Functions

Prof.Dr. Bahadır AKTUĞ

803400815021 Machine Learning with Python

**Compiled from sources given in the references.*

Functions

- Functions are code blocks which takes a certain number of parameters as input, processes them and outputs a certain number of values/variables to do repeated tasks in a program.
- Functions have different names in different programming languages depending on how they handle the output (whether they return value or not etc.) (subroutine, function, procedure vb.)
- The input variable of the functions are called "parameters" and the actual values of those parameters are called "arguments".
- If the number of parameters depends on the program flow, a function which takes a variable number of parameters can also be defined.
- In Python, functions are first-class objects. This means that they can be assigned to variables, returned from other functions and passed into functions. Classes are also first class objects

Functions

- Functions are available in nearly all the programming languages including Python.
- While functions do not return anything in some programming languages (e.g. subroutines of Fortran), a distinction is made between functions with and without output parameters (e.g. functions versus procedures in Pascal).
- In Python, there is only one type of function regardless of whether they return a value or not.
- One main advantage of functions in Python with respect to other programming languages is that they can return multiple values
- Definition of a function:

```
def fonksiyon(parameter1,parameter2,…):
```

Functions

- A function may not have any parameters at all. Even in that case the function must have paranthesis after the function name.
- An example function and code block which calls the this function can be given as below:

```
def mean(a,b,c):
    return (a+b+c)/3

x = 5
y = 6
z = 1
print('The mean of these three numbers is %4.2f' % (mean(x,y,z)))
The mean of these three numbers is 4.00
```

- Here, a,b and c are parameters, the values of these parameters (5, 6, 1) are arguments.

Optional Parameters

- Some of the parameters can be optional. When the optional parameter is not given, the default value is used.

```
# -*- coding=cp1254 -*-
def record(name,nationality="Turkish"):
    print('%s-%s' % (name,nationality))

record('John Smith','English')
record('Ahmet DEMİR')
```

```
John Smith-English
Ahmet DEMİR-Turkish
```

Keyword Parameters

- While working with optional parameters, the order of the parameters are important.
- In the example below, both "marital status" and "nationality" are optional. Please note that how an unwanted result is produced!

```
# -*- coding=cp1254 -*-
def record(name,maritalstatus='married',nationality="Turkish"):
   print('%s-%s-%s' % (record(name,maritalstatus,nationality))
record('John Smith','English')
```

```
John Smith-English-Turkish
```

Keyword Parameters

- To overcome such problems, there are keyword parameters similar to the optional parameters.
- The keyword parameters can be in any order in the function definition.

```
# -*- coding=cp1254 -*-
def record(name,maritalstatus='married',nationality="Turkish"):
    print('%s-%s-%s' % (name, maritalstatus, nationality))

record('John Smith',nationality='English')
```

```
John Smith-married-English
```

Returning Multiple Values

Functions in Python can return multiple values:

```
def multiples(x):
   return 2*x,3*x,4*x,5*x,6*x

x2,x3,x4,x5,x6 = multiples(5)
print(x2,x3,x4,x5,x6)
```

```
10 15 20 25 30
```

The scope of the variables

- The scope of the variables in Python are limited to block they are defined.
- The scope of the variables defined in a function is limited with the function.
- In the example below, the variable "text" can be printed within the function instead the main block.

```
def writefun():
  print(text)

text = 'EEE105'
writefun()
```

```
EEE105
```

The scope of the variables

- However, if a value is assigned to a variable in the function but the value is used before assignment, an exception is thrown.
- The error mesage is due to usage of the variable before assignment.

```
def writefun():
  print(text)
  text = 'EEE105'

text = 'EEE105'
writefun()
```

!Error message

The scope of the variables

- In the example below, a value is assignment to the "text" variable both in the main block and in the function.
- As seen, the very same variable can be used with different assignments without any problem.

```
def writefun():
   text = 'EEE105'
   print(text)

text = 'EEE106'
writefun()
print(text)
```

```
EEE105
EEE106
```

The scope of the variables

- To be able to use a variable which is defined within the main block elsewhere (e.g. in a function etc.), it is necessary to define it as "global".

```
def writefun():
  global text
  print(text)
  text = 'EEE105'
  print(text)

text = 'EEE106'
writefun()
print(text)
```

Program çıktısı:

```
EEE106
EEE105
EEE105
```

Varying number of function parameters

- The number of parameters may have to be changed depending on the input.
- Instead of writing a different function for each case, a single tuple of parameters which is identified by a preceeding '*' can be used.

```
def mean(*values):
  return sum(values)/len(values)

print(mean(5,2,4,6))
print(mean(1,9,42,5,2,4,6))
```

