wir haben bisher nur die Zeit als wesentliche Ressource gemessen. Man kann aber auch:
    * den Verbrauch an Speicherplatz bestimmen
    * den Kommunikationsaufwand (die Bandbreite) bestimmen

  - Es gibt außerdem weitere asymptotische Notationen, die in der Informatik weniger häufig auftauchen...

- **Komplexitätstheorie**
  - eine Stufe abstrakter:
    * Frage nach dem Laufzeitbedarf eines ProblemBeispiel: Welches Laufzeitwachstum hat der beste Algorithmus für das Suchen eines Elements in einer sortierten Liste
      $\longrightarrow$ Hier quantifizieren wir über alle möglichen Algorithmen für das Problem!!!
    * Das bekannte Milleniumsproblem, ob P = NP ist, entstammt diesem Gebiet

- **NP-Vollständigkeit**
  - Es gibt eine Menge von Problemen, bei denen es einfach ist zu überprüfen, ob eine gegebene Struktur eine Lösung ist. Es gibt aber keinen Algorithmus bekannt, der schnell eine Lösung findet
  - Beispiel:
    Ist eine gegebene Boolsche Formel erfüllbar (SAT), dh. gibt es eine Belegung der Boolschen Variablen, die Formel wahr macht: $(a \vee b) \wedge (c \vee \sim a)$
  - Bei gegebener Belegung (a=1, b=0, c=1) einfach überprüfbar. Eine erfüllende Belegung kann man (vermutlich nur) durch Ausprobieren finden
  - Solche Probleme kann man formal charakterisieren und bezeichnet sie als NP-vollständig
  - Wenn P = NP, dann kann man Lösungen in Polynomialzeit finden
  - Wenn $P \neq NP$, wird man nie effiziente Algorithmen finden

- **Quadratisch Falle**
  - quadratisch Laufzeiten sind dramatisch schlechter als lineare oder log-lineare
  - vermeiden!!!
  - Beispiel:
    * Es kommt ein Datenstrom mit monoton wachsenden Zahlen herein, der möglicherweise Doppelungen enthält
    * Alle auftretenden Zahlen sollen in einer Liste aufsteigend gespeichert werden
  - Mögliche Lösung

```
def record_data(newelement, li):
    if not newelement in li:
        li.append(newelement)
```

- Laufzeit in $O(n^2)$, da der `in`-Test linear list in der Länge
- Alternativ:
  andere Datenstruktur verwenden
- `Dict` und `set` haben konstante Zugriffszeit
- bessere Lösung

```
def record_data_fast(newelement, li):
    if newelement != li[-1]:
        li.append(newelement)
```

Funktionale Programmierung

- **Programmierparadigmen**
  - Imperativ:
    Man beschreibt, wie etwas erreicht werden soll
  - Deklarativ:
    Man beschreibt was erreicht werden soll
- **Imperative Programmierparadigmen**
  - Allen imperativen Programmierstilen ist gemeinsam, dass der Zustand der Berechnung explizit repräsentiert und modifiziert wird (Variablen und Zuweisungen)
  - Prozedurale Programmierung:
    - Die Aufgabe wird in kleinere Teile - Unterprogramme - zerlegt, die auf den Daten arbeiten
    - PASCAL, C
  - Objekt-orientierte Programmierung:
    - Im Gegensatz zur prozeduralen Programmierung bilden Daten und die darauf arbeitenden Unterprogramme eine Einheit
    - Ihre Struktur wird durch Klassen beschrieben, die die Wiederverwendbarkeit unterstützen
    - JAVA, C++
- **Deklarative Programmierparadigmen**
  - Allen deklarativen Programmierstilen ist gemeinsam, dass kein Berechnugszustand explizit repräsentiert wird
  - Logische Programmierung:
    - Man beschreibt das Ziel mit Hilfe einer logischen Formel

    - PROLOG
  - Funktionale Programmierung
    - Man beschreibt das Ziel durch Angabe von (mathematischen) Funktionen,
    - Haskell, ML, LISP
    - Abfragesprachen wie SQL oder XQuery sind auch deklarative Programmiersprachen, allerdings nur für Spezialzwecke einsetzbar
    - Gleiches gilt für viele Auszeichnungssprachen-Sprachen (markup-Sprachen) wie HTML

