ÖZYEĞİN UNIVERSITY

EE393 Python for Engineers

Dr. Orhan Gökçöl

orhan.gokcol@ozyegin.edu.tr

online

WEEK #5

2020-2021 Fall Semester

1

Agenda

- Short review
- Exception handling
- Tuples
- Dictionaries
- Advanced uses of functions
- Lambda functions
- Mapping

ipynb files

(Download to your local)

2

Review

- Encoding : ASCII, Unicode
  - chr, ord, unicode (UTF-8 & UTF-16), .encode, .decode
  - ASCII still important!!! (serial comm, configs, source codes etc.)
- Formatted output using .format method
  - How to use in print ( )
- Review for while & for
  - range, in, break, continue, else
- Lists
  - Common list methods
  - Iteration over lists
  - Enumeration
- List comprehension using “for”
- Simple file i/o
- Functions
  - Void
  - Value returned
  - Function scope

3

Libraries

To use a library, you can use a simple **import** statement, which should come at the beginning of your program. For an example, let’s look at a standard Python library, **math**. To access the functions within this module start with:

import math

Then, we can use methods included within the module such as sqrt as **math.sqrt(5.8)**

Alternatively, **from math import sqrt**

Then, we can use methods included within the module such as sqrt as **sqrt(5.8)**

Use with caution: **from math import ***

4

Defining an object

Libraries contain objects that we can call after importing the relevant library

Objects know things.

- Data that is internal to the object.
- We often call those instance variables.

Objects can do things.

- Behavior that is internal to the object.
- We call functions that are specific to an object methods.
  - But you knew that one already.

We access both of these using **dot notation**

- object.variable
- object.method()

```
math.pi
math.sqrt()
```

5

Making your own libraries

- Importing a code that you have written as a library works the same as in the examples above, provided that you have saved your code as a **.py** file.
- For example, if you will be using a function or set of functions in multiple programs, you can save them in a separate file as **myfunc.py**. Be sure to include in your module whatever import command are necessary for it to work. In another program, you can import it using:

  import myfunc

- Then, you may use your methods using **dot** notation

6

Error Handling in Pythons : Exceptions

- We can make certain mistakes while writing a program that lead to errors when we try to run it.
- A python program terminates as soon as it encounters an unhandled error. These errors can be broadly classified into two classes:
- Syntax errors
- Logical errors (Exceptions)

Exception handling

- Trying to open a file which does not exist
- Division by zero
- Square root of negative values
- Variable doesn't exist
- Value doesn't exist
- <...>

7

Exception Handling

An exception is an **error** that happens during the execution of a program (Very similar to Java). Look at the code. Let's analyse:

```
x = float(input("Your number: "))
inverse = 1.0 / x
print("The inverse: ", inverse)
```

exception_handling.ipynb

if x=0, normally it should raise an error. However, using the following modification:

```
#exception handling
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
    print("The inverse: ", inverse)
except ZeroDivisionError:
    print("No zero in denominator ...")
finally:
    print("Execution completed.")
```

```
Your number:  0
No zero in denominator ...
Execution completed.
```

To catch all errors, do not specify exception name!

8

Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).
When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.

Important Python Built-In Exceptions

- **EOFError :** Raised when the input() functions hits end-of-file condition.
- **FloatingPointError :** Raised when a floating point operation fails.
- **ImportError :** Raised when the imported module is not found.
- **IndexError :** Raised when index of a sequence is out of range.
- **KeyError :** Raised when a key is not found in a dictionary.
- **KeyboardInterrupt :** the user hits interrupt key (Ctrl+c or delete).
- **MemoryError :** Raised when an operation runs out of memory.
- **NameError :** a variable is not found in local or global scope.
- **OSError :** Raised when system operation causes system related error.
- **OverflowError :** Result of an arithmetic is too large to be represented.
- **RuntimeError :** An error does not fall under any other category.
- **SyntaxError :** Raised by parser when syntax error is encountered.
- **SystemExit :** Raised by sys.exit() function.
- **TypeError :**function or operation is applied to an object of incorrect type.
- **ValueError :** Raised when a function gets argument of correct type but improper value.
- **ZeroDivisionError :** Raised when second operand of division or modulo operation is zero.