Program output:

```
4.25
9.857142857142858
```

Modifying the value of the parameters within a function

- The variable input to the function can be modified depending on whether they are mutable or immutable types.
- Note the example below. Since the string type is immutable it cannot be modified within a function block.

```
def modify(s):
    print(s)
    s = "EEE111"
    print(s)

s = "EEE105"
print(s)
modify(s)
print(s)
```

Program output:

```
EEE105
EEE105
EEE111
EEE105
```

Modifying the value of the parameters within a function

- On the other hand, since lists are of mutable type, they can be modified within a function.

```
def modify(s):
   print(s)
   s[0] = 14
   print(s)

s = [5,2,4,6]
print(s)
modify(s)
print(s)
```

Program output:

```
[5, 2, 4, 6]
[5, 2, 4, 6]
[14, 2, 4, 6]
[14, 2, 4, 6]
```

Functions calling themselves (Recursion)

- Sometimes, it is needed for a function to call itself.
- This process is called "recursion".
- The prominent examples are factorials and fibonacci numbers where the value of the function relies on the output of the previous.

```
def factorial(number):
    if number == 1:
        return number
    else:
        return number * factorial(number-1)

print(faktorial(5))
```

Program output:

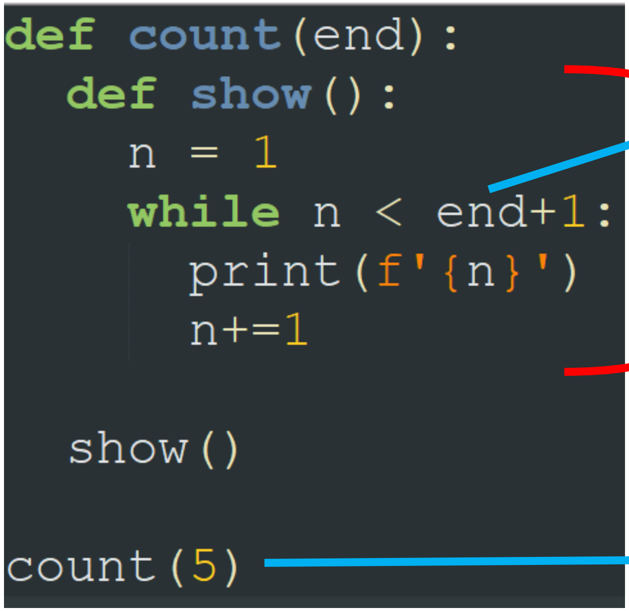
```
120
```

Scope

- We already talked about namespaces in relation with the functions.
- The assigned variable inside a function are always bound to the function's local namespace.
- One way to expose the variable to other parts of a program is to use «global» keyword.
- In nested functions, inner function can also access the scope of the outer function.
- After exiting the function, the variables (defined in function scope) are not accessible outside the function.

Nested Functions

- Python fully supports nested functions. Let’s see that in an example.



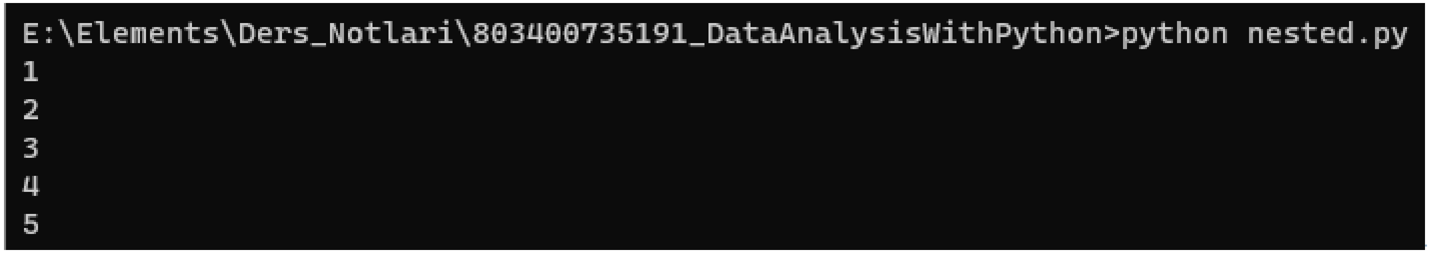
```python
def count(end):
  def show():
    n = 1
    while n < end+1:
      print(f'{n}')
      n+=1

  show()

count(5)
```

