Python notes:

6] Lists:: []
1)Lists. - **Copy with the Slice Operator**
list1 = ['a','b','c','d']
list2 = list1[:]

Slice copy the list is OK if the list copying do not have sublists.

2. For sublists: Deep Copy module

Using the Method Deepcopy from the Module copy

A solution to the described problems are provided by the module "copy". This module provides the method "deepcopy", which allows a complete or deep copy of an arbitrary list, i.e. shallow and other lists.
Let's use deepcopy for our previous list:

from copy import deepcopy
lst1 = ['a','b',['ab','ba']]
lst2 = deepcopy(lst1)
Changing the sublist element of one list won't change the other list.

7] Dictionaries:: {}
1)More theoretically, we can say that dictionaries are the Python implementation of an abstract data type, known in computer science as an associative array. Associative arrays consist - like dictionaries of (key, value) pairs, such that each possible key appears at most once in the collection. Any key of the dictionary is associated (or mapped) to a value. The values of a dictionary can be any type of Python data. So, dictionaries are unordered key-value-pairs. Dictionaries are implemented as hash tables, and that is the reason why they are known as "Hashes" in the programming language Perl.

2)it is not possible to access an element of the dictionary by a number, like we did with lists:

city_population[0] ==> key error

Values of two keys can be same but keys are unique.
Keys of a dictionary are unique. In casse a keys is defined multiple times, the value of the last "wins":

3)We can use arbitrary types as values in a dictionary, but there is a restriction for the keys. Only immutable data types can be used as keys, i.e. no lists or dictionaries can be used: If you use a mutable data type as a key, you get an error message:

```
dic = { [1,2,3]: "abc"}

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-af2a40fe8efa> in <module>
----> 1 dic = { [1,2,3]:"abc"}

TypeError: unhashable type: 'list'
```

4)Tuple as keys are okay, as you can see in the following example:

```
dic = { (1,2,3):"abc", 3.1415:"abc"}
dic
Output::
{(1, 2, 3): 'abc', 3.1415: 'abc'}
```

5)**Operators in Dictionaries**

Operator	Explanation
len(d)	returns the number of stored entries, i.e. the number of (key,value) pairs.
del d[k]	deletes the key k together with his value
k in d	True, if a key k exists in the dictionary d
k not in d	True, if a key k doesn't exist in the dictionary d

6)Accessing no existing keys gives keyerrok.
To prevent this: You can prevent this by using the "in" operator:

```
province = "Ottawa"

if province in locations:
    print(locations[province])
else:
    print(province + " is not in locations")
```

Ottawa is not in locations

7)Another method to access the values via the key consists in using the get() method. get() is not raising an error, if an index doesn't exist. In this case it will return None. It's also possible to set a default value, which will be returned, if an index doesn't exist:

8)A dictionary can be copied with the method copy():
copy()

```
words = {'house': 'Haus', 'cat': 'Katze'}
w = words.copy()
words["cat"]="chat"
print(w)
```

{'house': 'Haus', 'cat': 'Katz
This copy is a shallow copy, not a deep copy. If a value is a complex data type like a list, for example, in-place changes in this object have effects on the copy as well:

9)**clear()**
The content of a dictionary can be cleared with the method clear(). The dictionary is not deleted, but set to an empty dictionary:

```
w.clear()
```

10)**Update: Merging Dictionaries**
What about concatenating dictionaries, like we did with lists? There is someting similar for dictionaries: the update method update() merges the keys and values of

one dictionary into another, overwriting values of the same key:

knowledge = {"Frank": {"Perl"}, "Monica":{"C","C++"}}
knowledge2 = {"Guido":{"Python"}, "Frank":{"Perl", "Python"}}
knowledge.update(knowledge2
It will not update the duplicate key:value

11)**Iterating over a Dictionary ::**

```
for key in d:
    print(key)

for value in d.values():
```

12)Making lists from dictionaries:
t's possible to create lists from dictionaries by using the methods items(), keys() and values(). As the name implies the method keys() creates a list, which consists solely of the keys of the dictionary. values() produces a list consisting of the values. items() can be used to create a list consisting of 2-tuples of (key,value)-pairs:

13)Making dictionaries from lists::
we need the function zip().

```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"]
country_specialities_iterator = zip(countries, dishes)
```

Using dict(zip([][])) ==> converts 2 lists to one dictionary

8] Lists:-
1)set is defincd as a function set with ()

In the following example, a string is singularized into its characters to build the resulting set x:

```
x = set("A Python Tutorial")
```

2)Sets dont allow elements that mutable.
Sets are implemented in a way, which doesn't allow mutable objects. The following

example demonstrates that we cannot include, for example, lists as elements:

3)Frozensets are like sets except that they cannot be changed, i.e. they are immutable:

```
cities = frozenset(["Frankfurt", "Basel","Freiburg"])
cities.add("Strasbourg")
```

4)Set Operations::
add(element)
clear()
copy
Creates a shallow copy, which is returned.
difference()
This method returns the difference of two or more sets as a new set.
pop()
pop() removes and returns an arbitrary set element. The method raises a KeyError if the set is empty

9] input::

1)The input of the user will be returned as a string without any changes. If this raw input has to be transformed into another data type needed by the algorithm, we can use either a casting function or the eval function.

```
age = input("Your age? ")
print("So, you are already " + age + " years old, " + name + "!")
```

2. **You can find data type of the input as:**

```
cities_canada = input("Largest cities in Canada: ")
print(cities_canada, type(cities_canada))
```

```
cities_canada = eval(input("Largest cities in Canada: "))
print(cities_canada, type(cities_canada))
```

10] conditional statements:

```
if condition_1:
    statement_block_1
elif condition_2:
```

```
    statement_block_2

...

elif another_condition:
    another_statement_block
else:
    else_block
```

11] Loops:
Python supplies two different kinds of loops: the while loop and the for loop, which correspond to the condition-controlled loop and collection-controlled loop.

