Python - Data Analysis Essentials

Day 1

Giuseppe Accaputo

g@accaputo.ch

It's nice to have you here today

About You

- Your major / occupation
- Your programming experience
- Your goals for this course

About me

- Work
  - Software Engineer, Nexiot AG (since April 2018)
  - Research Assistant, ETH Zürich (2017 – 2018)
  - Teaching Assistant and Course Instructor, ETH Zürich (2014 – 2017)
  - Private Tutor, freelance (2016 – 2018)
  - Software Engineer, LTV Gelbe Seiten AG (2009 – 2011)

- Education
  - B.Sc. & M.Sc. ETH in Computational Science and Engineering (2011 – 2017)
  - B.Sc. FH in Computer Science, with a specialization in Software Engineering (2006 – 2009)
  - Vocational education as Systems Engineer (2002 – 2006)

Course Structure

- Course consists of a theoretical part (morning) and a practical part (afternoon)
- Trying out a new format based on experiences and feedback from my past courses
  - Attention is better in the morning
  - More time for practical exercises in the afternoon

Please Feel Free to Always Ask Questions

- Questions are a natural part of the learning process and you're always allowed to ask them
- **Asking questions is an integral part of this course**
- Even if you have a feeling that you're question might "not be good enough," or you don't understand a concept "even if it should be easy to do so," please ask the question nonetheless
  - For one, it gives me the possibility to try and come up with better / clearer explanations
- In case you have any questions after the course, please feel free to contact me at any time via mail at g@accaputo.ch

Learning By Doing (and Making Errors)

- **Programming is best learned by doing**
- Don't be afraid to try stuff out in Python and make errors
  - Errors are a vital part of the learning process and help you understand situations much better

- If you should get stuck on an error during a programming exercise, please feel always free to always call for my help or the help of the fellow students
- Also, don't be afraid to use pen and paper to solve the exercises or when you are trying to understand a specific concept
  - For one, it helps a lot to step away from the computer from time to time
  - It also helps a lot to write down the immediate steps when trying to understand a complicated concept

Feedback

- This is the very first installment of this course
- I'm very thankful for all the feedback I get (be it positive or negative), since I want you to feel comfortable and I love to improve my courses and my teaching skills
  - Course is moving too fast?
  - I'm not speaking clearly enough?
  - Please feel free to inform me about anything whenever you feel like it ☺

Entering the Building

– In case you can't enter the building, you can call me directly on the phone in this room by using the following phone number: **+41446356776**

Course Outline

1. A Short Python Primer
2. Data Structures (Lists, Tuples, Dictionaries, Sets)
3. Working with Files
4. Working with NumPy (Array Creation, Indexing, Slicing, and More)
5. Working with Pandas (Aggregating Data and More)
6. Parsing Data Files with NumPy and Pandas

Learning Objectives for This Course

– The main goal is to get a better picture on the essential Python methods and libraries for preparing, cleaning, transforming and aggregating your data for analysis

Course Outline for Today

1. A Short Python Primer
2. Data Structures (Lists, Tuples, Dictionaries, Sets)
3. Working with Files
4. Working with NumPy (Array Creation, Indexing, Slicing, and More)
5. Working with Pandas (Aggregating Data and More)
6. Parsing Data Files with NumPy and Pandas

A Short Python Primer

Learning Objectives

- You know what code, interpreter, and code execution means
- You have a better picture over the basics of the Python programming language
- You can create a Jupyter notebook to run Python code

Python, the Programming Language

- Goal: we want be able to give the computer instructions to do specific things, e.g. reading a file, computing the sum between two numbers, and so on

- Python is a *formal language* which we humans can read, type, and use to formulate instructions for the computer
  - "Formal language" means that there exists a specific set of rules we have to follow when writing code with it

- The Python *interpreter* then translates our code to *machine code*, which can be directly executed by our computer
  - The interpreter is the interface between a human and a computer

Simplified Life Cycle of a Python Program

Why Python?

