PRG1 – Zusammenfassung

Computer Def: (digitalelektronische) Maschine zur Speicherung + automatische Verarbeitung, durch flexible (programmierbare) Vorschriften.

Kerneigenschaften:
1.digitalelektronisch
2. Speicherung von Daten
3. Automatische Verarbeitung von Daten
4. Programmierbare (flexible) Rechenvorschriften, d.h. Software

Bedingungen für Turing-vollständige Maschine:

- das Nullsetzen eines Wertes,
- das Inkrement bilden (um 1 erhöhen – hochzählen) eines Wertes und
- eine datenabhängige Verzweigung:, z.B. der Form: Wenn ein (Zwischen-)Wert 0 ist, dann führe die Operationsfolge 1 aus, sonst die Operationsfolge 2.

(minimal notwendig um alle berechenbaren Funktionen zu errechnen (Church-Turing-These)

Neumann-Architektur:
5 Funktionseinheiten:
1. Speicher (M), Steuerwerk/Leitwerk (CC), Rechenwerk, Eingabe, Ausgabe
(Steuerwerk + Rechenwerk = Prozessor)

2. Im Speicher sind Daten und Programm abgelegt (kann nicht gleichzeitig zugegriffen werden)

3. Befehls-Ausführungszyklus:

1. Befehl aus dem Speicher holen	FETCH INSTRUCTION
2. Befehl im Steuerwerk interpretieren	DECODE
3. Operanden holen	FETCH OPERANDS
4. Befehl ausführen (von einer der Funktionseinheiten)	EXECUTE
5. (Erhöhen des Befehlszählers	UPDATE INSTRUCTION POINTER)

Eigenschaften Algorithmus:

Determiniertheit (bei gleichen Startwerten muss gleiches Ergebnis kommen), Deterministisch (nach jedem Schritt darf es nur einen möglichen nächsten Schritt geben), Statische Finitheit (Beschreibung des Algorithmus ist endlich), Dynamische Finitheit (Menge an Daten und Zwischenspeicherungen sind endlich), Terminiertheit (bricht nach endlicher Zeit kontrolliert ab)

Programmierspachen müssen Algorithmen (Bearbeitungsvorschriften) in einer für Computer und Menschen verständliche und lesbare Form ausdrücken.

Programmiersprachen:

1. Generation: Maschinensprache (kann Prozesse direkt ausführen)
2. Generation: Assembler (ersetzen Zahlencode durch symbolische Bezeichner = Mnemonics)
3. Generation: Höhere Programmiersprachen (Die meisten modernen Programmiersprachen + objektorierntierte Sprachen OO-Generation)
4. Generation: anwendungsbezogene (aplikative) Sprachen für Zugriff auf Datenbanken (meist SQL) oder Gestaltung von graphical user interfaces (GUIs)

Programmiersprachenparadigmen:

- **Imperative Programmierparadigmen**
  - Strukturierte Programmierung
  - **Prozedurale Programmierung**
  - **Modulare Programmierung**
  - Programmierung mit abstrakten Datentypen
  - **Objektorientierung**
- **Deklarative Programmierparadigmen**
  - Constraint Programmierung
  - **Funktionale Programmierung**
  - Logische Programmierung
- **Sonstige Ansätze**
  - Agentenorientierte Programmierung
  - Aspektorientierte Programmierung
  - Generische Programmierung
  - Datenstromorientierte Programmierung

Imperative Programmierung:
Computerprogramm als lineare abfolge von Befehlern

Deklarative Programmierung:
basierend auf mathematischen, rechnerunabhängigen Theorien (es wird nicht der Weg zum Ergebnis programmiert, sondern nur das gewünschte Ergebnis)

Funktionale Programmierung:
Aufgabenstellung wird als funktionaler Ausdruck formuliert. Selbstständiges anwenden von Funktionsersetzungen von Interpreter oder Compiler lösen Aufgabenstellung

Compiler: Übersetzt (kompiliert) Quellprogramm in Assemblersprache, Bytecode oder Maschinensprache
(während Kompilierung: Lexikalische, Syntaktische und Semantische Analyse, Zwischencodeerzeugung, Programmoptimierung, Codegenerierung) [Übersetzen – Binden – Laufen]

Interpreter: Übersetzt nicht, sondern liest Programm Zeile für Zeile und führt direkt aus
(Liest erstes Statement, prüft lexikalisch, syntaktisch, semantisch, generiert Maschinencode und führt dann Maschinencode aus. Beginnt dann bei nächstem Statement)

Bytecode (Programm, das aus Befehlen für virtuelle Maschine besteht). Prüfung von Programm mittels Compiler, aber dann keine Übersetzung in Maschinensprache, sondern in einfach strukturierten Zwischencode, der dann von Interpreter in virtueller Maschine ausgeführt wird