9

General structure

```
try:
    You do your operations here;
    ......................
except Exception I :
    If there is Exception I, then execute this block.
except Exception II :
    If there is Exception II, then execute this block.
    ......................
else:
    If there is no exception then execute this block.
```

finally **Finally code is run no matter what else happens.** It is useful for cleanup code that has to run.

```
myfile = open("test.txt", "w")
try:
    myfile.write("the Answer is: ")
    myfile.write(42)
finally:
    myfile.close() # will be executed before TypeError is propagated
```

10

Tuples

- A tuple is a sequence data type that is similar to the **list**. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.
- The main differences between lists and tuples are:
  - Lists are enclosed in brackets ([ ]), and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated.
  - Tuples can be thought of as **read-only** lists.

mytuple = (11, 22, 33)
mytuple[0] ➔ **11**
mytuple[-1] ➔ **33**
mytuple[0:1] ➔ **(11,)**
The comma is required!

For more info on Tuples:
https://www.tutorialspoint.com/python/pdf/python_tuples.pdf

11

Tuple operations

```
In [1]: mytuple = (11, 22, 33)

In [2]: mytuple[0]
Out[2]: 11

In [3]: mytuple[0:2]
Out[3]: (11, 22)

In [4]: mytuple2 = mytuple

In [5]: mytuple += (2,8)

In [6]: mytuple
Out[6]: (11, 22, 33, 2, 8)

In [7]: mytuple[1] = 29
Traceback (most recent call last):

  File "<ipython-input-7-2e36276d2d75>", line 1, in <module>
```

NOT ALLOWED!!!

```
In [8]: x = 4,5,-8

In [9]: x
Out[9]: (4, 5, -8)

In [10]: y = mytuple + x

In [11]: y
Out[11]: (11, 22, 33, 2, 8, 4, 5, -8)
```

12

Note that,

- a function always returns a “tuple” if there are more than one values

functions2.ipynb

```
returnMe = fun1()
print (returnMe)
print (returnMe[4][3])
returnMe[4][3]="Ozyegin"
print (returnMe[4][3])
returnMe[1]=8 #will give an error
```

```
(1, 2, 3, 'hello', [7, 8, True, (3+4j), 'OzU'])
(3+4j)
Ozyegin
---------------------------------------------------------
---------
TypeError                                 Traceba
all last)
<ipython-input-9-b21c0aac17d8> in <module>
```

13

Named Tuples

(Optional)

- Named tuples are basically easy-to-create, lightweight object types. Named tuple instances can be referenced using object-like variable dereferencing or the standard tuple syntax. They can be used similarly to struct or other common record types, except that they are immutable.

functions2.ipynb

```
from collections import namedtuple
Point = namedtuple('Point', 'x y')
pt1 = Point(1.0, 5.0)
pt2 = Point(2.5, 1.5)

from math import sqrt
# use index referencing
line_length = sqrt((pt1.x-pt2.x)**2 + (pt1.y-pt2.y)**2)
 # use tuple unpacking
x1, y1 = pt1
print (line_length, x1, y1)
```

```
3.8078865529319543 1.0 5.0
```

Good readibility and clear object notation

14

Sets

- A Set is an unordered collection data type that is iterable, mutable, and has **no duplicate elements**. Python's set class represents the mathematical notion of a set. The major advantage of using a set, as opposed to a list, is that it has a **highly optimized** method for checking whether a specific element is contained in the set.

```
In [23]: a = [2,3,"hello",-8,5,2]

In [24]: b = set(a)

In [25]: b
Out[25]: {-8, 2, 3, 5, 'hello'}
In [29]: thisset = {"apple", "banana", "cherry"}

In [30]: print (thisset)
{'apple', 'banana', 'cherry'}
```

