

# Lecture 2:

# Data Structures in Python

Introduction to Python  
efl Data Science Courses

Johannes Chen

# Table of Contents

## **1. Data Types vs. Data Structures**

## **2. Tuples**

## **3. Lists**

## **4. Sets**

## **5. Dictionaries**

# 1. Data Types vs. Data Structures

- You have just learned about the primitive data types:
  - int: `iVar = 3`
  - float: `dVar = 3.0`
  - string: `sVar = „3“` or `sVar = „three“`
  - boolean: `bVar = True`
- With (primitive) data types:
  - you declare how you want to use the variable
  - you tell the interpreter (which translates your code into machine readable code) how the variable should be treated
  - the data type constrains how the variable may look, or how it may be treated: E.g, `sVar / 2` → **TypeError: unsupported operand type(s) for /: 'str' and 'int'**

# 1. Data Types vs. Data Structures

- With (primitive) data types:
  - you declare how you want to use the variable
  - you tell the interpreter (which translates your code into machine readable code) how the variable should be treated
  - the data type constrains how the variable may look, or how it may be treated: E.g, `bVar / 2` → **TypeError: unsupported operand type(s) for /: 'str' and 'int'**
- Data structures:
  - Organize and manage data
  - Enable you to store, access and operate on data efficiently
  - Provide a set of procedures/functions to manipulate the data structure and the data inside it
  - Various types of data structures: arrays, lists, tuples, dictionaries...

# 1. Data Types vs. Data Structures

## Primitive Data Types

- Declare usage intention and interpretation
- Constrain look and operations

int
float
string
boolean

## Data Structures

- Data Organization, Management, Storage
- Operations to efficiently manipulate the data inside them

Tuple: (1, 2)
List: [1,2,3,4]
Set: {2,1,4,3}
Dictionary: {„key“:“value“}

# Table of Contents

## 1. Data Types vs. Data Structures

## 2. Tuples

## 3. Lists

## 4. Sets

## 5. Dictionaries

## 2. Tuples

- Heterogeneous sequence of elements
- Tuples are immutable (see: <https://docs.python.org/3/glossary.html#term-immutable>)
- Accessing the elements is usually done via indexing
- Can be used to assign multiple values, or retrieve multiple values

(1,2)

(1,2,A,S,5)

(1,2,[2,3,5])

## 2. Tuples – Assigning Values

- Tuples are constructed like this:

```
Texample = 1, 2  
print(Texample)  
# (1,2)
```

- We can also assign more values to a tuple structure

```
Texample2 = 1, 2, 3, 4, 5  
print(Texample2)  
# (1, 2, 3, 4, 5)
```

- We can also assign values of different data types to a tuple structure

```
Texample3 = 1, 2, 3.0, "hey", True  
print(Texample3)  
# (1, 2, 3.0, "hey", True)
```



## 2. Tuples - Indexing

- We can access tuple values via indexing.
- Indexing means that each element within a data structure is assigned a value, by which it is uniquely callable.
- Different Data Structures have different operations for indexing.
- Let's call the item with index 1 of our first tuple, by inserting **[x]** behind our variable, whereas x is the index.

```
Texample[1]
# 2
```

- Why did we get the second element, but not the first?
- This has something to do with how indexing works:
- In programming, we usually use zero-based indexing because of performance and memory allocation reasons.
- Zero-based indexing: We start our index counting at the position 0.
- Let's try to call the first element then by asking for index 0.

index	0	1	2	3	4	5
content	4	6	3	2	7	9

```
Texample[0]
# 1
```

## 2. Tuples – Indexing (2)

- Great! Now let's call index 4 of Texample2.

```
Texample2[4]  
# 5
```

- If we know that a certain element is within the data structure, we can also ask for the index position of the element.
- We do this by using the `.index()` function.
- Ask for the index of the value 4.

```
Texample2.index(4)  
# 3
```

- Good! But what if there are duplicates of this value in the data structure?

```
Texample4 = 1, 4, 3, 4, 4, 5  
Texample4.index(4)  
# 1
```

- Only the index of the first occurrence of the value is called.

