**Python noter til eksamen**

**Vores Python projekt - Color histogram**

**mapper:**

**data**

**data/fck**

**data/bif**

**data/histogram**

**get\_url.py:**

import webget

def download\_image(arr, folder):

for image\_path in arr:

webget.download(image\_path, './data/' + folder)

Vi bruger webget til at hente filer via url'er.

Vores download\_image funktion tager et array og et mappenavn. Så looper vi igennem det array den får som argument og bruger vores webget.download funktion og placere billedet i data mappen og i den mappe, den tager som argument (fck eller bif).

**resize.py**

import cv2

def resize\_image(arr):

for img\_str in arr:

image = cv2.imread(img\_str)

resize\_image = cv2.resize(image, (800,600), interpolation=cv2.INTER\_CUBIC)

cv2.imwrite(img\_str, resize\_image)

Interpolation er en algoritme i openCV. Det er den der skalere et billede ned eller op. Der er forskellige metoder til sådan en skalering. Vi bruge den der hedder INTER\_CUBIC . Her er URL’en med de forskellige metoder. (<http://tanbakuchi.com/posts/comparison-of-openv-interpolation-algorithms/>)

Vi bruger cv2 til at læse, skalere og lave et nyt billede med de nye proportioner.   
Vi har en funktion der hedder resize\_image. Den tager et array og looper det igennem og skalerer hvert element i arrayet med cv2 med faste værdier på 800x600px.

**histogram\_creater.py**

import cv2

import numpy as np

import matplotlib.pyplot as plt

from glob import glob

import os

# Create histograms over all images we download and two super histograms

def histogram\_saver(arr, superhist):

blue\_channels\_hist = []

green\_channels\_hist = []

red\_channels\_hist = []

for imagename in arr:

image = cv2.imread(imagename)

cv2.imshow("image", image)

chans = cv2.split(image)

colors = ("b", "g", "r")

plt.figure()

plt.title(imagename + "' - BGR Color Histogram")

plt.xlabel("Bins")

plt.ylabel("# of Pixels")

plt.axis([0, 256, 0, 6000])

features = []

for (chan, color) in zip(chans, colors):

# create a histogram for the current channel and

# concatenate the resulting histograms for each

# channel

hist = cv2.calcHist([chan], [0], None, [256], [0, 256])

if color == 'b':

blue\_channels\_hist.append(hist.T[0][:])

elif color == 'g':

green\_channels\_hist.append(hist.T[0][:])

elif color == 'r':

red\_channels\_hist.append(hist.T[0][:])

features.extend(hist)

# plot the histogram

.T står for transpose. Den transponere matrixen og vender arrayet om. Så i stedet for at få en liste med 256 forskellige lister , for vi en liste med 256 værdier   
<https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.ndarray.transpose.html>

plt.plot(hist, color = color)

plt.xlim([0, 256])

plt.savefig('./data/histogram/' + os.path.basename(imagename))

# Super histogram

b = np.mean(blue\_channels\_hist, axis=0)

g = np.mean(green\_channels\_hist, axis=0)

r = np.mean(red\_channels\_hist, axis=0)

plt.figure()

plt.title(superhist + "' - BGR Color Histogram")

plt.xlabel("Bins")

plt.ylabel("# of Pixels")

plt.axis([0, 256, 0, 6000])

plt.plot(b, color='b')

plt.plot(g, color='g')

plt.plot(r, color='r')

plt.xlim([0, 256])

plt.savefig('./data/histogram/' + os.path.basename(superhist))

# Create histogram on a specific image with different axis than the others

def oneimage(imagename):

image = cv2.imread(imagename)

cv2.imshow("image", image)

chans = cv2.split(image)

colors = ("b", "g", "r")

plt.figure()

plt.title(imagename + "' - One Image")

plt.xlabel("Bins")

plt.ylabel("# of Pixels")

plt.axis([20, 30, 0, 4000])

features = []

for (chan, color) in zip(chans, colors):

# create a histogram for the current channel and

# concatenate the resulting histograms for each

# channel

hist = cv2.calcHist([chan], [0], None, [256], [0, 256])

features.extend(hist)

# plot the histogram

plt.plot(hist, color = color)

plt.xlim([20, 30])

plt.savefig('./data/histogram/' + os.path.basename(imagename))

Vi bruger cv2. til at bearbejde billederne med, openCV har en masse funktioner til billeder. Vi bruger imread til at læse billederne, imshow til at vise billederne med,

Vi bruger numpy.   
**- NumPy** er den grundlæggende package til videnskabelig databehandling i Python. Det er et Python bibliotek.

Vi bruger matplotlib.pyplot. Det bruger vi til at plotte værdier på en graf i en GUI

Vi bruger glob glob.glob(*pathname*)

Returner en mulig tom liste over pathnames, der matcher patchen, som skal være en string, der indeholder en pathspecifikation. (../../Tools/\*/\*.gif)

Vi bruger os -gør det muligt hvis du bare vil læse eller skrive en fil, [**open()**](https://docs.python.org/2/library/functions.html#open), hvis du vil manipulere stier, se os.path-modulet, og hvis du vil læse alle linjerne i alle filerne på kommandolinjen, se [**fileinput**](https://docs.python.org/2/library/fileinput.html#module-fileinput) module . Hvis du vil oprette midlertidige filer og mapper, kan du se tempfile-modulet, og til filhåndtering og filhåndtering på højt niveau, se [**shutil**](https://docs.python.org/2/library/shutil.html#module-shutil) modulet.

Vi har 2 funktioner i **histogram\_saver(arr, superhist)** som tager et array og et navn til hver af de to superhistogramer, **oneimage(imagename)** som tager et billede navn.

I **histogram\_saver** har vi 3 arrays: blå, grøn og rød.

Så looper vi igennem vores array som vi får ind fra vores argument.

Så læser vi billedet med cv2.imread

Så viser vi billedet cv2.imshow men det bruger vi ikke til noget som sådan.

chans = bruger vi til at splitte billedet op i tre dele til hver farve kanal.

colors = bruger vi til hver farve: b, g og r.

Så laver vi vores graf med plt.figure

Så sætter vi en title på med navnet fra billedet og en tekst

Så sætter vi et label på y og x-aksen som er en tekst

Så sætter vi værdierne på y og x akserne

Så laver vi features arrayet features.extend(hist): Det extend gør er at den sætter det ind i det array som er det i arrayet hvor derimod append sætter objektet ind, så brugte man append ville man have flere objekter inde i objekter derfor bruger vi extend.

append: [1, 2, 3, [4, 5]]

extend: [1, 2, 3, 4, 5]

Så laver vi loop med zip på farver og kanaler: Det zip gør er at den tager to lister og samler dem i tuples, så det passer med at det første element i hver liste kommer i en tuple, og sådan fortsætter det.

**Histogram beregning i OpenCV**

Så nu bruger vi **cv2.calcHist ()** funktionen til at finde histogrammet. Lad os gøre bekendt med funktionen og dens parametre:

**cv2.calcHist** (billeder, kanaler, maske, histSize, intervaller [, hist [, accumulate]])

**billeder:** det er kildebilledet af typen uint8 eller float32. den skal angives i firkantede parenteser, dvs. "[img]".

**kanaler:** den findes også i firkantede parenteser. Det er indekset for kanalen, som vi beregner histogram for. Hvis input f.eks. Er gråtonebillede, er værdien [0]. For farvebillede kan du passere [0], [1] eller [2] for at beregne histogram af henholdsvis blå, grøn eller rød kanal.