- **Funktionale Programmierung**
  - Wichtige Eigenschaften:
    - Funktionen sind Bürger erster Klasse (first-class-citizens)
      Alles was man mit Daten machen kann, kann man auch mit Funktionen machen
    - Es gibt Funktionen höherer Ordnung
      Funktionen, die auf Funktionen operieren, die womöglich auf Funktionen operieren
    - Rekursion ist die wesentliche Art, den Kontrollfluss zu organisieren
    - In funktionalen Programmiersprachen gibt es oft keine Anweisungen, sondern nur auswertbare Ausdrücke
    - In reinen funktionalen Sprachen gibt es keine Zuweisungen (und damit auch keine Seiteneffekte)
      $\longrightarrow$ referentielle Transparenz:
      Eine Funktion gibt immer das gleiche Ergebnis bei gleichen Argumenten

- **FP in Python**
  - Funktionen sind Bürger erster Klasse
  - Funktionen höherer Ordnung werden voll unterstützt
  - Viele anderer Konzepte aus funktionalen Programmiersprachen werden unterstützt
    - z.B. Listen-comprehension
  - In vielen funktionalen Programmiersprachen ist Lazy Evaluation ein wichtiger Punkt
    - Die Auswertung von Ausdrücken wird solange verzögert bis das Ergebnis benötigt wird
    - Damit lassen sich unendliche Sequenzen repräsentieren

  - Das Letztere unterstützt Python (und andere Sprachen) durch Iteratoren und Generatoren

- **FP in Python: Defizite**
  - Referentielle Transparenz:
    - kann man natürlich selbst erzwingen:
      Keine globalen Variablen nutzen, keine Mutables ändern
  - Rekursion als wesentliche Steuerung des Kontrollflusses wird in Python nur eingeschränkt unterstützt:
    Keine Optimierung durch Endrekursion
    - Beachte: Maximale Rekursionstiefe kann mit `sys.setrecursionlimit(n)` geändert werden
    - Mit `sys.getrecursionlimit()` kann man sie abfragen
  - Ausdrücke statt Anweisungen:
    Wird in Python nicht unterstützt
    Allerdings gibt es konditionale Ausdrücke!
    - `true-value if cond else false-value`
      hat den Wert `true-value`, falls `cond` wahr ist
      Ansonsten hat der Ausdruck den Wert `false-value`

- **Exkurs: Konditionale Ausdrücke**
  - ```
    >>> "a if True else "b"
    'a'
    >>> "a" if False else "b"
    'b'
    >>> cond = True
    >>> 2 * 3 if cond else 2 ** 3
    8
    >>> res = 2 * 3 if cond else 2 ** 3
    >>> def mult_or_exp(cond):
    ...     return 2 * 3 if cond else 2 ** 3
    >>> mult_or_exp(False)
    8
    ```

- **Funktionen definieren und verwenden**
  - Funktionen existieren in dem Namensraum, in dem die definiert wurden
  - Eine Funktion ist ein normales Objekt (wie andere Pthon-Objekte)
  - Es kann zugewiesen werden, als Argument übergeben werden und als Funktionsresultat zurückgegeben werden
  - Wie stellt man fest, ob ein Objekt aufrufbar ist?
    ```

* Funktionen besitzen die magische Methode `__call__`
* Funktionsobjekte sind Instanzen einer bestimmten Klasse, nämlich `collections.Callable`

```
>>> hasattr(spam, '__call__')
True
>>> import collections
>>> isinstance(spam, collections.Callable)
True
```

* selbst Funktionen definieren:

```
>>> class CallMe:
...     def __call__(self, msg=None):
...         if msg: print("called:", msg)
...         else: print("called")
...
>>> c = CallMe()
>>> c()
called
>>> c("hi")
called: hi
```

- **Lambda-Notation**
  - statt mit Hilfe der `def`-Anweisung eine benannte Funktion zu definieren, kann man mit dem `lambda`-Operator eine kurze, namenlose Funktion definieren:

```
>>> lambda x, y: x * y # multipliziere 2 Zahlen
<function <lambda> at...
>>> (lambda x, y: x * y)(3, 8)
24
>>> mult = lambda x, y: x * y
```

  - Als Funktionskörper ist nur ein einziger Ausdruck (arithmetisch, Boolesch, ...) zulässig
  - Lambda-Funktionen benutzt man gerne für:
    * einfache Prädikatsfunktionen (Boolesche Tests)
    * einfache Konverter
    * Objektdestruktoren
    * Lazy Evaluation ermöglichen
    * Sortierordnug bestimmen