1)simple while loop
While (condition):
execute

2)While loop with else part:
3) while loop with else and break statement

12]For Loop::
1)the Python for loop is an iterator based for loop. It steps through the items of lists, tuples, strings, the keys of dictionaries and other iterables. The Python for loop starts with the keyword "for" followed by an arbitrary variable name, which will hold the values of the following sequence object, which is stepped through. The general syntax looks like this:

for in :

```
languages = ["C", "C++", "Perl", "Python"]
for language in languages:
    print(language)
```

2)for in :else with break - If a break statement has to be executed in the program flow of the for loop, the loop will be exited and the program flow will continue with the first statement following the for loop, if there is any at all.Usually break statements are wrapped into conditional statements,

```
edibles = ["bacon", "spam", "eggs", "nuts"]
```

```
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

3). For in : with continue -

```
edibles = ["bacon", "spam", "eggs","nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        continue
    print("Great, delicious " + food)

print("Finally, I finished stuffing myself")
```

4)for in with The range() function

The built-in function range() is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions:

```
for i in range(5):
    print(i)
```

==>range(n) generates an iterator to progress the integer numbers starting with 0 and ending with (n -1).

5)range() can also be called with two arguments:
range(begin, end)

6)range - with start, end and step -range(begin,end, step)

Example with step:

```
list(range(4, 50, 5))
```

Output::
[4, 9, 14, 19, 24, 29, 34, 39, 44, 49]

7)Iterating over Lists with range()

f you have to access the indices of a list, it doesn't seem to be a good idea to use the for loop to iterate over the lists. We can access all the elements, but the index of an element is not available. However, there is a way to access both the index of an element and the element itself. The solution lies in using range() in combination with the length function len():

```
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
for i in range(len(fibonacci)):
    print(i,fibonacci[i])
print()
```

13] Iterators and iterables:
1)we can loop or iterate over various Python objects like lists, tuples and strings using **for in**

2)Y**ou can iterate with a for loop over iterators and iterables. Every iterator is also an iterable, but not every iterable is an iterator. E.g. a list is iterable but a list is not an iterator! An iterator can be created from an iterable by using the function 'iter'. To make this possible the class of an object needs either a method 'iter', which returns an iterator, or a 'getitem' method with sequential indexes starting with 0.**

Iterators are objects with a '**next**' method, which will be used when the function 'next' is called.

3)The loop executes until StopIteration exception, if there are no further elements available

4. Example of utterable objects ae list, set, tuples, stringshe following function 'iterable' will return True, if the object 'obj' is an iterable and False otherwise.

```
def iterable(obj):
    try:
        iter(obj)
        return True
    except TypeError:
```

```
        return False

for element in [34, [4, 5], (4, 5), {"a":4}, "dfsdf", 4.5]:
    print(element, "iterable: ", iterable(element))
```

14]print function - print is a function

1)print Function

The arguments of the print function are the following ones:

```
 print(value1, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

The print function can print an arbitrary number of values ("value1, value2, ..."), which are separated by commas. These values are separated by blanks. In the following example, we can see two print calls. We are printing two values in both cases, i.e. a string and a float number:

```
a =  3.564
print("a = ", a)
```

2. **Use of Sep -keyword in print function-**It's possible to redefine the seperator between values by assigning an arbitrary string to the keyword parameter "sep", i.e. an empty string or a smiley:

```
print("a","b",sep=":-)")
```

```
a:-)b
```

15]formated output::
Format method to print nicer output

1)Using Local Variable Names in "format"

```
print("a={a}, b={b} and f={f}".format(**locals()))
```

2)formate string literals

Formatted String Literals

Python 3.6 introduces formatted string literals. They are prefixed with an 'f'. The formatting syntax is similar to the format strings accepted by str.format(). Like the format string of format method, they contain replacement fields formed with curly braces. The replacement fields are expressions, which are evaluated at run time,

and then formatted using the format() protocol. It's easiest to understand by looking at the following examples:

price = 11.23
f"Price in Euro: **{price}**"
Output::
'Price in Euro: 11.23'

16] Functions::
1)def function-name(Parameter list):
    statements, i.e. the function body

The mechanism for assigning arguments to parameters is called **argument passing**.

2. **Default Arguments: Default arguments in Python**
When we define a Python function, we can set a default value to a parameter. If the function is called without the argument, this default value will be assigned to the parameter. This makes a parameter optional. To say it in other words: Default parameters are parameters, which don't have to be given, if the function is called. In this case, the default values are used.

But it is very important to be careful passing mutable objects such as dictionaries as a default arguments because Default values will not be created when a function is called. Default values are created exactly once, when the function is defined, i.e. at compile-time.