## 2. Tuples – Assigning Multiple Values

- Tuples are quite useful, if you want to save multiple values at once.
- It is also quite easy to assign multiple values from a tuple to multiple variables at once.

```
Texample5 = 3.0,4.0,12.0
dVarA, dVarB, dVarC = Texample5
```

- Tuples provide many more functions, which you should definitely explore, since they may be useful for data analysis.
- One problem with tuples is that their values are immutable.

```
Texample6 = 4.0,8.0,16.0
Texample6[1] = 12.0
```

```
Traceback (most recent call last):
  File "<ipython-input-12-128d3d34ee60>", line 2, in <module>
    Texample6[1] = 12.0
TypeError: 'tuple' object does not support item assignment
```

# Table of Contents

**1. Data Types vs. Data Structures**

**2. Tuples**

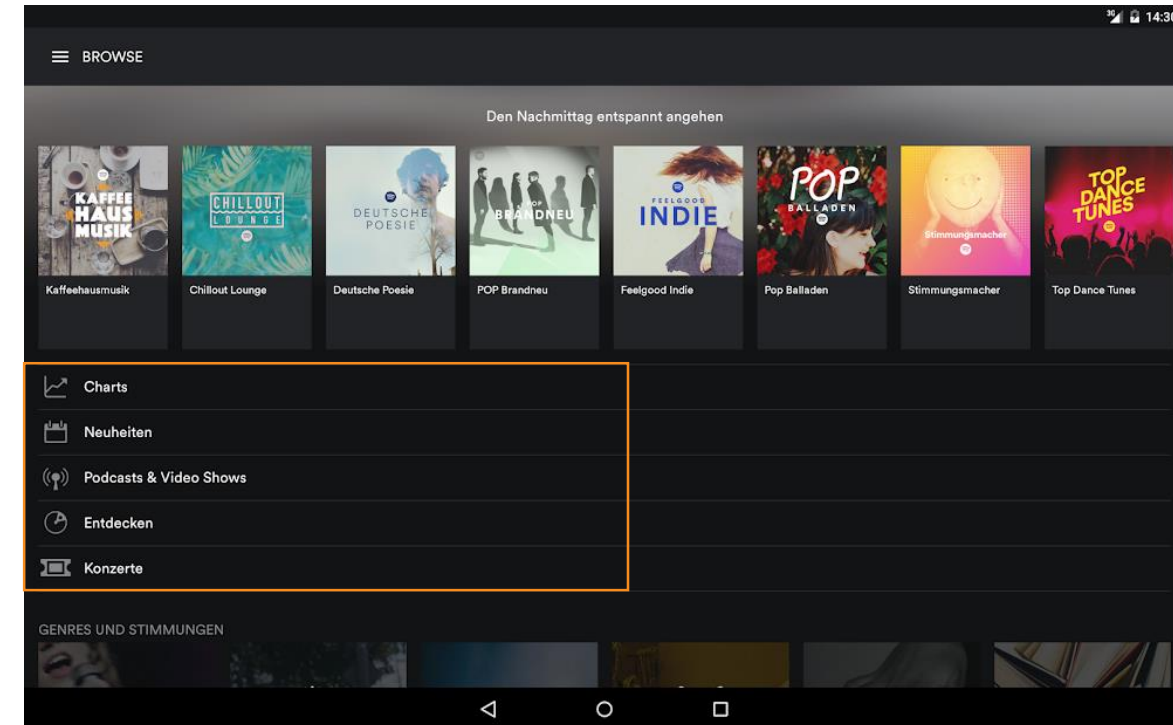
**3. Lists**

**4. Sets**

**5. Dictionaries**

# 3. Lists

- As the name implies, variables inside lists are stored in a list-like data structure
- Elements are usually homogeneous
- Ordered, countable values, e.g. [1,2,3,4]
- Because of the order, lists are indexable
- The same values may occur multiple times
- Lists are a finite sequence, which may be altered
- The contents are mutable
- Lists may be initialized via the command `list()`
- Lists are very important and used in almost every app you use!