For more info: https://www.geeksforgeeks.org/sets-in-python/

15

Mappings & Dictionaries

- Mapping is a scheme of defining **data** where each element is identified with a **key** called "hash tag." Therefore, the element can be accessed by referring to the key. One of the data types in this category is a **dictionary**.

```
In [87]: person={'age':21,'height':1.79, 'location':'Istanbul'}

In [88]: person
Out[88]: {'age': 21, 'height': 1.79, 'location': 'Istanbul'}

In [89]: person['age']
Out[89]: 21

In [90]: type(person)
Out[90]: dict
```

[ ] : List
( ) : Tuple
{ } : Dictionary

16

Dictionaries

- In data structure terms, a dictionary is better termed an *associative array*, *associative list* or a *map*.
- You can think if it as a **list of pairs**, where the first element of the pair, the ***key***, is used to retrieve the second element, the ***value***.
- Thus we ***map a key to a value***
- dictionaries are collections but they are not sequences such as lists, strings or tuples
  - there is no order to the elements of a dictionary
  - in fact, the order (for example, when printed) might change as elements are added or deleted.

17

Key Value pairs

- The **key** acts as an index to find the associated value.
  - Just like a dictionary, you look up a word by its spelling to find the associated definition
- A dictionary can be searched to locate the value associated with a key
- **Key** must be immutable (i.e. read-only) and **unique**
  - strings, integers, tuples are fine
  - lists are NOT
- **Value** can be anything

18

Dictionaries

- The **dict** type is not only widely used in our programs but also a fundamental part of the Python implementation.
- Module namespaces, class and instance attributes and function keyword arguments are some of the fundamental constructs where dictionaries are deployed. The built-ins live in **builtins.__dict__**
- Because of their crucial role, Python dicts are highly optimized.
  - Hash tables are the engines behind Python's high performance dicts.
- Try **print(builtins.__dict__)** to see built-in variables.

19

Dictionaries

- A dictionary is a container object (like strings, tuples and lists) in which **entries are accessed by a key rather than a nonnegative integer index**. For example if

```
# a dictionary one uses sqrt[1], sqrt[4], and sqrt[9] to get values.
sqrt = {1:1, 4:2, 9:3}
```

- Creating a dictionary is simple: Place **items** inside curly braces **{ }** separated by comma. An **item** has a (1) key and (2) the corresponding value expressed as a pair, **key: value**.
  - **Remember: value**s can be of any data type and can repeat
  - **Remember: key**s must be of immutable type (string, number or tuple with immutable elements) and **must be unique**.

20

Creating a dictionary:

There are several methods for creating a dictionary:

1- Standard way

```
myDict = {}    # empty dictionar
ymyDict = {1: 'ali', 2: 'veli'}      # dictionary with integer keys
myDict = {'name': 'Jon', 1: [2, 4, 3]} # dictionary with mixed keys
```

2- Using dict() function

```
myDict = dict({1:'OzU', 2:'Ozyegin'})
```

A=dict(m=8, n=9) ➔ {'m': 8, 'n': 9}

3- Using sequence having each item as a pair in a dict()

```
myDict = dict([(1,'Ozu'), (2,'Ozyegin')])
```

dict.ipynb

4- Comprehensions

```
mySquares = {x: x*x for x in range(6)}
print(mySquares) # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

which is equivalent to:
```

mySquares = {}for x in range(6): mySquares[x] = x*x

21

It is possible to use sequences as dictionary member

```
inventory = { 'gold' : 500, 'pouch' : ['flint', 'twine', 'gemstone'],
              'backpack' : ['xylophone','dagger', 'bedroll','bread loaf'] }


ali_r = { "name": "Ali",
          "homework": [90.0,97.0,75.0,92.0],
          "quizzes": [88.0,40.0,94.0],
          "tests": [75.0,90.0] }
veli_r = { "name": "Veli",
          "homework": [100.0, 92.0, 98.0, 100.0],
          "quizzes": [82.0, 83.0, 91.0],
          "tests": [89.0, 97.0] }
```

dict.ipynb

22

It is possible to use sequences as dictionary member

```
person = dict()
person['fname'] = 'Joe'
person['lname'] = 'Jones'
person['age'] = 50
person['spouse'] = 'Natalie'
person['children'] = ['Ralph', 'Betty', 'Joey']
person['pets'] = {'dog': 'Lucky', 'cat': 'Boni'}
```

dict.ipynb

23

Accessing items in a dictionary:

We can use **get()** method and keys:

```
myDict = {'name': 'Ali', 'age': 19}
print(myDict['name'])        # Output will be Ali -> it is a key
print(myDict.get('age'))     # Output will be 19 -> it is a value
```

Note that trying to access a non-existent key throws error (NameError)

Note that, Access requires `[ ]`, but the *key* is the index!

`my_dict={}`
- an empty dictionary

`my_dict['bill']=25`
- added the pair 'bill':25

`print(my_dict['bill'])`
- prints `25`

dict.ipynb

24

Dictionaries are mutable

- Like lists, dictionaries are a mutable data structure
  - you can change the object via various operations, such as index assignment

```
my_dict = {'bill':3, 'rich':10}
print(my_dict['bill'])     # prints 3
my_dict['bill'] = 100
print(my_dict['bill'])     # prints 100
```

dict.ipynb

25

Dictionary keys can be any immutable object

```
#Dictionary keys can be any immutable object
demo = {2: ['a','b','c'], (2,4): 27, 'x': {1:2.5, 'a':3}}
print (demo)
print (demo[2])
print (demo[(2,4)])
print (demo ['x'])
print (demo['x'][1])
```

```
{2: ['a', 'b', 'c'], (2, 4): 27, 'x': {1: 2.5, 'a': 3}}
['a', 'b', 'c']
27
{1: 2.5, 'a': 3}
2.5
```

dict.ipynb

26

Common operators

Like others, dictionaries respond to these

- `len(my_dict)`
  - number of key:value **pairs** in the dictionary
- `element in my_dict`
  - boolean, is `element` a **key** in the dictionary
- `for key in my_dict:`
  - iterates through the **keys** of a dictionary

27

fewer methods for Dictionaries

Here are some methods with dictionaries:

- `key in my_dict`
  does the key exist in the dictionary
- `my_dict.clear()` – empty the dictionary
- `my_dict.update(yourDict)`
  (for each key in `yourDict`, updates `my_dict` with that key/value pair) –see example
- `my_dict.copy` - shallow copy
- `my_dict.pop(key)` – remove key, return value

dict.ipynb

28

Dictionary content methods

- `my_dict.items()` – all the key/value pairs
- `my_dict.keys()` – all the keys
- `my_dict.values()` – all the values

```
#items, keys and values
person = dict()
person['fname'] = 'Joe'
person['lname'] = 'Jones'
person['age'] = 50
person['spouse'] = 'Natalie'
person['children'] = ['Ralph', 'Betty', 'Joey']
person['pets'] = {'dog': 'Lucky', 'cat': 'Boni'}
print (person.items()) #returns all items
for x,y in person.items(): access keys and values via items
    print (x,y)
```

```
dict_items([('fname', 'Joe'), ('lname', 'Jones'), ('age', 50
pouse', 'Natalie'), ('children', ['Ralph', 'Betty', 'Joey']]
ts', {'dog': 'Lucky', 'cat': 'Boni'})])
fname Joe
lname Jones
age 50
spouse Natalie
```

dict.ipynb

29

Dictionary views are iterable using for

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

- prints all the keys

```
for key,value in my_dict.items():
    print (key,value)
```

- prints all the key/value pairs

```
for value in my_dict.values():
    print (value)
```

- prints all the values

dict.ipynb

30

Deleting/removing an item from a dictionary

- Using **pop()**, we can remove a particular item and get the value of the removed item at the same time.

```
mySquares = {1:1, 2:4, 3:9, 4:16, 5:25}
a=mySquares.pop(4) #removes item with key=4 and returns the removed item
print (a) # Output is -> 16
print (mySquares) # Output is {1: 1, 2: 4, 3: 9, 5: 25}
```

popitem() can be used to remove the last item and return its value form the dictionary.

```
mySquares = {1:1, 2:4, 3:9, 4:16, 5:25}
a=mySquares.popitem() #removes item with key=4 and returns it
print (a) # Output is -> (5,25)
print (mySquares) # Output is {1: 1, 2: 4, 3: 9, 4: 16}
```