The solution consists in using the immutable value None as the default. This way, the function can set dynamically at run-time bag to an empty list:

def spammer(bag=**None**): instead of **def** spammer(bag=**[]**):

3)doctoring :: -The first statement in the body of a function is usually a string, which can be accessed with function_name.**doc** This statement is called Docstring. Example:

def hello(name="everybody"):
    """ Greets a person """

4)Return Values - function without a return statstement returns a None
Let's summarize this behavior: Function bodies can contain one or more return statement. They can be situated anywhere in the function body. A return statement ends the execution of the function call and "returns" the result, i.e. the

value of the expression following the return keyword, to the caller. If the return statement is without an expression, the special value None is returned. If there is no return statement in the function code, the function ends, when the control flow reaches the end of the function body and the value None will be returned.

5)Returning multiple values. - function can return exactly one object so to return multiple values say 3 integers - return in a object like a list or dictionary or tuple.

Return of the function can be one of these. - 1 integer, 1 float, 1 string or 1 list , 1 dict.....

6)local and global variables in function:: global keyword inside the function makes the variable global and anything we do to s inside of the function body of f is done to the global variable s outside of f.

```
def f():
    global s
    print(s)
    s = "dog"
    print(s)
s = "cat"
f()
print(s)
```

7)Arbitrary Number of Parameters

An arbitrary parameter number can be accomplished in Python with so-called tuple references. An asterisk "*" is used in front of the last parameter name to denote it as a tuple reference.

Example :

```
def arithmetic_mean(first, *values):
    """ This function calculates the arithmetic mean of a non-empty
        arbitrary number of numerical values """

    return (first + sum(values)) / (1 + len(values))
```

Values passed can be any no of values.

8. **Positional Vs Keyword arguments:**

Positional arguments passed to the functions as values in the order
e.g. def sum(3,5):
3 is assigned to the first variable and 5 to the second in the function

Keyword argument - can be passed in any order.
e.g. def sum(real=3, img=5): can be passed as def sum(img=5,real=3):

9)Arbitrary Number of Keyword Parameters::
is also possible to pass an arbitrary number of keyword parameters to a function. To this purpose, we have to use the double asterisk "**"

```
def f(**kwargs):
    print(kwargs)
```

E.g. keyword_dict = {'real': 3, 'imag': 5}
complex(**keyword_dict)

17]Recursive function - that repeats and has a base case that has a condition to terminate.

Recursion is a method of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition satisfies the condition of recursion, we call this function a recursive function.

Termination condition: A recursive function has to fulfil an important condition to be used in a program: it has to terminate. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case. A base case is a case, where the problem can be solved without further recursion. A recursion can end up in an infinite loop, if the base case is not met in the calls.

18]Parameters and Arguments:: functions have parameters and the passing to the function are arguments

1) When the mutable objects are passed as parameters and the function has the statements to alter like += or -= it can alter the object and there are side effects of it. Instead a good practice is to pass the arguments as a shallow copy e.g. pass a list names cities to a function as a shallow copy e.g test(cities[:]) rather than test(cities)

2)Command Line Arguments

If you call a Python script from a shell, the arguments are placed after the script name. The arguments are separated by spaces. Inside the script these arguments are accessible through the list variable sys.argv. The name of the script is included in this list sys.argv[0]. sys.argv[1] contains the first parameter, sys.argv[2] the second and so on. The following script (arguments.py

Import sys

```
# Module sys has to be imported:
import sys
```

Iteration over all arguments:

```
for eachArg in sys.argv:
        print(eachArg)
```

Example call to this script:
python argumente.py python course for beginners
This call creates the following output:
argumente.py
python
course
for
beginners

3. **Variable length of parameters::**

A function with an arbitrary number of arguments is usually called a variadic function in computer science.

```
def varpafu(*x):
```

Passing variable parameters to this func=tion :-

varpafu(34,"Do you like Python?", "Of course")

Variable x stores it in tuple.

Passing empty arguments to the function will be a empty tuple x.

4)*** in Function Calls**
A * can appear in function calls as well, as we have just seen in the previous exercise: The semantics is in this case "inverse" to a star in a function definition. An argument will be unpacked and not packed. In other words, the elements of the list or tuple are singularized:

```
e.g. def f(x,y,z):
    print(x,y,z)

p = (47,11,12)
f(*p)
```

47 11 12

5)Arbitrary Keyword Parameters
There is also a mechanism for an arbitrary number of keyword parameters. To do this, we use the double asterisk "**" notation:

```
>>> def f(**args):
    print(args)

f()

{}

f(de="German",en="English",fr="French")

{'de': 'German', 'en': 'English', 'fr': 'French'}
```

6)Double Asterisk in Function Calls
The following example demonstrates the usage of ** in a function call:

```
def f(a,b,x,y):
    print(a,b,x,y)

d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
```

```
f(**d)
```

append block extract yes

and now in combination with *:

```
t = (47,11)
d = {'x':'extract','y':'yes'}
f(*t, **d)
```