<https://play.google.com/store/apps/details?id=com.spotify.music&hl=de>

# 3. Lists

- Lists are constructed by assigning comma-separated values in brackets [ ] like this:

```
LNumbers = [1,2,3,4,5]
print(LNumbers)
# [1,2,3,4,5]
```

- We can also assign values of different data types to a list structure

```
LVarious = [1, 2, 3.0, "hey", True]
print(LVarious)
# [1, 2, 3.0, "hey", True]
```

- If we want to declare a variable as a list before filling it, we can use the list() operator

```
LEmpty = list()
print(LEmpty)
# []
```

# 3. Lists – Indexing and Slicing

- Indexing works similar to the way we did with tuples.

```
LVarious[1]
# 2
LVarious.index(2)
#1
```

- Yet, we can do more interesting things with lists: we can slice them to get a specific range of values.
- We can slice by indicating an index range like [1:4], which gets all elements between 1 and 4. Let's do this.

```
LVarious[1:4]
#[ 2, 3.0, "hey"]
```

- Slicing can be very useful for getting and working on specific pieces of data.
- Let's assign this slice to another list variable.

```
LVariousPart = LVarious[1:4]
print(LVariousPart)
#[ 2, 3.0, "hey"]
```

### 3. Lists – Assigning Values, append, pop

- Since lists are mutable, we can reassign values.
- Assign these values to the spotify list: Charts, Neuheiten, Podcasts & Video Shows, Entdecken, Konzerte

```
LSpotify = ["Charts", "Neuheiten", "Podcasts & Video Shows",  
"Entdecken", "Konzerte"]
```

- Now, reassign the value of list element with ID 0 to “Aktuelle Charts”

```
LSpotify[0] = "Aktuelle Charts"
```

- We can also append elements to the lists. Append the element „Deine Songs“ by using the list.append function.

```
LSpotify.append("Deine Songs")
```

- Lists also provide two methods to delete elements from the lists. Use the .pop method first, then print the list.

```
LSpotify.pop()  
print(LSpotify)
```

- Now use pop with index 3 like this: pop(3)

```
LSpotify.pop(3)  
print(LSpotify)
```



### 3. Lists – remove, insert

- **list.remove(x)** is the other option to remove elements from lists.
- For **x**, we provide the actual value (not index!) we want to remove from the list. Careful: if you have multiple values of the same kind in a list, only the first one is removed.

```
LSpotify.remove("Neuheiten")
print(LSpotify)
```

- Now we can see that the two elements we wanted to remove are gone.
- If we want to insert elements into the list, we can do this as well.
- The function **list.insert(i,x)** inserts the variable **x** at given index **i** into the list.
- Insert „Neuheiten“ at index 1.
- Insert „Entdecken“ at index 3.

```
LSpotify.insert(1, "Neuheiten")
LSpotify.insert(3, "Entdecken")
print(LSpotify)
```

### 3. Lists – len, count, reverse

- There are also useful helper functions if you want to get information about your lists or rearrange it
- Get the length of the spotify list by calling `list.len()`

```
len(LSpotify)
```

- Count the occurrences of „Neuheiten“ in the spotify list with `list.count(x)`

```
LSpotify.count("Neuheiten")
```

- Reverse the list with `list.reverse()` print it, then reverse it again and print it again.

```
LSpotify.reverse()  
print(LSpotify)  
LSpotify.reverse()  
print(LSpotify)
```

# 3. Lists & Strings

- Lists are very important and versatile data structures! Make good use of them.
- Further methods and information on lists may be in the python documentation.
- Fun fact: lists and strings have many common properties, such as indexing and slicing operations.
- Try it out!

```
sTestString = "Lists are awesome and so are Strings!"  
sTestString[3]  
sTestString[0:5]  
len(sTestString)
```

More info available at: <https://docs.python.org/3/tutorial/datastructures.html>

Abelson, H., Sussman, G. J., & Sussman, J. (1985). Structure and Interpretation of Computer Programs. *Cambridge: MIT Press and New York: McGraw-Hill, 1985.*