inner function

Note! inner function can access the variables in the outer function

calling the outer function

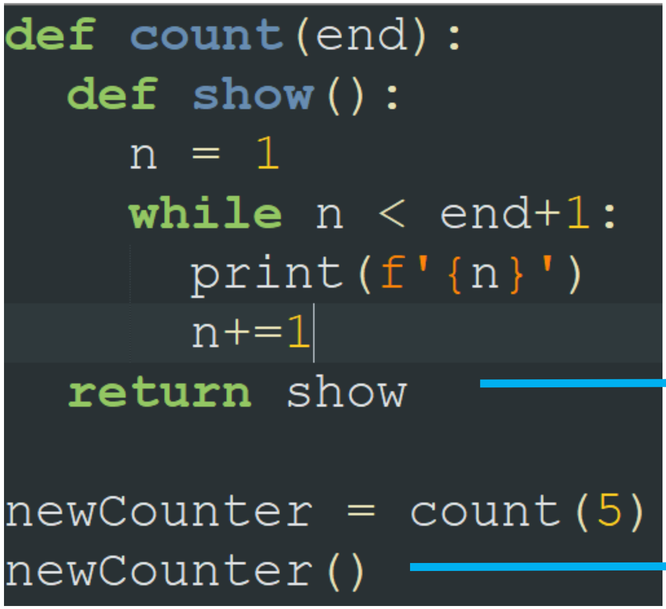
Output:



```
E:\Elements\Ders_Notlari\803400735191_DataAnalysisWithPython>python nested.py
1
2
3
4
5
```

Closures

- What would happen if the inner function returned show() function instead of calling it?

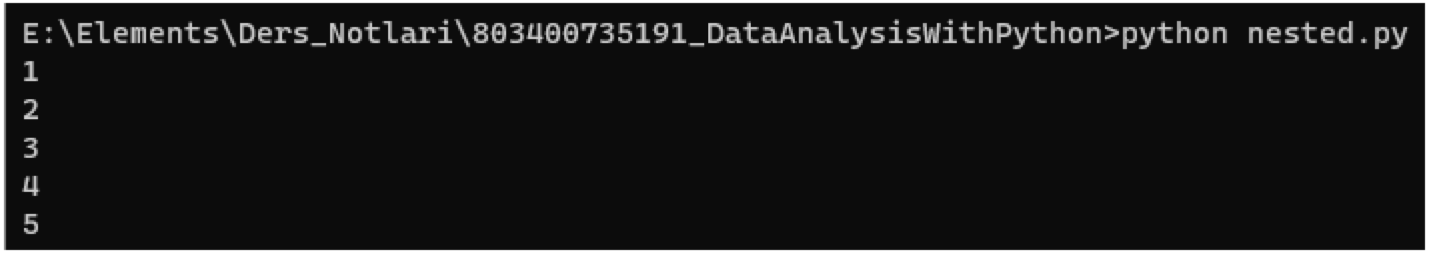


```
def count(end):
  def show():
    n = 1
    while n < end+1:
      print(f'{n}')
      n+=1
  return show

newCounter = count(5)
newCounter()
```

Note! inner function is returned not called.

Note! Outer function is called without arguments

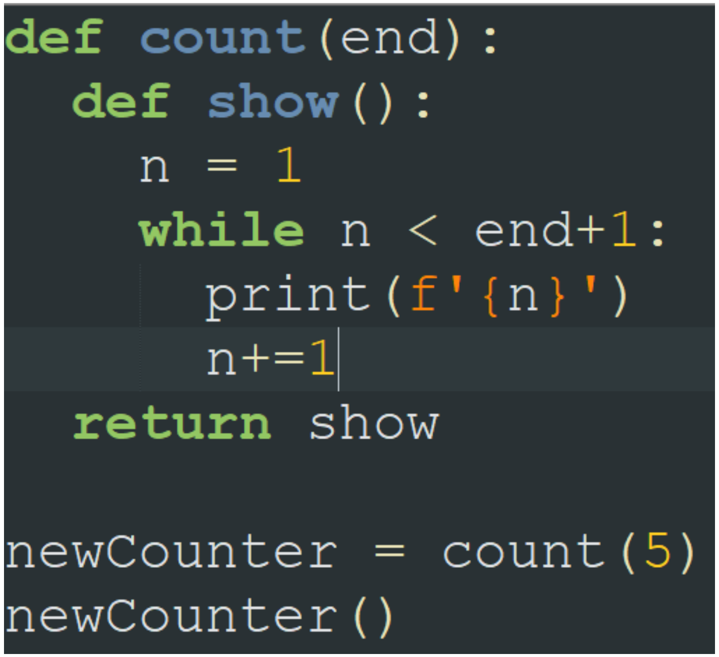
output



```
E:\Elements\Ders_Notlari\803400735191_DataAnalysisWithPython>python nested.py
1
2
3
4
5
```

Closures

- The count() function with argument is assigned as a new function (newCounter).



```
def count(end):
  def show():
    n = 1
    while n < end+1:
      print(f'{n}')
      n+=1
  return show

newCounter = count(5)
newCounter()
```

- The count() function with argument is assigned as a new function (newCounter) was called with the value 2 and the returned function was bound to the name counter1. when newCounter is executed it remembers the original argument (i.e. 5).

