1 Zeichen *(characters)*, Zeichenketten *(strings)* und Unicode

1.1 Zeichencodierungen (Frühgeschichte)

Es gab - abhängig von Hersteller, Land, Programmiersprache, Betriebsssystem,… - eine große Vielzahl von Codierungen.

Bis heute relevant sind:

1.1.1 ASCII

Der *American Standard Code for Information Interchange* wurde 1963 in den USA als Standard veröffentlicht.

- definiert $2^7 = 128$ Zeichen:
  - 33 Steuerzeichen, wie `newline`, `escape`, `end of transmission/file`, `delete`
  - 95 graphisch darstellbare Zeichen:
    - 52 lateinische Buchstaben `a-z, A-Z`
    - 10 Ziffern `0-9`
    - 7 Satzzeichen `.,:;?!"`
    - 1 Leerzeichen
    - 6 Klammern `[{()}]`
    - 7 mathematische Operationen `+-*/<>=`
    - 12 Sonderzeichen `#$%&'\^_|~`@`
- heute noch "kleinster gemeinsamer Nenner" im Codierungs-Chaos
- die ersten 128 Unicode-Zeichen sind identisch mit ASCII

1.1.2 ISO 8859-Zeichensätze

- ASCII nutzt nur 7 Bits.
- In einem Byte kann man durch Setzen des 8. Bits weitere 128 Zeichen unterbringen.
- 1987/88 wurden im ISO 8859-Standard verschiedene 1-Byte-Codierungen festgelegt, die alle ASCII-kompatibel sind, darunter:

Codierung	Region	Sprachen
ISO 8859-1 (Latin-1)	Westeuropa	Deutsch, Französisch,…,Isländisch
ISO 8859-2 (Latin-2)	Osteuropa	slawische Sprachen mit lateinischer Schrift
ISO 8859-3 (Latin-3)	Südeuropa	Türkisch, Maltesisch,…
ISO 8859-4 (Latin-4)	Nordeuropa	Estnisch, Lettisch, Litauisch, Grönländisch, Sami
ISO 8859-5 (Latin/Cyrillic)	Osteuropa	slawische Sprachen mit kyrillischer Schrift
ISO 8859-6 (Latin/Arabic)		
ISO 8859-7 (Latin/Greek)		
…		
ISO 8859-15 (Latin-9)		1999: Revision von Latin-1: jetzt mit Euro-Zeichen!

1.2 Unicode

- Ziel: einheitliche Codierung für alle Schriften der Welt
- Unicode Version 1 erschien 1991
- Unicode Version 14 erschien 2021 mit 144 697 Zeichen (das sind 838 mehr als Unicode 13), darunter:
  - 159 Schriften
  - mathematische und technische Symbole
  - Emojis und andere Symbole, Steuer- und Formatierungszeichen

- davon entfallen über 90 000 Zeichen auf die CJK-Schriften (Chinesisch/Japanisch/Koreanisch)

1.2.1 technische Details

- Jedem Zeichen wird ein `codepoint` zugeordnet. Das ist einfach eine fortlaufende Nummer.
- Diese Nummer wird hexadezimal notiert
  - entweder 4-stellig als `U+XXXX` (0-te Ebene)
  - oder 5…6-stellig als `U+XXXXXX` (weitere Ebenen)
- Jede Ebene geht von `U+XY0000` bis `U+XYFFFF`, kann also $2^{16} = 65\,534$ Zeichen enthalten.

- Vorgesehen sind bisher 17 Ebenen `XY=00` bis `XY=10`, also der Wertebereich von `U+0000` bis `U+10FFFF`.
- Damit sind maximal 21 Bits pro Zeichen nötig.
- Die Gesamtzahl der damit möglichen Codepoints ist etwas kleiner als 0x10FFFF, da aus technischen Gründen gewisse Bereiche nicht verwendet werden. Sie beträgt etwa 1.1 Millionen, es ist also noch viel Platz.
- Bisher wurden nur Codepoints aus den Ebenen
  - Ebene 0 = BMP *Basic Multilingual Plane* `U+0000 - U+FFFF`,
  - Ebene 1 = SMP *Supplementary Multilingual Plane* `U+010000 - U+01FFFF`,
  - Ebene 2 = SIP *Supplementary Ideographic Plane* `U+020000 - U+02FFFF`,
  - Ebene 3 = TIP *Tertiary Ideographic Plane* `U+030000 - U+03FFFF` und
  - Ebene 14 = SSP *Supplementary Special-purpose Plane* `U+0E0000 - U+0EFFFF` vergeben.
- `U+0000` bis `U+007F` ist identisch mit ASCII
- `U+0000` bis `U+00FF` ist identisch mit ISO 8859-1 (Latin-1)