**maske:** maske billede. For at finde histogrammet for det fulde billede, er det givet som "Ingen". Men hvis du vil finde et histogram af et bestemt område af billedet, skal du oprette et maskebillede for det og give det som maske. (Jeg vil vise et eksempel senere.)

**histSize:** Dette repræsenterer vores BIN-tæller. Skal angives i firkantede parenteser. For fuld skala passerer vi [256].

**Områder:** Dette er vores RANGE. Normalt er det [0,256].

Så spørger vi om det er blå, grøn eller rød og hvis det er den farve så ligger den vores udregning til det givne array blå, grøn eller rød. Det array ligger inde i et array derfor skal vi gå ind på index 0 også tager vi alt der ligger i det array. **hist.T[0][:]**: .**T** står for transpose. **Den transponere matrixen og vender arrayet om**. Så i stedet for at få en liste med 256 forskellige lister , for vi en liste med 256 værdier   
Så skal vi tegne vores graf med plt.plot() med vores udregning og vores nuværende farve.

Så sætter vi max værdi på x aksen

Så gemmer vi vores billedet i vores histogram mappe

**# Super histogram**

Nu laver vi vores super histogram

Så tager vi gennemsnittet af det blå, grøn og røde array med en akse på 0 dvs. enten er det vandret eller lodret alt efter hvordan det 2 dimensionelle array ligger.

Så laver vi vores graf med plt.figure

Sætter en title

Sætter label på y og x aksen

Sætter værdier på akserne

Så laver vi hver farve - b, g, r

Så sætter vi max værdi på x aksen

Så gemmer vi billedet i vores histogram

**oneimage(imagename)**

**Dette er for at teste på værdier med 20-30 på x og 0 til 4000 på Y på et specifik billedet**

Funktionen tager et billede navn

Loop i det array vi tager som argument

Læse billedet med cv2.imread

Viser billedet som vi ikke bruger til noget

Chans = så bruger vi cv2.split på billedet for at dele det op i de 3 farvekanaler

Så laver vi colors med b, g, r farver

Så laver vi vores graf

Så sætter vi en title på med billede navn og en tekst

Så sætter vi et label på x-aksen

Så sætter vi et label på y-aksen

Så sætter vi værdierne på y og a akserne som er 20-30 på X og 0 til 4000 på Y

Så laver vi features arrayet features.extend(hist)

Så laver vi loop med zip på farver og kanaler: Det zip gør er at den tager to lister og samler dem i tuples, så det passer med at det første element i hver liste kommer i en tuple, og sådan fortsætter det.

Så skal vi tegne vores graf med plt.plot() med vores udregning og vores nuværende farve.

Så sætter vi vores x aksen en fast værdi fra 20 til 30

Så gemmer vi vores billedet i vores histogram mappe

**run.py**

import cv2

import numpy as np

import matplotlib.pyplot as plt

from glob import glob

import os

import webget

## Our own py files

import get\_url

import resize

import histogram\_creater as hist

image\_array\_fck [xxx]

image\_array\_bif[xxx]

print('Downloading FCK')

get\_url.download\_image(image\_array\_fck, 'fck')

print('Downloading Brøndby')

get\_url.download\_image(image\_array\_bif, 'bif')

print('Globbing FCK')

fck\_images = glob('data/fck/\*.jpg')

print('Globbing Brøndby')

bif\_images = glob('data/bif/\*.jpg')

print('Resizes FCK')

resize.resize\_image(fck\_images)

print('Resizes Brøndby')

resize.resize\_image(bif\_images)

print('Creating histograms for FCK')

hist.histogram\_saver(fck\_images, 'Super FCK')

print('Creating histograms for Brøndby')

hist.histogram\_saver(bif\_images, 'Super BIF')

print('One image')

hist.oneimage('Fanside537-1.jpg')

hist.oneimage('Frederik-Bay\_-FCK.jpg')

**Her bruger vi alle vores funktioner i: webget.py, get\_url.py, resize.py, histogram\_creater.py.**

**predict.py**

def predict(arr\_blue, arr\_red, arr\_green):

#Two checks to conclude if picture contain BIF or FCK fan

checkBIF = False

#Not used yet as FCK range is not decided.

checkFCK = False

for x in range(50, 70):

if 1999 < arr\_blue[x-1] < 4201:

checkBIF = True

else:

checkBIF = False

if not checkBIF:

break

#Add another loop for the range of FCK, and change return

#logic to include checkFCK and return the final conclusion.

if checkBIF and not checkFCK:

return "Brøndby fan!"

else:

return "Ikke en brøndby fan..."

#Test data for en brøndby fan

result = predict([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300,

2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300,

2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 0, 0, 0, 0], [], [])

#Test data for en ikke brøndby fan

#result = predict([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

# 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

# 0, 0, 0, 0, 0, 0, 0, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 1900,

# 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300,

# 2300, 2300, 2300, 2300, 2300, 2300, 2300, 2300, 0, 0, 0, 0], [], [])

print(str(result))

**ShakalAli koden til noget predict som ikke virker.**

**Lecture note 5 - Basics**

print('Hello World!')

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Operation** | **Example** | **Evaluates to...** |
| \*\* | Exponent | 2 \*\* 3 | 8 |
| % | Modulus | 22 % 8 | 6 |
| // | Integer division | 22 // 8 | 2 |
| / | Division | 22 / 8 | 2.75 |
| \* | Multiplication | 3 \* 5 | 15 |
| - | Subtraction | 5 - 2 | 3 |
| + | Addition | 2 + 3 | 5 |

|  |  |
| --- | --- |
| **Data Type** | **Examples** |
| Integers | -2, -1, 0, 1, 2, 3, 4, 5 |
| Floating-point numbers | -1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25 |
| Strings | 'a', 'aa', 'aaa', 'Hello!', '11 things' |

name = 'ada lovelace'  
print(name.title())

name = 'Ada Lovelace'  
print(name.upper())  
print(name.lower())

message = 'Hello'  
message = 'World!'  
print(message)

message\_1 = 'Hello'  
print(message\_1)

|  |  |
| --- | --- |
| **Augmented Assignment Statement** | **Equivalent Assignment Statement** |
| value += 1 | value = value + 1 |
| value -= 1 | value = value - 1 |
| value \*= 1 | value = value \* 1 |
| value /= 1 | value = value / 1 |
| value %= 1 | value = value % 1 |

**Main**

def main():  
  
if \_\_name\_\_ == "\_\_main\_\_": main()

The Python approach to "main" is almost unique to the language(\*).

The semantics are a bit subtle. The \_\_name\_\_ identifier is bound to the name of any module as it's being imported. However, when a file is being executed then \_\_name\_\_ is set to "\_\_main\_\_" (the literal string: \_\_main\_\_).

This is almost always used to separate the portion of code which should be executed from the portions of code which define functionality.

