Einführung in die Programmierung

Andreas Hildebrandt

Arrays

- Bisher nur betrachtet: (fast) triviale Programme mit wenigen Variablen
- Für realistischere (d.h. schwerere) Probleme müssen wir anfangen, Daten zu **strukturieren**
- Dazu dienen **Datenstrukturen** (ach?)
- Auswahl der richtigen Datenstruktur und des richtigen Algorithmus ist Kernkompetenz des Informatikers
- Absolut entscheidend für erfolgreiche Softwareentwicklung

Arrays

- Informatik kennt viele teils sehr komplexe Datenstrukturen
- Einige davon später in DSEA besprochen
- In dieser Vorlesung nur absolut elementare Datenstrukturen behandelt
- Wohl grundlegendste Datenstruktur ist das eindimensionale **Array** (**Feld**)
- Arrays speichern **Sequenzen** von Objekten (in Python genauer: Objektreferenzen)
- Einzelne Objekte nennen wir **Elemente** des Arrays
- Elemente sind von 0 bis n-1 **durchnummeriert**
- Position eines Elements bekannt als sein **Index** (Zahl zwischen 0 und n-1)
- Nennen Element mit Index i dann auch kurz das **i-te Element** des Arrays

Arrays

- **Hinweis:** Arrays **sollten** nur Elemente vom gleichen Datentyp enthalten
- Vorstellung einer Sequenz von Daten gleichen Typs
- **Aber:** in Python **nicht** von Compiler/Laufzeitsystem erzwungen
- Arrays in Python *können* Elemente unterschiedlichen Typs enthalten...
- ...sollten das aber nur, wenn es wichtigen Grund dafür gibt
- heterogene Arrays sehr verwirrend, schwer zu verwenden, Quelle von Programmfehlern

Arrays

- Neben eindimensionalen Arrays auch mehrdimensionale möglich
- Hat dann mehrere Indizes (andere Betrachtung: ein mehrdimensionaler Index, Multiindex)
- Implementiert als "Array von Arrays"
  - Elemente des äußeren Arrays sind selbst wieder Arrays
  - Haben dann einen Index für das äußere Array ⇒ liefert inneres Array
  - Elemente des inneren Arrays haben selbst wieder eigene Indizes
  - Sind Elemente des inneren Array selbst wieder Elemente, geht es wie oben weiter

Arrays

- **Achtung, Verwirrungspotential:** Arrays heißen in Python **Listen**
- In der Praxis in Python beide Begriffe meist gleich verwendet, nicht genauer unterschieden
- **Aber:**
  - In Python existiert noch spezialisierte, selten verwendete Datenstruktur array
  - In der Informatik bezeichnen Listen und Arrays sehr unterschiedliche Dinge
- Eigentliche Implementierung als dynamisch wachsende Arrays im Sinne der Informatik (ähnlich z.B. std::vector in C++)
- Entscheidende Eigenschaften (charakteristisch für Arrays):
  - **random access** (**wahlfreier Zugriff**) - können auf jedes Element direkt und effizient mittels Index zugreifen
  - Daten liegen **hintereinander** im Speicher

Arrays

- Ist die Frage überhaupt wichtig?
- Ja! Präzise Notation entscheidend in der Programmierung
- Besonders aufpassen bei
  - Vergleich mit anderen Programmiersprachen
  - Diskussion mit Informatikern
- Am besten informatischen Sprachgebrauch angewöhnen

Arrays

- Wie legt man Arrays in Python an?
- Einfachste Möglichkeit für kleine Arrays mit bekanntem Inhalt: **Array-Literale**
- Geben dazu Array-Elemente als Literale an, getrennt durch , und umgeben von []
- **Beispiel:**

In [3]:
```
a = [1, 3, 5, 7]
print(a)

lexemes = ["Hello", ", ", "World", "!"]
print(lexemes)
```

```
[1, 3, 5, 7]
['Hello', ', ', 'World', '!']
```

Arrays

- **Syntaktischer Zucker:** Operator * mit Operanden vom Typ list und int verfielfacht Array
- **Beispiel:**

In [4]:
```
x = [1, 2, 3] * 3
print(x)
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Arrays