Python Code Is Often Quite Readable

– Idea for a program:

IDEE

1. Number 1 has value 2
2. Number 2 has value 10
3. Number 3 has value 18.3
4. Compute Number 1 * Number 2 + Number 3
5. Print the result

– Corresponding Python code:

CODE

```
number_1 = 2
number_2 = 10
number_3 = 18.3
result = number_1  * number_2 + number_3
print(result)
```

«Hello, World!» With Python

CODE

```
print('Hallo Welt!')
```

«Hello, World!» With Java

CODE

```
public class HelloWorld{
  public static void main(String args[]){
    System.out.println('Hello, World!');
  }
}
```

«Hello, World!» With C++

CODE

```
#include <iostream>
int main(){
  std::cout << 'Hello, World!' << std::endl;
  return 0;
}
```

Python Code Is Portable

– Python code can be interpreted and run / executed using any current operating system, e.g. Windows, OS X, and Linux

The Python Ecosystem Is Huge

- Python already comes with a lot of useful tools and libraries
- Nonetheless, there also exist thousands of third-party modules and libraries which can be used to accomplish various tasks, NumPy and Pandas being just two of them
  - https://awesome-python.com/

An Introduction to IPython and Jupyter

IPython: Interactive Python

- Interactive computing in Python
- Offers introspection: We can inspect values and errors, time our functions, and more
- Offers tab completion and history
- Offers a browser-based notebook interface with support for code, text, mathematical expressions and more (it's called *Jupyter* nowadays)
  - A notebook runs Python / IPython statements

IPython: Interactive Python

- We are going to run all the code in this course with IPython
- IPython supports Python 3.*

Help and Documentation in IPython

- How do I call a function? What arguments and options does It have?
- What does the source code of this Python value / object look like?
- What is in this package I imported?
- What variables / attributes or methods does this value / object have?

Help and Documentation in IPython

– We can access documentation with `?`

IPYTHON

```
In [1]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
```

– This notation works for about anything, including object methods and functions (as we will see later)

Help and Documentation in IPython

– We can access source code with `??`

IPYTHON

```
In [1]: def myfun(lst):
   ...:     for e in lst:
   ...:         print(e)
   ...:

In [2]: myfun??
Signature: myfun(lst)
Docstring: <no docstring>
Source:
def myfun(lst):
    for e in lst:
        print(e)
File:      ~/<ipython-input-9-42be41fecbd8>
Type:      function
```

Shell Commands in IPython

- The shell is a way to interact textually with your computer
  - Operating systems existed long before graphical user interfaces as we know and use today
- We can create folders, files, copy and delete them, and more with a shell
  - Basically, we can submit a lot of commands via shell to the computer

Shell Commands in IPython

- Common shell commands
  - `pwd`: Print the working directory (where we currently are in the file system)
  - `ls`: List working directory contents
  - `cd`: Change directory
  - `mkdir`: Make new directory
- In IPython we can use these shell commands by prefixing them with `!`

Running External Code with %run

– We can use a text editor to write code and use IPython to run it with %run

print.py

```
def fun(lst):
  for e in lst:
    print(e)

fun([1,2,3,4])
```

IPYTHON

```
In [1]: %run print.py
1
2
3
4
```

Some Important Basics of the Python Programming Language
(At Least for This Course)

Values and Data Types

- *Values* are fundamental things like the number `2` or `1.234`, or the string `Hello`
- A *data type* is a category for values, and a value always belongs to a single data type
  - Integer data type: `-1, -100, 0, 12, 34`
  - Float data type: `-1.324, 0.14123, 10.1, 100.0`
  - String data type: `'Hello', 'Word', 'Spaces are included'`

Storing Values in Variables

- A *variable* is like a box where you can store a single value
- Assigning a value to a variable is done with an *assignment statement*:

CODE
```
myNumber = 123
```

  - `myNumber` is the variable name, and `123` is the value stored within this variable
- Since a variable stores a value, a variable also belongs to a data type, which we can query with the `type` function:

CODE
```
type(myNumber)
```

Statements, Expressions, and Operators

- A *statement* is an instruction that the Python interpreter can execute
- An *expression* is a combination of values, variables, operators, and calls to functions
  - Expressions need to be *evaluated*
  - The evaluation of an expression always produces a single value
- An operator is a special token that represents a computation like an addition, multiplication, and division
  - Values that the operator works on are called *operands*