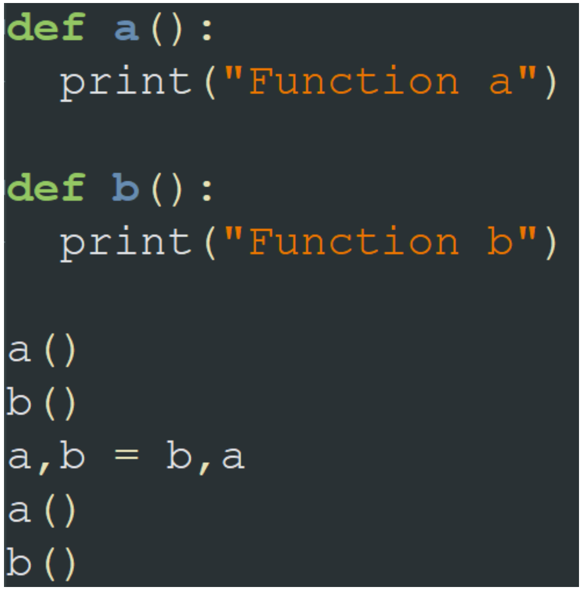
- This concept is called a «closure» and used frequently in other structures such as decorators.

Closures

- Closures look like nested functions. However, they are more than simply nested functions.
- A closure must be have a nested function.
- The inner function must use a variable (the value of which) definded in the enclosing function.
- The inner function must not be directly called but must return the nested function.
- The interesting feature of a closure is it recalls the arguments even the after exiting the original function.
- The feature is also useful for other objectives like lazy implementation.

Decorators

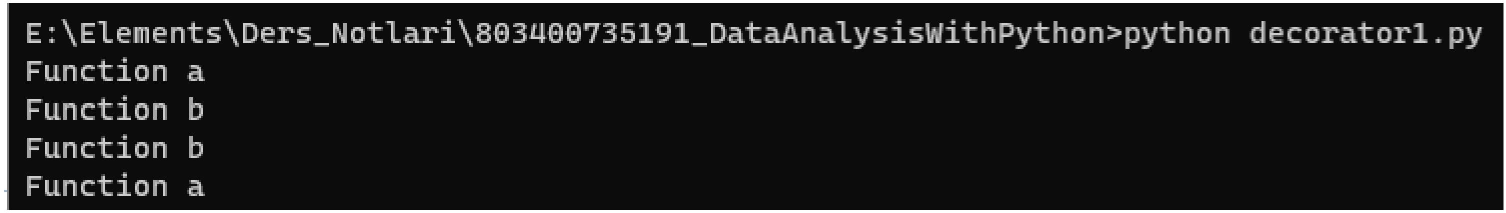
- The name of a function in Python is simply a reference to the definition of that function. It is possible to redirect the function name to another function definition.
- Let’s see it in an example:



```
def a():
    print("Function a")

def b():
    print("Function b")

a()
b()
a,b = b,a
a()
b()
```

Swapping just like simple variables!

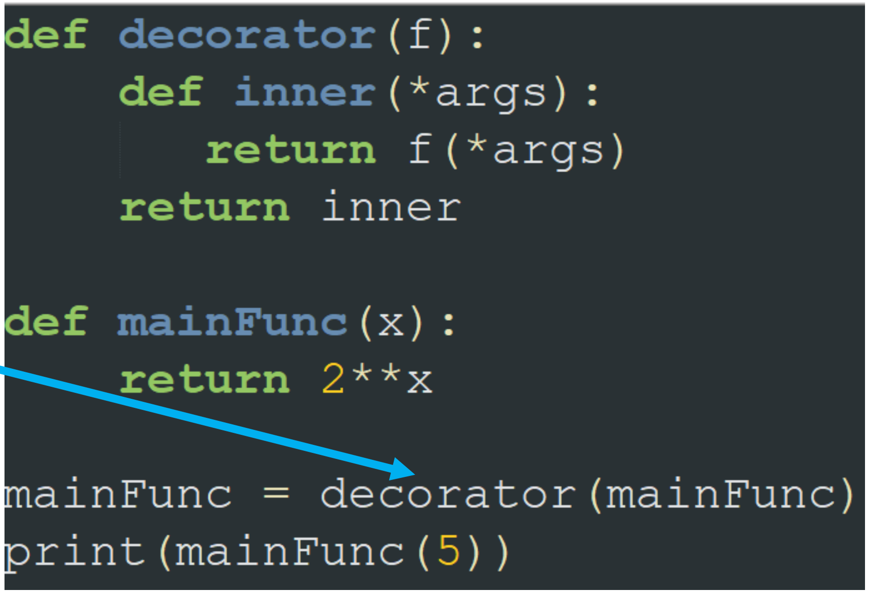
Output:



```
E:\Elements\Ders_Notlari\803400735191_DataAnalysisWithPython>python decorator1.py
Function a
Function b
Function b
Function a
```