```
a = [1, 3, 5, 7]
```

- Hier speichert Variable a Referenz auf gesamtes Array
- Zugriff auf einzelne Array-Elemente möglich über deren Index mittels **[]** - **Operator**

In [5]:
```
a = [1, 3, 5, 7]
print(a)
print(a[0])
print(a[1])
print(a[2])
print(a[3])
```

```
[1, 3, 5, 7]
1
3
5
7
```

Arrays

- **Achtung:** in der Informatik zählt man ab **0**, nicht ab **1**
- Damit ist Index des ersten Elements **0**, nicht **1**
- Letztes Element eines n-elementigen Arrays hat immer Index **n-1**, **nicht n**
- Sehr häufige Fehlerquelle (**off-by-one** - Fehler)!
- Wird aber aus guten Gründen so gemacht...

Arrays

- Anzahl der Elemente im Array bezeichnen wir auch als seine **Länge**
- Kann in Python durch Funktion len bestimmt werden
- **Beispiel:**

```
In [6]: a = [1, 3, 5, 7]
        print(len(a))
```

```
4
```

- **Hinweis:** letztes Element des Arrays a ist dann a[len(a)-1]

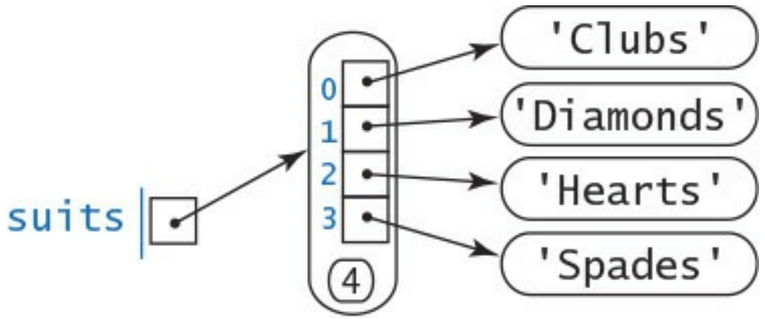
Arrays

- Zeigt x auf ein Array, schreibt man manchmal auch x[], wenn man über die Variable redet
- Wird aber nicht so im Quellcode verwendet!
- Dient nur als Hinweis in Kommentaren, Dokumentation, ...

Arrays

- **Wichtig:** Elemente in Python-Arrays sind nicht Objekte direkt, sondern Objekt-Referenzen
- **Beispiel:** Spielkarten

```
In [7]: suits = ['Clubs', 'Diamonds', 'Hearts', 'Spades'] # Kreuz, Karo, Herz, Pik
```



Array data structure

Arrays

- Eindimensionale Arrays oft als Repräsentation für 1D-Vektoren verwendet
- **Beispiel:** Berechnung des Skalarprodukts $\vec{x} \cdot \vec{y}$ für $\vec{x}, \vec{y} \in$ ' ^{n}
  - Skalarprodukt definiert als $\vec{x} \cdot \vec{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$

In [8]:
```
x = [1., 1., 1.]
y = [0., 1., 0.] # hier nur als Beispiele

if len(x) != len(y):
    print("Skalarprodukt nur für Vektoren gleicher Länge definiert!")
else:
    result = 0.
    for i in range(len(x)):
        result += x[i] * y[i]