```
# add cookies in order of most spezific
# (ie. longest) path first
cookies.sort(key=lambda arg: len(arg.path), reverse=True)
```

  * Mit Lambda-Notation aufgerufene Funktionen sind Seiteneffektfrei (wenn die aufgerufenen Funktionen es sind!!!)
  * Funktionen können ja Funktionen zurückgeben. Zur Erzeugung der Funktion kann man natürlich Lambda-Ausdrücke verwenden

```
>>> def gen_adder(c):
...     return lambda x: x + c
...
>>> add5 = gen_adder(5)
>>> add5(15)
20
```

- **Funktionen höhrer Ordnung:** `map`, `reduce` **und** `filter`
  - `map`
    * Hat mindestens zwei Argumente:
      Eine Funktion und ein iterierbares Objekt
    * liefert einen Iterator zurück, der über die Anwendungen der Funktion auf jedes Objekt des übergebenen Arguments iteriert

```
>>> list(map(lambda x: x**2, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

    * Wird mehr als ein iterierbares Objekt angegeben muss die Funktion entsprechend viele Argumente besitzen

```
>>> list(map(lambda x, y, z: x + y + z,
...   range(5), range(0, 40, 10), range(0, 400, 100)
[0, 111, 222, 333]
```

    * Anwendungsbeispiel:
      Wir wollen eine Liste `c_list` von Temperaturen von Celsius nach Fahrenheit konvertieren.

```
list(map(lambda c: ((9.0 / 5) * c + 32), c_list))
```

  - `reduce`
    * Reduzierung eines iterierbaren Objekts auf ein Element
    * Wendet eine Funktion mit zwei Argumenten auf ein iterierbares Objekt an
    * Es werden jeweils die ersten beiden Objekte genommen und zu einem Objekt reduziert, das dann das neue Anfangsobjekt ist
    * `reduce` wurde allerdings aus dem Sprachkern von Python 3 entfernt und findet sich nun im Modul `functools`

```
>>> from functools import reduce
>>> reduce(lambda x, y: x * y, range(1, 5))
24 # ((1 * 2) * 3 * 4)
```

- `filter`
  - Filtert nicht passende Objekte aus
  - erwartet als Argument eine Funktion mit einem Parameter und ein iterierbares Objekt
  - Es liefert einen Iterator zurück, der die Objekte aufzählt, bei denen die Funktion nicht `False` (oder äquivalente Werte) zurück gibt

    ```
    >>> list(filter(lambda x: x > 0, [0, 3, -7, 9, 2]))
    [3, 9, 2]
    ```

Listen-, Generator-, dict- und Mengen-Comprehension

- **ähnlich wie mit `lambda, map` und `filter` lassen sich Listen u.a. mit Comprehension deklarativ und kompakt beschreiben**
- **Stil ist ähnlich dem in der mathematischen Mengenschreibweise: {x ∈ U: f(x)} (alle x aus U, die die Bedingung f erfüllen)**

  ```
  >>> [str(x) for x in range(10) if x % 2 == 0]
  ["0", "2", "4", "6", "8"]
  ```

- **erstellt aus allen `str(x)` eine Liste, wobei `x` über das iterierbare Objekt `range(10)` läuft und nur die geraden Zahlen berücksichtigt werden**
- **Generelle Syntax von Listen-Comprehensions:**

  ```
  [ expression for expr1 in seq1 if cond1
               for expr2 in seq2 if cond2
               ...
               for exprn in seqn if condn ]
  ```

  - Die `if`-Klauseln sind dabei optional
  - Ist `expression` ein Tupel, muss es in Klammern stehen
  - Damit kann man ganz ähnliche Dinge wie mit `lamda, map, filter` erreichen
- **Geschachtelte Listen-Comprehensions**
  - Beispiel:
    Zweidimensionale Matrix der Art `[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]` konstruieren:

    ```
    >>> [[x for x in range(4)] for y in range(3)]
    [0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]
    ```

  - Beispiel:
    Wir wollen `[1, 2, 3], [4, 5, 6], [7, 8, 9]` konstruieren:

    ```
    >>> [[x+1 for x in range(y*3, y*3 + 3)] for y in range(3)]
    [1, 2, 3], [4, 5, 6], [7, 8, 9]
    ```