Decorators

- Let’s define a decorator function which The name of a function in Python is simply a reference to the definition of that function. It is possible to redirect the function name to another function definition.
- Let’s see it in an example:

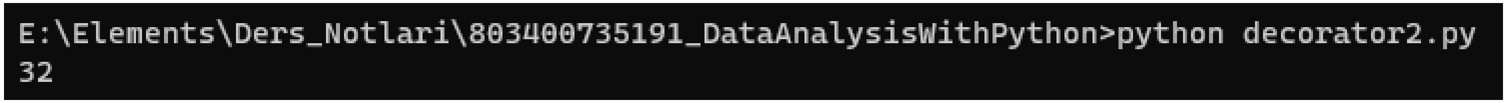


```
def decorator(f):
    def inner(*args):
        return f(*args)
    return inner

def mainFunc(x):
    return 2**x

mainFunc = decorator(mainFunc)
print(mainFunc(5))
```

decorating function

decorated function

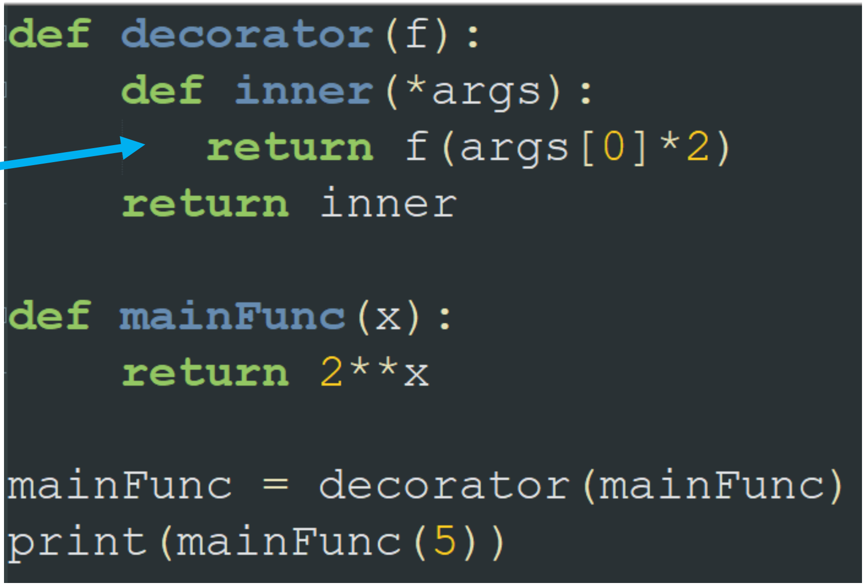
Output:



```
E:\Elements\Ders_Notlari\803400735191_DataAnalysisWithPython>python decorator2.py
32
```

Decorators

- This structure is very useful to add additional functionality of the original function (mainFunc).
- For instance we can add more code to be run before or after calling mainFunc(x).

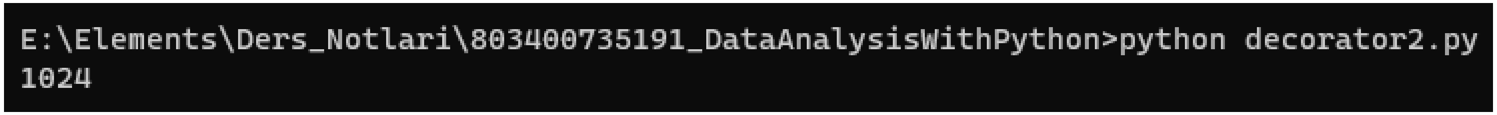


```python
def decorator(f):
    def inner(*args):
        return f(args[0]*2)
    return inner

def mainFunc(x):
    return 2**x

mainFunc = decorator(mainFunc)
print(mainFunc(5))
```

increasing the value by 2

Calling the function with argument 5. But function is evaluated by 10.

Output:

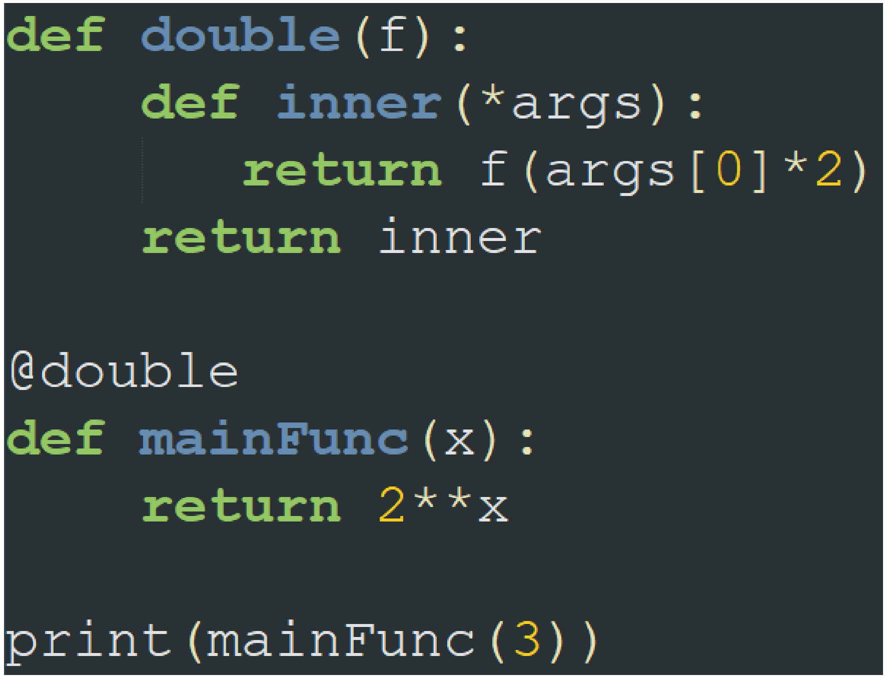


```
E:\Elements\Ders_Notlari\803400735191_DataAnalysisWithPython>python decorator2.py
1024
```

Decorators

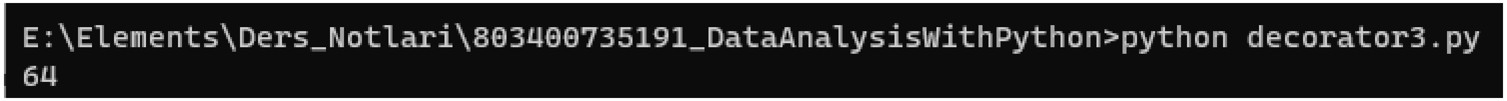
- There is a shorthand for defining decorator. Usually this is the form you'll encounter decorator.
- It's merely syntactic sugar

We omit the line:
mainFunc = decorator(mainFunc)



```
def double(f):
    def inner(*args):
        return f(args[0]*2)
    return inner

@double
def mainFunc(x):
    return 2**x

print(mainFunc(3))
```

Output:



```
E:\Elements\Ders_Notlari\803400735191_DataAnalysisWithPython>python decorator3.py
64
```

Modules and calling functions from modules

- Modules are files which contains Python functions and classes.
- In particular, in large programs, it is necessary to organize functions and classes in different files.
- To be able to a function/class in another file:
  - The module has to be in the search path of Python
  - The module has to imported with «import» command.
- When the "import" is used, a single function or all the functions in that module can be imported.
  - import mod1 ⍰ the module named mod1
  - from mod1 import func1 ⍰ function named func1 in the file named mod1
  - from mod1 import * ⍰ all the functions and classes in the module named mod1

Modules and calling functions from modules

- Whether the module we want to use is in the search path can be checked as follows:
  - >>> import sys
  - >>> print(sys.path)
  - >>> '', 'D:\\anaconda3\\lib’,'D:\\anaconda3\\lib\\site-packages’, …
  - >>> import numpy
  - >>> numpy.__file__
  - 'D:\\anaconda3\\lib\\site-packages\\numpy\\__init__.py’

Modules and calling functions from modules

- dir() can also ve used to return a list of all the functions and variables in a module:
- dir(numpy)
-

```
['ALLOW_THREADS', 'AxisError', 'BUFSIZE', 'CLIP', 'ComplexWarning', 'DataSource', 'ERR_CALL',
'ERR_DEFAULT', 'ERR_IGNORE', 'ERR_LOG', 'ERR_PRINT', 'ERR_RAISE', 'ERR_WARN',
'FLOATING_POINT_SUPPORT', 'FPE_DIVIDEBYZERO', 'FPE_INVALID', 'FPE_OVERFLOW',
'FPE_UNDERFLOW', 'False_', 'Inf', 'Infinity', 'MAXDIMS', 'MAY_SHARE_BOUNDS',
'MAY_SHARE_EXACT', 'MachAr', 'ModuleDeprecationWarning', 'NAN', 'NINF', 'NZERO', 'NaN',
'PINF', 'PZERO', 'RAISE', 'RankWarning', 'SHIFT_DIVIDEBYZERO', 'SHIFT_INVALID',
'SHIFT_OVERFLOW', 'SHIFT_UNDERFLOW', 'ScalarType', 'Tester', 'TooHardError', 'True_',
'UFUNC_BUFSIZE_DEFAULT', 'UFUNC_PYVALS_NAME', 'VisibleDeprecationWarning', 'WRAP',
'_NoValue', '_UFUNC_API', '__NUMPY_SETUP__', '__all__', '__builtins__', '__cached__',
'__config__', '__dir__', '__doc__', '__file__', '__getattr__', '__git_revision__', '__loader__',
'__mkl_version__', '__name__', '__package__', '__path__', '__spec__', '__version__',
'_add_newdoc_ufunc', '_distributor_init', '_globals', '_mat', '_pytesttester', 'abs', 'absolute',
'add', 'add_docstring', 'add_newdoc', 'add_newdoc_ufunc', 'alen', 'all', 'allclose', 'alltrue', 'amax',
'amin', 'angle', 'any', 'append', 'apply_along_axis', 'apply_over_axes', 'arange', 'arccos', 'arccosh',
'arcsin', 'arcsinh', 'arctan', 'arctan2', 'arctanh', 'argmax', 'argmin', 'argpartition', 'argsort',
'argwhere', 'around', 'array', 'array2string', 'array_equal', 'array_equiv', 'array_repr', 'array_split',
'array_str', 'asanyarray', 'asarray', 'asarray_chkfinite', 'ascontiguousarray', 'asfarray',
'asfortranarray', 'asmatrix', 'asscalar', 'atleast_1d', 'atleast_2d',
```