# Table of Contents

**1. Data Types vs. Data Structures**

**2. Tuples**

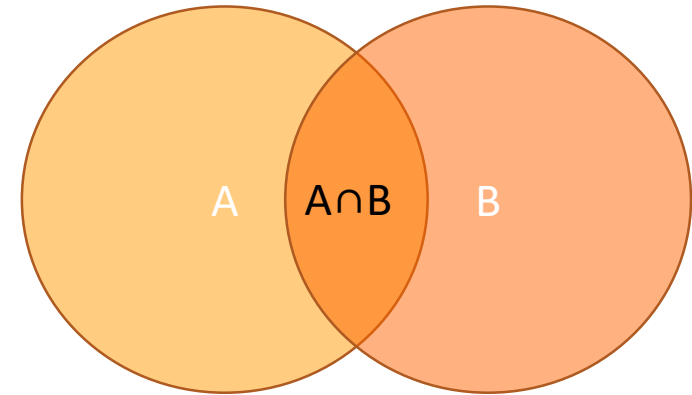
**3. Lists**

**4. Sets**

**5. Dictionaries**

# 4. Sets

- Sets contain a set of values: {1,53,21}
- In Contrast to lists:
  - The values are unordered
  - Are not indexable
  - Values may occur only once → unique values
  - We do not retrieve specific element, we check if it is part of a set
- Sets are a finite set, which may be altered
- The set is mutable (e.g. extendable), the contents are not
- Operations on the set are similar to those of mathematical sets, e.g., union, intersect
- Sets may be initialized via the command `set()`



## 4. Sets

- Sets are constructed by assigning comma-separated values in curly brackets `{}` like this:

```
SNumbers = {1,2,3,4,5,1}
print(SNumbers)
# Out: {1, 2, 3, 4, 5}
```

- As discussed earlier, sets do only contain unique values. If we try to add multiple variables of the same value, only one will remain in the set.
- We can also assign values of different data types to a set structure

```
SVarious = {1, 2, 3.0, "hey"}
print(SVarious)
#Out[55]: {1, 2, 3.0, 'hey'}
```

- If we want to declare a variable as a set before filling it, we can use the `set()` operator

```
SEmpty = set()
print(SEmpty)
#
```

## 4. Sets

- Since sets are unordered, we cannot perform indexing. Instead, we test for membership of a certain value in the set.
- Test the membership of 4 and 2 like this: ***value in SVarious***

```
4 in SVarious
Out: False
2 in SVarious
Out: True
```

- As we can see, this delivers us a boolean value.
- We could use this as a starting condition for some sort of algorithm. You will focus on this part in another lecture.

## 4. Sets – List to set

- Since one of the strengths of the set are the set operations you can perform, it is also possible in python to transform a list into a set (useful for certain occasions)

```
LtoSet = [1,2,3,5,2,4,12,523,123,21]
SfromList = set(LtoSet)
print(SfromList)
# {1, 2, 3, 4, 5, 523, 12, 21, 123}
```

- Be aware that your list loses its order when it is converted to a set!

```
LfromSet = list(SfromList)
print(LfromSet)
```

- See?



