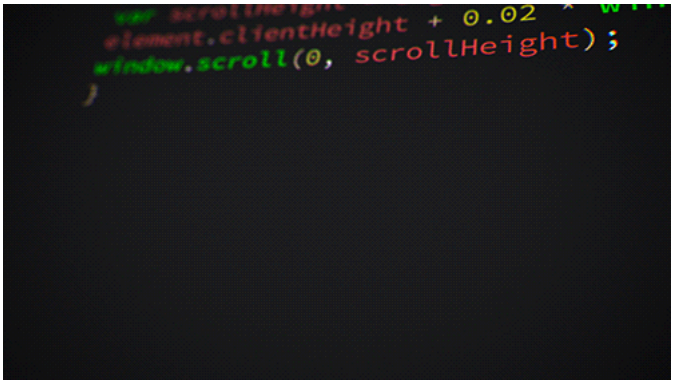
Objektorientierung / Visualisierung mit Python

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik`
Semester	`Wintersemester 2022/23`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Visualisierung mit Python`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/10_Datenvisualisierung.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf



Fragen an die heutige Veranstaltung ...

- Wie lassen sich die Konzepte der OOP in Python ausdrücken?
- Welche spezifischen Einschränkungen gibt es dabei?
- Welche Grundkonzepte stehen hinter der Programmierung von Grafiken?
- Wie geht man bei der Erschließung von unbekannten Methoden sinnvoll vor?

Organisatorisches

- Wer von Ihnen ist Hörerinnen und Hörer der Vorlesung *Einführung in die Informatik*?
- Bitte bringen Sie sofern möglich Ihre Notebooks zu den Übungen mit. Installieren Sie darauf bereits Python mittels Anaconda.
- In der letzten Übung wird eine Zusammenfassung der behandelten Inhalte angeboten. Dabei wird insbesondere auf die Objektorientierung unter C++ eingegangen.

Objektorientierung in Python

> Klassen werden verwendet, um benutzerdefinierte Datenstrukturen zu erstellen und definieren Funktionen, sogenannte Methoden, die das Verhalten und die Aktionen identifizieren, die ein aus der Klasse erstelltes Objekt mit seinen Daten ausführen kann.

Eine kurze Auffrischung Ihrer Erinnerungen zur objektorientierter Programmierung in C++ ...

Comparison.cpp

```cpp
#include <iostream>

class Rectangle {
  private:
    float width, height;
  public:
    Rectangle(int w, int h){
        if ((w > 0) & (h > 0)) {
                this->width = w;
                this->height = h;
        }else{
                this->width = w;
                this->height = h;
        }
    }
    float area() {return width*height;}
    Rectangle operator+=(Rectangle offset) {
        float ratio = (offset.area() + this->area()) / this->area();
        this->width = ratio *  this->width;
        return *this;
    }
};

int main () {
  Rectangle rect_a(3,4);
  Rectangle rect_b(1,3);
  std::cout << "Fläche a : " << rect_a.area() << "\n";
  std::cout << "Fläche b : " << rect_b.area() << "\n";
  rect_a += rect_b;
  std::cout << "Summe    : " << rect_a.area();

  return 0;
}
```

```
Fläche a : 12
Fläche b : 3
Summe    : 15
```

Zeile	Bedeutung
3-22	Definition der Klasse `Rectangle` (Schablone für Daten, Methoden, Operatoren)
5	Gekapselte Daten der Klasse, diese sind "von Außen" nicht sichtbar
7	Konstruktor mit Evaluation der übergebenen Parameter
16	Methode über den Daten der Klasse
17	Individueller Operator `+` mit einer spezifischen Bedeutung
25-28	Generierung von Objekten mittels Konstruktoraufruf und Parameterübergabe

Objektorientierte Programmierung (OOP) ist ein Paradigma, das über die Ideen der Prozeduralen Programmierung hinaus geht. Es definiert Objekte und deren Verhalten. Dabei baut es auf 3 zentralen Grundprinzipien auf:

1. **Kapselung** Objekte kapseln ihre Daten, Operatoren, Methoden usw. sofern diese nicht als "öffentlich" deklariert sind.

> *Was intern passiert bleibt intern!*

2. **Vererbung** Objekte können "Fähigkeiten" an andere, speziellere Objekte weitergeben.