**Summary: if \_\_name\_\_ == '\_\_main\_\_': has two primary use cases:**

* **Allow a module to provide functionality for import into other code while also providing useful semantics as a standalone script (a command line wrapper around the functionality)**
* **Allow a module to define a suite of unit tests which are stored within the same file as) the code to be tested and which can be executed independently of the rest of the codebase.**

**Lecture note 6 - Lists**

Guideline:

* name of your lists in plural, such as letters, digits, or names

Square brackets ([]) indicate a list, and individual elements in the list are separated by commas.

numbers = [1, 2, 3]  
type(numbers)

// List

words = ['cat', 'bat', 'rat', 'elephant']  
words

// ['cat', 'bat', 'rat', 'elephant']

mixed\_elements = ['hello', 3.1415, True, None, 42]  
mixed\_elements

// ['hello', 3.1415, True, None, 42]

animals = ['cat', 'bat', 'rat', 'elephant']

animals[3]

// ['bat', 'elephant']

animals[0] + 'woman and ' + animals[1] + 'man'

// 'catwoman and batman'

### **Negative Indexes**

animals[-1]

// 'elephant'

animals[-3]

// 'bat'

### **Getting Sublists with Slices**

* animals[2]is a list with an index (one integer)
* animals[1:4] is a list with a slice (two integers)

In a slice, the rst integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, but will not include, the value at the second index. A slice evaluates to a new list value.

animals[1:4]

// ['bat', 'rat', 'elephant']

animals[0:-1]

// ['cat', 'bat', 'rat']

animals[-2:-1]

// ['rat']

As a shortcut, you can leave out one or both of the indexes on either side of the colon in the slice. Leaving out the rst index is the same as using 0, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list.

animals[:3]

// ['cat', 'bat', 'rat']

animals[::2]  
// ['cat', 'rat']

### **Getting a List’s Length with len()**

fst\_sentence = ['Call', 'me', 'Ishmael']  
len(fst\_sentence)  
// 3

### **List Concatenation and List Replication**

* + operator combines two lists to create a new list value
* \* operator can also be used with a list and an integer value to replicate the list

fst\_sentence = ['Call', 'me', 'Ishmael']  
numbers = [1, 2, 3, 4]  
  
concat = fst\_sentence + numbers  
concat

// ['Call', 'me', 'Ishmael', 1, 2, 3, 4]

fst\_sentence \* 3

// ['Call', 'me', 'Ishmael', 'Call', 'me', 'Ishmael', 'Call', 'me', 'Ishmael']

### **Removing Values from Lists with del Statements**

The del statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index.

fst\_sentence = ['Call', 'me', 'Ishmael']  
del fst\_sentence[1]  
fst\_sentence

// ['Call', 'Ishmael']

### **Copying a List**

fst\_sentence = ['Call', 'me', 'Ishmael']  
new\_sentence = fst\_sentence[:]  
new\_sentence[1] = 'him'  
print(fst\_sentence)  
print(new\_sentence)

// ['Call', 'me', 'Ishmael']  
// ['Call', 'him', 'Ishmael']

### **Sorting the Values of a List with sorted()**

The sorted() function returns a sorted **copy** of the corresponding list.

fst\_sentence = ['Call', 'me', 'Ishmael']  
sorted\_sentence = sorted(fst\_sentence)  
print(sorted\_sentence)  
print(fst\_sentence)

// ['Call', 'Ishmael', 'me']  
// ['Call', 'me', 'Ishmael']

**Sequences**

In this course we consider sequences to be "list like". Technically, they are something different but this is beyound the scope of this course.

range returns a sequence of numbers from start to stop by a given step.

range(5)

// range(0, 5)

list(range(5))

// [0, 1, 2, 3, 4]

list(range(3, 10))

// [3, 4, 5, 6, 7, 8, 9]

list(range(3, 20, 4))  
// [3, 7, 11, 15, 19]

list(range(30, 10, -2))

// [30, 28, 26, 24, 22, 20, 18, 16, 14, 12]

### **Simple Statistics with a List of Numbers**

A few Python functions are specific to lists of numbers. For example, you can easily find the minimum, maximum, and sum of a list of numbers.

values = list(range(10))  
print(min(values))  
print(max(values))  
print(sum(values))

// 0

// 9

// 45

## **Methods**

A method is the same thing as a function, except it is “called on” a value. For example, if a list value were stored in fst\_sentence, you would call the index() list method on that list like so: fst\_sentence.index('hello'). The method part comes after the value, separated by a period. Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

### **Adding Values to Lists with the append() and insert() Methods**

To add new values to a list, use the append() and insert() methods. The append() method call adds the argument to the end of the list. The insert() method can insert a value at any index in the list. The first argument to insert() is the index for the new value, and the second argument is the new value to be inserted.

**OBS!** append() and insert() modify the given list inplace. That is, no copy is generated.

fst\_sentence = ['Call', 'me', 'Ishmael']  
fst\_sentence.append('Holgerson')  
fst\_sentence

// ['Call', 'me', 'Ishmael', 'Holgerson']

### **Removing Values from Lists with remove()**

The remove() method is passed the value to be removed from the list it is called on. Attempting to delete a value that does not exist in the list will result in a ValueError error.

fst\_sentence = ['Call', 'me', 'Ishmael']  
fst\_sentence.remove('Ishmael')  
fst\_sentence

// ['Call', 'me']

### **Sorting the Values of a List with the sort() Method**

Lists of number values or lists of strings can be sorted with the sort() method. You can also pass True for the reverse keyword argument to have sort() sort the values in reverse order.

There are three things you should note about the sort() method. First, the sort() method sorts the list in place; do not try to capture the return value by writing code like result = values.sort()

values = [2, 5, 3.14, 1, -7]  
values.sort()  
values

// [-7, 1, 2, 3.14, 5]

**You cannot sort lists that have both number values *and* string values in them, since Python does not know how to compare these values.**

### **Joining a List into a String**

Actually, join is a methond on strings. But you will see it often to join the values of a list into a single string.

fst\_sentence = ['Call', 'me', 'Ishmael']  
(';').join(fst\_sentence)

// 'Call;me;Ishmael'

## **List Comprehensions** A list comprehension allows you to generate arbitrary lists in just one line of code. A list comprehension combines the for loop and the creation of new elements into one line, and automatically appends each new element. List comprehensions are not always presented to beginners, but I have included them here because you’ll most likely see them as soon as you start looking at other people’s code, they may make your code more concise, and they have a better performance than normal loops.

squares = [value\*\*2 for value in range(0, 21)]  
squares

// [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]

# **List-like Types: Strings, Tuples, and Sets**

Lists are not the only data types that represent ordered sequences of values. For example, strings and lists are actually similar, if you consider a string to be a "list" of single text characters. Many of the things you can do with lists can also be done with strings: indexing; slicing; and using them with for loops, with len(), and with the in and not in operators.

## **Mutable and Immutable Data Types**

Lists and strings are different in an important way. A list value is a *mutable* data type: It can have values added, removed, or changed. However, a string is *immutable*: It cannot be changed. Trying to reassign a single character in a string results in a TypeError error.

Consequently, the proper way to "mutate" a string is to use slicing and concatenation to build a new string by copying from parts of the old string.

## **The Tuple Data Type**

The tuple data type is almost identical to the list data type, except in two ways. First, tuples are typed with parentheses, ( and ), instead of square brackets, [ and ].

The main way that tuples are different from lists is that tuples, like strings, are immutable. Tuples cannot have their values modified, appended, or removed.

If you have only one value in your tuple, you can indicate this by placing a trailing comma after the value inside the parentheses. Otherwise, Python will think you’ve just typed a value inside regular parentheses. The comma is what lets Python know this is a tuple value. (Unlike some other programming languages, in Python it is ne to have a trailing comma after the last item in a list or tuple.)