Modules and calling functions from modules

- The **globals()** and **locals()** functions can be used to return module names in the global and local namespaces.
  - >>> **globals()**

    -
      ```
      {'__name__': '__main__', '__doc__': None, '__package__': None,
      '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
      '__spec__': None, '__annotations__': {}, '__builtins__': <module
      'builtins' (built-in)>, 'Proj': <class 'pyproj.Proj'>, 'a': [2, 4, 5, 6],
      'data': ['class', 'Python', 'script', 'example'], 'sys': <module 'sys'
      (built-in)>, 'numpy': <module 'numpy' from
      'D:\\anaconda3\\lib\\site-packages\\numpy\\__init__.py’>}
      ```
- **reload() function**
  - A module can be imported only once. To re-execute the code in a module, the reload() can be used. This could be necessary if you have edited the already imported module externally

Modules and calling functions from modules

matematics.py

```
def factorial(number):
    if number == 1:
        return number
    else:
        return number * factorial(number-1)
```

main program

```
from matematics import factorial
print(factorial(5))
```

Anonymous Functions, Comprehensions and functions that operate directly over iterables

Prof.Dr. Bahadır AKTUĞ

803400815021 Machine Learning with Python

Lambda

- Lambda functions or Lambda operator is used to define anonymous functions.
- Lambda function is probably inherited into Python from other functional programming languages such as Lisp, Scheme.
- Lamdba functions are generally used to define functions for which no name is needed to refer later due to it limited and/or brief function during program execution.
- Lambda function can either be used on its own or as a parameter of other functions which take functions as input.
- Lambda functions are typically anonymous and a name can also be assigned.

Lambda

- Defining a Lambda function:

```
lambda a, b : a * b
```

- where a and b are function parameters, the part following colon is the operation which operate on these arguments
- If we choose to give this function a name, we can assign it a name as:

```
>>> multiply = lambda a, b : a * b
```

- After assignment, the lambda function can be directly called by its name:

```
>>> multiply (5,4)
>>> 20
```

Lambda

- A normal function with the same functionality as above can be written as:

```
def multiply (a, b) :
    return a * b
multiply (5,4)
```

- The same computation with lambda function:

```
(lambda a, b : a * b) (5,4)
```

- While both approaches are possible, as will be seen in the next part lambda functions are often handier to be used as a argument.

Map

- "map" function takes an iterable as input and applies that function to each element of the iterable.
- The general form of the "map" function is:

  map(function, iterable)

- In the earlies Python versions (< v3), the map function returns a list. Now, the "map" function returns an iterator.
- The output from map function can be easly converted into a list by using list() function or whatever data type is needed

Map

- Example:

```
def square (a):
    return a*a

x = range(5)

c = list(map(square,x))
print(c)
[0, 1, 4, 9, 16]
```

An iterator is returned by "map" function. The "list()" function is used to transform it into a list.

«map» function applies the «square» function to each element of x.

Map

- The same can be done through lambda functions:

```
x = range(5)
c = list(map(lambda a: a*a,x))
print(c)
[0, 1, 4, 9, 16]
```

The original square function is replaced by lambda function. This way of function definition is sometimes called «inline».

- Multiple inputs (arguments of lambda function) can be supplied as separate iterables:

```
x = [1, 5, 9, 15]; y = [2, 3, 7, 10]; z = [6, 4, 5, 20]
c = list(map(lambda a,b,c: a*b+c, x, y, z))
print(c)
[8, 19, 68, 170]
```

Filter

- "filter" function is similar to «map» in many ways. The general usage is as follows:

  filter (function, iterable)

- However, the function used with «filter» has to be a boolean function which returns «True/False» for each item of the iterable.
- In this respect, the individual elements of an iterable can be filtered depeding on the boolean value returned (only those which return True will be returned).
- It was discussed in previous lessons what variables are True/False. New filters can be derived based on this property.