print(result)
```

```
1.0
```

Arrays

- Operator + auch auf Arrays definiert
- Erzeugt **Array-Konkatenation**:
  - `c = a + b` ergibt neues Array c
  - Inhalt von c entspricht Inhalt von a gefolgt von Inhalt von b
  - `len(a+b) = len(a) + len(b)`
- Auch hier wieder Kurzschreibweise (syntaktischer Zucker):

  `a += b` entspricht `a = a + b`

Arrays

- Array-Konkatenation technisch nicht ganz einfach effizient umzusetzen
- Für dynamische Arrays wie in Python:

  Anzahl benötigter Operationen für Konkatenation ist proportional zur Länge des neuen Array

Arrays

- Eine Charakteristik von Arrays: Daten liegen zusammenhängend hintereinander im Speicher
- Vereinfachtes Bild (Details komplizierter um dynamisches Wachstum zu ermöglichen):

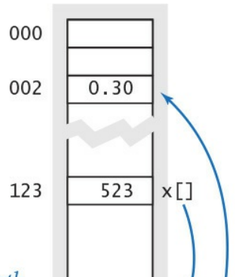
  Array-Referenz x zeigt auf Speicherbereich, in dem

  - die Länge des Arrays
  - die einzelnen Array-Elemente

  abgelegt sind

Arrays

- Vereinfachtes **Beispiel:** (Sedgewick et al., Introduction to Programming in Python)
  - Angenommen, Speicher hat Platz für 1000 Bytes (realistischer Speicher locker einige Millionen mal größer...)
  - Speicher aufgeteilt in Bytes, nummeriert von 0 bis 999 (Realität ist komplizierter; zusammenhängender Speicher nur vorgegaukelt, Alignment wird beachtet, ...)
  - Objektreferenzen interpretierbar als Zahlen zwischen 0 und 999 (Startadresse der Daten des referenzierten Objekts)
  - Betrachte Array-Referenz x[] für float - Array mit 3 Elementen, das an Stelle 523 beginnt
  - *Vorstellung:*
    - in Speicherzelle 523 liegt Länge des Arrays (3)
    - in Speicherzelle 524 liegt Speicheradresse des Elements x[0] (002)
    - in Speicherzelle 525 liegt Speicheradresse des Elements x[1] (998)
    - in Speicherzelle 526 liegt Speicheradresse des Elements x[2] (741)
  - *Achtung:* Modell ist vereinfacht, da Länge und Objektreferenzen mehr als ein Byte belegen; Byteanzahl aber bekannt und fest



Arrays

- Arrays damit sehr ähnlich aufgebaut wie Speicher selbst
- Ermöglicht sehr effiziente Zugriffe: um auf x[i] zuzugreifen, muss Python nur
  - Objektreferenz x[] auflösen (im Beispiel eben: 523) und 1 addieren (wg. Länge des Arrays)
  - Wert von i auf aufgelöste Objektreferenz addieren (524 + i) (genauer: i mal Anzahl Bytes für Objektreferenz; i * 4 in 32 bit - Version, i * 8 in 64 bit - Version)
  - Objektreferenz zurückgeben, die an dieser Stelle im Speicher liegt
- **Beachte:** in Python liegen nicht direkt Objekte im Array, sondern **Objektreferenzen** (Speicheradressen)
- Müssen zum Auslesen des gespeicherten Wertes noch der Referenz folgen

Arrays

- Code zum Zugriff auf Array-Element besteht immer aus einer Addition (genauer: einer Addition und einer Multiplikation)
- Unabhängig von Größe des Arrays und von Element, das gelesen werden soll
- Zugriff benötigt daher **konstante Zeit**
- Extrem wichtige Eigenschaft von Arrays!

Arrays

- Code zum Zugriff auf Array-Element besteht immer aus einer Addition (genauer: einer Addition und einer Multiplikation)
- Unabhängig von Größe des Arrays und von Element, das gelesen werden soll
- Zugriff benötigt daher **konstante Zeit**
- Extrem wichtige Eigenschaft von Arrays!
- ...

Arrays

- Code zum Zugriff auf Array-Element besteht immer aus einer Addition (genauer: einer Addition und einer Multiplikation)
- Unabhängig von Größe des Arrays und von Element, das gelesen werden soll
- Zugriff benötigt daher **konstante Zeit**
- Extrem wichtige Eigenschaft von Arrays!
- ...
- Ok, die Realität sieht wieder komplexer aus...
- Speicherhierarchien und Cache-Effekte verändern Zugriffszeiten abhängig von Zugriffs**reihenfolge**
- In der theoretischen Betrachtung normalerweise ignoriert, aber in der Praxis oft wichtig für performante Applikationen

Arrays

- Was passiert bei Zugriff auf Index, der außerhalb des Arrays liegt?
- **Beispiel:** x[len(x)] - Speicherzelle direkt hinter letztem Array-Element
- Würden dann Wert auslesen, der nicht mehr zum Array gehört
- Speicherzelle von Betriebssytem möglicherweise sogar anderem Programm zugeteilt, dessen Daten dort liegen
- Gigantische Sicherheitslücke und Fehlerquelle:
  - könnten Werte von anderen Programmen auslesen (Kreditkartendaten, Passwörter, ...)
  - könnten Werte von anderen Programmen überschreiben
- Möglich z.B. in Programmiersprache C - wohl häufigste Quelle von Sicherheitslücken
- Teilweise von Betriebssytem verhindert, aber kompletter Schutz so nicht möglich

Arrays

- Python erlaubt **keinen** Zugriff außerhalb der Array-Grenzen!
- **Bounds Checking:** Python löst exception aus, wenn Wert außerhalb des Arrays angefordert wird
- **Beispiel:**