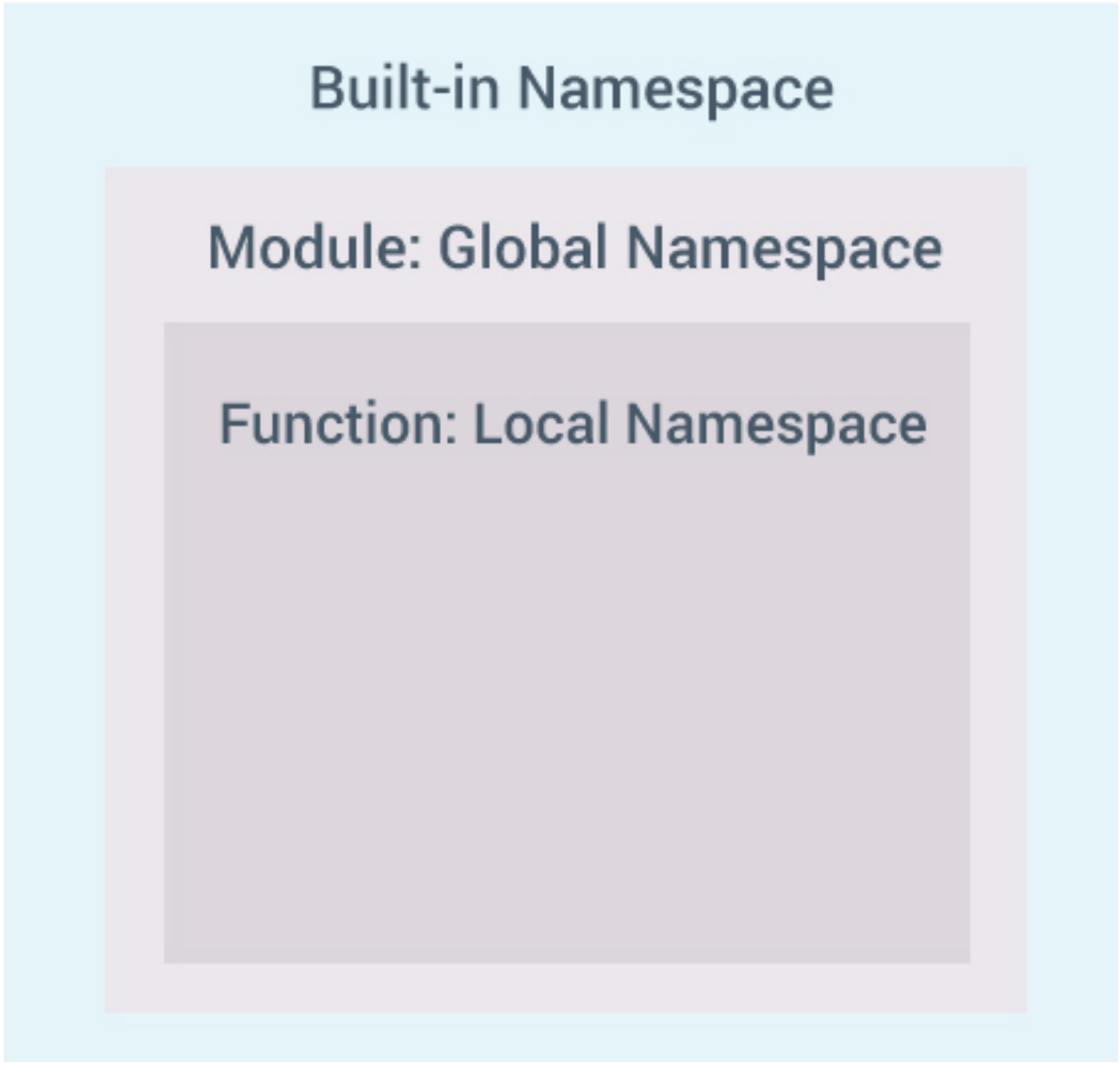
47 11 extract yes

19]Namespaces::

In Python, you can imagine a namespace as a mapping of every name you have defined to corresponding objects.

Scope of namespaces varies.

A namespace containing all the built-in names is created when we start the Python interpreter and exists as long as the interpreter runs.
This is the reason that built-in functions like id(), print() etc. are always available to us from any part of the program. Each module creates its own global namespace.



20]Global and local Variables in Functions:
To use a global variable inside function declare using
As **global s**

Nonlocal variables in function is declared as nonlocal x

21]zip function. - it takes 2 mutable objects like lists and returns the object that is utterable.

1) e.g. 2 lists if zipped will have an object that is an iterator.

```
a_couple_of_letters = ["a", "b", "c", "d", "e", "f"]
some_numbers = [5, 3, 7, 9, 11, 2]

print(zip(a_couple_of_letters, some_numbers))
```

```
<zip object at 0x00000216F4527588>
```

Use of zip is to get the tuple that will combine the 2 lists.
he application of zip returns an iterator, which is capable of producing tuples. It is combining the first items of each iterable (in our example lists) into a tuple, after this it combines the second items and so on. It stops when one of them is exhausted, i.e. there are no more items available.

