In [ ]:
```
# Sprachkurs Python3.6
```

In [1]:
```
print("Hallo zusammen!")
```

```
Hallo zusammen!
```

Die Grammatik des `print`-Befehls

Print nimmt beliebig viele Argumente. Je nach Argumenten verhält sich der Befehl unterschiedlich

In [203]:
```
# Einfaches print:
print("Ich bin ein Text, und ende mit einer neuen Zeile!")
# Print ohne neue Zeile:
print("Ich bin ein Text, ", end="")
# Print mit mehreren Argumenten:
print("und", "gehe", "weiter", "in", "der", "letzten", "Zeile.")
# Print mit beliebigem Separator:
print(1, 2, 3, 4, sep=",")
```

```
Ich bin ein Text, und ende mit einer neuen Zeile!
Ich bin ein Text, und gehe weiter in der letzten Zeile.
1,2,3,4
```

Zahlen

In Python gibt es zwei unterschiedliche Arten von Zahlen.

In [3]:
```
# Ganze Zahlen (Integer)
print(1, 19, 42)
```

```
1 19 42
```

In [4]:
```
# Gleitkommazahlen (Float)
print(1.11, 1.2e60, 3.1415)
```

```
1.11 1.2e+60 3.1415
```

Wahrheitswerte

In [5]:
```
# Wahr und Falsch
print(True, False)
```

```
True False
```

Zeichenketten (Strings)

Python kann auch Zeichenketten, wie zum Beispiel diesen Satz, darstellen.

In [6]:
```
# Zeichenkette mit "..."
print(
    "Python kann auch Zeichenketten, wie zum Beispiel diesen Satz, darst
ellen.")
# Zeichenkette mit '...'
print(
    'Python kann auch Zeichenketten, wie zum Beispiel diesen Satz, darst
ellen.')
# Zeichenkette mit """..."""
print(
    """Python kann auch Zeichenketten, wie zum Beispiel diesen Satz, dar
stellen.""")
```

```
Python kann auch Zeichenketten, wie zum Beispiel diesen Satz, darstellen.
Python kann auch Zeichenketten, wie zum Beispiel diesen Satz, darstellen.
Python kann auch Zeichenketten, wie zum Beispiel diesen Satz, darstellen.
```

Variablen

Variablen sind eine Möglichkeit, in Python Werte mit Namen zu verknüpfen.

In [9]:
```
# Ich weise einer Variable einen Wert zu:
ich_bin_eine_zahl = 22
print("In der Variable steht", ich_bin_eine_zahl)
# Ich weise ihr einen neuen Wert zu:
ich_bin_eine_zahl = ich_bin_eine_zahl + 10
# oder:
ich_bin_eine_zahl += 10
print("In der Variable steht jetzt", ich_bin_eine_zahl)
# Ich kann auch etwas ganz anderes in die Variable schreiben:
ich_bin_eine_zahl = "Hallo I bims ein String."
print("In der Variable steht jetzt", ich_bin_eine_zahl)
```

```
In der Variable steht 22
In der Variable steht jetzt 42
In der Variable steht jetzt Hallo I bims ein String.
```

Listen

Manchmal möchte man mehrere zusammenhängende Werte nicht einzeln aufbewahren, sondern zusammen, in einer bestimmten Reihenfolge. Dafür nimmt man dann eine Liste.

In [13]:
```
# Liste von Integers:
print([1, 2, 3, 77, 128])
# Liste von Floats:
print([1.1, 1.2, 1.3, 1.011178])
# Liste von Zeichenketten:
print(["Hallo", "I", "bims"])
# Liste von was auch immer:
print(["Hallo", "I", "bims", 1, "Program!", 3.1415, ["Lol", 111]])

# Ich schreibe jetzt eine Liste in eine Variable:
meine_liste = ["Hallo", "I", "bims", 1, "Program!", 3.1415, ["Lol", 111]
]
# dann kann ich auf die Elemente der Liste zugreifen:
print("Erstes Element:", meine_liste[0])
print("Zweites Element:", meine_liste[1])
print("Letztes Element:", meine_liste[-1])
print("Vorletztes Element:", meine_liste[-2])
# und sie auch verändern:
meine_liste[0] = 42
print("Erstes Element:", meine_liste[0])
# oder Unterabschnitte der Liste auswählen:
print("Liste vom zweiten bis zum fünften Element: ", meine_liste[1:5])
```

```
[1, 2, 3, 77, 128]
[1.1, 1.2, 1.3, 1.011178]
['Hallo', 'I', 'bims']
['Hallo', 'I', 'bims', 1, 'Program!', 3.1415, ['Lol', 111]]
Erstes Element: Hallo
Zweites Element: I
Letztes Element: ['Lol', 111]
Vorletztes Element: 3.1415
Erstes Element: 42
Liste vom zweiten bis zum fünften Element:  ['I', 'bims', 1, 'Program!']
```