```
In [9]: x = [1, 2, 3, 4]

        x[len(x)]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-f00f37849974> in <module>()
      1 x = [1, 2, 3, 4]
      2
----> 3 x[len(x)]

IndexError: list index out of range
```

Arrays

- Können mit for - Schleifen sehr einfach über Arrays iterieren:

In [10]:
```
x = [1, 2, 3, 4]
for i in range(len(x)):
    print(x[i], end=" ")
```

```
1 2 3 4
```

- Oft will man **rückwärts** durch das Array laufen:

In [11]:
```
x = [1, 2, 3, 4]
for i in range(len(x)):
    print(x[len(x)-(i+1)], end=" ")
```

```
4 3 2 1
```

Arrays

- Rückwärts-Zugriff so oft benötigt, dass Python syntaktischen Zucker für Array-Referenzierung anbietet
- Wird **negativer Index** $i \in [-n, -1]$ übergeben, liefert Python den Wert x[len(x) - |i|]

In [13]:
```
x = [1, 2, 3, 4]
for i in range(len(x)):
    print(x[-(i+1)], end=" ")
print()
# oder
for i in range(1, len(x)+1):
    print(x[-i], end=" ")
```

```
4 3 2 1
4 3 2 1
```

Arrays

- Wird bei Iteration nur der **Wert** benötigt, nicht der **Index**, bietet Python weiteren syntaktischen Zucker
- Alternative Form der for - Schleife: statt range direkt das Array angeben (auch **for each - Schleife** genannt)

In [14]:
```
x = [1, 2, 3, 4]
for v in x:
    print(v, end=" ")
```

```
1 2 3 4
```

- Python bindet Array-Elemente der Reihe nach direkt an Variable
- Nicht sinnvoll, wenn auch Array-Indizes benötigt werden:

In [15]:
```
x = [1, 2, 3, 4]
for i in range(len(x)):
    print(str(i) + ": " + str(x[i]))
```

```
0: 1
1: 2
2: 3
3: 4
```

Arrays

- Auch Rückwärts-Iteration in dieser Variante möglich
- Dazu dient Funktion reversed:

In [ ]:
```
x = [1, 2, 3, 4]
for v in reversed(x):
    print(v, end=" ")
```

- reversed sorgfältig implementiert
- Array wird nicht wirklich umsortiert (wäre sehr teuer!), verhält sich nur so als ob
- Gelöst über **Iterator** (betrachten wir später)

Arrays

- Benötigen sehr oft Teilarrays eines größeren Arrays
- Python bietet dafür syntaktischen Zucker namens **Slicing**
- **Array-Slice** a[i:j] liefert **neues Array**, in das Array-Elemente a[i] bis a[j-1] **kopiert** werden, falls $i >= 0, j <$ len(a)
- Genauer: kopiert werden die **Objektreferenzen**, nicht die dahinter stehenden Objekte
- Falls i oder $j >=$ len(a) wird nur der Teil kopiert, der in das Array fällt (kein Fehler, keine Exception!)
- Falls $i >= j$ ist Ergebnis leer

Arrays

- **Beispiel:**

In [16]:
```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(x[0:4])
print(x[0:20])
```

```
[0, 1, 2, 3]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Arrays

- Slices mit negativen Indizes etwas unintuitiv
- Negativer Index $i < 0$ wird ersetzt durch len(x) - Iil wie bei Arrayindizierung
- **Beispiel:**

In [17]:
```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(x[-1:-11])
print(x[-11:-1])
```

```
[]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

- Sehr oft verwendet, um nur erste k Elemente eines Arrays zu kopieren
- **Beispiel:**

In [18]:
```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(x[0:-1])
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Arrays