> *Von wem hat er das denn wohl?*

3. **Polymorphismus** Objekte werden durch Kapselung und Vererbung austauschbar!

> *Was bist denn Du für einer?*

Vorteile der objektorientierten Programmierung

- höhere Wartbarkeit durch Abstraktion
- Wiederverwendbarkeit von Code (je mehr desto kleiner und allgemeiner die Objekte gehalten sind)
- schlanker und übersichtlicher Code durch Vererbung

Warum also nicht immer objektorientiert entwickeln?

OOP verführt ggf. dazu, das eigentliche Problem durch eine aufwändigen Entwurf unnötig zu verkomplizieren. Dabei ist die Entwicklung der Gesamtstruktur eines komplexen Softwareprojektes aus n Objekten eine Kunst und braucht viel Übung! Erst, wenn man entsprechende Regeln kennt und sinnvoll anwendet, zeigen sich die Vorteile des Paradigmas.

... und in Python?

> **In Python ist alles ein Objekt!**

```
1  import inspect
2
3  i=5
4
5  for name, data in inspect.getmembers(i):
6      if name == '__builtins__':
7          continue
8      print(f'{name} - {repr(data)}')
```

```
__abs__ - <method-wrapper '__abs__' of int object at 0x7f0b61920170>
__add__ - <method-wrapper '__add__' of int object at 0x7f0b61920170>
__and__ - <method-wrapper '__and__' of int object at 0x7f0b61920170>
__bool__ - <method-wrapper '__bool__' of int object at 0x7f0b61920170>
__ceil__ - <built-in method __ceil__ of int object at 0x7f0b61920170>
__class__ - <class 'int'>
__delattr__ - <method-wrapper '__delattr__' of int object at 0x7f0b61920170>
__dir__ - <built-in method __dir__ of int object at 0x7f0b61920170>
__divmod__ - <method-wrapper '__divmod__' of int object at 0x7f0b61920170>
__doc__ - "int([x]) -> integer\nint(x, base=10) -> integer\n\nConvert a number or string to an integer, or
return 0 if no arguments\nare given.  If x is a number, return x.__int__().  For floating point\nnumbers, this
truncates towards zero.\n\nIf x is not a number or if base is given, then x must be a string,\nbytes, or
bytearray instance representing an integer literal in the\ngiven base.  The literal can be preceded by '+' or '-
' and be surrounded\nby whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.\nBase 0 means to
interpret the base from the string as an integer literal.\n>>> int('0b100', base=0)\n4"
__eq__ - <method-wrapper '__eq__' of int object at 0x7f0b61920170>
__float__ - <method-wrapper '__float__' of int object at 0x7f0b61920170>
__floor__ - <built-in method __floor__ of int object at 0x7f0b61920170>
__floordiv__ - <method-wrapper '__floordiv__' of int object at 0x7f0b61920170>
__format__ - <built-in method __format__ of int object at 0x7f0b61920170>
__ge__ - <method-wrapper '__ge__' of int object at 0x7f0b61920170>
__getattribute__ - <method-wrapper '__getattribute__' of int object at 0x7f0b61920170>
__getnewargs__ - <built-in method __getnewargs__ of int object at 0x7f0b61920170>
__gt__ - <method-wrapper '__gt__' of int object at 0x7f0b61920170>
__hash__ - <method-wrapper '__hash__' of int object at 0x7f0b61920170>
__index__ - <method-wrapper '__index__' of int object at 0x7f0b61920170>
__init__ - <method-wrapper '__init__' of int object at 0x7f0b61920170>
__init_subclass__ - <built-in method __init_subclass__ of type object at 0x562f46b85320>
__int__ - <method-wrapper '__int__' of int object at 0x7f0b61920170>
__invert__ - <method-wrapper '__invert__' of int object at 0x7f0b61920170>
__le__ - <method-wrapper '__le__' of int object at 0x7f0b61920170>
__lshift__ - <method-wrapper '__lshift__' of int object at 0x7f0b61920170>
__lt__ - <method-wrapper '__lt__' of int object at 0x7f0b61920170>
__mod__ - <method-wrapper '__mod__' of int object at 0x7f0b61920170>
__mul__ - <method-wrapper '__mul__' of int object at 0x7f0b61920170>
__ne__ - <method-wrapper '__ne__' of int