(Vorteil: Plattformunabhängigkeit, dynamische Optimierung möglich; Nachteil: langsam)

Einer Variablen zugewiesener Wert: Literal

Prioritäten der Operatoren:

Operatoren	**Kurzbeschreibung**	**String / Unicode**	**Float**	**Integer**	**Boolean**	**In erweiterter Zuweisung anwendbar**
`(...)`	(Vorrang) Klammerung	x	x	x	x	

`s[i]`
`s[i:j]`
`f(...)`
	Indizierung (bei Sequenztypen)
	Teilbereiche (bei Sequenztypen)
	Funktionsaufruf
		x
		x
		x
			
			x
				
				x
					
					x	

`+x, -x,` `~x`	Einstellige Operatoren Invertiere x		x	x x		
`x ** y`	Exponential-Bildung x^y (Achtung: rechts-assoziativ)		x	x		x

`x * y`
`x / y`
`x % y`
 	
`x // y`
	Multiplikation (Wiederholung)
	Division
	Modulo (-Division) = (Ganzzahliger) Rest
	Restlose Division [2])
		x	x
			x
			x
			
			x
				x
				x
				x
				
				x
						x
						x
						x
						
						x

`x + y` `x - y`	Addition (Konkatenation) Subtraktion	x	x x	x x		x
`x << y, x >> y`	Bitweises Schieben (nur bei Integer)			x		x
`x & y`	Bitweises Und (nur bei Integer)			x		x
`x ^ y`	Bitweises exklusives Oder (nur bei Integer)			x		x
`x | y`	Bitweises Oder (nur bei Integer)			x		x

`x >= y, x == y, x != y`
`[x <> y]`
 	
`x is y, x is not y`
`x in s, x not in s`
	Vergleichsoperatoren liefern als Ergebnis **`True`** oder **`False`** [1])
	Test auf Identität
	Tests auf Enthaltensein in Sequenzen
		x
		
		x
		x
			x
			
			x
				x
				
				x
					x
					
					x	

`not x`	Logische Negation				x	
`x and y`	Logisches Und				x	
`x or y`	Logisches Oder				x	

In Python reservierte Variablennamen:

and	def	finally	in	or	while
as	del	for	is	pass	with
assert	elif	from	lambda	raise	yield
break	else	global	None	return	
class	except	if	nonlocal	True	
continue	False	import	not	try	

Einer-Komplement:
Erste Stelle der Binärzahl gibt Vorzeichen an (0 = +, 1 = -)
Darstellung von positiver Zahl x in binär (0011_2)
Darstellung von negativer Zahl -x mit allen Zahlen von x invertiert (1100_2)

Nachteile: Doppelkodierung (+0 / -0)
Ändert Addition von +1 die Vorzeichenstelle, so muss Ergebnis durch Addition von +1 korrigiert werden

Zweier-Komplement:
Gleiches Spiel wie Einer-Komplement für positive Zahlen und maximale negative Zahl. Alle anderen negativen Zahlen werden wie Einer-Komplement invertiert und dann +1 auf die Binärzahl gerechnet.
Von Dezimal in Zweier-Komplement:
positiv: Wandle in binär um --> fertig
Negativ: Wandle in positive Binärzahl um, gehe von rechts nach links, nimm erste 1, lass sie stehen und invertiere restliche Zahlen

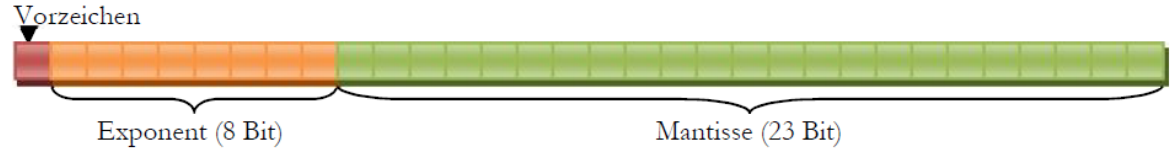
Von Zweier-Komplement in Dezimal:
Wenn hinterste Zahl 0, dann rechne in Dezimal um
Wenn hinterste Zahl 1, dann rechne -1 in Binär und invertiere Zahl, dann rechne zu Dezimal um

Operationen mit Bitzahlen:

~X	Bitweises Invertieren von X
X & Y X | Y X ^ Y	Bitweises logisches UND, ODER, XOR zwischen X und Y; siehe auch Datentyp Boolean.
X << Y	Bitweises Verschieben von X um Y Binärstellen nach links, zieht 0 nach, Vorzeichen bleibt erhalten!
X >> Y	Bitweises Verschieben von X um Y Binärstellen nach rechts, zieht Vorzeichenbit nach, Vorzeichen bleibt erhalten!