### **Looping Through All Values in a Tuple**

You can loop over all the values in a tuple using a for loop, just as you did with a list.

earnings = ('Moby Dick', 150, 556)  
  
for element in earnings:  
 print(element)

// Moby Dick

// 150

// 556

### **Converting Types with the list() and tuple() Functions**

Just like how str(42) will return '42', the string representation of the integer 42, the functions list() and tuple() will return list and tuple versions of the values passed to them.

## **Sets**

Sets are like lists, only that every element is unique.

set(['Hej', 1, 2, 3, 4, 4, 'Hej',])

// {1, 2, 3, 4, 'Hej'}

**Lecture note 7 - Loops**

# **for Loops**

## **Iterating over Lists and Sequences**

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

// 0, 1, 2, 3

for i in range(4):  
 print(i)

// 0, 1, 2, 3

## **Iterating over Lists and Sequences with Indices**

In case you want to iterate over a list or sequence of values in and you need to have access to the index of each element, you could do the following:

values = range(4, 0, -1)  
for idx in range(len(values)):  
 print(idx, values[idx])

// 0 4

// 1 3

// 2 2

// 3 1

However, **do not do it as shown above**! First, it is more complicated and second, it is not "Pythonic". Instead do the following:

for idx, value in enumerate(range(4, 0, -1)):  
 print(idx, value)

// 0 4

// 1 3

// 2 2

// 3 1

# **while Loops**

The for loop takes a collection of items and executes a block of code once for each item in the collection. In contrast, the while loop runs as long as, or while, a certain condition is True.

current\_number = 0  
while current\_number <= 5:  
 print(current\_number)  
 current\_number += 1

// 0 1 2 3 4 5

## **Checking Lists for Element Containment**

1 in [0, 1, 2, 3]

// true

'Hej' in [0, 1, 2, 3]

// false

**Lecture note 8 - Conditional Statements**

titles = ['moby-dick; or, the whale', 'dracula', 'adventures of huckleberry finn',   
 'the adventures of sherlock holmes', 'alice**\'**s adventures in wonderland']  
  
**for** title **in** titles:  
 **if** 'moby-dick' **in** title:  
 print(title.upper())  
 **else**:  
 print(title.title())

name = 'Ishmael'  
  
*# Check for equality*  
**if** name == 'Ishmael':  
 print('Reading Moby Dick.')  
   
*# Check for inequality*  
**if** name != 'Ishmael':  
 print('Reading something else.')

Operator Meaning

==Equal to

!=Not equal to

<Less than

>Greater than

<=Less than or equal to

>=Greater than or equal to

## **Checking Whether a Value is in a List**

1 **in** [0, 1, 2, 3]

1 **not** **in** [0, 1, 2, 3]

**if** 5 > 4:  
 print('Yep, right!')  
 print('Still, right!')

itle = 'Moby-Dick; or, the Whale'  
  
**if** 'Moby-Dick' **in** title:  
 print('by Herman Melville')  
**else**:  
 print('hmm, I do not know the author.')

year = 1871  
  
**if** year == 1851:  
 message = 'First print.'  
**elif** year == 1855:  
 message = 'Second print.'  
**elif** year == 1863:  
 message = 'Third print.'  
**elif** year == 1871:  
 message = 'Fourth print.'  
**else**:  
 message = 'Hmm, I do not know this year...'  
   
print(message)

authors = ['Herman Melville', 'Arthur Conan Doyle'] *# , 'Mark Twain']*  
  
books\_a = ['Moby Dick; Or, The Whale', 'The Apple-Tree Table and Other Sketches',   
 'Bartleby, the Scrivener: A Story of Wall-Street', 'The Piazza Tales',  
 'The Confidence-Man: His Masquerade']

books\_b = ['The Adventures of Sherlock Holmes', 'A Study in Scarlet',   
 'The Hound of the Baskervilles', 'The Sign of the Four',  
 'The Return of Sherlock Holmes']  
   
books\_c = ['Adventures of Huckleberry Finn', 'The Adventures of Tom Sawyer',  
 'The Mysterious Stranger, and Other Stories',   
 'A Connecticut Yankee in King Arthur**\'**s Court', 'The Innocents Abroad']  
   
library = []  
   
**if** 'Herman Melville' **in** authors:   
 library += books\_a  
**if** 'Arthur Conan Doyle' **in** authors:  
 library += books\_b   
**if** 'Mark Twain' **in** authors:   
 library += books\_c  
  
print('Your library contains:')  
print(library)

library = []  
  
**if** library:  
 print('List is not empty.')  
**else**:  
 print('List is empty.')

**Lecture note 9 - Dictionaries**

# **The Dictionary Data Type**

The dictionary data type provides a flexible way to access and organize data.

Like a list, a dictionary is a collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called keys, and a key with its associated value is called a key-value pair.

In code, a dictionary is typed with Curl-braces, {}.

image = {'color': 'greyscale', 'size': 289983, 'type': 'jpg', 'address': 'illustration.jpg'}

for at få data ud skriver man bare Variablens navn og så navnet på den Key man skal have Value fra.

image[‘color’]

Man kan også tilføje til en Dictionary sådan her:

image[‘someKey’] = ‘SomeValue’

Hvis man vil ændre på en value gør man sådan:

image[‘color’] = ‘red’

Hvis man vil Delete noget bruger man det keyword “ del “

del image[‘color’]

Der er 3 methoder man bruger til Dictionarys, keys() , values() og items()

for key in image.keys():  
 print(key)

for value in image.values():  
 print(value)

for key, value in image.items():  
 print(key)  
 print('\t -' + value)

Checke om en Key eller value er i en Dictionarty, det returnere enten en true eller false.

'color' in image.keys()

‘red’ in image.values()

It is tedious to check whether a key exists in a dictionary before accessing that key’s value. Fortunately, 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.

color\_val = image.get('color', 'unknown')

Der er også en setdefault() method.

You will often have to set a value in a dictionary for a certain key only if that key does not already have a value.

The setdefault() method offers a way to do this in one line of code. The rst argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the setdefault() method returns the key’s value.

Man kan også gemme flere lister i et array, det er ret nemt

article\_images = [dick\_01, dick\_02, dick\_03]

Man kan godt neste en dictionary i en anden dictionary , der gør giver man den første dictionary en key og så value’en laver man til endnu en dictionary.

If you import the pprint module into your programs, you will have access to the pprint() and pformat() functions that will “pretty print” a dictionary’s values. This is helpful when you want a cleaner display of the items in a dictionary than what print() provides.

import pprint  
 pprint.pprint(users)

**Lecture note 10 - Functions and modules**

Python provides several built-in functions like these, but you can also write your own functions. A function is like a mini-program within a program.

If you need to perform that task multiple times throughout your program, you do not need to type all the code for the same task again and again; you just call the function dedicated to handling that task, and the call tells Python to run the code inside the function.

**def** print\_sentence():  
 *"""Display the first sentence of Moby Dick"""*  
   
 fst\_sentence = 'Call me Ishmael.'  
 print(fst\_sentence)  
   
print\_sentence()

**def** modify\_sentence(name):  
 *"""Display a modified first sentence of Moby Dick*  
  
 *:param name: str*  
 *Name to insert in the sentence.*  
 *"""*  
   
 fst\_sentence = 'Call me ' + name + '.'  
 print(fst\_sentence)  
   
modify\_sentence('Ahab')

//Call me Ahab.