object at 0x7f0b61920170>
__neg__ - <method-wrapper '__neg__' of int object at 0x7f0b61920170>
__new__ - <built-in method __new__ of type object at 0x562f46b85320>
__or__ - <method-wrapper '__or__' of int object at 0x7f0b61920170>
__pos__ - <method-wrapper '__pos__' of int object at 0x7f0b61920170>
__pow__ - <method-wrapper '__pow__' of int object at 0x7f0b61920170>
__radd__ - <method-wrapper '__radd__' of int object at 0x7f0b61920170>
__rand__ - <method-wrapper '__rand__' of int object at 0x7f0b61920170>
__rdivmod__ - <method-wrapper '__rdivmod__' of int object at 0x7f0b61920170>
__reduce__ - <built-in method __reduce__ of int object at 0x7f0b61920170>
__reduce_ex__ - <built-in method __reduce_ex__ of int object at 0x7f0b61920170>
__repr__ - <method-wrapper '__repr__' of int object at 0x7f0b61920170>
__rfloordiv__ - <method-wrapper '__rfloordiv__' of int object at 0x7f0b61920170>
__rlshift__ - <method-wrapper '__rlshift__' of int object at 0x7f0b61920170>
__rmod__ - <method-wrapper '__rmod__' of int object at 0x7f0b61920170>
__rmul__ - <method-wrapper '__rmul__' of int object at 0x7f0b61920170>
__ror__ - <method-wrapper '__ror__' of int object at 0x7f0b61920170>
__round__ - <built-in method __round__ of int object at 0x7f0b61920170>
__rpow__ - <method-wrapper '__rpow__' of int object at 0x7f0b61920170>
__rrshift__ - <method-wrapper '__rrshift__' of int object at 0x7f0b61920170>
__rshift__ - <method-wrapper '__rshift__' of int object at 0x7f0b61920170>
__rsub__ - <method-wrapper '__rsub__' of int object at 0x7f0b61920170>
__rtruediv__ - <method-wrapper '__rtruediv__' of int object at 0x7f0b61920170>
__rxor__ - <method-wrapper '__rxor__' of int object at 0x7f0b61920170>
__setattr__ - <method-wrapper '__setattr__' of int object at 0x7f0b61920170>
__sizeof__ - <built-in method __sizeof__ of int object at 0x7f0b61920170>
__str__ - <method-wrapper '__str__' of int object at 0x7f0b61920170>
__sub__ - <method-wrapper '__sub__' of int object at 0x7f0b61920170>
__subclasshook__ - <built-in method __subclasshook__ of type object at 0x562f46b85320>
__truediv__ - <method-wrapper '__truediv__' of int object at 0x7f0b61920170>
```

```
__trunc__ - <built-in method __trunc__ of int object at 0x7f0b61920170>
__xor__ - <method-wrapper '__xor__' of int object at 0x7f0b61920170>
as_integer_ratio - <built-in method as_integer_ratio of int object at 0x7f0b61920170>
bit_count - <built-in method bit_count of int object at 0x7f0b61920170>
bit_length - <built-in method bit_length of int object at 0x7f0b61920170>
conjugate - <built-in method conjugate of int object at 0x7f0b61920170>
denominator - 1
from_bytes - <built-in method from_bytes of type object at 0x562f46b85320>
imag - 0
numerator - 5
real - 5
to_bytes - <built-in method to_bytes of int object at 0x7f0b61920170>
```

Klassen in Python

Alle Klassendefinitionen beginnen mit dem Schlüsselwort `class`, gefolgt vom Namen der Klasse und einem Doppelpunkt. Jeder Code, der unterhalb der Klassendefinition eingerückt ist, wird als Teil des Klassenhauptteils betrachtet.

Analog zu C++ nutzt Python für die Interaktion mit den Klassenelementen ein *dot notation*.

OOPclass.py

```python
import inspect

class Dog:    # Schlüsselwort "class"
    family = "Canidae"
    name = "Bello"
    age = 5

i = Dog()
print(i.species)
i.name = "Russel"
print(i.name)

for name, data in inspect.getmembers(i):
    if name == '__builtins__':
        continue
    print(f'{name} - {repr(data)}')

```

```
Traceback (most recent call last):
  File "/tmp/tmp_7bb6kcu/main.py", line 9, in <module>
    print(i.species)
AttributeError: 'Dog' object has no attribute 'species'
```

Aufgabe: Erläutern Sie die Ausgabe folgenden Codes. Wie müssen wir das Ergebnis interpretieren?