Gleichtkommazahlen (IEEE 754, bzw. IEEE 754-2008)
Standartdarstellung für binäre Gleitkommazahlen
Format der Darstellung: $x = (-1)^s * m * 2^e$
s = Vorzeichenbit, m = Mantisse, e = Exponent
Darstellung von Gleitkommazahlen in 16 (10 M, 5 E), 32 (single; 23 M, 8 E), 64 (double; 52 M, 11 E) und 128 (112 M, 15 E) Bit

Die Anordnung der Bits einer Gleitpunktzahl in 32 bit *(float)* zeigt die nachfolgende Abbildung.[4]



Besondere Codewörter in IEEE 754:

Exponent	Mantisse	Bedeutung
111...111binär	000...000binär	+/– Unendlich, je nach Vorzeichen
111...111binär	≠ 000...000binär	"Keine Zahl" (NaN = Not a Number)
000...000binär	000...000binär	+/– 0. (Null)
000...000binär	≠000...000binär	Denormalisierte Gleitpunktzahl als „subnormal“ nach IEEE 754r bezeichnet

Unendlich ist synonym für nicht darstellbare Zahl
Keine Zahl stellt ungültige (oder nicht definierte) Ergebnisse dar, z.B. Wurzel aus negativer Zahl
Denormalisierte Zahl ist Zahl die zu klein ist um Exponent nicht als von Null verschieden zu speichern
--> Veränderte Interpretation:

$$W = \pm 1.m \cdot 2^{e} = \pm 1.m \cdot 2^{E-B}$$

Wird zu

$$W = \pm 0.m \cdot 2^{e_m}$$

Denormalisierte Zahlen haben geringere Genauigkeit (Präzision) als normalisierte Zahlen
e_m ist kleinster möglicher Exponent (negativer bias, abhängig von Anzahl der Bits)

Bsp: Umwandlungen: Dezimalzahl ⇔ Gleitpunktzahl
($-166.125_{(10)} = -10100110.001_{(2)}$

Schritt 1: Man wandele die Zahl Vorpunkt- und Nachpunktstellen getrennt in Dualzahlen
Beispiel: $-166.125_{(10)} = -10100110.001_{(2)}$

Schritt 2: Normalisieren (Verschieben des „Punkts“) also in eine Darstellung 1.xxx.... Die Anzahl der Verschiebungen ergeben e.
Die Mantisse M erhält wir, indem wir die Vorpunkt 1 und das Komma weglässt. Um auf 23 Bit zu kommen wird (falls nötig) hinten mit Nullen aufgefüllt:
Beispiel: $-10100110{,}001_{(2)} = -1.0100110001_{(2)} \cdot 2^{7_{(1)}}$
M = 0100110 00100000 00000000

Schritt 3: Bestimmung der Exponentenrepräsentation: E = e + Bias

Umgekehrt:

Da das erste Bit auf 0 gesetzt ist, sehen wir sofort dass es sich um eine positive Zahl handelt. Den mit einem Bias behaftete Exponenten e dekodieren wir als

$$e = 10000110_2 = 134_1 - 127_1 = 7_1$$

Für die Mantisse m ergibt sich schließlich

$$m = 10001000111110010101100_2 = 0.5350547_1 + 1_1 = -1.5350547_1 .$$

Insgesamt erhalten wir also

$$\begin{aligned} x &= (-1)^s m \cdot 2^e \\ &= (-1)^s \cdot 1.5350547 \cdot 2^7 \\ &= 1 \cdot 1.5350547 \cdot 128 \\ &= 196.4870016_1 \end{aligned}$$

Gleitpunktzahlen haben unterschiedliche Genauigkeit (Präzision), abhängig von der Länge der Mantisse

Vergleichsoperatoren:

Operator	Beschreibung
`X < Y`	echt kleiner als
`X <= Y`	kleiner oder gleich
`X > Y`	echt größer als
`X >= Y`	größer oder gleich
`X == Y`	gleicher Wert
`X != Y`	Ungleicher Wert (
`(X <> Y)`	Ungleicher Wert (Schreibweise nicht empfohlen)
`X is Y`	Gleiches Objekt (Variable)
`X is not Y`	Negierte Objektgleichheit

Zeichendarstellung:
ASCII-Code: Darstellung von Zeichen mittels 7 oder 8 Bit (viele Verschiedene Varianten)
[American Standard Code for Information Interchange]
Unicode: 16 Bit pro Zeichen (international anerkannt) (U+00000 bis U+10FFFF, ersten beiden Ziffern identifizieren eien von 17 „Ebenen“, die letzten vier Ziffern ein Zeichen in der jeweiligen Ebene)

Wichtige internationale Zeichensätze			
ASCII	Einer der ältesten Computer-Zeichensätze – 7 Bit	1963 (1968)	Sehr weit verbreitet, in der Regel der Basis-Zeichensatz für Programmiersprachen, Internet-Adressen, etc.
Unicode ISO/IEC 10646 – 1991	Ein internationale Standard – (7) 8, 16 oder 32 Bit	1991	Universell, mit verschiedenen Code-längen. nimmt an Bedeutung stark zu.