**def** apply\_division(a, b):  
 *"""Divide a by b*  
  
 *:param a: number*  
 *Dividend of devision operation.*  
  
 *:param b: number*  
 *Divisor of devision operation.*  
 *"""*  
 result = a / b  
 print(result)  
   
apply\_division(5, 4)

//1.25

**def** apply\_division(dividend, divisor):  
 result = dividend / divisor  
 print(result)  
   
apply\_division(dividend=5, divisor=4)  
apply\_division(divisor=4, dividend=5)

//1.25  
//1.25

**def** apply\_division(dividend, divisor=2):  
 result = dividend / divisor  
 print(result)  
   
apply\_division(5, 4)

//1.25

**def** hire\_crew(\*sailors):  
 *"""Print the list of hired crew members."""*  
   
 **for** sailor **in** sailors:  
 print('- ' + sailor)  
  
hire\_crew('Ahab')  
hire\_crew('Ahab', 'Ishmael', 'Queequeg')

// Ahab  
// Ahab  
// Ishmael  
// Queequeg

**import** **random** **as** **r**  
  
r.choice([1, 2, 3, 4, 5, 6])

//2

**from** **random** **import** choice  
  
choice([1, 2, 3, 4, 5, 6])

//4

**import** **webget**  
  
moby\_dick\_url = 'http://www.gutenberg.org/files/2701/2701-0.txt'  
bones\_in\_london\_url = 'http://www.gutenberg.org/cache/epub/27525/pg27525.txt'  
  
webget.download(moby\_dick\_url)

**Lecture note 11 - CLI Programs (Command line)**

# 

# **Running Programs from the Command Line**

When you call your programs from the CLI, you can pass arguments to it. In the most basic form this looks like:

$ python your\_program.py arg1 arg2

**import** **argparse**  
parser = argparse.ArgumentParser()  
parser.parse\_args()

**Lecture note 12 - Object Oriented Programming**

Object-oriented programming is one of the most common ways to write and structure software. In object-oriented programming you write classes that represent real-world things and situations, and you create objects based on these classes. When you write a class, you define the general behavior that a whole category of objects can have.

A function that’s part of a class is a *method*. Everything you learned about functions applies to methods as well; the only practical difference for now is the way we will call methods.

The \_\_init\_\_() method is a special method Python runs automatically whenever we create a new instance based on the Book class.

**class** **Book**():  
 *"""A simple book model consisting of chapters, which in*   
 *turn consist of paragraphs."""*  
  
 **def** \_\_init\_\_(self, title, author, chapters=[]):  
 *"""Initialize title, the author, and the chapters."""*  
 self.title = title   
 self.author = author  
 self.chapters = chapters   
   
 **def** read(self, chapter=1):  
 *"""Simulate reading a chapter, by calling the reading*   
 *method of a chapter."""*   
 self.chapters[chapter - 1].read()  
   
 **def** open\_book(self, chapter=1 :int) -> Chapter:  
 *"""Simulate opening a book, which returns a chapter*   
 *object."""*   
 **return** self.chapters[chapter - 1]

We define the \_\_init\_\_() method to have four parameters: self, title, author, and chapters=.   
The self parameter is **required**in the method definition, and it must come **first** before the other parameters. It must be included in the definition because when Python calls this \_\_init\_\_() method later (to create an instance of Book), the method call will automatically pass the selfargument. Every method call associated with a class automatically passes self, which is a reference to the instance itself; it gives the individual instance access to the attributes and methods in the class. When we make an instance of Book, Python will call the \_\_init\_\_() method from the Book class. We will pass Book() a title, an author, and chapters= as arguments; self is passed automatically, so we do not need to pass it. Whenever we want to make an instance from theBook class, we will provide values for only the last three parameters.

Think of a class as a set of instructions for how to make an instance. The class Book is a set of instructions that tells Python how to make individual instances representing specific books.

empty\_book = Book('The Empty Book', 'Helge')  
  
print(empty\_book.author)  
print(empty\_book.title)  
print(len(empty\_book.chapters))

//Helge  
The Empty Book  
0

We tell Python to create a book whose title is 'The Empty Book' and whose author is 'Helge'. When Python reads the line, empty\_book = Book('The Empty Book', 'Helge') it calls the \_\_init\_\_() method in Book with the arguments 'The Empty Book' and 'Helge'. The \_\_init\_\_() method creates an instance representing this particular book and sets the author and titleattributes using the values we provided. The \_\_init\_\_() method has no explicit return statement, but Python automatically returns an instance representing this book.

After we create an instance from the class Book, we can use 'dot' notation to call any method defined in Book. To call a method, give the name of the instance (in this case, book) and the method you want to call, separated by a dot. When Python reads book.read(chapter=1), it looks for the method read(chapter) in the class Book and runs that code.

chapter\_1 = Chapter(1, 'Bones and Big Business', get\_text(82, 762))  
chapter\_2 = Chapter(2, 'Hidden Treassure', get\_text(769, 1455))  
book = Book('Bones in London', 'Edgar Wallace', [chapter\_1, chapter\_2])  
  
empty\_book = Book('The Empty Book', 'Helge')  
  
book, empty\_book

you can create as many instances from a class as you need. Even if we used the same title, author, and chapters for the second book, Python would still create a separate instance from the Book class. You can make as many instances from one class as you need, as long as you give each instance a unique variable name or it occupies a unique spot in a list or dictionary.

**class** **Book**():  
 *"""A simple book model consisting of chapters, which in*   
 *turn consist of paragraphs."""*  
  
 **def** \_\_init\_\_(self, title, author, chapters=[]):  
 *"""Initialize title, the author, and the chapters."""*  
 self.title = title   
 self.author = author  
 self.chapters = chapters   
   
 **def** \_\_repr\_\_(self):  
 **return** 'Book(**%r**, **%r**, **%r**)' % (self.title, self.author,   
 self.chapters)  
   
 **def** \_\_str\_\_(self):  
 **return** '**{name}** by **{by}** has **{nr\_chap}** chapters.'.format(  
 name=self.title, by=self.author, nr\_chap=len(self.chapters))  
   
   
 **def** read(self, chapter=1):  
 *"""Simulate reading a chapter, by calling the reading*   
 *method of a chapter."""*   
 self.chapters[chapter - 1].read()  
   
 **def** open\_book(self, chapter=1):  
 *"""Simulate opening a book, which returns a chapter*   
 *object."""*   
 **return** self.chapters[chapter - 1]  
   
  
**class** **Chapter**():  
  
 **def** \_\_init\_\_(self, number, title, paragraphs):  
 *"""A chapter consists of multiple paragraphs."""*  
 self.number = number  
 self.title = title  
 self.paragraphs = []  
 **for** paragraph\_lines **in** paragraphs:  
 new\_pragraph = Paragraph(paragraph\_lines)  
 self.paragraphs.append(new\_pragraph)  
  
 **def** \_\_repr\_\_(self):  
 **return** 'Chapter(**%r**, **%r**, **%r**)' % (self.number, self.title,   
 self.paragraphs)  
   
 **def** read(self, paragraph\_idx=**None**):  
 *"""A paragraph can be read."""*   
 **if** paragraph\_idx:  
 self.paragraphs[paragraph\_idx].read()  
 **else**:  
 **for** paragraph **in** self.paragraphs:  
 paragraph.read()  
   
**class** **Paragraph**():  
 *"""A paragraph consists of a list of lines."""*  
  
 **def** \_\_init\_\_(self, lines):  
 *"""Initialize the paragraph with its lines of text."""*  
 self.lines = lines  
   
 **def** \_\_repr\_\_(self):  
 **return** 'Paragraph(**%r**)' % (self.lines)  
   
 **def** \_\_str\_\_(self):  
 **return** '**{}**...'.format(self.lines[0][0:35])  
  
 **def** read(self):  
 *"""Simulate reading a paragraph by printing its contents."""*   
 **for** line **in** self.lines:  
 print(line)

The \_\_repr\_\_ special method is called by the repr built-in to get string representation of the object for inspection. If we did not implement \_\_repr\_\_, book instances would be shown in the console like <Book at 0x7f43aea894a8>. Note that in our \_\_repr\_\_ implementation we used %r to obtain the standard representation of the attributes to be displayed. This is good practice.  
Contrast \_\_repr\_\_ with \_\_str\_\_, which is called by the str() constructor and **implicitly used by the print function**. \_\_str\_\_should return **a string suitable for display to end-users**. If you only implement one of these special methods, choose \_\_repr\_\_, because when no custom \_\_str\_\_ is available, Python will call \_\_repr\_\_ as a fallback.