OOPclass.py

```python
import inspect

class Dog:
    family = "Canidae"
    name = "Bello"
    age = 5

i = Dog()
j = Dog()

print(i == j)
```

```
False
```

OOP Grundelemente in Python

Frage: Für welche Aufgaben ist der Konstruktor in einer Klasse verantwortlich?

OOPclass.py

```
class Dog:
    family = "Canidae"
    def __init__(self, name, age):
        self.name = name
        self.age = age

i = Dog("Rex", 5)
print(i.name, i.family, i.age)
```

```
Rex Canidae 5
```

Instanzmethoden sind Funktionen, die innerhalb einer Klasse definiert sind und nur von einer Instanz dieser Klasse aufgerufen werden können. Genau wie bei `__init__()` ist der erste Parameter einer Instanzmethode immer self.

OOPclass.py

```
class Dog:
    family = "Canidae"
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def makeSound(self):     # : nicht vergessen!
        print(f"{self.name} says Wuff")

i = Dog("Rex", 5)
i.makeSound()
```

```
Rex says Wuff
```

Aufgabe: Schreiben Sie eine Methode, so dass eine Instanz von Dog in Abhängigkeit von ihrem Alter schläft. Recherchieren Sie dazu unter `python delay` die notwendigen Methoden der `time` Klasse.

Wie Sie bereits bie der Inspektion der `list`, `int` aber auch der `Dog` Klassen gesehen haben, existiert eine Zahl von vordefinierten Funktionen - die sogenannten *dunder Methods*. Das Wort *dunder* leitet sich von *double underscore* ab.

Methode	Typ	implementiert
`__init__()`	Konstruktor	
`__str__()`	Methode	Generiert einen String aus den Objektdaten
...		
`__add__()`	Operator Obj + Obj	Arithmetische Operation
...		
`__eq__()`	Operator Obj == Obj	Logische Operation
`__lt__()`	Operator Obj <= Obj	
...		

Eine gute Einführung und detailierte Erklärung liefert Link

OOPclass.py

```
class Dog:
    family = "Canidae"
    def __init__(self, name, age):
        self.name = name
        self.age = age

i = Dog("Rex", 5)
print(i)
```

```
<__main__.Dog object at 0x7f17469dc310>
```

Kapselung

Python nutzt zwei führende Unterstriche, um Methoden und Variablen als *private* zu markieren.

private.py

```
class A:
  def method_public(self):
    print("This is a public method")

  def __method_private(self):
    print("This is a private method")

obj = A()
obj.method_public()
```

```
This is a public method
```

Auf private Methoden einer Klasse kann weder außerhalb der Klasse noch von irgendeiner Basisklasse aus zugegriffen werden kann.

> Wie können wir die private Methode überhaupt aufrufen?

Vererbung

> Was stört Sie an folgendem Codebeispiel?

RedundandCode.py

```
class Student:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print("Student -", self.firstname, self.lastname)


class StaffMember:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

Humboldt = Student("Alexander", "Humboldt")
Cotta = StaffMember("Bernhard", "von-Cotta")

Humboldt.printname()
Cotta.printname()
```

```
Student - Alexander Humboldt
Bernhard von-Cotta
```

Vererbung überträgt das Verhalten einer Basisklasse auf eine abgeleitete Klasse. Dadurch wird redundanter Code gespart.

Inheritance.py

```python
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

class Student(Person):
    pass

class StaffMember(Person):
    pass

Humboldt = Student("Alexander", "Humboldt")
Cotta = StaffMember("Prof. - " "Bernhard", "von-Cotta")

Humboldt.printname()
Cotta.printname()
```

```
Traceback (most recent call last):
  File "/tmp/tmp240lvm1t/main.py", line 15, in <module>
    Humboldt.printname()
AttributeError: 'Student' object has no attribute 'printname'
```

Python und C++ mit Blick auf OOP Konzepte

- Das Konzept der Überladung wird in Python nicht nativ unterstützt!