The best way to see what it creates is to use it in a for loop.

```
for t in zip(a_couple_of_letters, some_numbers):
    print(t)
```

```
('a', 5)
('b', 3)
('c', 7)
('d', 9)
('e', 11)
('f', 2)
```

2)zip can have an arbitrary number of iterable arguments :

```
location = ["Helgoland", "Kiel",
            "Berlin-Tegel", "Konstanz",
            "Hohenpeißenberg"]
air_pressure = [1021.2, 1019.9, 1023.7, 1023.1, 1027.7]
temperatures = [6.0, 4.3, 2.7, -1.4, -4.4]
altitude = [4, 27, 37, 443, 977]

for t in zip(location, air_pressure, temperatures, altitude):
    print(t)
```

```
('Helgoland', 1021.2, 6.0, 4)
('Kiel', 1019.9, 4.3, 27)
('Berlin-Tegel', 1023.7, 2.7, 37)
('Konstanz', 1023.1, -1.4, 443)
('Hohenpeißenberg', 1027.7, -4.4, 977)
```

3)The use of zip is not restricted to lists and tuples. It can be applied to all iterable objects like lists, tuples, strings, dictionaries, sets, range and many more of course.

```
food = ["ham", "spam", "cheese"]

for item in zip(range(1000, 1003), food):
    print(item)
```

```
(1000, 'ham')
(1001, 'spam')
(1002, 'cheese')
```

4)zip with Parameters with Different Lengths:
zip can be called with an arbitrary number of iterable objects as arguments. So far the number of elements or the length of these iterables had been the same. This is not necessary. If the Lengths are different, zip will stop producing an output as soon as one of the argument sequences is exhausted. "stop producing an output" actually means it will raise a StopIteration exception like all other iterators do.

5)zip can be used to unpack one list that has tuples into 2 tuples - reverse of example2 - use zip with * to unpack the list into 2 tuples.
E.g.
e have a list with the six largest cities in Switzerland. It consists of tuples with the pairs, city and population number:

```
cities_and_population = [("Zurich", 415367),
                         ("Geneva", 201818),
                         ("Basel", 177654),
                         ("Lausanne", 139111),
                         ("Bern", 133883),
                         ("Winterthur", 111851)]
```

The task consists of creating two lists: One with the city names and one with the population numbers. zip is the solution to this problem, but we also have to use the star operator to unpack the list:

```
cities, populations = list(zip(*cities_and_population)) #unpack
print(cities)
print(populations)
```

```
('Zurich', 'Geneva', 'Basel', 'Lausanne', 'Bern', 'Winterthur')
```

(415367, 201818, 177654, 139111, 133883, 111851)

6)Converting two Iterables into a Dictionary
zip also offers us an excellent opportunity to convert two iterables into a dictionary. Of course, only if these iterables meet certain requirements. The iterable, which should be used as keys, must be unique and can only consist of immutables. We demonstrate this with the following morse code example.

e.g.
abc = "abcdef". #string is itterable and immutable that can be used as a key
morse_chars = [".-", "-...", "-.-.", "-..", ".", "..-."]. #List is iterable

text2morse = dict(zip(abc, morse_chars)) #zip can give a dictionary from 2 iterables and key as immutable and iterable
print(text2morse)

Output::
{'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.', 'f': '..-.'}

22]Decorators::

1. **Two types of decorators:**

Even though it is the same underlying concept, we have two different kinds of decorators in Python:

- Function decorators
- Class decorators

A decorator in Python is any callable Python object that is used to modify a function or a class. A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class. The modified functions or classes usually contain calls to the original function "func" or class "C".

2)Functions as Parameters:: function call another function —>
the fact that every parameter of a function is a reference to an object and functions are objects as well, we can pass functions - or better "references to functions" - as parameters to a function.

The 'real' name of func is access by function.__name__, as it contains this name:

def g():

```
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()
    print("func's real name is " + func.__name__)  # reference to the g


f(g)
```

```
Hi, it's me 'f'
I will call 'func' now
Hi, it's me 'g'
Thanks for calling me
func's real name is g
```

3)Functions returning Functions : The output of a function is also a reference to an object. Therefore functions can return references to function objects.

Use Cases for Decorators
Checking Arguments with a Decorator
Recursive function factorial can be modified to check negative argument and raise exception before calling factorial.
E.g.

```
def argument_test_natural_number(f):
    def helper(x):
        if type(x) == int and x > 0:
            return f(x)
        else:
            raise Exception("Argument is not an integer")
    return helper

@argument_test_natural_number
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

for i in range(1,10):
```

```
    print(i, factorial(i))
```

print(factorial(-1)). ==>calls decorator first argument_test_natural_no and decorator calls factorial if number is not negative . In this case factorial is not called and exception is raised.

4)Function Calls with Decorators:

decorators for functions can be with exactly one parameter:

Can use the *args and **kwargs notation to write decorators which can cope with functions with an arbitrary number of positional and keyword parameters.

```
def call_counter(func):
    def helper(*args, **kwargs):
        helper.calls += 1
        return func(*args, **kwargs)
    helper.calls = 0

    return helper

@call_counter
def succ(x):
    return x + 1

@call_counter
def mul1(x, y=1):
    return x*y + 1

print(succ.calls)
for i in range(10):
    succ(i)
mul1(3, 4)
mul1(4)
mul1(y=3, x=2)

print(succ.calls)
print(mul1.calls)
```

5)decorators with parameters:

@greetings(“adasdawewqe”)

5)Using wraps from functools

The attributes

- __name__ (name of the function),
- __doc__ (the docstring) and
- __module__ (The module in which the function is defined)

of the original functions can be saved by a) importing wraps from functools module and b) decorating the functionbwith wrap. c) also its a common practice to have the decorator in a separate file.
E.g.
from greeting_decorator_manually import greeting