**class** **Paragraph**():  
 *"""A paragraph consists of a list of lines."""*  
  
 **def** \_\_init\_\_(self, lines):  
 *"""Initialize name and age attributes."""*  
 self.lines = lines  
 self.reading\_position = 0  
   
 **def** read(self):  
 *"""Simulate reading a paragraph by printing its contents."""*   
 *# for line in self.lines:*  
 *# print(line)*  
 print(self.lines[self.reading\_position])  
   
 **def** get\_reading\_position(self):  
 **return** self.reading\_position

Every attribute in a class needs an initial value, even if that value is 0 or an empty string. In some cases, such as when setting a default value, it makes sense to specify this initial value in the body of the \_\_init\_\_() method; if you do this for an attribute, you do not have to include a parameter for that attribute.

Let’s add an attribute called reading\_position that always starts with a value of 0. We’ll also add a method get\_reading\_position() that returns a reader's current position in the text.

You can change an attribute’s value in three ways:

* you can change the value directly through an instance,
* set the value through a method,
* or increment the value (add a certain amount to it) through a method.

Let’s look at each of these approaches.

**class** **Paragraph**():  
 *"""A paragraph consists of a list of lines."""*  
  
 **def** \_\_init\_\_(self, lines):  
 *"""Initialize name and age attributes."""*  
 self.lines = lines  
 self.reading\_position = 0  
   
 **def** read(self):  
 *"""Simulate reading a paragraph by printing its contents."""*   
 *# for line in self.lines:*  
 *# print(line)*  
 print(self.lines[self.reading\_position])  
  
 **def** get\_reading\_position(self):  
 **return** self.reading\_position  
   
 **def** update\_reading\_position(self, new\_position):  
 **if** new\_position <= len(self.lines) - 1:  
 self.reading\_position = new\_position  
   
   
snd\_paragraph = Paragraph([  
 'Her second husband had begun life at the bottom of the ladder as a',   
 'three-card trickster, and by strict attention to business and the',   
 'exercise of his natural genius, had attained to the proprietorship of a',   
 'bucket-shop.'])

It can be helpful to have methods that update certain attributes for you. Instead of accessing the attribute directly, you pass the new value to a method that handles the updating internally.

We can extend the method update\_reading\_position(self, new\_position) to do additional work every time the reading position is modified. Let’s add a little logic to make sure no one tries to set the reading position beyond a paragraph's limit.

In Python, there is no concept of private fields. That is, the interpreter will not stop you of modifying whatever you are 'touching'.

A good practice is to *mark* your attributes and methodss as private by a leading underscore. However, this is only a visual marker for others using your code that you do not intend them to access the corresponding fields.

ou do not always have to start from scratch when writing a class. If the class you are writing is a specialized version of another -readily available- class, you can use inheritance to implement your extensions.

When one class inherits from another, it automatically takes on all the attributes and methods of the first class. The original class is called the *parent class*, and the new class is the *child class*. The child class inherits every attribute and method from its parent class but is also free to define new attributes and methods of its own.

## **Checking *instance of* Relation Ships**

Sometimes, during development, you would like to know of which class a certain variable is instance of. You can do so with the built-in method isinstance() as illustrated in the following. However, usually you will not need this function in your programs as Python supports *duck-typing*.

# **Importing Classes**

As you add more functionality to your classes, your files can get long, even when you use inheritance properly. In keeping with the overall philosophy of Python, you will want to keep your files as uncluttered as possible. To help, Python lets you store classes in modules and then import the classes you need into your main program.

You can store as many classes as you need in a single module, although each class in a module should be related somehow.

## **Importing a Single Class or Multiple Classes**

We include a module-level docstring that briefly describes the contents of this module. You should write a docstring for each module you create.

Now we make a separate file called book.py. From this file will import the Book class and then create an instance from that class.

You can import as many classes as you need into a program file. If we want to make a regular car and an electric car in the same file, we need to import both classes, Car and ElectricCar

**from** **book** **import** Book

**from** **book** **import** Book, Chapter, Paragraph  
  
chapter\_1 = Chapter(1, 'Bones and Big Business', get\_text(82, 762))  
chapter\_2 = Chapter(2, 'Hidden Treassure', get\_text(769, 1455))  
book = Book('Bones in London', 'Edgar Wallace', [chapter\_1, chapter\_2])  
  
empty\_book = Book('The Empty Book', 'Helge')  
  
empty\_book

You can import every class from a module using the following syntax:

**from** **module\_name** **import** \*

This method is **not** recommended for two reasons:

* It is helpful to be able to read the import statements at the top of a file and get a clear sense of which classes a program uses. With this approach it is unclear which classes you are using from the module. This approach can also lead to confusion with names in the file. If you accidentally import a class with the same name as something else in your program file, you can create errors that are hard to diagnose. I show this here because even though it is not a recommended approach, you are likely to see it in other people’s code.

If you need to import many classes from a module, you are better off importing the entire module and using the module\_name.class\_name syntax.

**Lecture note 13 - Working with files**

# **Reading from a File**

An incredible amount of data is available in text files. Text files can con- tain weather data, traffic data, socioeconomic data, literary works, and more. Reading from a file is particularly useful in data analysis applications, but it’s also applicable to any situation in which you want to analyze or modify information stored in a file. For example, you can write a program that reads in the contents of a text file and rewrites the file with formatting that allows a browser to display it. When you want to work with the information in a text file, the first step is to read the file into memory. You can read the entire contents of a file, or you can work through the file one line at a time.

## **Reading an Entire File**

To begin, we need a file with a few lines of text in it. Let’s start with a file that contains pi to 30 decimal places with 10 decimal places per line:

**import** **webget**  
  
*# Download the file in case we do not have it already*  
url = 'https://raw.githubusercontent.com/ehmatthes/pcc/master/chapter\_10/pi\_30\_digits.txt'   
webget.download(url)  
  
**with** open('pi\_30\_digits.txt') **as** file\_object:  
 contents = file\_object.read()  
 print(contents)

// Downloading file to ./pi\_30\_digits.txt

// 3.1415926535

// 8979323846

// 2643383279

Let’s start by looking at the open() function. To do any work with a file, even just printing its contents, you first need to open the file to access it. The open() function needs one argument: the name of the file you want to open. Python looks for this file in the directory where the program that is currently being executed is stored.

The open() function returns an object representing the file. Here, open('pi\_30\_digits.txt') returns an object representing pi\_30\_digits.txt. Python stores this object in file\_object, which we will work with later in the program.

## **Reading Line by Line**

When you are reading a file, you will often want to examine each line of the file. You might be looking for certain information in the file, or you might want to modify the text in the file in some way.