OOPclass.py

```python
class Dog:
  family = "Canidae"
  def __init__(self, *args):
    if len(args)>0:
      if isinstance(args[0], str):
        self.name = args[0]
      else:
        print("Der Datentyp passt nicht für die Variable Name!")
    else:
      self.name = "-"

i = Dog()
print(i.name, i.family)
j = Dog("Fido")
print(j.name, j.family)
```

```
- Canidae
Fido Canidae
```

- Private ist nicht wirklich private

NameMangling.py

```python
class A:
  def fun(self):
    print("This is a public method")

  def __fun(self):
    print("This is a private method")

obj = A()
obj.fun()
obj._A__fun()   # <- Name Mangling  "_classname__function"
```

```
This is a public method
This is a private method
```

OOP Beispiel

Nehmen wir an, dass wir eine Liste von Vorname erzeugen wollen. Dabei soll sichergestellt werden, dass diese unabhängig von den Eingaben der Bediener vergleichbar sind. Zudem sollen fehlerhafte Eingaben, die zum Beispiel Zahlen enthalten erkannt und gefiltert werden.

newListClass.py

```python
class NameList(list):
  def __init__(self):
    super().__init__()

  def append(self, item):
    if isinstance(item, str):
      if item.isalpha():
        super().append(item.lower())
      else:
        print("Wrong data type!")

  def uniques(self):
    return set(self)

A = NameList()
A.append("Jannes")
A.append("linda")
A.append("Moritz")
A.append("MORITZ")
print(A)
print(A.uniques())
```

```
['jannes', 'linda', 'moritz', 'moritz']
{'moritz', 'linda', 'jannes'}
```

Dafür schreiben wir eine abgeleitet Listenklasse mit einer eigenen Implementierung von `append()`.

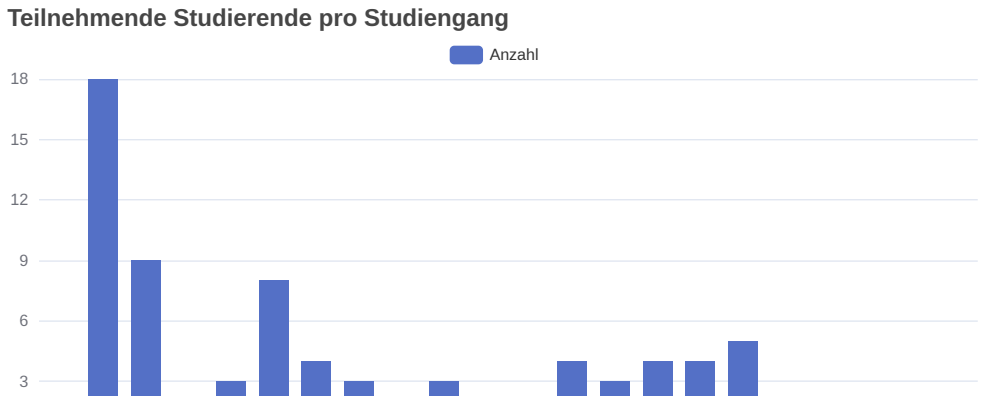
Aufgabe Erweitern Sie die Implementierung auf die `extend()` Methode der Listen.

Datenvisualisierung

In der vergangenen Woche haben wir Ihre Zugehörigkeit zu verschiedenen Studiengängen eingelesen und analysiert Link L09.

Auf die Frage hin, welche Häufigkeiten dabei auftraten, beantwortete unser Skript mit einem Dictonary:

```
{'S-UWE': 1, 'S-WIW': 18, 'S-GÖ': 9, 'S-VT': 2, 'S-BAF': 3, 'S-WWT': 8, 'S-NT': 4
  ,
 'S-ET': 3, 'S-MB': 1, 'S-FWK': 3, 'F1-INF': 2, 'S-BWL': 2, 'S-MAG': 4, 'F2-ANCH'
   : 3,
 'S-ACW': 4, 'S-GTB': 4, 'S-GBG': 5, 'S-GM': 2, 'S-ERW': 1, 'S-INA': 1, 'S-MORE':
  1,
  'S-CH': 1}
```



Die textbasierte Ausgabe ist nur gering geeignet, um einen raschen Überblick zu erlangen. Entsprechend suchen wir nach einer grafischen Ausgabemöglichkeit für unsere Python Skripte.