- i und j können jeweils weggelassen werden
- i wird dann ersetzt durch 0, j durch len(x)

```
In [19]: x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
         print(x[:4])
         print(x[5:])
         print(x[:])
```

```
[0, 1, 2, 3]
[5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **Hinweis:** x[:] legt vollständige **Kopie** des Arrays x an!

Arrays

- Noch mehr syntaktischer Zucker: **extended Slices**
- Ermöglichen neben Angabe von Start- und Endindex auch noch Wahl der **Schrittweite**
- **Notation:** x[i:j:k]
- Erzeugt neues Array und kopiert Objektreferenzen der Einträge x[i], x[i + k], …, x[j-1] (jedes k-te Element von i bis j)
- Index des letzten kopierten Elements immer kleiner als j
- Wie bei einfachen Slices nur der Teil kopiert, der innerhalb des Arrays liegt
- i, j und k können wieder weggelassen werden
- i dann wieder ersetzt durch 0, j durch len(x), k durch 1

Arrays

- **Beispiel:**

In [20]:
```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(x[1:len(x):2])
print(x[::2])
```

```
[1, 3, 5, 7, 9]
[0, 2, 4, 6, 8]
```

Arrays

- Bei extended slices auch **negative Schrittweite** möglich
- i sollte dann größer sein als j

In [21]:
```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
x[10:5:-1]
```

Out[21]: [9, 8, 7, 6]

Arrays

- **Achtung:** könnten bei extended slices mit negativer Schrittweite erstes Array-Element eigentlich nicht mitkopieren:
  - Setzen wir j auf 0, stoppt Iteration bei 1 (**vor** j, obere Grenze nicht inkludiert)
  - Setzen wir j auf -1, wird es ersetzt durch len(x) - 1
- Daher hier besondere Konvention: wird bei negativer Schrittweite j weggelassen, wird alles bis zum ersten Element kopiert!
- Unerwartet und potentielle Fehlerquelle!

In [22]:
```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(x[len(x):0:-1])
print(x[len(x)::-1])
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Arrays

- **Hinweis:** Array-Indizierung und Slicing auch für str möglich
- Prinzip bleibt das gleiche
- str hier ähnlich zu Arrays aus einzelnen Zeichen

In [23]:
```
s = "Hallo, Welt!"
print(s[1])
print(s[1:-1])
print(s[::2])
```

```
a
allo, Welt
Hlo et
```

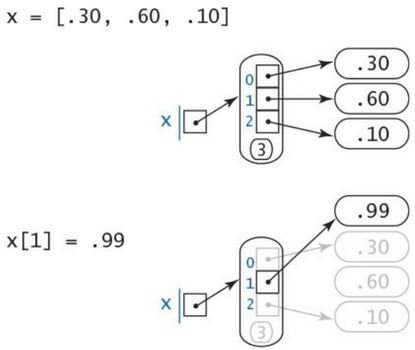
Mutability

- **Definition:** ein Datentyp heißt **veränderlich** (**mutable**), wenn Elemente ihre Werte ändern können
- Arrays in Python sind veränderlich:
  - können vergrößert werden
  - gespeicherte Objektreferenzen können ausgetauscht werden

Mutability

- **Definition:** ein Datentyp heißt **veränderlich** (**mutable**), wenn Elemente ihre Werte ändern können
- Arrays in Python sind veränderlich:
  - können vergrößert werden
  - gespeicherte Objektreferenzen können ausgetauscht werden
- **Beispiel:** (Sedgewick et al., Introduction to Programming in Python)

```
x = [0.30, 0.60, 0.10]
x[1] = .99
```



Reassigning an array element

Mutability

- **Definition:** ein Datentyp heißt **veränderlich** (**mutable**), wenn Elemente ihre Werte ändern können
- Arrays in Python sind veränderlich:
  - können vergrößert werden
  - gespeicherte Objektreferenzen können ausgetauscht werden
- Scheint selbstverständliche Eigenschaft von Datenstrukturen zu sein...
- ...ist es aber nicht
- Mutability hat Nachteile (Fehlerquellen, weniger automatische Optimierung, ...), viele Datenstrukturen daher **unveränderlich** (**immutable**)
- **Beispiel:** str ist immutable

```
In [24]: s = "Hallo, Welt!"
         s[1] = "o"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-2b58817e52b5> in <module>()
      1 s = "Hallo, Welt!"
----> 2 s[1] = "o"

TypeError: 'str' object does not support item assignment
```