In [204]:
```
# Listen und Strings besitzen einen gemeinsamen Satz an Operationen:
# 1. Zugriff:
mein_string = "Text"
meine_liste = [1, 2, 3, 4]
print("Erster Buchstabe:", mein_string[0])
print("Erstes Element:", meine_liste[0])
# 2. Multiplikation:
print(mein_string * 10)
print(meine_liste * 10)
# 3. Verkettung:
print("#" + mein_string)
print(meine_liste + [5])
# 4. Substring:
print(mein_string[1:3])
print(meine_liste[1:3])
```

```
Erster Buchstabe: T
Erstes Element: 1
TextTextTextTextTextTextTextTextTextText
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
#Text
[1, 2, 3, 4, 5]
ex
[2, 3]
```

Dictionaries

Wenn wir Werte nicht nur in einer bestimmten Reihenfolge zusammenfassen wollen, sondern mit einem Satz anderer Werte verknüpft, können wir Dictionaries verwenden.

In [15]:
```
# Dictionary mit zwei Einträgen:
mein_dictionary = {
    "Hallo": "I bims.",
    42: "Die Lösung"
}
print(mein_dictionary)

# Auch hier kann ich auf Werte im Dictionary zugreifen:
print("Wert für \"Hallo\":", mein_dictionary["Hallo"])
# oder sie verändern:
mein_dictionary["Hallo"] = "Welt!"
print("Wert für \"Hallo\":", mein_dictionary["Hallo"])
# oder neue hinzufügen:
mein_dictionary["Foo"] = "Bar"
print(mein_dictionary)
```

```
{42: 'Die Lösung', 'Hallo': 'I bims.'}
Wert für "Hallo": I bims.
Wert für "Hallo": Welt!
{42: 'Die Lösung', 'Foo': 'Bar', 'Hallo': 'Welt!'}
```

Funktionen

Ihr findet es doch sicher auch ätzend, alles tausend mal wiederholen zu müssen. Glücklicherweise gibt es in den meisten Programmiersprachen Wege, um Dinge nur einmal sagen zu müssen. Diese werden je nach Sprache als Routines, Subroutines, Procedures oder Functions bezeichnet. Wir nennen diese Bausteine Funktionen.

In [17]:
```
# Wir wollen Leute begrüßen, aber nicht jedes mal sagen müssen:
print("Hallo, was auch immer dein Name ist!")
# deswegen definieren wir uns eine Funktion,
# der wir nur noch den Namen geben müssen und die
# den Rest für uns erledigt:
def begrüße(name):
    print("Hallo " + name + "!")
# dann können wir diese Funktion immer wieder verwenden,
# wenn wir jemanden begrüßen wollen:
begrüße("Welt")
begrüße("Python")
begrüße("Freckenhorst")
```

```
Hallo, was auch immer dein Name ist!
Hallo Welt!
Hallo Python!
Hallo Freckenhorst!
```

In [208]:
```
# Funktionen können auch Dinge zurückgeben,
# die sich Variablen zuweisen lassen.

# Wir wollen zum Beispiel das Quadrat einer Zahl berechnen:
def quadriere(zahl):
    return zahl * zahl

# oder:
def quadriereV2(zahl):
    ergebnis = zahl * zahl
    return ergebnis

print("42 im Quadrat:", quadriere(42))
```

```
42 im Quadrat: 1764
```

Kontrollstrukturen

Manchmal wollen wir Programme nicht volkommen geradlinig ausführen, sondern für unterschiedliche Eingaben unterschiedliche Verhaltensweisen anwenden, Programmteile mehrfach ausführen, oder Programmteile bei Programmfehlern zur Laufzeit schmerzlos verlassen. Dazu verwenden wir sogenannte Kontrollstrukturen. Mehr dazu im Codebeispiel.