- Lets see what exactly it means to *evaluate* an expression **{Live Coding}**

Functions

CODE

```
def hello():
  print('Hello World')

hello()
```

- A *function* is defined by using the def keyword
- The code in the block that follows the def statement is called the *function body*
  - This code is only executed when the function gets called, not when it's first defined
- The `hello()` after the function definition is a *function call*
  - A function call is just a functions name followed by parentheses, possibly with some arguments in between the parentheses

Functions

- We can define functions that take in *arguments*, which are typed between the parentheses
  - For example, the `print()` function takes an argument, namely the string we want to have printed on the screen

CODE

```
def hello(name):
  print('Hello, ' + name)

hello('Giuseppe')
```

Functions

- Functions can evaluate to a value, which is called the *return value* of the function
  - For example, if we pass the argument `'Hello'` to the `len()` function, it will evaluate to the integer value `5`, which is the length of the string we passed
- We can specify what a function should return by using the return statement followed by the value we want to return:

CODE

```
def sqr(x):
  return x*x

sqr_of_two = sqr(2)
print(str(sqr_of_two))
```

- *Note*: Functions without return value always evaluate to `None`

The Boolean Data Type

- Boolean data type: `True`, `False` (only two possible values)
- *Comparison operators* compare two values and evaluate to a single Boolean value
  - The comparison operators are `==`, `!=`, `<`, `>`, `<=`, and `>=`
- *Boolean operators* are used to compare Boolean values
  - The Boolean operators are `or`, `and`, and `not`
- We can mix Boolean and comparison operators to create *conditions*

- Lets see the Boolean and comparison operators in action **{Live Coding}**

Flow Control Statements

- `if` Statement: Its clause (the block following the if statement) will only execute if the statement's condition is `True`
  - "If this condition is true, execute the code"
- `else` Statement: An `if` statement can optionally be followed by an `else` statement. The `else` clause is executed only when the `if` statement's condition is `False`
- `elif` Statement: Is an "else if" statement that always follows an `if` or another `elif` statement.

- Lets see flow control statements in action

{Live Coding}

Learning Objectives

- You know what code, interpreter, and code execution means
- You have a better picture over the basics of the Python programming language
- You can create a Jupyter notebook to run Python code

Data Structures: Lists, Dictionaries, Sets, and Tuples

Lists

Learning Objectives

- You know
  - what a list is
  - how to read and change elements of a list by using an index
  - how to traverse a list with the help of a `for` loop
  - how slicing works

The List: Our First Sequence Type

– A *list* contains multiple values in an ordered sequence and may look like this:

CODE
```
first_list = ["one", "two", 3, 4, 5]
```

– A list starts with an opening square bracket [and ends with a closing square bracket]

CODE
```
empty_list = []
```

– Values inside a list are called *items* and are separated with commas (they are *comma-delimited*)
  – The items in a list can be of different or equal types

The List: Our First Sequence Type

- A list is also a *value*, i.e. it can be stored in a variable or passed to a function like any other value
- Just like other values (e.g. `1.0` or `"abc"`), a list also has a *type*

CODE

```
list_1 = ["ab", "cd"]
list_2 = [1, 2, 3, 4, 5, 6]
list_3 = [1, 2, "three"]

print(str(type(list_1)))
print(str(type(list_2)))
print(str(type(list_3)))
```

INTERPRETER

OUTPUT

```
<class 'list'>
<class 'list'>
<class 'list'>
```

Getting Individual Values in a List with Indexes

– Lets say we have the following list stored in a variable:

CODE
```
numbers = ["one", "two", "three", "four"]
```

– `numbers[0]` would evaluate to `"one"`
– `numbers[1]` would evaluate to `"two"`, and so on
– The integer inside the square brackets is called an *index*
  – The first value in the list is at index `0`
  – The second value is at index `1`
  – The third value is at index `2`, and so on

Getting Individual Values in a List with Indexes

– If you use an index that exceeds the number of values in your list, Python will give you an error:

CODE

```
list_1 = ["ab", "cd"]
list_1[100000]
```

INTERPRETER

OUTPUT

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Getting Individual Values in a List with Indexes

– Indexes can be only integer values, so floats are not allowed:

CODE

```
list_1 = ["ab", "cd"]
list_1[1.5]
```

INTERPRETER

OUTPUT

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be
integers or slices, not float
```

Structure of a List: Values and Indices

Index:	0	1	2	3
Element:	"one"	"two"	"three"	"four"
Type:	`str`	`str`	`str`	`str`

Negative Indexes

- Negative indexes are allowed
- `-1` refers to the last element
- `-2` refers to the second-to-last element in a list, and so on

CODE

```
n = ["one", "two", "three", "four"]
print(n[-1])
print(n[-2])
```

INTERPRETER

OUTPUT

```
four
three
```

Negative Indexes

Index:	-4	-3	-2	-1
Element:	"one"	"two"	"three"	"four"
Type:	`str`	`str`	`str`	`str`

Getting Slices of a List

- An *index* can get a *single value* from a list
- A *slice* can get *several values* from a list

- `numbers[1]` is a list with an index
- `numbers[0:2]` is a list with a slice
  - First integer is the index where the slice starts
  - Second integer is the index where the slice ends
  - *Note*: The slice goes up to, but will not include, the value at the second index

Getting Slices of a List

– A slice will evaluate to a list:

CODE

```
n = ["one", "two", "three", "four"]
slice = n[0:2]

print(slice)
print(str(type(slice)))
```

INTERPRETER

OUTPUT

```
['one', 'two']
<class 'list'>
```

Getting Slices of a List

- Leaving out the first index is the same as using 0 (beginning of the list)
- Leaving out the second index is the same as using the length of the list (will slice to the end of the list)

CODE

```
n = ["one", "two", "three", "four"]
s_1 = n[:2]
s_2 = n[1:]

print(s_1)
print(s_2)
```

INTERPRETER

OUTPUT

```
['one', 'two']
['two', 'three', 'four']
```

A List's Length: Counting the Number of Elements in a List

– The `len()` function will return the number of values in a list passed to it

CODE

```
n = ["one", "two", "three", "four"]
print(len(n))
```

INTERPRETER

OUTPUT

```
4
```

Changing Values in a List

– We can use an index of a list to change the value at that index, e.g. `n[0] = "uno"` means "Set the value at index `0` in the list `n` to the string `"uno"`":

CODE

```
n = ["one", "two", "three", "four"]

n[0] = "uno"
n[-1] = "quattro"

print(n)
```

INTERPRETER

OUTPUT

```
['uno', 'two', 'three', 'quattro']
```

List Concatenation and Replication

- *Concatenation*: We can combine two lists into a new list by using the + operator
- *Replication*: The * operator with an integer value can be used to replicate a list

CODE

```
l1 = [1,2,3]
l2 = ["A", "B", "C"]
l3 = ["a", "b"]

print(l1 + l2)
print(3 * l3)
```

INTERPRETER

OUTPUT

```
[1, 2, 3, 'A', 'B', 'C']
['a', 'b', 'a', 'b', 'a', 'b']
```

Removing Elements from Lists

- We can use the `del` statement to delete values at an index in a list
- `del n[0]` will delete the value at index `0` in the list `n`
- `del n[1]` will delete the value at index `1` in the list `n`, and so on
- *Important*: Deleting an element will move all the elements coming after it to the left

CODE

```
n = ["one", "two", "three", "four"]

del n[0]
del n[1] # Watch out!

print(n)
```

INTERPRETER

OUTPUT

```
['two', 'four']
```

Iterating over a List Using a `for` Loop

– The following `for` loop loops through its clause with the variable `i` set to a successive value in the `[0, 1, 2, 3]` list in each iteration (4 iterations in total):

CODE

```
for i in [0, 1, 2, 3]:
  print(i)
```

INTERPRETER

OUTPUT

```
0
1
2
3
```

Iterating over a List Using a `for` Loop

– Another `for` loop:

CODE

```
letters = ["a", "b", "c", "d"]

for letter in letters:
  print(letter)
```

INTERPRETER

OUTPUT

```
a
b
c
d
```

Question • Understanding `for` Loops [Exercise]

– What gets printed on the screen after the following program has been executed?

CODE

```
sum = 0

for num in [0,1,2,3,4,5]:
  sum = sum + num

print(sum)
```

Check If a Value Exists in a List

– We can use the `in` and `not in` operators to determine whether a value is or isn't in a list

CODE

