FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

DIE HOCHSCHULE, DIE MEHR KANN.

ILV Einführung in das Programmieren
Datenstrukturen

Mohamed Goha, BSc. | WS 2024/25

APPLIED LIFE SCIENCES | MASTERSTUDIENGANG BIOINFORMATIK

FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

Kapitelübersicht

- Motivation
- Sequenzielle Datentypen
- Ungeordnete Sammlungen
- Assoziative Datenstrukturen
- (Im)Mutability

Reminder elementare Datentypen und Variablen

- ganze Zahlen (**int**), Gleitkommazahlen (**float**), etc.
- Zugriff auf Daten durch **Variablen**, denen Daten zugewiesen wurden
  - ```
    name = "Patrick" # creates a string
    ```
  - ```
    age = 23 # creates an int
    ```
  - ```
    gdp = 3.65 # creates a float
    ```
- **Static typing**: datentyp ist mit variable assoziiert
- **Dynamic typing**: datentyp ist mit Wert assoziiert, nicht mit der variable
- Variablen in Python sind **dynamically typed** und eine Referenz auf ein **Objekt** im Speicher!
    ```

Motivation Datenstrukturen

- Elementare Datentypen für Speichern von einfachen Werten ausreichend
- Oft möchten wir jedoch eine **Sammlung an Werten** speichern, ohne für jeden einzelnen Wert eine Variable zu definieren
- Beispiel: Telefonbuch
  - `name1, name2, name3 = "Felix", "Sara", "Alexander"`
  - `number1, number2, number3 = "+436601234567" , "+436601234567", "+436601234567"`
  - Bei 1000 Einträgen → 2000 Variablen
  - Nicht praktikabel
- Eine Sammlung von Werten = **Datenstruktur**
  - `names = ["Felix", "Sara", "Alexander"] # Zugriff auf alle Namen über einzelne variable "names"`

Datenstrukturen

- Sequence types (Sequenzielle Datentypen)
- Unordered collections (Ungeordnete Mengen)
- Mapping types (Assoziative Datenstrukturen)
- Classes (Klassen) -> “Objektorientiertes Programmieren”

Sequenzielle Datentypen: Lists (Listen)

- Folge gleichartiger oder verschiedener Elemente
- Elemente haben **definierte Reihenfolge**
- Zugriff auf Elemente mittels **Index**
  - Index fängt in den meisten Programmiersprachen bei 0 an
  - Beispiel: Zugriff auf drittes Element in der Liste mit Index 2

```
names = ["Felix", "Sara", "Alexander"] # Erzeugung mittels eckiger Klammer
print(names[0]) # gibt "Felix" aus
print(names[2]) # gibt "Alexander" aus
```

Sequenzielle Datentypen: Lists (Listen)

- Python Listen sind **mutable**
- Python Listen können alle möglichen Elemente enthalten
- Können **nested** sein - also andere Listen enthalten
- Strings verhalten sich in Python ähnlich zu Listen

Sequenzielle Datentypen: Lists (Listen)

```python
# verschiedene Datentypen innerhalb einer Liste möglich
mixed_list = [2, 5.7, "a string", ["another_string", 4]]

# tic-tac-toe board mittels nested list
tic_tac_toe = [
    ['X', 'O', 'X'],
    ['O', 'X', 'O'],
    ['X', ' ', 'O']
]

print(tic_tac_toe[0][1])  # Output: 'O' (erste Reihe, zweites Element)
print(tic_tac_toe[2][0])  # Output: 'X' (dritte Reihe, erstes Element)

# über board iterieren und ausgeben
for row in tic_tac_toe:
    print(row)
```

(Im)Mutability

- Objekte können **mutable** und **immutable** sein
- **Immutable:**
  - z.B int, float, str, tuple, bytes
  - bedeutet, dass das **Objekt** selbst nicht verändert werden kann
  - Beispiel:

```
x = 5 # x ist Referenz auf int object mit Wert 5
y = x
x = 9 # x ist Referenz auf *neues* int object mit Wert 9

# Frage: Was ist der Inhalt der Variable "y"?
```

(Im)Mutability

- **Mutable:**
  - z.B list, dict, set, bytearray
  - Bedeutet, dass das **Objekt** selbst verändert werden kann
  - Werte können hinzugefügt, entfernt oder modifiziert werden
  - Beispiel:

```
my_fruit = ["apple", "pear", "cherry"] # Inhalt my_fruit: ["apple", "pear", "cherry"]
another_reference = my_fruit

my_fruit[0] = "banana" # Inhalt my_fruit: ['banana', 'pear', 'cherry']
my_fruit.remove("cherry") # Inhalt my_fruit: ['banana', 'pear']
my_fruit.append("orange") # Inhalt my_fruit: ['banana', 'pear', 'orange']
# insert(index, element) fügt Element vor einem dem angegebenen index hinzu
my_fruit.insert(0, "Mango") # Inhalt my_fruit: ['Mango', 'banana', 'pear', 'orange']

# Frage: Was ist der Inhalt der Variable "another_reference"?
```

(Im)Mutability

> Immutability: Um Wert zu ändern, wird **neues Objekt** im Speicher angelegt

> Mutability: Variable **referenziert gleiche Speicherstelle** nach Modifizierung, lediglich Wert(e) des Objekts an dieser Speicherstelle ist/sind modifiziert

(Im)Mutability

> y referenziert int Objekt
>
> mit Wert 5

> x referenziert neues objekt

```
x = 5 # x ist Referenz auf int object mit Wert 5
y = x
x = 9 # x ist Referenz auf *neues* int object mit Wert 9

# Frage: Was ist der Inhalt der Variable "y"?
```

> another_reference referenziert
>
> gleiches Objekt wie my_fruit

```
my_fruit = ["apple", "pear", "cherry"] # Inhalt my_fruit: ["apple", "pear", "cherry"]
another_reference = my_fruit

my_fruit[0] = "banana" # Inhalt my_fruit: ['banana', 'pear', 'cherry']
my_fruit.remove("cherry") # Inhalt my_fruit: ['banana', 'pear']
my_fruit.append("orange") # Inhalt my_fruit: ['banana', 'pear', 'orange']
# insert(index, element) fügt Element vor einem dem angegebenen index hinzu
my_fruit.insert(0, "Mango") # Inhalt my_fruit: ['Mango', 'banana', 'pear', 'orange']

# Frage: Was ist der Inhalt der Variable "another_reference"?
```

Sequenzielle Datentypen: Tuples (Tupel)

> Vergleichbar zu Listen: werden mittels Anzahl an Werten erstellt, die durch Beistriche getrennt sind

```
# Initialisierung eines tuple
my_tuple = 3, 9.6, "a string", ["a", "list", 17]
my_tuple = (3, 9.6, "a string", ["a", "list", 17])
my_tuple = 2, # Beistrich auch bei nur einem Wert
my_tuple = tuple([3, 9.6, "a string", ["a", "list", 17]])
```

Sequenzielle Datentypen: Tuples (Tupel)

> **Unterschied zu Listen:** tuples sind **immutable**!

> Es ist aber möglich, tuples mit mutable Objekten zu erstellen.

Ungeordnete Mengen: Sets

> Sets sind **ungeordnet** und enthaltene Elemente sind **unique** (keine Duplikate)

> Initialisierung in Python mittels `set()` oder geschwungener Klammer

```
my_set = set() # leeres set
my_set = {37, 2.4, "hello"}
my_set.add(10) #mutable
```

> Sets sind **mutable**

Ungeordnete Mengen: Sets

- **Operatoren:**
  - Union
  - Intersection
  - Difference

```
my_set1 = set([37, 12]) # leeres set
my_set2 = {37, 2.4, "hello"}

my_set1 | my_set2 # union: {12, 2.4, 37, 'hello'}
my_set1 & my_set2 # intersection: {37}
my_set1 - my_set2 # difference: {12}
```

- Vollständige Liste:

  https://docs.python.org/3/library/stdtypes.html#set

Assoziative Datenstrukturen: Dictionaries

- Beispiel Telefonbuch: Wir möchten ein Telefonbuch implementieren, also Namen mit Telefonnummern assoziieren
- Möglichkeit: Namen in einer Liste, Telefonnummern in einer zweiten Liste
- Nicht ideal: Wir müssten uns Reihenfolge und Positionen der Namen/Telefonnummern merken
- Es wäre besser, den Namen als “Schlüssel” verwenden zu können, um auf die Telefonnummer zuzugreifen. (--> **mapping**)

Assoziative Datenstrukturen: Dictionaries

- Python dictionaries sind **mappings**
- Bestehen aus **key - value pairs** (name->Telefonnummer)
- Jedes **hashable** Objekt kann als key verwendet werden
- Sind **mutable** und **ordered**

Dictionaries: Code-Beispiel

```
phone_book = {
    "Max Mustermann": "+49 123 456789",
    "Erika Musterfrau": "+49 987 654321",
    "Hans Müller": "+49 234 567890",
}
print(phone_book["Max Mustermann"]) # output: +49 123 456789
```