Aliasing

- Arrays verhalten sich bei Zuweisung so wie andere Datentypen auch
- Zuweisung erzeugt Bindung an Objekt, gibt Objekt einen Namen
- Mehrere Variablen können auf das gleiche Array zeigen (mehrere Namen)
- Man nennt verschiedene Namen auch **Aliase** des gleichen Objekts
- Wichtige Folge bei veränderlichen Datentypen:
  - Aliase zeigen auf **identisches** Objekt
  - wird dieses verändert, sehen alle Aliase die Änderung

Aliasing

- Was tun, wenn Aliasing unerwünscht?
- Müssen dann Array **kopieren**
- Geht z.B. über Schleife

```
y = []
for v in x:
    y += [v]
```

- Oder eleganter über Slices (erzeugen immer **Kopien**)

```
y = [:]
```

Nützliche Funktionen auf Arrays

- Python bringt viele Funktionen auf Arrays in Standardbibliothek mit
- **Beispiele:**

Name	Beschreibung
len	Länge des Arrays
min	Kleinstes Element, falls Array numerische Werte enthält
max	Größtes Element, falls Array numerische Werte enthält
sum	Summe der Arrayelemente, falls Array numerische Werte enthält
random.shuffle	Zufällige Permutation des Arrays

- **Beispiel:** `avg = sum(x) / len(x)` Mittelwert

Alternativen zu list

- list ist Python-Standard für Arrays
- bequem, aber nicht immer effizient
- mehrere Alternativen verfügbar, z.B. Modul array oder Modul numpy
- insbesondere numpy sehr beliebt in wissenschaftlichen Applikationen
- aber nicht ganz so komfortabel wie list

Häufige Array-Anwendungen

- Arrays verwendet, um Programm Parameter (**Argumente**) beim Start mitzugeben
- Programmstart von program.py auf Kommandozeile bisher mit python program.py
- Nach Dateiname können Argumente - getrennt durch Leerzeichen - angegeben werden
- Werden dann von Python in Array sys.argv aus Modul sys als String-Array abgelegt
- **Achtung:** erster Eintrag (sys.argv[0]) immer Name des aufgerufenen Programms, Argumente ab sys.argv[1]

Häufige Array-Anwendungen
Einführung in das Ausführungsmodell

- **Beispiel**:

  add_numbers.py

```python
import sys

if len(sys.argv) != 3:
    print("Verwendung: python " + sys.argv[0] + " a b")
else:
    print(int(sys.argv[1]) + int(sys.argv[2]))
```

- Ausgabe:

```
[andreas@localhost] $ python add_numbers.py
                     Verwendung: python add_numbers.py a b
[andreas@localhost] $ python add_numbers 1 2
                     3
```

Häufige Array-Anwendungen

- Arrays oft verwendet, um lange if - Blöcke zu vereinfachen
- **Beispiel:**

In [25]:
```
month = 3 # April (wir zählen immer bei 0 los!)

if   month == 0:  days = 31
elif month == 1:  days = 28
elif month == 2:  days = 31
elif month == 3:  days = 30
elif month == 4:  days = 31
elif month == 5:  days = 30
elif month == 6:  days = 31
elif month == 7:  days = 31
elif month == 8:  days = 30
elif month == 9:  days = 31
elif month == 10: days = 30
elif month == 11: days = 31

print(days)
```

```
30
```

Häufige Array-Anwendungen

- Vereinfachung durch passendes Array:

In [26]:
```
month = 3 # April (wir zählen immer bei 0 los!)

days_per_month = [ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ]

print(days_per_month[month])
```

```
30
```

Häufige Array-Anwendungen

- Andere wichtige Anwendung von Arrays: Speichern von vorberechneten Werten
- **Beispiel:** Berechnung der **harmonischen Zahlen** $H_k := \sum_{i=1}^{k} 1/i$
- Naive Berechnung der harmonischen Zahlen von 1 bis n:

In [27]:
```
n = 10
harmonic_numbers = [0.0] # H_0 = 0
for i in range(1, n+1):
    harmonic_numbers += [0.0]
    for k in range(1, i+1):
        harmonic_numbers[i] += 1/k
print(harmonic_numbers)
```

```
[0.0, 1.0, 1.5, 1.8333333333333333, 2.083333333333333, 2.283333333333333, 2.4499999999999997, 2.5928571428571425, 2.7178571428571425,
2.8289682539682537, 2.9289682539682538]
```