1.2.2 Eigenschaften von Unicode-Zeichen

Im Standard wird jedes Zeichen beschrieben duch

- seinen Codepoint (Nummer)
- einen Namen (welcher nur aus ASCII-Großbuchstaben, Ziffern und Minuszeichen besteht) und
- verschiedene Attributen wie
  - Laufrichtung der Schrift
  - Kategorie: Großbuchstabe, Kleinbuchstabe, modifizierender Buchstabe, Ziffer, Satzzeichen, Symbol, Seperator,….

Codepoint und Name:

```
...
U+0041 LATIN CAPITAL LETTER A
U+0042 LATIN CAPITAL LETTER B
U+0043 LATIN CAPITAL LETTER C
U+0044 LATIN CAPITAL LETTER D
...
U+00E9 LATIN SMALL LETTER E WITH ACUTE
U+00EA LATIN SMALL LETTER E WITH CIRCUMFLEX
...
U+0641 ARABIC LETTER FEH
U+0642 ARABIC LETTER QAF
...
U+21B4 RIGHTWARDS ARROW WITH CORNER DOWNWARDS
...
```

[1]:
```
# Wie sieht 'RIGHTWARDS ARROW WITH CORNER DOWNWARDS' aus?

'\U21b4'
```

[1]:
```
'↴': Unicode U+21B4 (category So: Symbol, other)
```

[2]:
```
"""
printuc(c, n):
print n characters from unicode table, starting with character c
"""
function printuc(c, n)
    for i in 0:n-1
        print(c + i)
    end
end
```

[2]:
```
printuc
```

Kyrillisch

[3]:
```
printuc('\U0400', 100)
```

```
ÈËЂЃЄЅІЇЈЉЊЋЌЍЎЏАБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмнопрстуфхцчшщъыьэюя
ѐёђѓєѕіїјљњћќѝўџѠѡѢѣ
```

Tamilisch

[4]:
```
printuc('\U0be7',20)
```

```
௧௨௩௪௫௬௭௮௯௰௱௲௳௴௵௶௷௸
```

Schach

[5]:
```
printuc('\U2654', 12)
```

```
♔♕♖♗♘♙♚♛♜♝♞♟
```

mathematische Operatoren

[6]:
```
printuc('\U2200', 255)
```

```
∀∁∂∃∄∅∆∇∈∉∊∋∌∍∎∏∐∑−∓∔∕∖∗∘∙√∛∜∝∞∟∠∡∢∣∤∥∦∧∨∩∪∫∬∭∮∯∰∱∲∳∴∵∶∷∸∹∺∻∼∽∾∿≀≁≂≃≄≅≆≇≈≉≊≋≌≍≎≏
≐≑≒≓≔≕≖≗≘≙≚≛≜≝≞≟≠≡≢≣≤≥≦≧≨≩≪≫≬≭≮≯≰≱≲≳≴≵≶≷≸≹≺≻≼≽≾≿⊀⊁⊂⊃⊄⊅⊆⊇⊈⊉⊊⊋⊌⊍⊎⊏⊐⊑⊒⊓⊔⊕⊖⊗⊘⊙⊚⊛⊜⊝⊞⊟
⊠⊡⊢⊣⊤⊥⊦⊧⊨⊩⊪⊫⊬⊭⊮⊯⊰⊱⊲⊳⊴⊵⊶⊷⊸⊹⊺⊻⊼⊽⊾⊿⋀⋁⋂⋃⋄⋅⋆⋇⋈⋉⋊⋋⋌⋍⋎⋏⋐⋑⋒⋓⋔⋕⋖⋗⋘⋙⋚⋛⋜⋝⋞⋟⋠⋡⋢⋣⋤⋥⋦⋧⋨⋩⋪⋫⋬⋭⋮⋯
⋰⋱⋲⋳⋴⋵⋶⋷⋸⋹⋺⋻⋼⋽⋾⋿
```