Starke Typisierung:
Es existiert eine einmalig durchgeführte Bindung zwischen Variable und Datentyp. Verschiedene Datentypen können nicht miteinander „interagieren“ (integers können nicht zu strings addiert werden, etc.). Umwandlungen von Datentypen (wenn möglich) müssen explizit durch einen Befehl durchgeführt werden.
(Vorteil: Schnellere Compilerarbeit, da keine Typprüfungen nötig)
Vs.
Schwache Typisierung:
Es existiert keine feste Bindung zwischen Variable und Datentyp. Unterschiedliche Datentypen können ggf. miteinander interagieren.

Statische Typisierung:
Zuweisung des Datentyps zu Variablen erfolgt durch Deklaration (Code der festlegt, welcher Datentyp mit einer Variable verbunden ist).
Vs.
Dynamische Typisierung:
Zuweisung von Datentyp zu Variable erfolgt automatisch während Laufzeit des Programms. Keine Typisierung von Hand während Programmierung notwendig

Python ist eine starke, dynamische Sprache

Coercion (implizite Typkonvertierung): Automatische Umwandlung von einem Datentyp in einen anderen, bei logisch sinnvollen Operationen
z.B. ein integer --> Float, wenn das integer mit einem Float addiert wird. Ergebnis ist logischerweise ein Float, ohne das Information verloren geht

Cast(ing) (explizite Typkonvertierung): Explizite Umwandlung von einem Datentyp in einen Anderen.
Z.B. ein Integer in einen String

Coercion in Python:
Bei numerischen Ausdrücken: bool --> Int --> long --> float --> complex
Bei Sting-Ausdrücken nach Schema: str --> unicode

Direkte Rekursion: Rekursion, die sich auf sich selbst bezieht
Indirekte Rekursion: Aktion, die sich auf eine andere Aktion bezieht, die sich auf die erste Aktion bezieht (Menge der Aktionen egal, solange sich der Kreis schließt)

Primitive Rekursion: Rekursion, die durch eine Iteration ersetzt werden kann

Unterprogramme können Argumente haben (=Parameter), in Programmdefinition werden sie formale Parameter genannt. Bei Aufruf der Funktion werden sie durch aktuelle Parameter ersetzt

Mechanismen zur Parameterübergabe:
Call by reference: Compiler oder Interpreter übergibt Adresse des Speicherbereichs übergibt --> übergibt einen Zeiger (=Referenz). Änderungen am formalen Parameter ändert auch aktuellen Paramerter

Call by value: Compiler oder Interpreter übergeben Wert (bei Variable). Änderungen an diesem Wert wirken sich nicht auf die ursprüngliche Variable aus.

Call by name: Werte die nicht bei Übergabe, sondern erst bei Benutzung berechnet werden (in modernen Sprachen unüblich) [sog. call by need merkt sich bereits ausgewertete Variablen und behebt damit Problem der mehrfachen Berechnung von Werten in call by name)

	Referenzparameter	**Wertparameter**
Formale Parameter	Einfache Variablen und strukturierte Variablen	Einfache Variablen und strukturierte Variablen
Aktuelle Parameter	Nur Variablen, Felder, Feldelemente, Strukturelemente. Keine Konstanten und Ausdrücke	Beliebige Ausdrücke wie *1.0*, *2*X*, *sin(x)*, y[i]
Übergabe	Als *Adresse* übergeben (geringer Aufwand bei Feldern)	Als *Kopie* (hoher Aufwand bei großen Datenstrukturen)
Zuweisung innerhalb des Unterprogramms	möglich	möglich oder verboten (je nach Programmiersprache)
Rückgabe des Wertes bei Unterprogrammende	ja	nein