```
numbers = [1,2,3,4]
c1 = 1 in numbers
c2 = 23 not in numbers

print(c1)
print(c2)
```

INTERPRETER

OUTPUT

```
True
True
```

Learning Objectives

- You know
  - what a list is
  - how to read and change elements of a list by using an index
  - how to traverse a list with the help of a `for` loop
  - how slicing works

Methods

Learning Objectives

- You know
  - how to call a method
  - how to use tab-completion in IPython to help you with methods
  - that different data types may provide different methods

Methods

- A *method* is the same thing as a function, except it is called on a value
  - Function call: `my_fun(a,b,c)`
  - Method call: `my_list.index("k")`
    - We called the `index` method on the value of `my_list`, which is of type `list`
- Each data type (`str`, `list`, `dict`, etc.) has its own set of methods
  - The `list` data type has several useful methods for finding, adding, removing, and manipulating values in a list
- A method always acts on the value it has been called on
  - `list1.index("k")` → `index("k")` acts on the value of `list1`
  - `list2.index("e")` → `index("e")` acts on the value of `list2`

Help on Methods in IPython

– We can check the documentation for specific methods with ? in IPython

IPYTHON

```
In [1]: lst = [1,2,3]
In [2]: lst.index?
Docstring:
L.index(value, [start, [stop]]) -> integer -- return first index of value.
Raises ValueError if the value is not present.
```

– IPython also provides tab-completion, meaning it will show all available methods for a specific value

- Lets check out the tab-completion in IPython

{Live Coding}

Finding a value in a List: The `index()` Method

– The list data type provides an `index()` method, to which we can pass a value. If that value exists in the list, the index of the value is returned, else Python produces a `ValueError` error

CODE

```
n = ["one", "two", "three", "four"]

ind1 = n.index("two")
print("Index of 'two': " + str(ind1))

ind2 = n.index("five")
```

INTERPRETER

OUTPUT

```
Index of 'two': 1

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'five' is not in list
```

Adding Values to a List: The `append()` and `insert()` Methods

- We can add new values to a list by calling the `append()` and `insert()` methods
- The `append()` method call adds the argument to the end of the list
- The `insert()` method call requires two arguments: the first argument is the index for the new value, and the second argument is the new value to be inserted

Adding Values to a List: The `append()` and `insert()` Methods

CODE

```
alpha = ["a", "b", "c"]

alpha.append("d")

print(alpha)
```

INTERPRETER

OUTPUT

```
['a', 'b', 'c', 'd']
```

Adding Values to a List: The `append()` and `insert()` Methods

– Lets append a new value at the end of a list:

CODE

```
alpha = ["a", "b", "c"]

alpha.append("d")

print(alpha)
```

INTERPRETER

OUTPUT

```
['a', 'b', 'c', 'd']
```

Adding Values to a List: The `append()` and `insert()` Methods

– Lets add a new element at index `1` of the list:

CODE

```
alpha = ["a", "b", "c"]

alpha.insert(1, "w")

print(alpha)
```

INTERPRETER

OUTPUT

```
['a', 'w', 'b', 'c']
```

– *Note*: After adding the new element, all previously existing elements at index 1, 2, and above are moved to the right. This can be a costly operation if we insert elements in very large lists like this

Adding Values to a List: The `append()` and `insert()` Methods

- Note: It's not `alpha = alpha.append("d")` or `alpha = alpha.insert(1, "w")`
  - Both functions do not return the modified list `alpha` (both calls evaluate to `None`)
  - The list `alpha` is rather modified *in place* (a list is a *mutable* data type)

Different Methods for Different Data Types

- Methods belong to a single data type
  - `append()` and `insert()` are list methods and can be called only on lists, not on other values such as strings and integers

CODE

```
num = 1023

num.insert(1, "w")
```

INTERPRETER

OUTPUT

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError:
'int' object has no attribute 'insert'
```

Removing Values from Lists: The `remove()` Method

– We can pass a value we want to be removed to the `remove()` method of a specific list:

CODE

```
alpha = ["a", "b", "c"]

alpha.remove("a")

print(alpha)
```

INTERPRETER

OUTPUT

```
['c', 'd']
```

– Note: If you know the index of the value we want to remove, we can still use the `del` operator for the removal; if you know the value, just use the `remove()` method