Häufige Array-Anwendungen

- Naiver Algorithmus sehr langsam: berechnen immer wieder die gleichen Zahlen mit, denn

  $H_k = 1/k + H_{k-1}$

- Motiviert schnellere Version:

In [28]:
```
n = 10
harmonic_numbers = [0.0] # H_0 = 0
for i in range(1, n+1):
    harmonic_numbers += [1/i + harmonic_numbers[i-1]]
print(harmonic_numbers)
```

```
[0.0, 1.0, 1.5, 1.8333333333333333, 2.083333333333333, 2.283333333333333, 2.4499999999999997, 2.5928571428571425, 2.7178571428571425,
2.8289682539682537, 2.9289682539682538]
```

Plots mit matplotlib

- numerische Arrays können leicht graphisch dargestellt werden
- verwenden dazu Modul matplotlib
- sehr viele verschiedene Plot-Funktionen und Optionen...
- oft ausreichend: numerisches Array an Funktion matplotlib.pyplot.plot übergeben

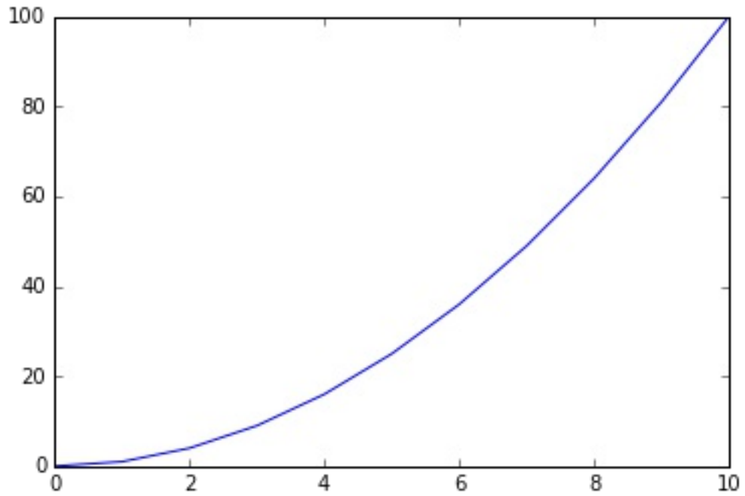
Plots mit matplotlib

- **Beispiel:**

In [29]:
```
import matplotlib

x = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
matplotlib.pyplot.plot(x)
```

Out[29]: [<matplotlib.lines.Line2D at 0x10cb8dbe0>]



Plots mit matplotlib

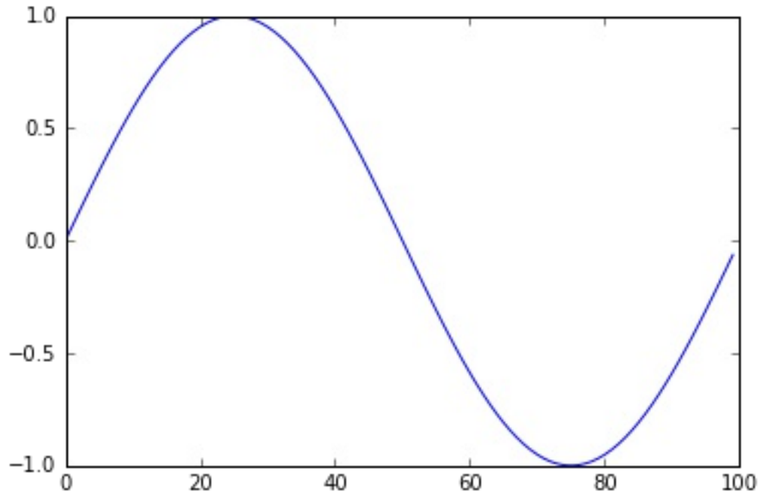
- **Beispiel:**

In [30]:
```
import matplotlib
import math

x = []

for i in range(100):
    x += [math.sin(i * 2 * math.pi / 100.)]
matplotlib.pyplot.plot(x)
```

Out[30]: [<matplotlib.lines.Line2D at 0x10d1e18d0>]



Arrays

- Nicht-triviales Beispiel: **Kartenspiel**
- Karten haben **Farbe** (**Suit**) und **Wert** (**Rank**)
- Dargestellt durch zwei Arrays:

```
farben = ['Kreuz', 'Pik', 'Herz', 'Karo']
werte  = ['2', '3', '4', '5', '6', '7', '8', '9', '10',
          'Bube', 'Dame', 'Koenig', 'Ass']
```