  - Beispiel:
    Wir wollen das kartesische Produkt aus [0, 1, 2] und [”a”, ”b”, ”c”] erzeugen:

    ```
    >>> [(x,y) for x in range(3) for y in ["a", "b", "c"]]
    [(0, "a"), (0, "b"), (0, "c"), (1, "a"), (1, "b"), (1, "c"),
    (2, "a"), (2, "b"), (2, "c")]
    ```

- **Generator-Comprehension**
  - Variante der Listen-Comprehension, die die Liste nicht explizit aufbaut, sondern einen Iterator erzeugt, der alle Objekte nacheinander generiert
  - Einziger Unterschied zur Listen-Comprehension:
    $\longrightarrow$ Runde statt eckige Klammern
  - Diese können weggelassen werden, wenn der Ausdruck in einer Funktion mit nur einem Argument angegeben werden

    ```
    >>> sum(x**2 for x in range(11))
    385
    ```

  - Ist speicherplatzschonender als `sum([x**2 for x in range(11)])`

- **Comprehensions für Dictionarys und Mengen**

```
>>> evens = set(x for x in range(0, 20, 2))
>>> evenmultsoftthree = set(x for x in evens if x % 3 == 0)
>>> evenmultsoftthree
{0, 18, 12, 6}
>>> text = "Lorem ipsum"
>>> res = set(x for x in text if x >= "a")
>>> print(res)
{"e", "i", "m", "o", "p", "r", "s", "u"}
>>> d = dict((x, x**2) for s in range(1, 10))
>>> print(d)
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>> sqnums = set(x for (_, x) in d.items())
>>> print(sqnums)
{64, 1, 36, 81, 9, 16, 49, 25, 4}
>>> dict((x, (x**2, x**3)) for x in range(1, 10))
{1: (1, 1), 2:(2, 4), 3: (9, 27), 4: (16, 64), 5: (25, 125), 6: (36, 216),
7: (49, 343), 8: (64, 512), 9: (81, 729)}
>>> dict((x, x**2) for x in range(10)
...     if not x**2 < 0.2 * x**3)
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

- Mit `all` und `any` kann man über die Elemente eines iterierbaren Objekts oder eines Iterators quantifizieren
  - `all(iterable)`
    Evaluiert zu `True` genau dann wenn alle Elemente äquivalent zu `True` sind (oder das `iterable` leer ist)
  - `any(iterable)`
    Ist `True` wenn ein Element äquivalent zu `True` ist

```
>>> text = "Lorem ipsum"
>>> all (x.strip() for x in text if x < "b")
False
>>> any(x.strip() for x in text if x < "b")
True
>>> all(x for x in text if x > "z") # iterable leer
True
>>> any(x for x in text if x > "z")
False
```

Dekoratoren

- **Funktionen (Callables), die Funktionen (Callables) als Parameter nehmen und zurückgeben**
- **werden verwendet um andere Funktionen oder Mehtoden zu "umhüllen"**
- **Dekoratoren die uns schon früher begegnet sind:**
  - `staticmethod, classmethodm property` etc.
- **Spezielle Syntax (`wrapper`-Syntax):**
  - Falls der Dekorator `wrapper` definiert wurde:

```
def confused_cat(*args):
    pass # do some stuff
confused_cat = wrapper(confused_cat)
```

    Wir können dann auch schreiben:

```
@wrapper
def confused_cat(*args):
pass # do some stuff
```

  - `property, staticmethod` in der `wrapper`-Syntax:

```
class C:
    def __init__(self, name):
        self.name = name
```

```
@property
def name(self):
    return self._name

# @name.setter
# def name(self, x):
#     self.name = 2 * x

@staticmethod
def hello()
    print("Hello world")
```

- **Dekorator selbst definieren:**
  - Beispiel:
    Multiplikation, bei Funktionsaufruf soll in der Konsole etwas angezeigt werden:
    - zunächst in hässlich:

```
verbose = True

def mult(x, y):
    if verbose:
        print("---- a nice header ----")
        print("--> call mult with args: %s, %s" %x, y)
    res = x * y
    if verbose:
        print("---- a nice footer ----")
    return res
```

    - Und nun Elegant:

```
def decorator(f):
    def wrapper(*args, **kwargs):
        print("---- a nice header ----")
        print("--> call %s with args: %s %
              (f.__name__, ",".join(map(str, args))))
        res = f(*args, **kwargs)
        print("---- a nice footer ----")
        return res
    # print("--> wrapper now defined")
    return wrapper