Sorting the Values in a List: The `sort()` Method

– We can sort lists of strings or numbers by calling the sort() method on a specific list:

CODE

```
alpha = ["c", "a", "b"]
alpha.sort()
print(alpha)

num = [3.14, 10, 1, -23, 0.4]
num.sort()
print(num)
```

INTERPRETER

OUTPUT

```
['a', 'b', 'c']
[-23, 0.4, 1, 3.14, 10]
```

Sorting the Values in a List: The `sort()` Method

– We can also pass True for the reverse keyword argument to the sort() method; this will sort the values in reverse order:

CODE

```
alpha = ["c", "a", "b"]
alpha.sort(reverse=True)
print(alpha)

num = [3.14, 10, 1, -23, 0.4]
num.sort(reverse=True)
print(num)
```

INTERPRETER

OUTPUT

```
['c', 'b', 'a']
[10, 3.14, 1, 0.4, -23]
```

Learning Objectives

- You know
  - how to call a method
  - how to use tab-completion in IPython to help you with methods
  - that different data types may provide different methods

Tupels

Learning Objectives

- You know
  - what a tuple is
  - the difference between an immutable and mutable data type

Mutable and Immutable Data Types

– A list is a *mutable* data type: We can add, remove, or change values in a list

CODE

```
alpha = ["c", "a", "b"]
alpha[1] = "d"
print(alpha)
```

INT.

OUTPUT

```
['c', 'd', 'a']
```

– A string instead is an *immutable* data type: It cannot be changed

CODE

```
hw = "Hello, World!"
hw[7] = "h"
print(hw)
```

INTERPRETER

OUTPUT

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not
support item assignment
```

Mutable and Immutable Data Types

- Changing a value of a mutable data type changes the value in place, meaning that the value changes
- We can use immutable data types to ensure that a value doesn't get changed
  - For example, we can ensure that a function doesn't change the content of the value

The Tuple Data Type

- The *tuple* data type is almost identical to the list data type, except:
  - Tuples are typed with parentheses, (and)
  - Tuples are immutable

CODE

```
data = ("c", 1.23, "def")
print(data[0:2])
```

INT.

OUTPUT

```
('c', 1.23)
```

CODE

```
data = ("c", 1.23, "def")
data[0] = "a"
```

INTER.

OUTPUT

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not
support item assignment
```

Converting Lists and Tuples

– The `list()` and `tuple()` functions will return list and tuple versions of the values passed to them:

CODE

```
data = ("c", 1.23, "def")
data_mutable = list(data)
print(str(type(data_mutable)))

data_mutable[0] = "a"
print(data_mutable)
print(data)
```

INTERPRETER

OUTPUT

```
<class 'list'>
['a', 1.23, 'def']
('c', 1.23, 'def')
```

– Note: the tuple that we passed to the `list()` function remains still unchanged; the `list()` function copies the elements of the tuple into a new list

Learning Objectives

- You know
  - what a tuple is
  - the difference between an immutable and mutable data type

Dictionaries

Learning Objectives

- You know
  - what a dictionary, a key, and value are
  - how to read and change elements of a dictionary by using a key
  - how to traverse the keys, values, or key-value pairs of a dictionary with the help of a `for` loop

The Dictionary Data Type

- Dictionaries are typed with braces, `{` and `}`

CODE
```
menu = {"Burger": 10, "Pommes": 6, "Schnitzel": 12}
```

- In the code above:
  - We assign a dictionary to the variable menu
  - The dictionary's keys are `"Burger"`, `"Pommes"`, and `"Schnitzel"`
  - The values for these keys are `10`, `16`, and `12`, respectively

- An empty dictionary can be created by using `{}`

CODE
```
empty_set = {}
```

Accessing Elements in a Dictionary

– We can access the values by using their keys:

CODE

```
menu = {"Burger": 10, "Fries": 6, "Schnitzel": 12}
preis_burger = menu["Burger"]
print("A burger costs " + str(preis_burger) + " CHF")
```

INTERPRETER

OUTPUT

```
A burger costs 10 CHF
```

Dictionaries vs. Lists

- Lists are ordered
  - First item in a list is located at the index 0
  - We can slice lists
  - Trying to access an index that is out of range results in an error message
- Dictionaries are unordered
  - There is no "first" item, since we can only access items using keys
  - We cannot slice dictionaries
  - Trying to access a key that does not exist results in an error message