You can use a for loop on the file object to examine each line from a file one at a time.

**with** open('pi\_30\_digits.txt') **as** file\_object:  
 **for** line **in** file\_object:  
 print(line)

**Lecture note 14 - Plotting**

plt.figure() = figuren

plt. title() = titlen på grafen

plt.xlabel() = label på x-aksen

plt.ylabel = label på y-aksen

plt.axis = målestoksforhold på x og y aksen

plt.plot() = lave grafen (plotting)

plt.xlim() = indstil x grænserne for de aktuelle akser.

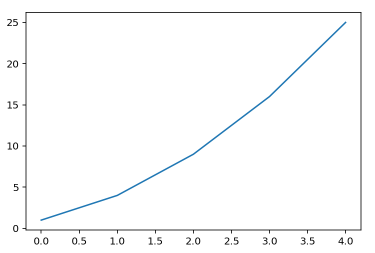
plt.savefig() = gemmer grafen som billedet

# **Plotting Line Graphs**

Let’s plot a simple line graph using matplotlib, and then customize it to create a more informative visualization of our data.

Just provide matplotlib with the numbers as shown below, and matplotlib should do the rest.

**import** **matplotlib.pyplot** **as** **plt**  
%matplotlib notebook  
  
squares = [x \*\* 2 **for** x **in** range(1,6)] *# [1, 4, 9, 16, 25]*  
plt.plot(squares)  
*# in a program you would have to call plt.show()*



**OBS:** The first line %matplotlib inline is needed once per notebook when using Jupyter Notebooks. In case you write stand-alone programs or you are coding in an interactive environment on the CLI, then you do not type this line! Instead, you would call the plt.show() method when you want to show your plot. In the following is an example of a stand-alone program plot\_example.py, generating the exact same plot.

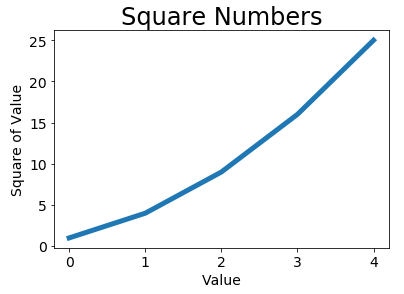
**import** **matplotlib.pyplot** **as** **plt**  
  
  
squares = [x \*\* 2 **for** x **in** range(1,6)] *# [1, 4, 9, 16, 25]*  
plt.plot(squares)  
plt.show()

## **Changing the Label Type and Graph Thickness**

Although the plot shown above illustrates that the numbers are increasing, the label type is too small and the line is too thin. Fortunately, matplotlib allows you to adjust every feature of a visualization.

The linewidth parameter of the plot function controls the thickness of the plotted line. The title() function sets a title for the chart. The fontsize parameters, which appear repeatedly throughout the code, control the size of the text on the chart. The xlabel()and ylabel() functions allow you to set a title for each of the axes, and the function tick\_params() styles the tick marks. The arguments shown here affect the tick marks on both the x- and y-axes (axes='both') and set the font size of the tick mark labels to 14 (labelsize=14).

**import** **matplotlib.pyplot** **as** **plt**  
  
squares = [x \*\* 2 **for** x **in** range(1,6)]  
  
plt.plot(squares, linewidth=5)  
*# Set chart title and label axes.*   
plt.title("Square Numbers", fontsize=24)  
plt.xlabel("Value", fontsize=14)  
plt.ylabel("Square of Value", fontsize=14)  
*# Set size of tick labels.*  
plt.tick\_params(axis='both', labelsize=14)



# **Saving Plots**

If you want your program to save the plot to a file, call plt.savefig():

plt.savefig('filename.png', bbox\_inches='tight')

The first argument is a filename for the plot image, which will be saved in the same directory. The second argument trims extra whitespace from the plot. If you want the extra whitespace around the plot, you can omit this argument.

**Lecture note 16 - Numpy**

# **Initialization of NumPy Arrays**

You can initialize a one-dimensional NumPy array, for example, by passing a list or a tuple to np.array.

**import** **numpy** **as** **np**  
  
  
a = np.array([0, 1, 2, 3, 4])  
b = np.array((0, 1, 2, 3, 4))  
  
print(a)  
print(b)

// [0 1 2 3 4]  
// [0 1 2 3 4]

Alternatively, you can create a one-dimensional NumPy array with the np.arange function, which is similar to its counterpart range in the standard library.

c = np.arange(9, 30, 3)  
print(c)

// [ 9 12 15 18 21 24 27]

## **Indexing Elements**

c =   
print(c)  
print(c[0]) *# Get element at index position 0*  
print(c[1]) *# Get element at index position 1*  
print(c[2:6]) *# Get subarray from index pos. 2 to excluding index pos. 6*  
print(c[1:-1:2]) *# Get every second element from subarray*

// [ 9 12 15 18 21 24 27]

// 9

// 12

// [15 18 21 24]

// [12 18 24]

# **Multi-Dimensional Arrays**

You can initialize multi-dimensional arrays either explicitely as in the following.

two\_dim = np.array([[ 0, 1, 2, 3],  
 [ 4, 5, 6, 7],  
 [ 8, 9, 10, 11]])  
  
three\_dim = np.array([[[ 0, 1, 2, 3],  
 [ 4, 5, 6, 7],  
 [ 8, 9, 10, 11]],  
  
 [[12, 13, 14, 15],  
 [16, 17, 18, 19],  
 [20, 21, 22, 23]],  
  
 [[24, 25, 26, 27],  
 [28, 29, 30, 31],  
 [32, 33, 34, 35]]])

print(two\_dim.shape)  
print(three\_dim.shape)

// (3, 4)

// (3, 3, 4)

# **Why do we care about NumPy?[¶](https://render.githubusercontent.com/view/ipynb?commit=53d0a4614b15b7c90f03e8f6baab6b031d532333&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f646174736f66746c796e6762792f6461743473656d32303138737072696e672d707974686f6e2f353364306134363134623135623763393066303365386636626161623662303331643533323333332f6c6563747572655f6e6f7465732f31362d496e74726f253230746f2532304e756d70792e6970796e62&nwo=datsoftlyngby%2Fdat4sem2018spring-python&path=lecture_notes%2F16-Intro+to+Numpy.ipynb&repository_id=119373481&repository_type=Repository#Why-do-we-care-about-NumPy?)**

Because we want to get things done. As you remember from the last session *14-Intro to Plotting.ipynb*, we had to write quite a bit of code to compute the histograms for the age distribution within Copenhagen citizens per year.

By representing our input data as a matrix, i.e., a two-dimensional array, and using boolean indexing, we can generate for example the histograms way more concisely and with way fewer lines of code.

**Lecture note 17 - Pandas**

# **Pandas for Time Series and Data Frames**

Pandas is -similar to NumPy- another library offering high-level data structures, which enable fast data analyzis. For us, the most important are probably the types Series and DataFrame, both of which are introduced in the following.

## 

## **Series**

A Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index.