@decorator
def mult(x,y):
    return x * y
```

- Beispiel:
  Wir wollen messen, wie lange die Ausführung einer Funktion dauert:

```
import time

def timeit(f):
    def wrapper(*args, **kwargs):
        print("--> Start timer")
        t0 = time.time()
        res = f(*args, **kwargs)
        delta = time.time() - t0
        print("--> End timer: %s sec." % delta)
        return res
    return wrapper
```

- Dekoratoren hintereinander schalten:

```
@dekorator
@timeit
def sub(x, y):
    return x - y

print(sub(3, 5))
```

  liefert Z.B:

```
---- a nice header ----
--> call wrapper with args: 3,5
--> Start timer
--> End timer: 2.14576......e-06 sec.
---- a nice footer ----
-2
```

- `docstrings und __name__`
  - Beim Dekorieren ändert sich ja der interne Name und der `docstring`
  - Allerdings könnte man ja die Funktionsattribute übernehmen:

```
def decorator(f):
    def wrapper(*args, **kwargs):
        print("---- a nice header ----")
        print("--> call %s with args: %s %
              (f.__name__, ",".join(map(str, args))))
        res = f(*args, **kwargs)
        print("---- a nice footer ----")
        return res
    wrapper.__name__ = f.__name__
    wrapper.__doc__ = f.__doc__
    return wrapper
```

- Zur Übernahme der Attribute gibt es natürlich schon einen Python-Dekorator:

```
import functools
def decorator(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        print("---- a nice header ----")
        print("--> call %s with args: %s %
              (f.__name__, ",".join(map(str, args))))
        res = f(*args, **kwargs)
        print("---- a nice footer ----")
        return res
    return wrapper
```

- **Parametrische Dekoratoren:**
  - Beispiel:
    Wir wollen einen Dekorator schreiben, der alle Stringausgaben auf 5 beschränkt:

```
def trunc(f):
    def wrapper(*args, **kwargs):
        res = f(*args, **kwargs)
        return res[:5]
    return wrapper

@trunc
def data():
return "foobar"
```

    Aktueller Aufruf:

```
>>> data()
"fooba"
```

  - Beispiel:
    Manchmal sollen es 3 Zeichen sein, manchmal 6:

```
def limit(length):
    def decorator(f):
        def wrapper(*args, **kwargs):
            res = f(*args, **kwargs)
            return res[:length]
        return wrapper
    return decorator

@limit(3)
def data:a():
```

```
    return "limit to 3"

@limit(6)
def data_b():
    return "limit to 6"
```

Aktueller Aufruf:

```
>>> data_a()
"lim"
>>> data_b()
"limit "
```

Funktionsschachtelung, Namensräume und Skopus

- **geschachtelte Funktionsaufrufe**
  - Es ist nicht immer klar, auf was sich ein bestimmter Variablenname bezieht
  - Um das zu verstehen müssen wir zunächst die Begriffe Namensraum (name space) und Skopus oder Gültigkeitsbereich (scope) verstehen
  - dabei ergeben sich zum Teil interessante Konsequenzen für die Lebensdauer einer Variablen
- **Namensräume**
  - Ein Namensraum ist eine Abbildung von Namen auf Werte (innerhalb von Python oft durch ein `dict` realisiert)
  - innerhalb von Python gibt es:
    - den `Build-in`-Namensraum(`__bultins__`)
      enthält alle vordefinierten Funktionen und Variablen
    - den Namensraum von Modulen, die importiert werden
    - den globalen Namensraum (des Moduls `__main__`)
    - den lokalen Namensraum innerhalb einer Funktion
    - Namensräume haben verschiedene Lebensdauern
    - Der lokale Namensraum einer Funktion existiert z.B. normalerweise nur während ihres Aufrufs
      $\longrightarrow$ Namensräume sind wie Telefonvorwahlbereiche. Sie sorgen dafür, dass gleiche Namen in verschiedenen Bereichen nicht verwechselt werden
    - Auf gleiche Variablennamen in verschiedenen Namensräumen kann meist mit der Punkt-Notation zugegriffen werden
- **Skopus / Gültigkeitsbereiche**

- Der Skopus (oder Gültigkeitsbereich) einer Variablen ist der textuelle Bereich in einem Programm, in dem die Variable ohne Punkt-Notation zugegriffen werden kann- dh. wo sie sichtbar ist
- Es gibt eine Hierarchie von Gültigkeitsbereichen, wobei der innere den äußeren normalerweise überschreibt!
- Wird ein Variablennam gefunden. so wird nacheinander versucht:
  * ihn im lokalen Bereich aufzulösen
  * ihn im nicht-lokalen Bereich aufzulösen
  * ihn im globalen Bereich aufzulösen
  * ihn im `Builtin`-Namensraum aufzulösen
- Gibt es eine Zuweisung im aktuelen Skopus, so wird von einem lokalen Namen ausgegangen (außer es gibt andere Deklarationen):
  * "`global varname`"
    Bedeutet, dass in der globalen Umgebung gesucht werden soll
  * "`non local varname`"
    bedeutet, dass `varname` in der nicht-lokalen Umgebung gesucht werden soll, dh. in den umgebenden Funktionsdefinitionen
- Gibt es keine Zuweisung wird in den umgebenden Namensräumen gesucht
- Kann ein Name nicht aufgelöst werden, dann gibt es eine Fehlermeldung
- Beispiel:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test-spam"
    do_local()
    print("After llocal assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)
```