All the items can be removed at once using the clear() method.

```
mySquares.clear()
```

dict.ipynb

31

Deleting/removing an item from a dictionary

We can also use the **del** keyword to remove individual items or the entire dictionary itself.

```
# delete a particular item
# create a dictionary of squares
mySquares = {1:1, 2:4, 3:9, 4:16, 5:25}
del mySquares[2]    # (2,4) is deleted.
print (mySquares)    # Outputs {1: 1, 3: 9, 4: 16, 5: 25}
del mySquares        # deleted entire dictionary
print (mySquares)    #gives a NameError
```

dict.ipynb

32

Examples for some Important dictionary methods

get() and **pop()** can be used with an additional argument. If key is not present, default value is returned.

mySquares.get(key, defaultval)

mySquares.pop(key,defaultval)

fromkeys() is used to return a new dictionary with keys from another dictionary and value equal to v (defaults to None).

```
mySquares = {1:1, 2:4, 3:9, 4:16, 5:25}
bb={ }.fromkeys(mySquares)
print (bb) # Outputs {1: None, 2: None, 3: None, 4: None, 5: None}

mySquares = {1:1, 2:4, 3:9, 4:16, 5:25}
bb={}.fromkeys(mySquares, 'hello')
print (bb) # Outputs {1: 'hello', 2: 'hello', 3: 'hello', 4: 'hell
```

```
{1: None, 2: None, 3: None, 4: None, 5: None}
{1: 'hello', 2: 'hello', 3: 'hello', 4: 'hello', 5: 'hello'}
```

33

Checking if a value is in the dictionary:

```
mySquares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
print(1 in squares) #True
print(2 not in squares) #True
if 49 in squares:  #False


mySquares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
for i in mySquares: # i is 1, 3, 5, 7, 9 in the loop sequences
    print(mySquares[i])
```

We can use for loop to iterate through a dictionary

Some built-in Functions

- all() : Return True if all keys of the dictionary are true (or if the dictionary is empty). It is False if any of the values is not defined (None)
- any() : Return True if any key of the dictionary is true. If the dictionary is empty, return False.
- len() : Return the length (the number of items) in the dictionary.
- cmp() : Compares items of two dictionaries.

sorted() :Return a new sorted list of keys in the dictionary.

34

Zipping iterables

- The **zip()** function takes iterables (can be zero or more), aggregates them in a tuple, and return it.

```
#zipping iterables
number_list = [1, 2, 3]
str_list = ['one', 'two', 'three']

# No iterables are passed
result = zip(number_list, str_list)

# Converting iterator to list
result_list = list(result)
print(result_list)
```

```
[(1, 'one'), (2, 'two'), (3, 'three')]
```

dict.ipynb

35

Functions –Extra Material

Functions may have a variable number of arguments

- We can have both normal and keyword variable number of arguments.

```
def myFun(*argv):
    for x in argv:
        print (x)
myFun('Hello', 'OzU', 'and', 'World')
```

```
Hello
OzU
and
World
```

functions2.ipynb

```
def myFun2(arg1, *argv):
    print ("First argument :", arg1)
    for x in argv:
        print("Next argument through *argv :", x)
myFun2('Hey', 'Hello', 'OzU', 'to', 'World', 'TR')
```

```
First argument : Hey
Next argument through *argv : Hello
Next argument through *argv : OzU
Next argument through *argv : to
Next argument through *argv : World
Next argument through *argv : TR
```

36

kwargs

functions2.ipynb

- The special syntax ****kwargs*** in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name *kwargs* with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).
  - You can use **kwargs to let your functions take an arbitrary number of keyword arguments ("kwargs" means "keyword arguments")
  - A keyword argument is where you provide a name to the variable as you pass it into the function.

```
def myFun3(**kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

myFun3(first ='OzU', mid ='is', last='Great')
```

```
first == OzU
mid == is
last == Great
```