In [24]:
```
# Wollen wir Programmteile konditional ausführen,
# so verwenden wir die `if` Kontrollstruktur:
if 0 + 1 == 1:
    # Wird ausgeführt, wenn 0 + 1 == 1
    print("Die Mathematik funktioniert!")
else:
    # Wird ausgeführt, wenn 0 + 1 != 1
    print("Oh nein, die Welt geht unter!!!")
```

```
Die Mathematik funktioniert!
```

In [215]:
```
# Wollen wir mehr als eine Alternative behandeln,
# verwenden wir `if` mit `elif` (else, if):
import random
a = random.randint(0,4)
if a == 0:
    print("Null")
elif a == 1:
    print("Eins")
else:
    print("Ist mir doch egal!!!")
```

```
Null
```

In [216]:
```
# Als komplexeres Beispiel wollen wir den Postfix
# für die Aufzählung von Zahlen bestimmen und
# uns die Aufzählung zurückgeben lassen:
def aufzählen_postfix(zahl):
    if zahl == 1:
        return "st"
    elif zahl == 2:
        return "nd"
    elif zahl == 3:
        return "rd"
    elif zahl < 20:
        return "th"
    elif zahl < 100:
        return aufzählen_postfix(zahl % 10)
    else:
        return aufzählen_postfix(zahl % 100)

def aufzählen(zahl):
    return str(zahl) + aufzählen_postfix(zahl)

print("111th?", aufzählen(111))
```

```
111th? 111th
```

In [217]:
```
# Wollen wir ein Teil des Programms ausführen,
# solange eine Bedingung wahr ist, nutzen wir
# `while`:

a = 0
while a < 10:
    a += 1
    print(aufzählen(a))
```

```
1st
2nd
3rd
4th
5th
6th
7th
8th
9th
10th
```

In [218]:
```
# Wollen wir einen Teil des Programms für
# alle Werte in einer Liste oder einem
# Iterator ausführen, nutzen wir `for`:

print("Mit einer Liste:")
aufzählbar = [0, 1, 2, 3]

for element in aufzählbar:
    # Tun wir etwas mit jedem Element einer Liste.
    print(element)

print("Mit einem Dictionary:")
aufzählbar = {"a": 1, "b": 2}

for element in aufzählbar:
    # Sieht ganz ähnlich aus ...
    # Macht es, was ihr erwartet?
    print(element)

print("Mit `items`:")
for schlüssel, wert in aufzählbar.items():
    # Das ist eher das, was man erwarten würde.
    print(schlüssel, "=", wert)

print("Als Zählschleife:")
for idx in range(0, 10):
    # Läuft einmal für jeden Wert zwischen 0 und 10,
    # 10 nicht eingeschlossen.
    print(idx)

print("Mit einem Iterator:")
def aufzählbar(bis_zahl):
    for idx in range(1, bis_zahl + 1):
        # Während Funktionen einen Wert ausgeben und
        # dann die Kontrolle mit `return` zurückgeben,
        # ermöglichen Iteratoren über `yield`,
        # mehrere Werte hintereinander zurück zu geben,
        # über welche in einer `for`-Schleife iteriert
        # werde kann.
        yield aufzählen(idx)

for element in aufzählbar(10):
    # Sieht doch ganz schön aus...
    print(element)
```

```
Mit einer Liste:
0
1
2
3
Mit einem Dictionary:
a
b
Mit `items`:
a = 1
b = 2
Als Zählschleife:
0
1
2
3
4
5
6
7
8
9
Mit einem Iterator:
1st
2nd
3rd
4th
5th
6th
7th
8th
9th
10th
```

Dekoratoren (weiterführend)

Manchmal wollen wir eine modulare Möglichkeit haben, Funktionen abzuändern. Diese ist uns durch Dekoratoren geboten.

In [117]:
```
# Rekursion ist teuer...
# Merken wir uns lieber, welcher Wert bei einer Funktion heraus kommt...
def memoize(a):
    def helferfunktion(*x):
        if x not in helferfunktion.memo:
            helferfunktion.memo[x] = a(*x)
        return helferfunktion.memo[x]
    helferfunktion.memo = {}
    return helferfunktion

@memoize
def fib(x):
    # Diese Funktion würde uns ziemlich schnell explodieren,
    # wenn sie komplett rekursiv wäre.
    # Mit unserem sich Dinge merkendem Dekorator ist dieses
    # Problem behoben.
    if x == 0:
        return 1
    if x == 1:
        return 1
    else:
        return fib(x - 1) + fib(x - 2)

print(fib(100))
```

```
573147844013817084101
```

Ein/Ausgabe (IO)

Normalerweise wollen wir ein Programm nicht auf spezielle Werte zuschneiden, sondern es dem Nutzer ermöglichen, seine eigenen Eingaben zu liefern. Das ist in Python sehr einfach.