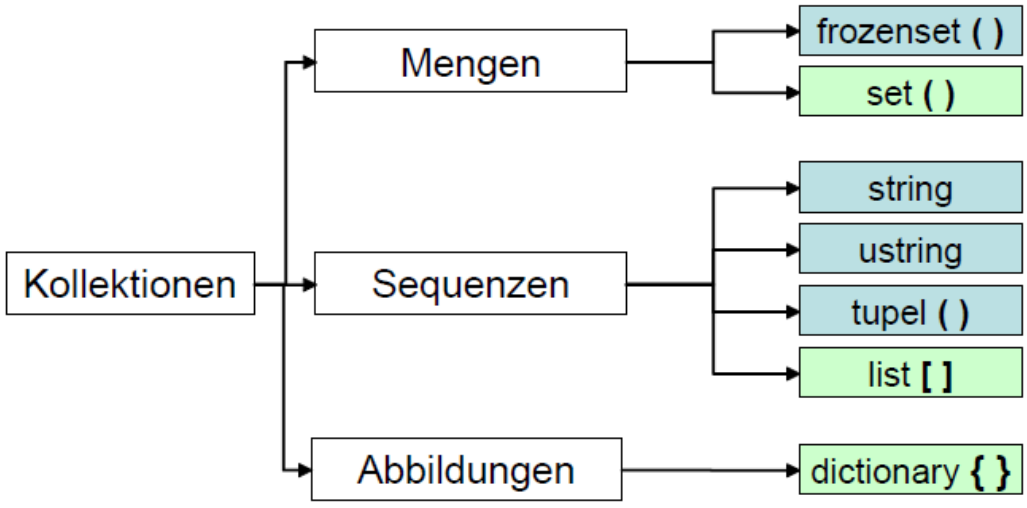
Funktionen: Errechnen einen Wert, der an das rufende Programm zurückgegeben wird.
Prozeduren: Führen eine bestimmte Aktion aus (Verändern entweder interne Variablen oder über Referenzparameter Änderungen an Variablen im rufenden Programm vorgenommen werden.

Wirkungen: Alles was in einem Programm an Aktionen stattfindet
Nebenwirkung: Alles was passiv in einem Programm durch Wirkungen verändert wird

In Python werden Parameter, die an eine Funktion weitergegeben werden über call by reference aufgerufen (können also verändert werden)

Funktionen rufen neue lokale Namensräume auf

Aggregierte Datentypen (Python)



Set, list und dictionary sind mutable (veränderlich)
Frozenset, string, ustring und tupel sind unmutable (unveränderlich)

Listen: Eckige Klammern, Inhalt durch Komma getrennt

[]	eine leere Liste
liste = [1, 2, 3]	eine Liste mit drei Integer-Elementen
liste[1]	holt das Element mit Index=2
liste = ['spam', [1, 2,3], 3.141]	eine verschachtelte Liste: liste [1][0] holt das Element 1
liste = list('spam')	erzeugt eine Liste durch Aufruf des Typkonstruktors

Tupel: Runde Klammern, Inhalt durch Komma getrennt

()	ein leeres Tupel
tuple = (1, 2, 3)	ein Tupel mit drei Integer-Elementen
tuple(1)	holt das Element 2
Tuple = ('spam', (1, 2,3), 3.141)	eine verschachteltes Tupel: tupel [1][0] holt das Element 1
tuple = tuple('spam')	erzeugt ein Tupel durch Aufruf des Typkonstruktors
tuple = 1, 2, 3	das gleiche Tupel wie in anderer Schreibweise, ist aber in Funktionsaufrufen nicht erlaubt, weil nicht zwischen Parametern und Tupel-Elementen unterschieden werden kann.
(1,)	ein Tupel mit einem Element und kein geklammerter Ausdruck. Dies wird durch das Komma deutlich gemacht.

Sequenz - Operationen für Stings, Listen und Tupel:

Operationen auf Sequenz-datentypen	Beschreibung	Klassenmethoden
`X in S` `X not in S`	Test auf Enthaltensein	`__contains__`
`S + T`	Verkettung	`__add__`
`S*i` `i*S`	Wiederholung N ist ein Integer	`__mul__`
`S[i]`	Indizierung mit Zahl	`__getitem__`
`S[i:j]`	Teilbereich (Slicing)	`__getslice__` `__getitem__`
`len(S)`	Länge	`__le__`
`min(S)`	minimales Element	
`max(S)`	maximales Element	
`iter(S)`	Iterator-Objekt	`__iter__`
`for X in S`	Iteration	`__getitem__` `__iter__`

Operationen speziell für Listen (da mutable):

Operationen auf Listen	Beschreibung	Klassenmethoden
`S[i] = x`	Zuweisung an das i-te Element	`__setitem__`
`S[i,j,k] = T`	Teilbereichszuweisung, S wird von i bis j durch T ersetzt, mit einer optionalen Schrittweite k	`__setslice__` `__setitem__`
`del S[i]`	Indizierten Eintrag löschen	`__delitem__`
`del S[i,j,k]`	Teilbereichslöschung, sonst wie oben	`__delslice__` `__delitem__`
`del S`	löscht S gänzlich, eine spätere Referenzierung von S ist ein Fehler	