Mit Ausgabe:

```
>>> scope_test()
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
>>> print("In global scope:", spam)
In global scope: global spam
```

Closures

- **In Python ist eine Closure einfach eine von einer anderen Funktion zurückgegebene Funktion (die nicht-lokale Referenzen enthält)**

```
>>> def add_x(x):
...     def adder(num):
...         # adder is a closure
...         # x is a free variable
...         return adder
...
>>> add_5 = add_x(5); add_5
<function adder at...>
>>> add_5(10)
15
```

- **Closures in der Praxis**
  - Closures treten immer auf, wenn Funktionen von anderen Funktionen erzeugt werden
  - Manchmal gibt es keine Umgebung, die für die erzeugte Funktion wichtig ist
  - Oft wird eine erzeugte Funktion aber paramtetrisiert, wie in unserem Beispiel oder bei den parametrisierten Dekoratoren
  - innerhalb von Closures kann auch zusätzlich der Zustand gekapselt werden, wenn auf nonlocal Variablen schreibend zugegriffen wird
  - In den beiden letzteren Fällen wird die Lebenszeit eines Namensraums nicht notwendig bei Verlassen einer Funktion beendet

Iteratoren

- **iterierbare Objekte**
  - unter anderem Container-Objekte, über deren Elemente wir in `for`-Schleifen iterieren können

  - Sequenzen und ähnliche Objekte, wie Tupel, Listen, Strings, dicts und Mengen gehören dazu

- **Das Iterator-Protokoll**
  - Ein Objekt ist iterierbar, wenn es das Iterator-Protokoll unterstützt
  - D.h. es muss due magische Methode `__iter__` besitzen, die einen neuen Iterator zurück liefert
  - Ein Iterator ist ein Objekt, das ebenfalls eine magische Methode `__iter__` besitzt, die self zurück gibt
  - Außerdem muss es eine magische Methode `__next__` besitzen, die das nächste Element zurück liefert
  - Gibt es kein nächstes Element soll die Exception StopIteration ausgelöst werden
  - Die `__iter__`-Methode kann auch mit der Funktion `iter(obkject)` aktiviert werden
  - Ebenso kann die `__next__`-Methode mit der Funktion `next(object)` aktiviert werden
  - Die Implementation der `for`-Schleife:

    ```
    for el in seq:
        do_something(el)
    ```

    wird zu:

    ```
    iterator = iter(seq) # erzeuge Iterator
    while True:          #durchlaufe Schleife
        try:
            el = next(iterator) #nächstes Element
            do_something(el)    #mache etwas damit
        except StopIteration:   #falls Ende Ausnahme
            break               #verlasse die Schleife
    ```

  - Das Iterator-Protokoll bei der Arbeit

    ```
    >>> seq = ["spam", "ham"]
    >>> iter_seq = iter(seq)
    >>> iter_seq
    <list_iterator object at ...>
    >>> print(next(iter_seq))
    spam
    >>> print(next(iter_seq))
    ham
    >>> print(next(iter_seq))
    Traceback (most recent call last):...
    StopIteration
    ```