In [221]:
```
# Wir können Zeichenketten von der Konsole mittels `input()` einlesen.
# Begrüßen wir mal Personen, die ihren Namen in die Konsole eingeben:
print("Nenne mir Deinen Namen!")
name = input()
begrüße(name)
```

```
Nenne mir Deinen Namen!
Karl-Heinz
Hallo Karl-Heinz!
```

In [222]:
```
# Wir nehmen in diesem Beispiel so eine Eingabe und machen daraus
# eine ganze Zahl (Integer), indem wir `int` darauf anwenden.
print("Sag mir bitte eine Zahl...")
for element in range(1, int(input()) + 1):
    print(aufzählen(element))
```

```
Sag mir bitte eine Zahl...
10
1st
2nd
3rd
4th
5th
6th
7th
8th
9th
10th
```

Aufgabe: Tag der (Fahrtkosten-)Abrechnung

Eure Aufgabe ist es, die Fahrtkostenabrechnung für die Ferienakademie zu digitalisieren. Ihr schreibt dafür zunächst ein Programm, welches die nötigen Informationen von der Konsole einliest:

- Den vollständigen Namen der Person.
- Deren Reisekosten.

Das Einlesen von Daten soll durch die Eingabe von `exit` beendet werden können. Das Programm soll dann für alle eingegebenen Datenpunkte die mittleren Fahrtkosten (mean) und die jeweilige Differenz dazu ausrechnen und tabelliert ausgeben.

In [164]:
```
def gelddrucken(name, kosten, differenz):
    return "{0:<30}{1:>20.2f} €{2:>20.2f} €".format(name, kosten, differ
enz)

def tagderabrechnung():
    mittlere_fahrtkosten = 0.0
    fahrtkosten = {}
    differenzen = {}
    print("Willkommen bei der Abrechnung!")
    while input() != "exit":
        print("Vor- und Zuname: ", end="")
        name = input()
        print("")
        print("Kosten: ", end="")
        kosten = float(input())
        print("")
        print("Nächster ...")
        fahrtkosten[name] = kosten
    for elem in fahrtkosten:
        mittlere_fahrtkosten += fahrtkosten[elem]
    mittlere_fahrtkosten /= len(fahrtkosten)
    for elem in fahrtkosten:
        differenzen[elem] = fahrtkosten[elem] - mittlere_fahrtkosten
    print("\n")
    print("-" * 74)
    print("  Fahrtkostenabrechnung Akademie 42")
    print("-" * 74)
    print("{0:<30}{1:>22}{2:>22}".format("Vor- und Zuname", "Fahrtkosten
", "Differenz"))
    for elem in fahrtkosten:
        print(gelddrucken(elem, fahrtkosten[elem], differenzen[elem]))
    print("Mittlere Fahrtkosten: ", mittlere_fahrtkosten, " €")

tagderabrechnung()
```

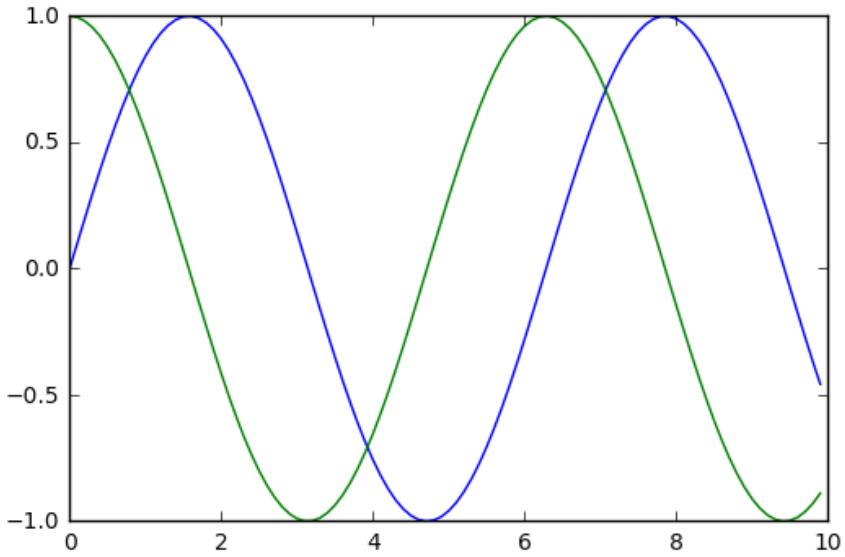
```
Willkommen bei der Abrechnung!

Vor- und Zuname: max mustermann

Kosten: 100.0

Nächster ...

Vor- und Zuname: melinda mustermann

Kosten: 120

Nächster ...
exit


--------------------------------------------------------------------------
-
  Fahrtkostenabrechnung Akademie 42
--------------------------------------------------------------------------
-
Vor- und Zuname                           Fahrtkosten             Differen
z
max mustermann                               100.00 €              -10.00
€
melinda mustermann                           120.00 €               10.00
€
Mittlere Fahrtkosten:  110.0  €
```