## 4. Sets

- As stated earlier, sets support the functions for mathematical sets. For examples, see here.

```
>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                     # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                                 # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                                 # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                                 # letters in both a and b
{'a', 'c'}
>>> a ^ b                                 # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

<https://docs.python.org/3/tutorial/datastructures.html#sets>

# Table of Contents

**1. Data Types vs. Data Structures**

**2. Tuples**

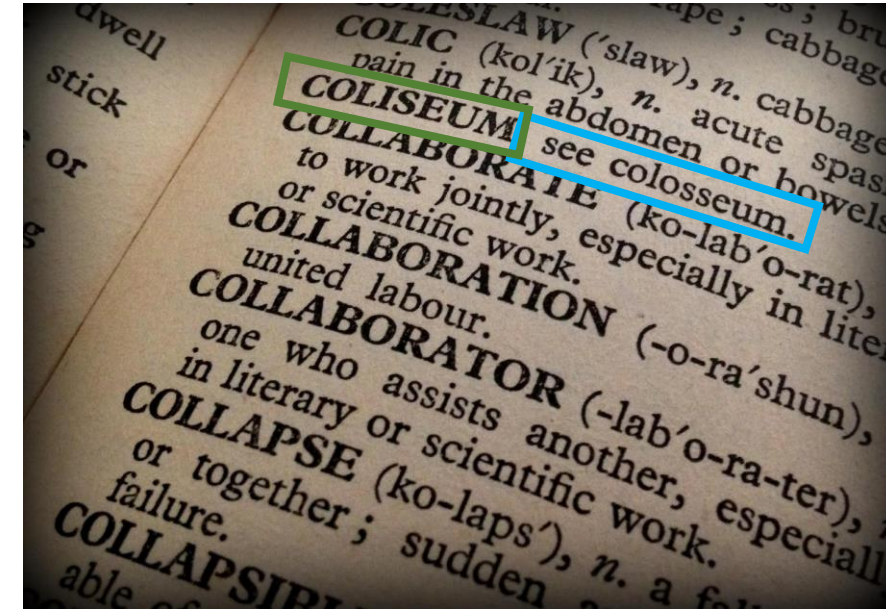
**3. Lists**

**4. Sets**

**5. Dictionaries**

# 5. Dictionaries

- Are built on top of sets
- Dictionaries are sets of key-value pairs, with the first element being the key, the second element being the value: {key : value}
- Keys must be of an immutable type, unique within the dictionary
- In Contrast to lists:
  - The values are unordered
  - Are not indexed via a simple index
  - keys may occur only once → unique
- In Contrast to sets:
  - Indexing is done via key
  - We can retrieve specific key-value pairs or keys\* and values alone.
  - the values of key-value pairs are mutable
- Think of real-world dictionaries!
- dicts may be initialized via the command dict() or {}



\*keys for specific values are not retrievable by a provided function, yet can be retrieved by simple algorithms. See here: <https://stackoverflow.com/questions/8023306/get-key-by-value-in-dictionary>

Abelson, H., Sussman, G. J., & Sussman, J. (1985). Structure and Interpretation of Computer Programs. Cambridge: MIT Press and New York: McGraw-Hill, 1985.

<https://docs.python.org/3/tutorial/datastructures.html#dictionaries>

# 5. Dictionaries – Keys and Values

- Dicts are constructed by assigning a number of key-value pairs in curly brackets {}, separated by commas like this:

```
DNumbers = {"One":1,"Two":2,"Three":3}
print(DNumbers)
# {'One': 1, 'Two': 2, 'Three': 3}
```

- Keys can be of type int, string or float. Let's use the respective int values as keys

```
DNumbers_nKeys = {1:1,2:2,3:3}
print(DNumbers_nKeys)
# {1: 1, 2: 2, 3: 3}
```

- Values can be of any type. Let's create a dict with int keys and string values.

```
DNumbers_sVals = {1:"One",2:"Two",3:"Three"}
print(DNumbers_sVals)
# {1: 'One', 2: 'Two', 3: 'Three'}
```

# 5. Dictionaries – Get and Change Values

- Values can be retrieved easily via commands.
- Retrieve the values from `DNumbers_sVals`, where the key is 3 and 1 like this: `Dvar[3]`

```
DNumbers_sVals[3]  
DNumbers_sVals[1]
```

- Since you know how to access these values, you can manipulate them.
- Change the value of the key-value pair with the key 3 to „I made this“

```
DNumbers_sVals[3] = "I made this."  
print(DNumbers_sVals[3])  
# I made this.
```

- Another method to only get values from dictionary keys is `dict.get()`. Use this method to get the value of the keys 1 and then 11.