Python Visualisierungstools

Python stellt eine Vielzahl von Paketen für die Visualisierung von Dateninhalten bereit. Diese zielen auf unterschiedliche Visionen oder Features:

- einfache Verwendbarkeit
- große Bandbreite von Diagrammarten und Adaptionsmöglichkeiten
- interaktive Diagramme
- Vielzahl von Exportschnittstellen

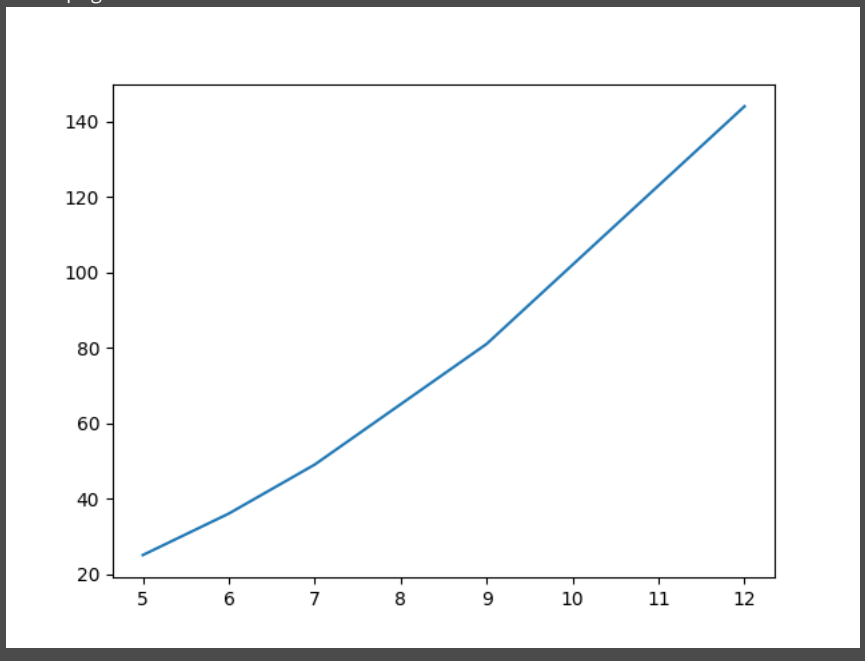
Package	Link	Besonderheiten
plotly	Link	Fokus auf interaktive Diagramme eingebettetet in Webseiten
seaborn	Link	Leistungsfähige Darstelung von statistischen Daten
matplotlib	Link	
...		

Matplotlib Grundlagen

Beispiel.py

```
1  import matplotlib.pyplot as plt
2
3  a = [5,6,7,9,12]
4  b =[x**2 for x in a]   # List Comprehension
5  plt.plot(a, b)
6  plt.show()
7
8  #plt.show()
9  plt.savefig('foo.png') # notwendig für die Ausgabe in LiaScript
```

foo.png



Anpassung	API	
Linientyp der Datendarstellung	pyplot.plot	`plt.plot(a, b, 'ro:')`
Achsenlabel hinzufügen	pyplot.xlabel	`plt.xlabel('my data', fontsize=14, color='red')`
Titel einfügen	pyplot.title	`plt.title(r'$\sigma_i=15$')`
Gitter einfügen	pyplot.grid	`plt.grid()`
Legende	pyplot.legend	`plt.plot(a, b, 'ro:', label="Data")`
		`plt.legend()`
Speichern	pyplot.savefig	`plt.savefig('foo.png')`