Fortunately, we don't have to add all this code to our decorators to have these results. We can import the decorator "wraps" from functools instead and decorate our function in the decorator with it:

```
from functools import wraps

def greeting(func):
    @wraps(func)
    def function_wrapper(x):
        """ function_wrapper of greeting """
        print("Hi, " + func.__name__ + " returns:")
        return func(x)
    return function_wrapper
```

6)class can be used as decorator —> by definition decorator is simply a callable object that takes a function as an input parameter.
Class has a callable method __call__ which is called when the object is instantiated . For that reason class can be a callable object and used as a decorator.
Example of class as a decorator ::
lass decorator2:

```
    def __init__(self, f):
        self.f = f

    def __call__(self):
        print("Decorating", self.f.__name__)
        self.f()
```

```
@decorator2
def foo():
    print("inside foo()")

foo()
```

Decorating foo

23]Memoization with Decorators:
We can have a function to save repeated data and use this as a decorator for the function that calculates the repetitive data. e.g. fibonci no.

24]File Management::

25]Modular Programming and Modules:
every file, which has the file extension .py and consists of proper Python code, can be seen or is a module!

1)Importing Modules : import math
2)import more than one module names - import math, random
3)import only certain objects for the module
from math import sin, pi. ==> can be used sin instead of math.sin
4) to import everything in the namespace of the importing module. :
from math import *
sin(3.01) + tan(cos(2.1)) + e
5)A module in Python is just a file containing Python definitions and statements. To get the fibooncci module just save the fibboncci code in the file fibonacci.py,. Now the fibbonci module can be imported to other modules.
Import fibboncci
6)Each module can only be imported once per interpreter session or in a program or script. If you change a module and if you want to reload it, you must restart the interpreter again.
7)Names from a module can directly be imported into the importing module's symbol table:
from fibonacci import fib, ifib
8)Executing Modules as Scripts:
Essentially a Python module is a script, so it can be run as a script:
python fibo.py

The module which has been started as a script will be executed as if it had been imported, but with one exception: The system variable **name** is set to "**main**".

if __name__ == "__main__":

9)Renaming a Namespace
While importing a module, the name of the namespace can be changed:

import math as mathematics

10)Kinds of Modules:
- It's possible to get a complete list of these modules:

import sys
print(sys.builtin_module_names)

11)Module Search Path:
If you import a module, let's say "import xyz", the interpreter searches for this module in the following locations and in the order given:
The directory of the top-level file, i.e. the file being executed.
The directories of PYTHONPATH, if this global environment variable of your operating system is set.
standard installation path Linux/Unix e.g. in /usr/lib/python3.5.

It's possible to find out where a module is located after it has been imported:

import numpy
numpy.**file**
'/usr/lib/python3/dist-packages/numpy/**init**.py'

11)Content of a Module: dir(math)
With the built-in function dir() and the name of the module as an argument, you can list all valid attributes and methods for that module.

import math
dir(math)

12) Calling dir() without an argument, a list with the names in the current local scope is returned: - dir()

import math

cities = ["New York", "Toronto", "Berlin", "Washington", "Amsterdam", "Hamburg"]
dir()

13) It's possible to get a list of the Built-in functions, exceptions, and other objects by importing the builtins module: dir(builtins)

import builtins
dir(builtins)

26] Packages ::
A package is basically a directory with Python files and a file with the name __init__.py. This means that every directory inside of the Python path, which contains a file named __init__.py, will be treated as a package by Python. It's possible to put several modules into a Package.

1. **Creating a package manually - put all the modules in the same directory say test. Test dir has a and b modules (py files a.py and b.py) and a file named as __intit__.py. __inti__.py can be empty or have info about importing a and b modules.**

If __init__.py is empty —> from test import a , b
If __init__.py has ==> import test.a
Import test.b
Then import test would import both a and b

2)Python provides a mechanism to give an explicit index of the subpackages and modules of a packages, which should be imported. For this purpose, we can define a list named __all__. This list will be taken as the list of module and package names to be imported when from package import * is encountered.

We will add now the line
__all__ = ["formats", "filters", "effects", "foobar"]
to the __init__.py file of the sound directory. We get a completely different result now

27] Regular Expressions::
1)Simple Regular Expression :: import the re module, which provides methods and functions to deal with regular
expressions.
checks a string s for an occurrence of a substring which matches the regular expression expr. The first substring (from left), which satisfies this condition will

be returned. If a match has been possible, we get a so-called match object as a result, otherwise the value will be None
Import re

2)Syntax of Regular Expression— re.search(expr,s)

x = re.search("cat", "A cat and a rat can't be friends.")
print(x)

<re.Match object; span=(2, 5), match='cat'>

3)Character Classes:

Square brackets, "[" and "]", are used to include a character class. [xyz] means e.g. either an "x", an "y" or a "z"

r"M[ae][iy]er". +> firstletter M , second letter a or e ; third lerre I or y and ends with er

"any uppercase letter" into regular expressions. {A-Z]

"any lower case or uppercase letter" [A-Za-z]

the expression [-az] is only the choice between the three characters "-", "a" and "z"

other special character inside square brackets (character class choice) is the caret "^" is to negate. [^0-9] denotes the choice "any character but a digit"

[^abc] means anything but an "a", "b" or "c" [a^bc] means an "a", "b", "c" or a "

4)Predefined Character Classes:
The special sequences consist of "\\" and a character from the following list:
\d Matches any decimal digit; equivalent to the set [0-9].
\D The complement of \d. It matches any non-digit character; equivalent to the set [^0-9].
\s Matches any whitespace character; equivalent to [\t\n\r\f\v].
\S The complement of \s. It matches any non-whitespace character; equiv. to [^ \t\n\r\f\v].
\w Matches any alphanumeric character; equivalent to [a-zA-Z0-9_]. With LOCALE, it will match the set [a-zA-Z0-9_] plus characters defined as letters for the current locale.
\W Matches the complement of \w.