```
DNumbers_sVals.get(1, 'This is the message, if no such key is in the dict.')  
# 'One'  
DNumbers_sVals.get(11, 'This is the message, if no such key is in the dict.')  
# 'This is the message, if no such key is in the dict.'
```

## 5. Dictionaries – Get Error

- Values can be retrieved easily via commands.
- Retrieve the values from `DNumbers_sVals`, where the key is 4 like this: `Dvar[4]`

```
DNumbers_sVals[4]
```

- What just happened? We received this error message:

```
Traceback (most recent call last):
  File "<ipython-input-31-bcbf0f01b928>", line 1, in <module>
    print(DNumbers_sVals[4])
KeyError: 4
```

- We got this error because we asked for a key(-value-pair) that is not existent within the dictionary.
- Be aware of this error when handling dictionaries.

## 5. Dictionaries – Data Structures as Values

- Values of dicts can be literally any type, they can even be data structures like lists or dicts.
- Create a `DSomeDicts` dict that holds `DNumbers` and `DNumbers_nKeys`. The keys should be strings containing the names of the two.

```
DSomeDicts = {"DNumbers":DNumbers, "DNumbers_nKeys": DNumbers_nKeys}
print(DSomeDicts)
# {'DNumbers': {'One': 1, 'Two': 2, 'Three': 3}, 'DNumbers_nKeys': {1: 1, 2: 2, 3: 3}}
```

- If we try to use the indexing method with brackets, or the `get` method, we can retrieve the data structure that is part of our dictionary. Get the data structure `DNumbers`.

```
DSomeDicts["DNumbers"]
DSomeDicts.get("DNumbers")
# {'One': 1, 'Two': 2, 'Three': 3}
```

- Change the value of the key `DNumbers` to this list: `[1,2,3]`. Then print `DSomeDicts`.

```
DSomeDicts["DNumbers"] = [1,2,3]
print(DSomeDicts)
# {'DNumbers': [1, 2, 3], 'DNumbers_nKeys': {1: 1, 2: 2, 3: 3}}
```

## 5. Dictionaries – Add and Remove K-V Pairs

- Similar to changing key-value pairs by assigning a new value to a key, you can add elements to a dict with ease.
- Use `dict[key] = value` to create a new value in `DNumbers`. Use „SomeInt“ as key and 1337 as a value.

```
DNumbers["SomeInt"] = 1337
print(DNumbers)
# {'One': 1, 'Two': 2, 'Three': 3, 'SomeInt': 1337}
```

- You can also remove elements from the dictionary by removing a key.
- Use the `dict.pop(<key>)` method to remove the key-value pair with the key „SomeInt“ from `DNumbers`.

```
DNumbers.pop("SomeInt")
print(DNumbers)
# {'One': 1, 'Two': 2, 'Three': 3}
```

- There are many more functions for dictionaries, such as merging two dicts!
- Check these out to get the best out of your dictionary usage: <https://docs.python.org/3/library/stdtypes.html#dict>



# Wrap-Up

Data Structures are used for Data Organization, Management, Storage and efficient manipulation of data with special operations that are provided by them.

We have learned about 4 data structures:

- Tuples: Can be used to assign multiple values, or retrieve multiple values
- Lists: Ordered, indexable, flexible; provides a lot of functionality and will be of use in many situations.
- Sets: Unordered, operations on the set are similar to those of mathematical sets.
- Dicts: Unordered. Data is stored in key-value (K-V) pairs. K-V may be easily retrieved and manipulated.  
Flexible data structure, that will be equally important to the list.

	Mutable	Ordered	Indexing	Duplicate Elements
Tuple		X	X	X
List	X	X	X	X
Set	X			
Dictionaries	X		X	

# References

Abelson, H., Sussman, G. J., & Sussman, J. (1985).

Structure and Interpretation of Computer Programs. *Cambridge: MIT Press and New York: McGraw-Hill, 1985.*

Dijkstra, E. W. Why numbering should start at zero, EWD 831, EW Dijkstra Archive, University of Texas at Austin, 1982

Ernesti, J.; Kaiser, P.: Python 3 – Das umfassende Handbuch. 4. 2015.

<https://docs.python.org/3/tutorial/>

<https://docs.python.org/3/library/stdtypes.html>