- **Iterierbare Objekte vs Iteratoren**
  - Ein iterierbares Objekt ist ein Objekt, das (bei Aufruf von `iter()`) einen Iterator erzeugt, aber selbst keine `__next__`-Methode besitzt
  - Bei jedem Aufruf von `iter()` wird ein neuer Iterator erzeugt
  - Ein Iterator dagegen erzeugt keine neuen Iteratoren, aber liefert bei jedem Aufruf von `next()` ein neues Objekt aus dem Container
  - Da Iteratoren auch die `__iter__`-Methode besitzen, können Iteratoren an allen Stellen stehen an denen ein iterierbares Objekt stehen kann (z.B. `for-Schleifen`
  - Beim `iter()`-Aufruf wird der Iterator selbst zurück gegeben
  - `map` z.B. liefert ja beispielsweise einen Iterator und kann in `for`-Schleifen genutzt werden
  - Iteratoren (z.B. `map`) können an Stellen stehen an denen ein iterierbares Objekt (z.B. eine Liste) stehen kann, aber es passiert etwas anderes!
  - Iteratoren sind nach einem Durchlauf, der mit `StopIteration` abgeschlossen wurde erschöpft wie im nächsten Beispiel:

```
>>> iterator = map(lambda x: x+1, range(2))
>>> for x in iterator:
...     for y in iterator:
...         print(x, y)
...
1 2
>>>
```

  - wird bei jedem Schleifendurchlauf ein neuer Iterator erzeugt läuft alles wie erwartet:

```
>>> for x in map(lambda x: x+1, range(2)):
...     for y in map(lambda x: x+1, range(2)):
...        print(x, y)
...
1 1
1 2
2 1
2 2
>>>
```

  - Die range Funktion liefert ein range-Objekt, das iterierbar ist
  - dh. das Pbjekt liefert bei jedem `iter()`-Aufruf einen neuen Iterator

```
>>> range_obj = range(10)
>>> range_obj
```

```
range(0, 10)
>>> range_iter = iter(range_obj)
>>> range_iter
<range_iterator object at ...>
```

- **Warum sind Iteratoren interessant?**
  - Sie bieten:
    1. eine einheitliche Schnittstelle zum durchlaufen von Elementen
    2. die Möglichkeit eine Menge von Elementen zu durchlaufen ohne eine Liste aufbauen zu müssen (Speicherschonend)
    3. die Möglichkeit, unendliche Mengen zu durchlaufen (natürlich nur endliche Anfangsstücke)
  - Iteratoren können natürlich auch selbst definiert werden
  - Beispiel:
    Iterator zum Aufzählen aller Fibonacci-Zahlen (oder die Länge beschränken)

```
class FibIterator():
    def __init__(self, max_n=0)
        self.max_n = max_n
        self.n, self.a, self.b = 0, 0, 1

    def __iter__(self):
        return self

    def __next__(self):
        self.n += 1
        self.a, self.b = self.b, self.a + self.b
        if not self.max_n or self.n <= self.max_n:
            return self.a
        else:
            raise StopIteration
```

Und nach Ausführung:

```
>>> f = FibIterator(10)
>>> list(f)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> list(f)
[]
>>> for i in FibIterator(): print(i)
...
1
1
2
```

```
3
5
8
...
```

Generatoren

- **Generatoren bieten die Möglichkeit Iteratoren mit Hilfe einer einfachen Funktionsdefinition zu erzeugen**
- **Dazu wird innerhalb der Funktionsdefinition das Schlüsselwort `yield` benutzt**
- **An dieser Stelle wird die Ausführung interbrochen und ein Wert zurückgegeben.**
- **Danach wird beim nächsten `next()`-Aufruf direkt an dieser Stelle weiter gemacht**
- **dh. Generatoren speichern den Zustand in Form der Wertebelegung der lokalen Variablen und den aktuellen Ausführungspunkt**
- **Beispiel**

```
>>> def gen(i): #sieht aus wie normale Funktion
...     i += 1
...     yield i # ist aber ein Generator
...     i += 1
...     yield i
...
>>> g = gen(5); g # Erzeuge Iterator
<generator objegt gen at ...>
>>> next(g) # erstes Element
6
>>> next(g) # zweites Element
7
>>> next(g) # Schluss!
Traceback...
StopIteration
```