Tutorial von Rizky Maulana Nurhidayat auf medium

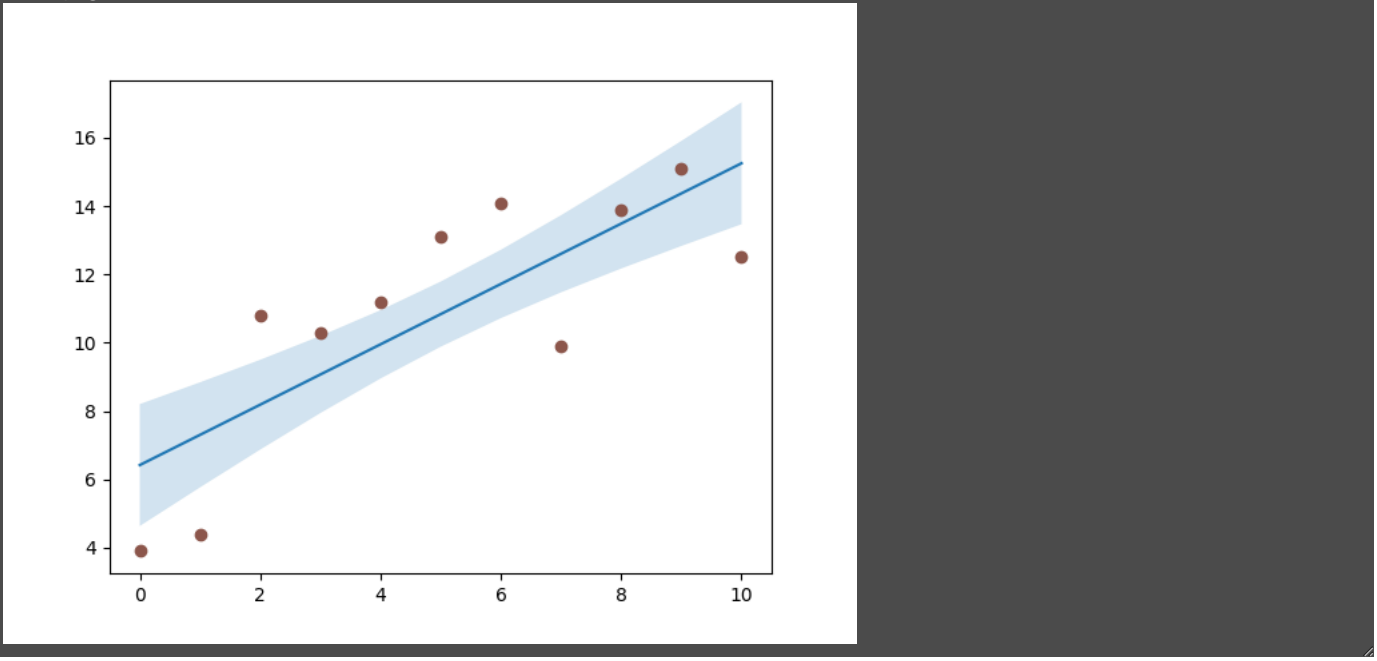
Weiter Tutorials sind zum !?[MatplotlibTutorial](https://www.youtube.com/watch?v=UO98lJQ3QGI)

Matplotlib Beispiele

MultipleDiagrams.py

```python
import numpy as np
import matplotlib.pyplot as plt

N = 21
x = np.linspace(0, 10, 11)
y = [3.9, 4.4, 10.8, 10.3, 11.2, 13.1, 14.1,  9.9, 13.9, 15.1, 12.5]

# fit a linear curve an estimate its y-values and their error.
a, b = np.polyfit(x, y, deg=1)
y_est = a * x + b
y_err = x.std() * np.sqrt(1/len(x) +
                          (x - x.mean())**2 / np.sum((x - x.mean())**2))

fig, ax = plt.subplots()
ax.plot(x, y_est, '-')
ax.fill_between(x, y_est - y_err, y_est + y_err, alpha=0.2)
ax.plot(x, y, 'o', color='tab:brown')

#plt.show()
plt.savefig('foo.png') # notwendig für die Ausgabe in LiaScript
```