Runen

[7]:
```
printuc('\U16a0', 40)
```

```
ᚠᚡᚢᚣᚤᚥᚦᚧᚨᚩᚪᚫᚬᚭᚮᚯᚰᚱᚲᚳᚴᚵᚶᚷᚸᚹᚺᚻᚼᚽᚾᚿᛀᛁᛂᛃᛄᛅ
```

Scheibe (Diskus) von Phaistos

- Diese Schrift ist nicht entziffert.
- Es ist unklar, welche Sprache dargestellt wird.
- Es gibt nur ein einziges Dokument in dieser Schrift: die Tonscheibe von Phaistos aus der Bronzezeit

[8]:
```
printuc('\U101D0', 46 )
```

1.2.3 Unicode transformation formats: UTF-8, UTF-16, UTF-32

Unicode transformation formats legen fest, wie eine Folge von Codepoints als eine Folge von Bytes dargestellt wird.

Da die Codepoints unterschiedlich lang sind, kann man sie nicht einfach hintereinander schreiben. Wo hört einer auf und fängt der nächste an?

- **UTF-32**: Das einfachste, aber auch speicheraufwändigste, ist, sie alle auf gleiche Länge zu bringen. Jeder Codepoint wird in 4 Bytes = 32 Bit kodiert.
- Bei **UTF-16** wird ein Codepoint entweder mit 2 Bytes oder mit 4 Bytes dargestellt.
- Bei **UTF-8** wird ein Codepoint mit 1,2,3 oder 4 Bytes dargestellt.

- **UTF-8** ist das Format mit der höchsten Verbreitung. Es wird auch von Julia verwendet.

UTF-8

- Für jeden Codepoint werden 1, 2, 3 oder 4 volle Bytes verwendet.
- Bei einer Codierung mit variabler Länge muss man erkennen können, welche Bytefolgen zusammengehören:
  - Ein Byte der Form 0xxxxxxx steht für einen ASCII-Codepoint der Länge 1.
  - Ein Byte der Form 110xxxxx startet einen 2-Byte-Code.
  - Ein Byte der Form 1110xxxx startet einen 3-Byte-Code.
  - Ein Byte der Form 11110xxx startet einen 4-Byte-Code.
  - Alle weiteren Bytes eines 2-,3- oder 4-Byte-Codes haben die Form 10xxxxxx.
- Damit ist der Platz, der für den Codepoint zur Verfügung steht (Anzahl der x):
  - Ein-Byte-Code: 7 Bits
  - Zwei-Byte-Code: $5 + 6 = 11$ Bits
  - Drei-Byte-Code: $4 + 6 + 6 = 16$ Bits
  - Vier-Byte-Code: $3 + 6 + 6 + 6 = 21$ Bits
- Damit ist jeder ASCII-Text automatisch auch ein korrekt codierter UTF-8-Text.
- Sollten die bisher für Unicode festgelegten 17 Ebenen = 21 Bit = 1.1 Mill. mögliche Zeichen mal erweitert werden, dann wird UTF-8 auf 5- und 6-Byte-Codes erweitert.