37

Functions are first-class objects

Functions can be used as any other datatype, eg:

- Arguments to function
- Return values of functions
- Assigned to variables
- Parts of tuples, lists, etc

functions2.ipynb

```
#functions are first class objects
def square(x):
    return x*x
def applier(q,x):
    return q(x)

myVal=applier(square,7)
print (myVal)
```

```
49
```

38

Local Variables

- Local variable: variable that is assigned a value inside a function
  - Belongs to the function in which it was created
    - Only statements inside that function can access it, error will occur if another function tries to access the variable
- Scope: the part of a program in which a variable may be accessed
  - For local variable: function in which created
- Local variable cannot be accessed by statements inside its function which precede its creation
- Different functions may have local variables with the same name
  - Each function does not see the other function's local variables, so no confusion

39

Global Variables and Global Constants

- Global variable: created by assignment statement written outside all the functions
  - Can be accessed by any statement in the program file, including from within a function
  - If a function needs to assign a value to the global variable, the global variable must be redeclared within the function
    - General format: `global variable_name`
- Reasons to avoid using global variables:
  - Global variables making debugging difficult
    - Many locations in the code could be causing a wrong variable value
  - Functions that use global variables are usually dependent on them
    - Makes function hard to transfer to another program
  - Global variables make a program hard to understand

functions2.ipynb

```
global x,y
def ff():
    return x*y
```

```
x=10
y=21
print (ff())
```

Global!

```
210
```

40

Lambda Function

An important Pythonic style!

- It's like **inline functions** available in some of the other languages (for ex. C++)
  - A small, user-defined function to do a simple task
- In Python, a lambda function is known as an **anonymous** function.
- It can take any number of arguments, but can only have **one expression**.
- Similar functionality is possible using comprehensions; but lambda is extensively used in Python programs
- See some of the examples

It is possible to do same tasks using list comprehensions

(Lambda may be removed from Python in the future)

(Do it yourself)

41

Sample uses of Lambda

```
x = lambda a : a**2
t=x(5) ➔ the value of t is 5^2=25

x = lambda a, b : a * b
t=x(5, 6) ➔ the value of t is 5*6=30
```

There are popular and trickier uses ıf lambda.
See functions2.ipynb !!!

(Lambda may be removed from Python in the future)

42

Python mapping and filtering

- See the example

It is possible to do same tasks using list comprehensions
(Do it yourself)

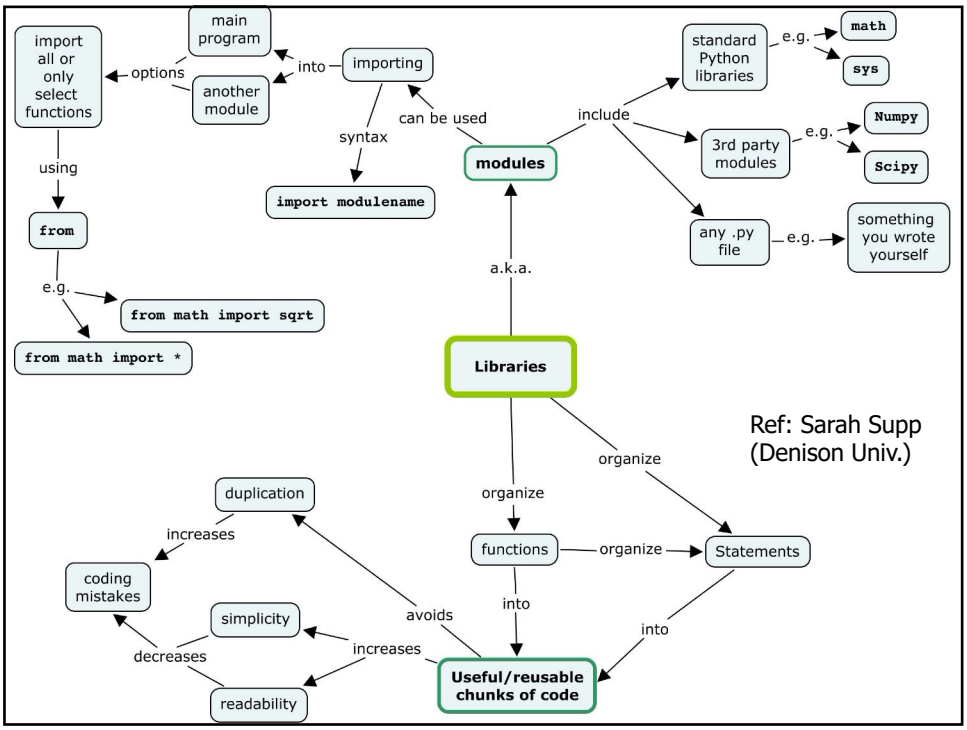
```
#map
def add1(x): return x+1
def odd(x): return x%2 == 1
def add(x,y): return x + y
result = map(add1, [1,2,3,4])
print (list(result))
result2 = list (map(add,[11,20,13,4],[100,200,300,400]))
print (result2)
result3 = filter(odd, [1,2,3,4])
print (list(result3))
```

```
[2, 3, 4, 5]
[111, 220, 313, 404]
[1, 3]

In [15]:
```