Plotting

Immer wieder hat man Daten die man darstellen möchte. In Python gibt es Bibliotheken, die das einfach machen.

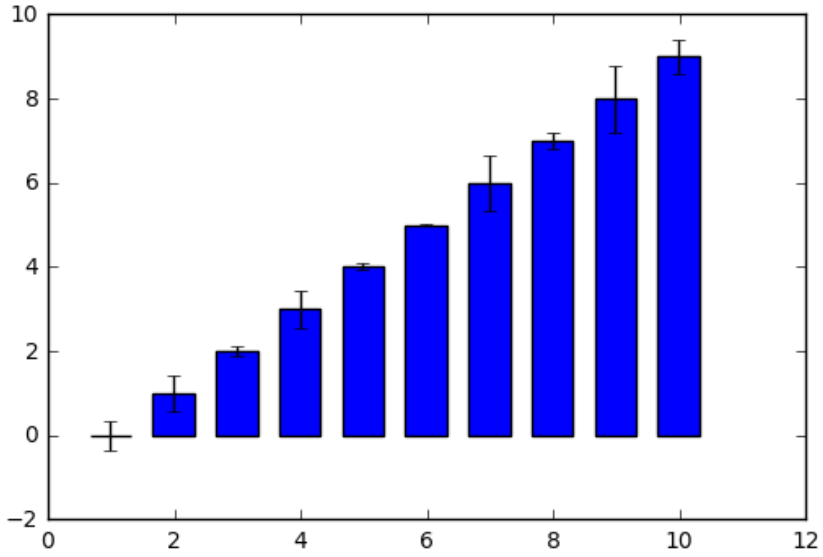
In [160]:
```
%matplotlib inline

# Wir importieren hier eine Plotbibliothek
from matplotlib import pyplot as plt
from math import sin
import numpy as np

# Jetzt plotten wir mal einen Graphen
x = np.arange(0, 10, 0.1)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
```

Out[160]:
```
[<matplotlib.lines.Line2D at 0x7f2e25c9d438>]
```



In [193]:
```
# Man kann auch andere Arten von Graphen erstellen:
left = np.arange(0.66, 10.66, 1)
width = 0.66
height = np.arange(0, 10, 1)
errorbar = np.asarray([random.random() for _ in range(10)])
plt.bar(left, height, width, yerr = errorbar, ecolor = "black")
```

Out[193]: <Container object of 10 artists>



In [183]:
```
# Wir bauen erstmal unsere x- und y-Achsen:
x = np.arange(0, 10, 0.1)
y = np.arange(0, 10, 0.1)
# Dann blasen wir sie auf 2D auf:
meshX, meshY = np.meshgrid(x, y)
# Wir wollen die Funktion f(x, y) := sin(x) * sin(y) plotten.
# Darum bauen wir uns eine Matrix dafür, unter Verwendung
# unserer x- und y-Achsen:
image = np.sin(meshX) * np.cos(meshY)

# Jetzt, da wir eine Matrix haben, können wir sie plotten:
fig, ax = plt.subplots()
ax.imshow(image)
ax.set_xticks(np.arange(0, 110, 10))
ax.set_yticks(np.arange(0, 110, 10))
ax.set_xticklabels(np.arange(0, 11, 1))
ax.set_yticklabels(np.arange(0, 11, 1));
```