Methoden auf Listen	Beschreibung	Klassenmethoden
`S.insert(i,x)`	Fügt ein Element x an einer bestimmten Stelle i ein. Das erste Argument ist der Index, vor dem eingefügt werden soll.	
`S.append(T)`	Hängt die Liste T an S an: äquivalent zu S.insert(len(S), T)	
`S.index(x)`	Gib den Index des ersten Elements in der Liste zurück, dessen Wert gleich x ist. Falls kein solcher existiert, so ist dies ein Fehler.	
`S.remove(x)`	Entferne das erste Element der Liste, dessen Wert x ist. Falls kein solches existiert, so ist dies ein Fehler.	
`S.sort()`	Sortiere die Elemente der Liste, in der Liste selbst.	
`S.reverse()`	Invertiere die Reihenfolge der Listenelemente, in dieser selbst.	
`S.count(x)`	Gib die Anzahl des Auftretens des Elements x in der Liste zurück.	
`S.pop([i])`	Entfernt Element an Position i undgibt diesen zurück (default (ohne Angabe von i = letzte Element	

Vergeleich von Objekten desselben typs:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

Zuerst wird jeweils erster Eintrag verglichen, wenn gleich, wird zweiter Eintrag verglichen, etc. Wenn beide Sequenzen gleich sind, ist die Kürzere kleiner.

Schlüssel von Dictionaries können Strings, Zahlen und Tupel sein. Listen nicht

Operationen für Dictionaries:

```
D[k]                Indizierung mit Schlüssel k
D[k]=X              Wertzuweisung über Schlüssel k
del D[k]            Lösche Eintrag über Schlüssel k
len (D)             Länge (Anzahl der Einträge)
k in D D.has_key(k) Test auf Enthaltensein des Schlüssels k
k not in D
```

```
D.clear()           Löscht alle Einträge in D
D.copy()            Ergibt eine flache Kopie von D
D.get(k[,d])        liefert D[k], wenn k in D, sonst d (wenn angegeben)
oder None .
D.items()           Liste von D's (key, value)-Paaren, als 2-Tuple
dictD.iteritems()   -> an iterator over the (key, value) items of D
D.iterkeys()        -> an iterator over the keys of D
D.itervalues()      -> an iterator over the values of D
D.keys()            -> list of D's keys
D.pop(k[,d]) -> v, remove specified key and return the corresponding value
D.popitem()         -> (k, v), remove and return some (key, value) pair
                    as a 2-tuple; but raise KeyError if D is empty
D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D
D.update(E, **F)    -> None.  Update D from E and F: for k in E: D[k] =
                    E[k](if E has keys else: for (k, v) in E: D[k] = v)
                    then: for k in F: D[k] = F[k]
D.values()          -> list of D's values
```

Funktionen:

Filter(Prüfung, Sequenz); gibt Liste aller Werte von Sequenz aus, die True für Prüfung sind
Map(Funktion, Sequenz); Gibt Liste aller Werte von Sequenz aus, nachdem Funktion darauf angewendet wurde
Reduce(Funktion, Sequenz); Gibt einzelnen Wert aus, der entsteht, wenn man Funktion auf nacheinander auf die Elemente von Sequenz anwendet. --> Ergebnis1 = Funktion(Sequenz[0], Sequenz[1]) --> Ergebnis2 = Funktion(Ergebnis1, Sequenz[2]) --> etc.

Objekt, dass auf eine anderes Objekt verweist = Container

Python:
Bei Zuweisung von Variablen a = b, wird kopie von a erstellt, wenn a eine Zahl oder ein String ist. Ist a allerdings eine Liste oder ein Dictionary, so ist b nur eine Referenz zu a --> Veränderungen an a führen auch zu Veränderungen an b und umgekehrt.

Flache Kopie: Kopie eines Objekts, aber Liste innerhalb des Objekts sind nur Referenzen

```
b = [1, 2, [3, 4]]
a = b[:]           # Erzeuge eine flache Kopie von b.
a.append(100)      # Füge Element an a hinzu.
print(b)           # Ergibt '[1, 2, [3, 4]]'. b unverändert.
a[2][0] = -100     # Ändere ein Element von a.
print b            # Ergibt '[1, 2, [-100, 4]]'.
```

Tiefe Kopie: Kopie eines Objekts und rekursive Kopie aller darin enthaltenen Objekte

```
import copy
b = [1, 2, [3, 4]]
a = copy.deepcopy(b)
```

Begriffe zu Wissen:
Signal (analog/digital)
Nachricht/Daten (Speicherzustand von Information in Rechner o.ä.)
Information (Zeichenfolge, die Wissen enthält)
Wissen

$$S \xrightarrow{\eta} \begin{matrix} N \\ D \end{matrix} \xrightarrow{\alpha} I \xrightarrow{\beta} W$$

Kodierung: Zuordnung (Abbildung) von Werten eines Zeichenvorrats auf einen anderen Zeichenvorrat.