- **Generatoren vs Funktionen**
  - Generatoren sehen aus wie Funktionen, geben aber Werte per `yield` (statt `return`) zurück
  - Wird ein Generator aufgerufen, so liefert er keinen Funktionswert, sondern einen Iterator zurück
  - Dieser gibt dann bei den folgenden `next()`-aufrufen die `yield`-Werte zurück

  - Kommt derr Iterator zum Ende (bzw. wird ein `return` ausgeführt), dann wird die `StopIteration`-Ausnahme ausgelöst
  - Beispiel:
    Fibonacci-Generator:

```
def fibgen(max_n=0):
    n, a. b = 0, 0, 1
    while max_n == 0 or n < max_n:
        n += 1
        a, b = b, a + b
        yield a
```

    Und nach Ausführung:

```
>>> list(fibgen(10))
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> for i in fibgen(): print(i, end=" ")
...
1 1 2 3 5 8 13 21 34 55 89...
```

- **Reursive Generatoren**
  - Genauso wie Funktionen können auch Generatoren rekursiv definiert werden
  - Beispiel:
    Alle Permutationen (Anordnungen) erzeugen

```
>>> def perm(seq):
...     if not seq: yield []
...     else:
...         for i in range(len(seq)): # alle Indices
...             for cc in perm(seq[:i]+seq[i+1:]): # Rekursion ohne i!!!
...                 yield [seq[i]] + cc # i_tes
...
>>> list(perm("lion"))
[["l", "i", "o", "n"], ["l", "i", "n", "o"], ...]
```

- **Das Modul `itertools`**
  - Das Modul `itertools` bietet viele Generatoren an, die man standardmäßig benötigt
  - Außerdem gibt es Kombinationen und Modifikationen von Iteratoren
  - Generell werden immer Iteratoren zurückgegeben
  - Beispiele
    * `accumulate(iterable, func=oprerator.add)`:
      Akkumuliert über einen Iterator.
      Man kann auch statt der Addition eine andere 2-Stellige Funktion nutzen:

  - `accumulate([1, 2, 3, 4]` $\longrightarrow$ 1 3 6 10

* `chain(*iterables`:
  Verkettet iterierbare Objekte.
  Beispiel:
  - `chain("ABC", "DEF"` $\longrightarrow$ "A", "B", "C", "D", "E", "F"
* `combinations(iterable, r)`:
  Erzeugt alle Kombinationen der Länge r
  Beispiel:
  - `combinations("ABC", 2)` $\longrightarrow$ ("A", "B"), ("A", "C"), ("A", "D"), ("B", "C"), ("B", "D"), ("B", "D"), ("C", "D")
  - Elemente werden dabei hier und im Folgenden auf Grund ihrer Position identifiziert, nicht auf Grund ihres Wertes
* `combinations_with_replacement(iterable, r)`:
  Kombinations mit Wiederholungen.
  Beispiel:
  - `combinations_with_replacement("ABC", 2)` $\longrightarrow$ ("A", "A"), ("A", "B"), ("A", "C"), ("B", "B"), ("B", "C"), ("C", "C")
* `cycle(iterable)`:
  Erzeugt einen unendlichen Iterator, der immer wieder über `iterable` iteriert
  Beispiel:
  - `cycle("ABC"` $\longrightarrow$ "A" "B" "C" "A" "B" "C" ...
  - `islice(iterable, stop)`
    `islice(iterable, start, stop)`
    `islice, iterable, start, stop, step)`
    Slice Funktion für Iteratoren
    Beispiel:
    `islice("ABCDEF", 2, 4)` $\longrightarrow$ "C" "D"
  - `permutations(iterable, r=None)`
    Permutationen der Länge r (bzw. aller Elemente)
  - `product(*iterable, repeat=1)`
    Kartesisches Produkt bzw. `repeat`-faches Produkt
  - `repeat(object, times=None)`
    Erzeugt Iterator, der Objekt `times`-fach oder unbegrenzt oft wiederholt
    Beispiel:
    `map(pow, range(5), repeat(2))` $\longrightarrow$ 0 1 4 9 16

· `startmap(function, iterable)`
Ähnlich wie `map`, allerdings für den Fall, dass die Argumente für `function` bereits in Tupel zusammengefasst wurden
Beispiel:
`startmap(lambda x, y: x+" "+y, [("nice", "restaurant"), ("dirty", "knife")])` ⟶ "nice restaurant" "dirty knife"