**import** **pandas** **as** **pd**  
**import** **numpy** **as** **np**  
**import** **matplotlib.pyplot** **as** **plt**  
  
filename = './KoreanConflict.csv'  
bef\_stats\_df = pd.read\_csv(filename)  
dd = bef\_stats\_df.as\_matrix()  
  
  
*# 4509 MARINE CORPS (12%)*  
*# 29856 ARMY (82%)*  
*# 1552 AIR FORCE (4%)*  
*# 657 NAVY (2%)*  
*# 36574 TOTAL (100%)*  
  
  
*# Q1: How many soldiers entered from a marine corps branch?*  
  
mask = (dd[:,3] == 'MARINE CORPS')  
np.sum(mask)  
  
*# 4509 MARINE CORPS*  
  
*# Data to plot*  
labels = 'MARINE CORPS', 'ARMY', 'AIR FORCE', 'NAVY'  
sizes = [4509, 29856, 1552, 657]  
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']  
explode = (0.1, 0, 0, 0) *# explode 1st slice*  
   
*# Plot*  
plt.pie(sizes, explode=explode, labels=labels, colors=colors,  
 autopct='**%1.1f%%**', shadow=**True**, startangle=140)  
plt.title('Q1: How many soldiers entered from a marine corps branch?', fontsize=20)   
plt.axis('equal')  
fig = plt.gcf()  
fig.set\_size\_inches(10,10)   
plt.show()

**Lecture note 19 - Requests**

Requests will allow you to send HTTP/1.1 requests using Python. With it, you can add content like headers, form data, multipart files, and parameters via simple Python libraries. It also allows you to access the response data of Python in the same way.

import requests

req = requests.get('<http://www.tutsplus.com/>')

req.encoding # returns 'utf-8'

req.status\_code # returns 200

req.elapsed # returns datetime.timedelta(0, 1, 666890)

req.url # returns '<https://tutsplus.com/>'

req.history

# returns [<Response [301]>, <Response [301]>]

req.headers['Content-Type']

# returns 'text/html; charset=utf-8'

**Lecture note 21 - Regular expressions**

*Regular expressions* allow you to specify a pattern of text to search for.

Regular expressions are huge time-savers, not just for software users but also for programmers. In fact, tech writer Cory Doctorow argues that even before teaching programming, we should be teaching regular expressions:

Knowing [regular expressions] can mean the difference between solving a problem in 3 steps and solving it in 3,000 steps. When you’re a nerd, you forget that the problems you solve with a couple keystrokes can take other people days of tedious, error-prone work to slog through.

## **Finding Patterns of Text with Regular Expressions**

Regular expressions, called regexes for short, are descriptions for a pattern of text. For example, a \d in a regex stands for a digit character, that is, any single numeral 0 to 9. The regex \d\d \d\d \d\d \d\d could be used by Python to match a Danish telefon number, a string of eight numbers separated by whitespaces.

But regular expressions can be much more sophisticated. For example, adding a 2 in curly brackets ({2}) after a pattern is like saying, "Match this pattern two times." So the regex \d{2} \d{2} \d{2} \d{2} also matches the correct phone number format. It could be shortened even more to (\d{2} ){3}\d{2}.

### **Creating Regex Objects**

All the regex functions in Python are in the re module.

Passing a string value representing your regular expression to re.compile() returns a Regex pattern object (or simply, a Regex object). Note, since regular expressions frequently use backslashes in them, it is convenient to pass raw strings to the re.compile()function instead of typing extra backslashes. Typing r'\d{2} \d{2} \d{2} \d{2}' is much easier than typing '\\d{2} \\d{2} \\d{2} \\d{2}'.

**import** **re**  
phone\_num\_reg = re.compile(r'\d**{2}** \d**{2}** \d**{2}** \d**{2}**')

### **Matching Regex Objects**

A Regex object’s search() method searches the string it is passed for any matches to the regex. The search() method will return None if the regex pattern is not found in the string. If the pattern is found, the search() method returns a Match object. Match objects have a group() method that will return the actual matched text from the searched string.

**import** **re**  
phone\_num\_reg = re.compile(r'\d**{2}** \d**{2}** \d**{2}** \d**{2}**')  
address\_entry = """Møller   
20 86 46 44   
Herningvej 8  
4800  
Nykøbing F"""  
  
mo = phone\_num\_reg.search(address\_entry)  
mo.group()

// '20 86 46 44'

**Lecture note 25 - OpenCV**

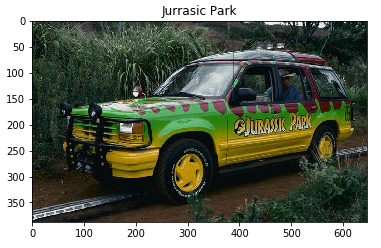
# **Intro to OpenCV**

*OpenCV* (Open Source Computer Vision Library) is an open source computer vision and machine learning software library with more than 2500 optimized algorithms. The library is written in optimized C/C++ and has interfaces for various languages.

## **Reading an Image from Disk**

You can read an image as in the following. Note, that the swap of color channels (cv2.COLOR\_BGR2RGB) is only necessary for inlining a picture with matplotlib.

**import** **cv2**  
**import** **matplotlib.pyplot** **as** **plt**  
  
  
image = cv2.imread('./jurassic-park-tour-jeep.jpg')  
*# the swap of color channels is only necessary for inlining a picture with matplotlib*  
image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)  
plt.title('Jurrasic Park')  
plt.imshow(image, interpolation='none')

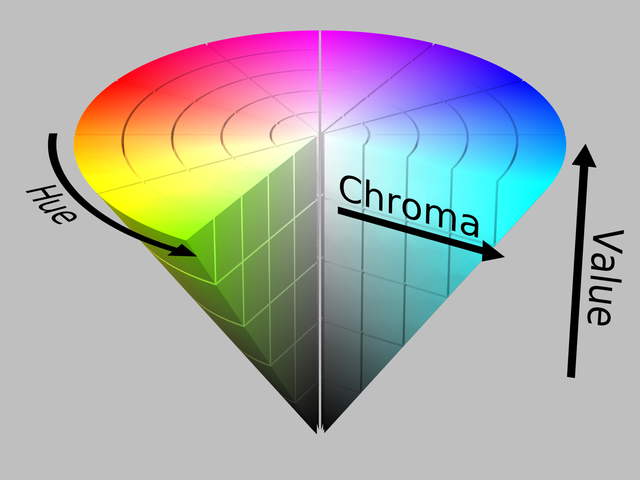


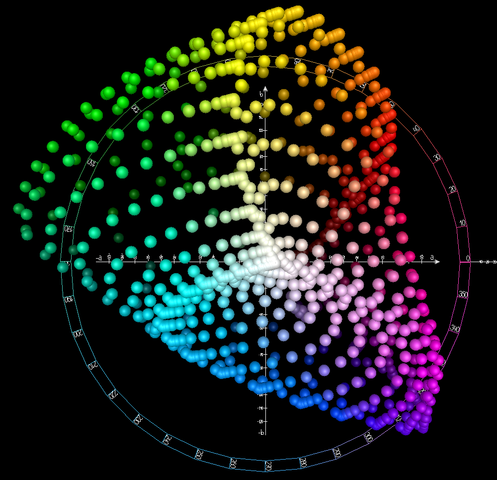
## **Color Spaces**

We have seen the RGB color space so far, where three channels contain the information about how much red, green, and blue contribute to the color of a pixel.

A disadvantage with the RGB color space is, that pixel values change quite much for the same color under different light conditions.

Alternatives to the RGB color space are for example the HSV (Hue, Saturation, Value) color space and the Lab color space, where *L*stands for lightness and *a* and *b* for the color opponents green–red and blue–yellow.





Using the HSV color space, we can find for example the tennis ball quite easily. We create a mask for a certain range of color values, which are characterisitc for a tennis ball.