\b Matches the empty string, but only at the start or end of a word.
\B Matches the empty string, but not at the start or end of a word.
\\ Matches a literal backslash.

the dollar sign ($), which is used to mark the end of a string,

5)Matching Beginning and End

Beginning - **match(re_str, s)** checks for a match of re_str merely at the beginning of the string
print(re.match(r"M[ae][iy]er", s1))

6)Optional Items - with ? In search - preceding char may or may not occur
e.g
e is optional
r"M[ae][iy]e?r"

6)Quantifiers
 * — r″[0-9]*″. ->A star following a character or a subexpression group means that this expression or character may be repeated arbitrarily, even zero times.

7)Grouping: We can group a part of a regular expression by surrounding it with parenthesis (round brackets). This way we can apply operators to the complete group instead of a single character.

8)A Closer Look at the Match Objects:
 the re.search() returns a match object if it matches and None otherwise . The match object contains lot of data, about what has been matched, positions etc.
A match object contains the methods group(), span(), start() and end(),.
span() returns a tuple with the start and end position, i.e. the string index where the regular expression started matching in the string and ended matching. The methods start() and end() are in a way superfluous as the information is contained in span(), i.e. span()[0] is equal to start() and span()[1] is equal to end(). group(), if called without argument, it returns the substring, which had been matched by the complete regular expression.

mo = re.search("[0-9]+", "Customer number: 232454, Date: February 12, 2011")
mo.group()
Output::
'232454'

mo.span()

Output::
(17, 23)

9)Back References:
If there are more than one pair of parenthesis (round brackets) inside the expression, the backreferences are numbered \1, \2, \3, in the order of the pairs of parenthesis.

```
import re

l = ["555-8396 Neu, Allison",
     "Burns, C. Montgomery",
     "555-5299 Putz, Lionel",
     "555-7334 Simpson, Homer Jay"]

for i in l:
    res = re.search(r"([0-9-]*)\s*([A-Za-z]+),\s+(.*)", i)
  print(res.group(3)) ==> will print the first names with \s of last ().
```

10)Named Backreferences — the back references can have the meaningful name for the group.

The syntax for a named group is one of the Python-specific extensions: (?P<name>...). name is, obviously, the name of the group. Named groups behave exactly like capturing groups, and additionally associate a name with a group.

11)Finding all Matched Substrings :
re.findall(pattern, string[, flags])
If one or more groups are present in the pattern, findall returns a list of groups. This will be a list of tuples if the pattern has more than one group

We have a long string with various Python training courses and their dates. With the first call to findall, we don't use any grouping and receive the complete string as a result. In the next call, we use grouping and findall returns a list of 2-tuples, each having the course name as the first component and the dates as the second component:

```
import re
courses = "Python Training Course for Beginners: 15/Aug/2011 - 19/Aug/
2011;Python Training Course Intermediate: 12/Dec/2011 - 16/Dec/2011;Python Text
Processing Course:31/Oct/2011 - 4/Nov/2011"
items = re.findall("[^:]*:[^;]*;?", courses). ==> pattern has no group
```

items
Output::
['Python Training Course for Beginners: 15/Aug/2011 - 19/Aug/2011;',
 'Python Training Course Intermediate: 12/Dec/2011 - 16/Dec/2011;',   ==> list of matched
 'Python Text Processing Course:31/Oct/2011 - 4/Nov/2011']

items = re.findall("([^:]*):([^;]*;?)", courses). ==> pattern has a 2 groups with () in pattern
items
Output::
[('Python Training Course for Beginners', ' 15/Aug/2011 - 19/Aug/2011;'),
 ('Python Training Course Intermediate', ' 12/Dec/2011 - 16/Dec/2011;'), ==> output is a List of tuples .
 ('Python Text Processing Course', '31/Oct/2011 - 4/Nov/2011')]

12)a choice between several regular expressions. It's a logical "or" and that's why the symbol for this construct is the "|" symbol.

13)Compiling Regular Expressions
If you want to use the same regexp more than once in a script, it might be a good idea to use a regular expression object, i.e. the regex is compiled.
The general syntax:
re.compile(pattern[, flags])
compile returns a regex object, which can be used later for searching and replacing. The expressions behaviour can be modified by specifying a flag value.