Dictionaries vs. Lists

– Lists are ordered; the order of the elements matters:

```
l1 = [1,2,3,4]
l2 = [2,1,3,4]

print(l1 == l2)
```

CODE → INTERP. → OUTPUT

```
False
```

– Dictionaries are unordered; the order of the elements does not matter:

```
d1 = {"a":13, "b":14}
d2 = {"b":14, "a":13}

print(d1 == d2)
```

CODE → INTERP. → OUTPUT

```
True
```

Accessing the Keys, Values, and Key-Value Pairs of a Dictionary

- The `keys()`, `values()`, and `items()` methods return the dictionary's keys, values, and key-value pairs respectively
  - The values returned by these methods are not real lists, but we can still use a `for` loop to traverse them

CODE

```
d1 = {"a":13, "b":14, "c": 100}

for k in d1.keys():
  print(k)
```

INTERP.

OUTPUT

```
a
b
c
```

CODE

```
d1 = {"a":13, "b":14, "c": 100}

for v in d1.values():
  print(v)
```

INTERP.

OUTPUT

```
13
100
14
```

Accessing the Keys, Values, and Key-Value Pairs of a Dictionary

– The values returned by the `items()` method are tuples of the key and value:

CODE

```
d1 = {"a":13, "b":14, "c": 100}

for i in d1.items():
  print(i)
```

INTERP.

OUTPUT

```
('a', 13)
('c', 100)
('b', 14)
```

– We can use a multiple assignment in a `for` loop to assign the key and value to separate variables:

CODE

```
d1 = {"a":13, "b":14, "c": 100}

for k,v in d1.items():
  print("Key: " + k + " | Val: " + str(v))
```

INTERP.

OUTPUT

```
Key: a | Val: 13
Key: c | Val: 100
Key: b | Val: 14
```

Checking If a Key or Value Exists in a Dictionary

– We can use the `in` and `not in` operators to determine whether a certain key or value exists in a dictionary:

CODE

```
d1 = {"a":13, "b":14, "c": 100}

print("a" in d1.keys())
print(120 in d1.values())
```

INTERP.

OUTPUT

```
True
False
```

Trying to Get A Value: The `get()` Method

- If we try to access a key's value with a key that does not exists, an error will be generated
- Dictionaries have a `get()` method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist:

CODE

```
d1 = {"a":13, "b":14, "c": 100}

print(d1.get("k", 999))
print(d1.get("b", 999))
```

INTERP.

OUTPUT

```
999 ← Fallback value
14
```

Learning Objectives

- You know
  - what a dictionary, a key, and value are
  - how to read and change elements of a dictionary by using a key
  - how to traverse the keys, values, or key-value pairs of a dictionary with the help of a `for` loop

Universität Zürich[UZH]

IT Training and Continuing Education

Sets

Learning Objectives

- You know
  - the basic properties of a set (unordered and contains no duplicates)
  - what union, intersection, and complement mean

The Set Data Type

- A *set* is an unordered collection with no duplicate elements
- Sets are typed with braces, `{` and `}`, just like dictionaries

CODE
```
guests = {"Marco", "Javier", "Giuseppe"}
```

- *Important*: An empty set is created by using the `set()` function; `{}` will create an empty dictionary

CODE
```
empty_set = set()
```

Iterating Over the Elements of a Set

– We can again use the for loop to iterate over all elements in a set:

CODE

```
s = {1,2,3,4}

for i in s:
  print(i)
```

INTERPRETER

OUTPUT

```
1
2
3
4
```

Removing Duplicates

– We can use the set data type to remove duplicates:

CODE

```
lst = [1,2,3,3,4,5,6,3,3,5,5,5,6]
s = set(lst)

print(s)
```

INTERP.