Filter

- A filter command which returns the odd numbers out of a list between 1 and 20 can be written as:

```
>>> numbers =  range(20)
>>> list(filter(lambda x: x % 2, numbers))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

- Pay attention to how the "modulo" operator is used as a boolean function to return "True/False« (any number other than zero is True in Python).

- Return numbers smaller than 10:

```
>>> numbers = [14, 6, 15, 17 , 9, 6 ]
>>> list(filter(lambda x: x > 10, numbers))
[14, 15, 17]
```

List Comprehension

- List Comprehension is one of the most flexible tools provided by Python.
- List Comprehension is the fastest way to derive a list and is often faster than lambda functions.
- While it is mostly used for lists, in fact, «comprehension» can be applied to other data types as well (sets, dictionaries etc.)
- General Usage:

[f(x) for x in array] OR [f(x) for x in array if g(x)]

List Comprehension

- For example, let’s convert a list of lenghts in inches to cm:

```
>>> inches =  [1, 2, 5, 3, 6]
>>> [2.54*x for x in inches]
[2.54, 5.08, 12.7, 7.62, 15.24]
```

- To convert and return only those larger than 3 inches;

```
>>> [2.54*x for x in inch if x > 3]
[12.7, 15.24]
```

List Comprehension

- We can generate an output similar to Matlab’s meshgrid function by using the list comprehension as follows:

```
>>> x = [5, 15, 25]
>>> y = [50, 55, 60]
>>> [(a,b) for a in x for b in y]
[(5, 50), (5, 55), (5, 60), (15, 50), (15, 55), (15, 60), (25, 50), (25, 55), (25, 60)]
```

List Comprehension

- The general form of list comprehension including nested structures can be given as follows:

```
[expression for x in array1 for y in array2]
```

The expanded form of the list comprehension above is as follows:

```
sonuc=[]
for x in array1:
    for y in array2:
        sonuc.append(expression)
```

List Comprehension

- For another nested structure:

```
[[ifade for x in dizi1] for y in dizi2]
```

the equivalent nested loop is:

```
sonuc=[]
for y in dizi2:
  icsonuc = []
  for x in dizi1:
      icsonuc.append(ifade)
  sonuc.append(icsonuc)
```

Set Comprehension

- Set comprehension is also similar to list comprehension:

```
{j for i in array1}
```

veya

```
{x for x in array1 for y in array2}
```

- Beware that no duplicate elements are allowed in a set. This also applies to set comprehension:

```
>>> set1 = {y for x in range(5) for y in range(1,x)}
>>> print(set1)
{1, 2, 3}
```

Dictionary Comprehension

- Dictionary comprehension can be used both for the keys and the values:

```
{x:2*x for x in array1}
```

OR

```
{x:y for x in array1 for y in array2}
```

- Note that the keys in a dictionary are unique:

```
>>> {x:2*x for x in range(4)}
>>> {0: 0, 1: 2, 2: 4, 3: 6}
```

Dictionary Comprehension

- Similarly, dictionary comprehension can be used as a «zip» command (note that any key will be overwritten if another item with the same key is added)

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> dictionary = {x:y for x in a for y in b}
>>> print(dictionary)
{1: 6, 2: 6, 3: 6}
```

- Reversing a dictionary via dictionary comprehension:

```
>>> {y:x for x,y in sozluk.items()}
{6: 3}
```

References

1	*Wentworth, P., Elkner, J., Downey, A.B., Meyers, C. (2014). How to Think Like a Computer Scientist: Learning with Python (3nd edition).*
2	*Pilgrim, M. (2014). Dive into Python 3 by. Free online version: DiveIntoPython3.org ISBN: 978-1430224150.*
3	*Summerfield, M. (2014) Programming in Python 3 2nd ed (PIP3) : - Addison Wesley ISBN: 0-321-68056-1.*
4	*Summerfield, M. (2014) Programming in Python 3 2nd ed (PIP3) : - Addison Wesley ISBN: 0-321-68056-1.*
5	*Jones E, Oliphant E, Peterson P, et al. SciPy: Open Source Scientific Tools for Python, 2001-, http://www.scipy.org/.*
6	*Millman, K.J., Aivazis, M. (2011). Python for Scientists and Engineers, Computing in Science & Engineering, 13, 9-12.*
7	*John D. Hunter (2007). Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering, 9, 90-95.*
8	*Travis E. Oliphant (2007). Python for Scientific Computing, Computing in Science & Engineering, 9, 10-20.*
9	*Goodrich, M.T., Tamassia, R., Goldwasser, M.H. (2013). Data Structures and Algorithms in Python, Wiley.*
10	*http://www.diveintopython.net/*
11	*https://docs.python.org/3/tutorial/*
12	*http://www.python-course.eu*
13	*https://developers.google.com/edu/python/*
14	*http://learnpythonthehardway.org/book/*