2 Zeichen und Zeichenketten in Julia

2.1 `Char`

- einzelnes Zeichen in einfachen Anführungszeichen: `'a'`

- 4 Bytes Speicher
- represäntieren Unicode-Codepoints
- kann von/zu `UInt`s umgewandelt werden
- Integer-Wert = Unicode-codepoint

2.2 `String`

- doppelte Anführungszeichen: `"a"`
- UTF-8-codiert, d.h., ein Zeichen kann zwischen 1 und 4 Bytes lang sein

[9]:
```
@show typeof('a') sizeof('a') typeof("a") sizeof("a");
```

```
typeof('a') = Char
sizeof('a') = 4
typeof("a") = String
sizeof("a") = 1
```

[10]:
```
UInt('a')
```

[10]:
```
0x0000000000000061
```

[11]:
```
b = Char(0x2656)
```

[11]:
```
'♖': Unicode U+2656 (category So: Symbol, other)
```

Bei einem Nicht-ASCII-String unterscheiden sich Anzahl der Bytes und Anzahl der Zeichen:

[12]:
```
astr = "Hello World!"
@show length(astr) ncodeunits(astr);
```

```
length(astr) = 12
ncodeunits(astr) = 12
```

[13]:
```
str = "😄 Hellö 🎶"
@show length(str) ncodeunits(str);
```

```
length(str) = 9
ncodeunits(str) = 16
```

Iteration über einen String iteriert über die Zeichen:

[14]:
```
for i in str
    println(i, "  ", typeof(i))
end
```

```
😄  Char
   Char
H  Char
e  Char
l  Char
l  Char
ö  Char
   Char
🎶  Char
```

2.2.1 Verkettung von Strings

"Strings mit Verkettung bilden ein nichtkommutatives Monoid."

[15]:
```
str * astr * str
```

[15]:
```
"😄 Hellö 🎶Hello World!😄 Hellö 🎶"
```

[16]:
```
str^3
```

[16]:
```
"😄 Hellö 🎶😄 Hellö 🎶😄 Hellö 🎶"
```

2.2.2 Stringinterpolation

wird oft in print() usw. genutzt

[17]:
```
a = 33.4
b = "x"

s = "Das Ergebnis für $b ist gleich: $a\n"
```

[17]:
```
"Das Ergebnis für x ist gleich: 33.4\n"
```

2.2.3 Backslash escape sequences

Julia benutzt die von C und anderen Sprachen bekannten backslash-Codierungen:

[18]:
```
s = "So bekommt man \'Anführungszeichen\" und ein \$-Zeichen und einen\nZeilenumbruch und ein \\ usw.
  ↪.. "
print(s)
```

```
So bekommt man 'Anführungszeichen" und ein $-Zeichen und einen
Zeilenumbruch und ein \ usw…
```

2.2.4 Triple-Quotes

In dieser Form bleiben Zeilenumbrüche und Anführungszeichen erhalten:

[19]:
```
s = """
 Das soll
ein "längerer"
  'Text' sein.
"""
```

```
print(s)
```

```
 Das soll
ein "längerer"
  'Text' sein.
```

2.2.5 Raw strings

In einem `raw string` sind alle backslash-Codierungen ausser `\"` abgeschaltet:

[20]:
```
s = raw"Ein $ und ein \ und zwei \\ und ein 'bla'..."
print(s)
```

```
Ein $ und ein \ und zwei \\ und ein 'bla'…
```

2.3 Weitere Funktionen für Zeichen und Strings (Auswahl)

2.3.1 Tests für Zeichen

[21]:
```
@show isdigit('0') isletter('Ψ') isascii('\U2655') islowercase('α') isnumeric('½') iscntrl('\n')␣
↪ispunct(';');
```

```
isdigit('0') = true
isletter('Ψ') = true
isascii('♕') = false
islowercase('α') = true
isnumeric('½') = true
iscntrl('\n') = true
ispunct(';') = true
```

2.3.2 Anwendung auf Strings

Diese Tests lassen sich z.B. mit `all()`, `any()` oder `count()` auf Strings anwenden:

[22]:
```
all(ispunct, ";.:")
```

[22]:
```
true
```

[23]:
```
any(isdigit,"Es ist 3 Uhr! 🕒" )
```

[23]:
```
true
```

[24]:
```
count(islowercase,"Hello, du!!")
```

[24]:
```
6
```

noch ein paar mehr...