OUTPUT

```
{1, 2, 3, 4, 5, 6}
```

Mathematical Operations on Sets: The Union Operator |

- In the following slides on mathematical operations, A and B are both sets
- Union operator |: Returns a set which includes all elements in A *and* B

CODE

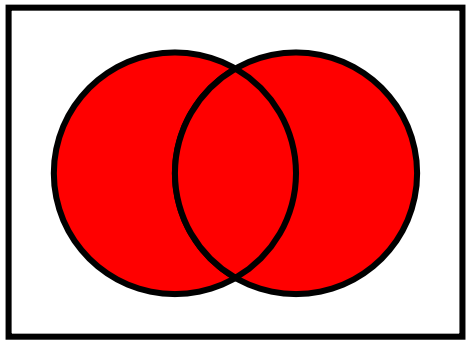
```
A = {1, 2, 3}
B = {1, 2, 3, 4, 5, 6}

print(A | B)
```

INTERP.

OUTPUT

```
{1, 2, 3, 4, 5, 6}
```



Source: Wikipedia

Mathematical Operations on Sets: The Intersection Operator &

– Intersection operator &: Returns a set which includes *only* elements that are both in *A* and *B*

CODE

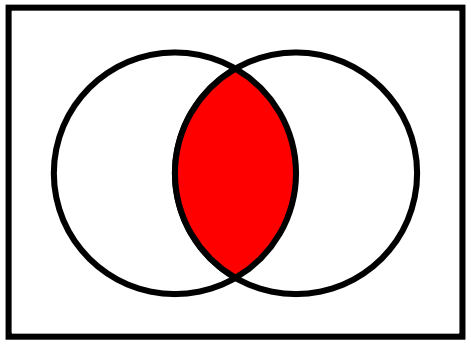
```
A = {1, 2, 3}
B = {1, 2, 3, 4, 5, 6}

print(A & B)
```

INTERP.

OUTPUT

```
{1, 2, 3}
```



Source: Wikipedia

Mathematical Operations on Sets: The Difference Operator -

– Difference operator `-`: Returns a set which includes elements that are *only in* A *but not in* B (basically, remove all elements that are in B from A)

CODE

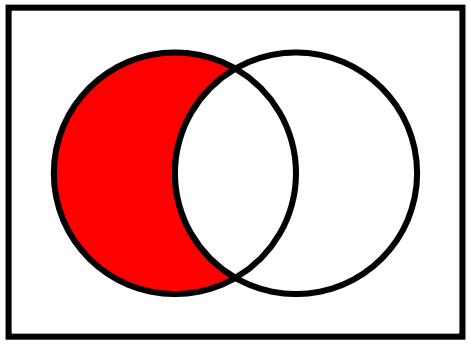
```
A = {1, 2, 3}
B = {1, 2, 3, 4, 5, 6}

print(A - B)
```

INTERP.

OUTPUT

```
set() ← Empty set
```



Source: Wikipedia

Mathematical Operations on Sets: Symmetric Difference Operator ^

– Symmetric difference operator ^: Returns a set which includes elements that are *either in* A *or* B*, but not in both*

CODE

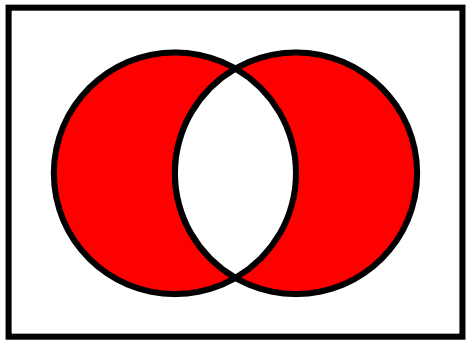
```
A = {1, 2, 3, 7, 8}
B = {1, 2, 3, 4, 5, 6}

print(A ^ B)
```

INTERP.

OUTPUT

```
{4, 5, 6, 7, 8}
```



Source: Wikipedia

Checking If a Set Is Contained Within Another Set

– We can use the operator `<` to check if a set A is completely contained within another set B

CODE

```
A = {1, 2, 3}
B = {1, 2, 3, 4, 5, 6}

print(A < B)
```

INTERP.

OUTPUT

```
True
```

– We can use the subset operator `<=` to additionally check if A is the same as B,
i.e. `<=` returns `True` if A is completely contained within B, or both contain the exact same elements

Learning Objectives

- You know
  - the basic properties of a set (unordered and contains no duplicates)
  - what union, intersection, and complement mean

Questions

– If you have any questions, information, or more about any topic of today's course, feel free to write me at g@accaputo.ch

Next Week

– We will meet again next Saturday, at 08:50 AM in front of the building Y10