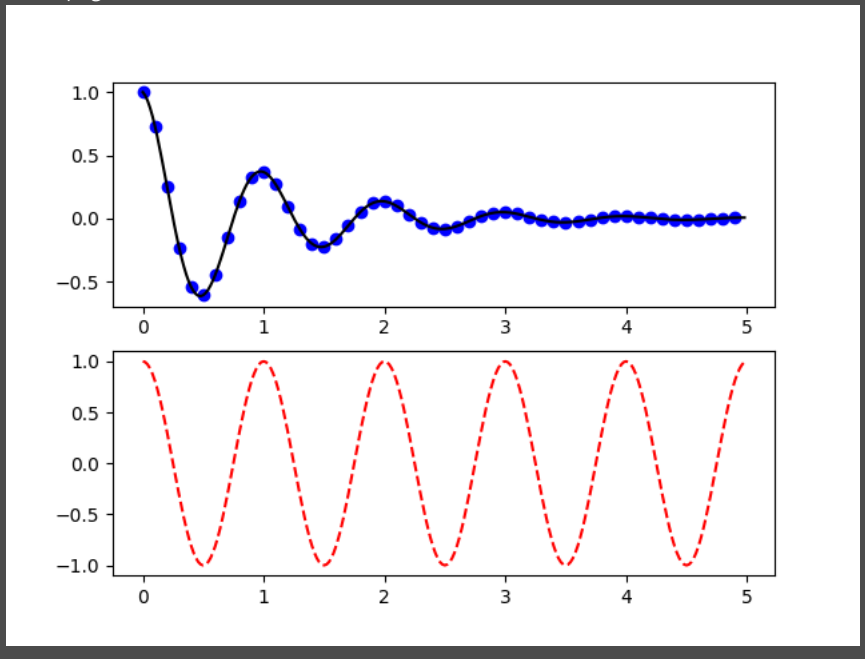
foo.png



MultipleDiagrams.py

```python
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure()
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
#plt.show()
plt.savefig('foo.png') # notwendig für die Ausgabe in LiaScript
```

foo.png

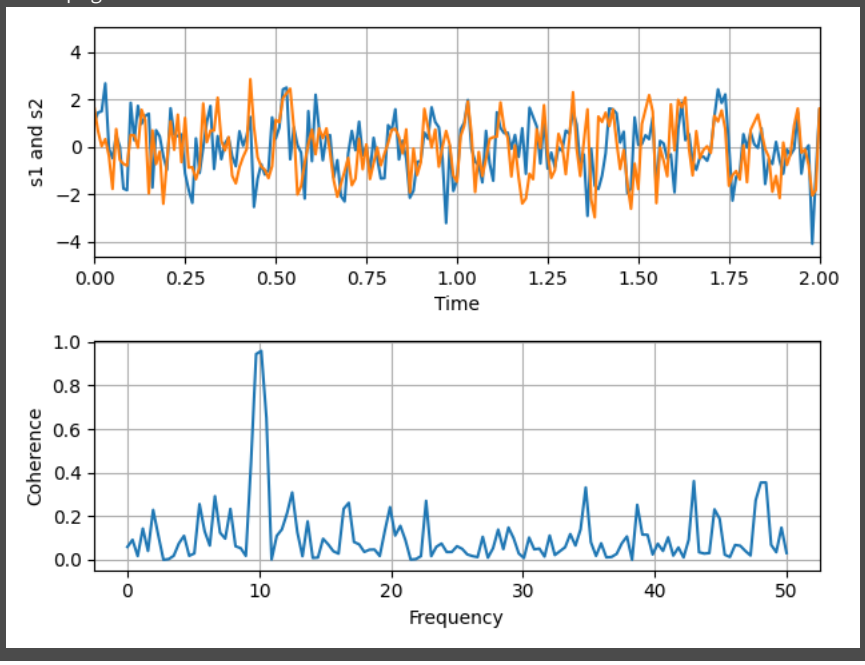


Beispiel der Woche

Beispiel.py

```python
import numpy as np
import matplotlib.pyplot as plt

# Fixing random state for reproducibility
np.random.seed(19680801)

dt = 0.01
t = np.arange(0, 30, dt)
nse1 = np.random.randn(len(t))                 # white noise 1
nse2 = np.random.randn(len(t))                 # white noise 2

# Two signals with a coherent part at 10 Hz and a random part
s1 = np.sin(2 * np.pi * 10 * t) + nse1
s2 = np.sin(2 * np.pi * 10 * t) + nse2

fig, axs = plt.subplots(2, 1)
axs[0].plot(t, s1, t, s2)
axs[0].set_xlim(0, 2)
axs[0].set_xlabel('Time')
axs[0].set_ylabel('s1 and s2')
axs[0].grid(True)

cxy, f = axs[1].cohere(s1, s2, 256, 1. / dt)
axs[1].set_ylabel('Coherence')

fig.tight_layout()

#plt.show()
plt.savefig('foo.png')
```

foo.png


s1 and s2
4
2
0
−2
−4
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
Time
Coherence
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
Frequency