[25]:
```
@show startswith("Lampenschirm", "Lamp")  occursin("pensch", "Lampenschirm") ␣
↪endswith("Lampenschirm", "irm");
```

```
startswith("Lampenschirm", "Lamp") = true
occursin("pensch", "Lampenschirm") = true
endswith("Lampenschirm", "irm") = true
```

[26]:
```
@show uppercase("Eis") lowercase("Eis")  titlecase("eiSen");
```

```
uppercase("Eis") = "EIS"
lowercase("Eis") = "eis"
titlecase("eiSen") = "Eisen"
```

[27]:
```
# remove newline from end of string

@show chomp("Eis\n")  chomp("Eis");
```

```
chomp("Eis\n") = "Eis"
chomp("Eis") = "Eis"
```

[28]:
```
split("π ist irrational.")
```

[28]:
```
3-element Vector{SubString{String}}:
 "π"
 "ist"
 "irrational."
```

[29]:
```
replace("π ist irrational.", "ist" ⇒ "ist angeblich")
```

[29]:
```
"π ist angeblich irrational."
```

2.3.3 Strings und Indizes

- Strings sind indizierbar
- Besonderheiten:
  - der Index ist ein Byte-Index
  - bei einem nicht-ASCII-String sind nicht alle Indizes gültig
  - ein gültiger Index adressiert immer ein Unicode-Zeichen

[30]:
```
str
```

[30]:
```
"😄 Hellö 🎶"
```

[31]:
```
# das erste Zeichen

str[1]
```

[31]:
```
'😄': Unicode U+1F604 (category So: Symbol, other)
```

[32]:
```
# da es 4 bytes lang ist, sind 2,3,4 alles ungültige Indizes:

str[2]
```

```
StringIndexError: invalid index [2], valid nearby indices [1]=>'😄', [5]=>' '

Stacktrace:
 [1] string_index_err(s::String, i::Int64)
   @ Base ./strings/string.jl:12
 [2] getindex_continued(s::String, i::Int64, u::UInt32)
   @ Base ./strings/string.jl:233
 [3] getindex(s::String, i::Int64)
   @ Base ./strings/string.jl:226
 [4] top-level scope
   @ In[32]:3
 [5] eval
   @ ./boot.jl:373 [inlined]
 [6] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base ./loading.jl:1196
```

[33]:
```
# erst das 5. Byte ist ein neues Zeichen:

str[5]
```

[33]:
```
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

[34]:
```
# auch bei der Adressierung von Substrings müssen Anfang und Ende
# jeweils gültige Indizes sein
```

```
str[1:7]
```

[34]: "😄 He"

[35]:
```
# eachindex() liefert einen Iterator über die gültigen Indizes:

for i in eachindex(str)
    c = str[i]
    println("$i: $c")
end
```

```
1: 😄
5:
6: H
7: e
8: l
9: l
10: ö
12:
13: 🎶
```

[36]:
```
# ... und wie üblich macht collect() aus einem Iterator einen Vektor:

collect(eachindex(str))
```

[36]:
```
9-element Vector{Int64}:
  1
  5
  6
  7
  8
  9
 10
 12
 13
```

[37]:
```
# und nextind()  liefert den nächsten gültigen Index

@show nextind(str, 1) nextind(str, 2);
```

```
nextind(str, 1) = 5
nextind(str, 2) = 5
```

Warum? Effizienz!

- In einem langen String, zB einem Buchtext, ist die Stelle `s[123455]` mit einem Byte-Index schnell zu finden.
- Ein Zeichen-Index müsste in der UTF-8-Codierung den ganzen String durchlaufen, um das n-te Zeichen zu finden, da die Zeichen 1,2,3 oder 4 Bytes lang sein können.

Einige Funktionen liefern Indizes oder Ranges als Resultat. Sie liefern immer gültige Indizes:

[38]:
```
findfirst('l', str)
```

[38]: 8

[39]:
```
findfirst("Hel", str)
```

[39]: 6:8

[40]:
```
str2 = "αβγδε"^3
```

[40]: "αβγδεαβγδεαβγδε"

[41]:
```
n = findfirst('γ', str2)
```

[41]: 5

[42]:
```
# Weitersuchen ab dem nächsten nach n=5 gültigen Index:

findnext('γ',str2, nextind(str2, n))
```

[42]: 15

[ ]:
```

```