Abbreviation	Full name	Description
re.I	re.IGNORECASE	Makes the regular expression case-insensitive
re.L	re.LOCALE	The behaviour of some special sequences like \w, \W, \b,\s, \S will be made dependent on the current locale, i.e. the user's language, country etc.
re.M	re.MULTILINE	^ and $ will match at the beginning and at the end of each line and not just at the beginning and the end of the string

re.S	re.DOTALL	The dot "." will match every character **plus the newline**
re.U	re.UNICODE	Makes \w, \W, \b, \B, \d, \D, \s, \S dependent on Unicode character properties
re.X	re.VERBOSE	Allowing "verbose regular expressions", i.e. whitespace are ignored. This means that spaces, tabs, and carriage returns are not matched as such. If you want to match a space in a verbose regular expression, you'll need to escape it by escaping it with a backslash in front of it or include it in a character class. # are also ignored, except when in a character class or preceded by an non-escaped backslash. Everything following a "#" will be ignored until the end of the line, so this character

import re
regex = r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]**{2}**"
address = "BBC News Centre, London, W12 7RJ"
compiled_re = re.compile(regex) ===> save it the compiled to eb able to reuse the regex in the script
res = compiled_re.search(address)
print(res)

<re.Match object; span=(25, 32), match='W12 7RJ'>

13)Regular Expression Split :

String method split can’t avoid the special characters etc and sp[ilts on the “white spaces”
Better solution is to use split function from the re module.

import re
metamorphoses = "OF bodies chang'd to various forms, I sing: Ye Gods, from whom these miracles did spring, Inspire my numbers with coelestial heat;"
re.split("\W+",metamorphoses) ==> gives the list of all the characters and doesn’t display special chars

14)Search and Replace with sub
re.sub(regex, replacement, subject)
Every match of the regular expression regex in the string subject will be replaced by the string replacement.

28]Lambda, filter, reduce and map::
1)lambada()
The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without a name. These functions are throw-away functions, i.e. they are just needed where they have been created. Lambda functions are mainly used in combination with the functions filter(), map() and reduce()

The general syntax of a lambda function is quite simple:
lambda argument_list: expression
The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. You can assign the function to a variable to give it a name.
The following example of a lambda function returns the sum of its two arguments:

sum = **lambda** x, y : x + y
sum(3,4)

2)map()
map() is a function which takes two arguments:
r = map(func, seq)
The first argument func is the name of a function and the second a sequence (e.g. a list) seq. map() applies the function func to all the elements of the sequence seq.
map() returns an iterator.

C = [39.2, 36.5, 37.3, 38, 37.8]

F = list(map(**lambda** x: (float(9)/5)*x + 32, C)). ==> map applies lambda function to the list C - every element of the list.
print(F)

map() can be applied to more than one list. The lists don't have to have the same length. map() will apply its lambda function to the elements of the argument lists, i.e. it first applies to the elements with the 0th index, then to the elements with the 1st index until the n-th index is reached:

b = [17, 12, 11, 10]
c = [-1, -4, 5, 9]

list(map(**lambda** x, y, z : x+y+z, a, b, c)). ==> x uses a, y uses b and z uses c and the 3 elements are added
Output::
[17, 10, 19, 23]

f one list has fewer elements than the others, map will stop when the shortest list has been consumed:

Mapping a List of Functions:
The map function of the previous chapter was used to apply one function to one or more iterables. We will now write a function which applies a bunch of functions, which may be an iterable such as a list or a tuple, for example, to one Python object.
rom math import sin, cos, tan, pi

```
def map_functions(x, functions):
    """ map an iterable of functions on the the object x """
    res = []
    for func in functions:
       res.append(func(x))
    return res

family_of_functions = (sin, cos, tan)
print(map_functions(pi, family_of_functions)). ==> map  variable to iterable  tuple of functions
```

3)Filtering
The function
filter(function, sequence)
offers an elegant way to filter out all the elements of a sequence "sequence", for

which the function function returns True. i.e. an item will be produced by the iterator result of filter(function, sequence) if item is included in the sequence "sequence" and if function(item) returns True.
In other words: The function filter(f,l) needs a function f as its first argument. f has to return a Boolean value, i.e. either True or False. This function will be applied to every element of the list l. Only if f returns True will the element be produced by the iterator, which is the return value of filter(function, sequence).
In the following example, we filter out first the odd and then the even elements of the sequence of the first 11 Fibonacci numbers:

```
fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
odd_numbers = list(filter(lambda x: x % 2, fibonacci)). ==> filter (lambda function, fib sequence) filter the odd nos
print(odd_numbers)
```

[1, 1, 3, 5, 13, 21, 55]

4. **Python3 does not support reduce() from functools**

29]list comprehensions::
List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```
>>>
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

30]Generators and Iterators:
1)How the iterators work behind the for loop?

```
cities = ["Paris", "Berlin", "Hamburg",
          "Frankfurt", "London", "Vienna",
```

```
        "Amsterdam", "Den Haag"]
for location in cities:
   print("location: " + location)
```

==>The function 'iter' is applied to the object following the 'in' keyword, e.g. for i in o:. Two cases are possible: o is either iterable or not. If o is not iterable, an exception will be raised, saying that the type of the object is not iterable. On the other hand, if o is iterable the call iter(o) will return an iterator, let us call it iterator_obj The for loop uses this iterator to iterate over the object o by using the next method. The for loop stops when next(iterator_obj) is exhausted, which means it returns a StopIteration exception.

```
ther_cities = ["Strasbourg", "Freiburg", "Stuttgart",
               "Vienna / Wien", "Hannover", "Berlin",
               "Zurich"]

city_iterator = iter(other_cities)
while city_iterator:
   try:
      city = next(city_iterator)
      print(city)
   except StopIteration:
      break
```

For dictionaries the iterator runs on the keys.

NOT DONE — go back to this section

31]