Arrays

- Ein **Satz** Spielkarten besteht aus einer Karte für jeden Wert für jede Farbe

In [31]:
```
farben = ['Kreuz', 'Pik', 'Herz', 'Karo']
werte  = ['2', '3', '4', '5', '6', '7', '8', '9', '10',
          'Bube', 'Dame', 'Koenig', 'Ass']

kartensatz = []
for farbe in farben:
    for wert in werte:
        kartensatz += [farbe + " " + wert]

print(kartensatz[:5])
print(len(kartensatz))
```

```
['Kreuz 2', 'Kreuz 3', 'Kreuz 4', 'Kreuz 5', 'Kreuz 6']
52
```

Arrays

- **Mischen** eines Kartensatzes entspricht Auswahl zufälliger Permutation des Kartensatz-Arrays

In [32]:
```
import random

n = len(kartensatz) # Speichern der Variable macht Schleife effizienter
for i in range(n):
    r = random.randrange(i, n)    # wähle zufällige Karte, die mit i vertauscht wird
    temp = kartensatz[r]          # swappe Karten
    kartensatz[r] = kartensatz[i]
    kartensatz[i] = temp

print(kartensatz[:5])
```

```
['Pik Bube', 'Karo Bube', 'Herz 9', 'Herz 10', 'Karo 8']
```

- Code (aus Sedgewick et al., Introduction to Programming in Python) ist sorgfältig entworfen
- Wahl zufälliger Karte zwischen i und n stellt sicher, dass Permutation gleichverteilt gezogen wird

Arrays

- **Geben** der Karten entspricht Auswahl von Teilarrays aus gemischtem Kartensatz

In [33]:
```
spieler_1 = kartensatz[0:5]
spieler_2 = kartensatz[5:10]
spieler_3 = kartensatz[10:15]

print(spieler_1)
print(spieler_2)
print(spieler_3)
```

```
['Pik Bube', 'Karo Bube', 'Herz 9', 'Herz 10', 'Karo 8']
['Kreuz 4', 'Pik 6', 'Pik Ass', 'Pik Koenig', 'Kreuz 3']
['Pik 9', 'Karo 3', 'Kreuz Koenig', 'Herz Bube', 'Herz 5']
```

Arrays

- **Hinweis:** Python bringt Funktion random.shuffle zur Erzeugung zufälliger Permutation mit
- Mischen dann kurz über `random.shuffle(kartensatz)`
- Zufälliges Ziehen von k Elementen aus Array a ohne Zurücklegen über random.sample(a, k)
- Kartengeben dann über `alle_spieler = random.sample(kartensatz, 15)` möglich
- **Achtung:** nicht dreimal hintereinander 5 Karten ziehen, Karten könnten dann doppelt auftauchen!

Arrays

- Nicht-triviales Beispiel: **Primzahlerkennung**
- **Fragestellung:** gegeben natürliche Zahl n, wieviele Primzahlen $<= n$ gibt es?
- Bekannt als **prime counting function** $\pi(n)$
- Wichtige Fragestellung in der Zahlentheorie, Kryptographie, ...
- Wichtiger Algorithmus: **Sieb des Eratosthenes**

Arrays

- Idee des Algorithmus:
  - gehe natürliche Zahlen von 2 (kleinste Primzahl) bis n der Reihe nach durch
  - alle Vielfachen der aktuell betrachteten Zahl können nicht Prim sein!
  - ist aktuelle Zahl nicht Prim, sind die Vielfachen schon durch Vielfache der Primfaktoren abgedeckt

Arrays

- Implementierung:

In [34]:
```
n = 25

is_prime = []
for i in range(n+1):
    is_prime += [True]

is_prime[0] = False
for i in range(2, n): # gehe der Reihe nach übrige Zahlen durch
    if (is_prime[i]):
        # Alle Vielfache von i können nicht Prim sein
        for j in range(2, n//i + 1):
            is_prime[i*j] = False

# Fertig. Zähle Primzahlen
count = 0
for i in range(2, n+1):
    if (is_prime[i]):
        count += 1
print(count)
```

```
9
```