Shannonsche Informationstheorie
Unterschiedliche Zeichen kommen unterschiedlich häufig in Sprachen vor
--> aus dem Auftreten von selteneren Zeichen, kann man mehr Information über das gesamte Wort ziehen (Enthält ein unbekanntes Wort ein x, dann schränkt das die Anzahl der im deutschen möglichen Wörter deutlich ein)

abrakadabra

	absolute Häufigkeit $h_n(A)$ = Anzahl des Auftretens im Beispieltext	**relative Häufigkeit** im Beispieltext $H_n(A) = \frac{h_n(A)}{n}$
a	5	5/(11*100)=0,45
b	2	0,18
d	2	0,18
k	1	0,09
r	1	0,09
Summe ($n=\|\Omega\|$)	11	1,00

Ereignis	Interpretation	Relative Häufigkeit	Wahrscheinlichkeit
Ω	Gesamtereignis (das sichere Ereignis)	$H_n(\Omega) = 1$	$P(\Omega) = 1$
$\varnothing$	Unmögliches Ereignis	$H_n(\varnothing) = 0$	$P(\varnothing) = 0$
A	Beliebiges Ereignis	$0 \le H_n(A) \le 1$	$0 \le P(A) \le 1$
$A \cup B$	Es treten die Ereignisse A oder B auf, die „Summe“ von Ereignissen	$H_n(A \cup B) = H_n(A) + H_n(B) - H_n(A \cap B)$	$P(A \cup B) = P(A) + P(B) - P(A \cap B)$
$A \cap B$	Es treten A und B ein		
A \ B	Es tritt A und nicht B ein	$H_n(A \cup B) = H_n(A) + H_n(B)$	$P(A \cup B) = P(A) + P(B)$
$\overline{A}$	Das Ereignis A tritt nicht ein - das komplementäre Ereignis	$H_n(\overline{A}) = 1 - H_n(A)$	$P(\overline{A}) = 1 - P(A)$
$A \subseteq B$	Tritt das Ereignis A ein, dann tritt auch B ein	$H_n(A) \le H_n(B)$	$P(A) \le P(B)$
	Die Ereignisse A und B sind unabhängig voneinander	$H_n(A \cap B) = H_n(A) \cdot H_n(B)$	$P(A \cap B) = P(A) \cdot P(B)$

Informationsgehalt von Nachrichten:

- Mit steigender Wahrscheinlichkeit für das Auftreten einer Nachricht sinkt deren Informationsgehalt.
- Da die Wahrscheinlichkeit einen Wertebereich von 0 bis 1 aufweist, ergibt sich immer ein positiver Wert für den Informationsgehalt einer Nachricht.
- Für Nachrichten mit gegen Null gehender Wahrscheinlichkeit steigt der Informationsgehalt stark an und geht für $P(x) \rightarrow 0$ gegen unendlich.
- **Bei voneinander unabhängigen Nachrichten addiert sich der Informationsgehalt.**
- Der Informationsgehalt gibt die günstigste (kürzeste) Codelänge für die Darstellung des Zeichens in Bit an.
- Wenn die Wahrscheinlichkeit für das Auftreten eines Zeichens den Wert 0,5 annimmt, ist der Informationsgehalt gleich 1. Ein idealer Code würde für dieses Zeichen eine Kodelänge von einem Bit verwenden.

Entropie H einer Nachrichtenquelle macht Aussage über „Maß an Zufälligkeit“ ihres Inhalts

$$H(I) = -\sum_{j=1}^{|\Omega|} p_j \cdot \log_2 p_j$$

Omega = Zeichenvorrat
I = Informationsquelle (Nachrichtenquelle)
pj = Wahrscheinlichkeit mit der Nachricht auftritt

Bsp für abrakadabra:

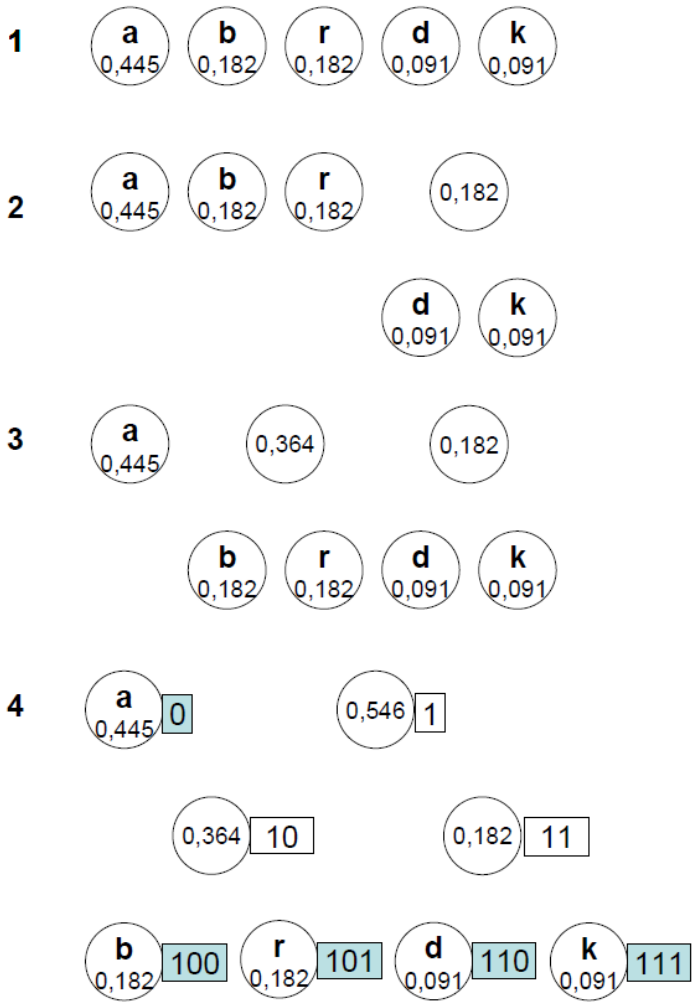
Zeichen A	**relative Häufigkeit** im Beispieltext $H_n(A) = \frac{h_n(A)}{n}$	- p(A) $\log_2$ p(A) = - $H_n(A)$ $\log_2$ $H_n(A)$
a	5/(11*100)=0,455	0,517
b	0,182	0,447
d	0,182	0,447
k	0,091	0,314
r	0,091	0,314
Summe (**n=** $\|\Omega\|$)	1,00	2,040

Ergebnis ist Summe aller: (–1) * (relative Häufigkeit(A)) * log2(relative Häufigkeit(A))
A = Zeichen in Wort (Nachricht)
Ergebnis sind bit (hier 2.040 bit)

Für Binäre Nachrichtenquelle gilt: P(A) = 1 – P(B)

abrakadabra enthält 5 verschiedene Zeichen --> 3 Bit für Codierung notwendig
Retundanz = 3bit – 2.040 bit = 0.96 bit
0.96 bit werden nicht genutzt

Variable Codelänge um Retundanz zu minimieren:
Bsp Huffman Code



Schritt 1: Wir notieren die zu kodierenden Zeichen mit ihren Häufigkeiten als Knoten.

Schritt 2: Wir fassen die zwei Knoten (allgemeiner zwei Knoten) mit den geringsten Häufigkeiten zusammen zu einem „virtuellen“ Knoten (Bedeutung hier d oder k), der mit der Summenhäufigkeit auftritt.

Schritt 3: Wie Schritt 2

Schritt 4: Wie Schritt 2; Abbruch, wenn auf oberster Ebene nur noch zwei Knoten vorhanden sind.

Abschluß: Wir ordnen allen Knoten Codewerte zu:

Auf 1. Ebene 0 und 1
2. Ebene 10 und 11
3. Ebene 100, 101, 110 und 111

jeweils, soweit noch „Nachfolger“ vorhanden sind.

Huffman Code Ergebnis:

Zeichen A	**relative Häufigkeit** im Beispieltext $H_n(A) = \frac{h_n(A)}{n}$	- p(A) $\log_2$ p(A) = - $H_n(A)$ $\log_2$ $H_n(A)$	kanonischer Binärcode	Huffman Code	Gewichtete Codelänge $p(A) \cdot \lvert C_n \rvert$
a	5/(11*100)=0,455	0,517	000	0	0,455
b	0,182	0,447	001	100	0,545
d	0,182	0,447	010	110	0,545
k	0,091	0,314	011	111	0,273
r	0,091	0,314	100	101	0,273
Summe (**n=** $\lvert\Omega\rvert$)	1,00	2,040 bit	3 bit		2,091 bit

Ergebnis bei Huffman Code 2.091
--> Retundanz = 2.091 bit – 2.040 bit = 0.051 bit
--> Deutlich niedrigere Retundanz, da Codes, die mit 0 beginnen immer sofort als a interpretiert werden
Hoffman Kodierung ist Entropiecodierung

Nyquist-Frequenz (Irgendwas mit Signalübertragung und Informationsverlust)