(map and filter may be removed from Python in the future)

43



44

Nested functions

(Extra material)

- A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.
- In Python, these non-local variables are read only by default and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them.

```
#nested functions
def print_msg(msg):
# This is the outer enclosing function
    def printer():
# This is the nested function
        print(msg)
    printer()
# We execute the function
# Output: Hello
print_msg("Hello world\n")
```

45

Defining a closure function

(Extra material)

```
def print_msg(msg):
# This is the outer enclosing function

    def printer():
# This is the nested function
        print(msg)

    return printer  # this got changed

# Now let's try calling this function.
# Output: Hello
another = print_msg("Hello")
another()
```

46

Rules for coding!

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer. So, make it human readable!
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. **All input is evil**, unless proven otherwise.
8. **A function should do one thing.**

47

Mapping Operators to Functions

Operation	Syntax	Function
Addition	a + b	add(a, b)
Concatenation	seq1 + seq2	concat(seq1, seq2)
Containment Test	obj in seq	contains(seq, obj)
Division	a / b	truediv(a, b)
Division	a // b	floordiv(a, b)
Bitwise And	a & b	and_(a, b)
Bitwise Exclusive Or	a ^ b	xor(a, b)
Bitwise Inversion	~ a	invert(a)
Bitwise Or	a | b	or_(a, b)
Exponentiation	a ** b	pow(a, b)
Identity	a is b	is_(a, b)
Identity	a is not b	is_not(a, b)
Indexed Assignment	obj[k] = v	setitem(obj, k, v)
Indexed Deletion	del obj[k]	delitem(obj, k)
Indexing	obj[k]	getitem(obj, k)
Left Shift	a << b	lshift(a, b)
Modulo	a % b	mod(a, b)
Multiplication	a * b	mul(a, b)
Matrix Multiplication	a @ b	matmul(a, b)
Negation (Arithmetic)	- a	neg(a)
Negation (Logical)	not a	not_(a)
Positive	+ a	pos(a)
Right Shift	a >> b	rshift(a, b)
Slice Assignment	seq[i:j] = values	setitem(seq, slice(i, j), values)
Slice Deletion	del seq[i:j]	delitem(seq, slice(i, j))
Slicing	seq[i:j]	getitem(seq, slice(i, j))
String Formatting	s % obj	mod(s, obj)
Subtraction	a - b	sub(a, b)
Truth Test	obj	truth(obj)
Ordering	a < b	lt(a, b)
Ordering	a <= b	le(a, b)
Equality	a == b	eq(a, b)
Difference	a != b	ne(a, b)
Ordering	a >= b	ge(a, b)
Ordering	a > b	gt(a, b)

Example:

```
In [4]: import operator

In [5]: operator.add(2,7)
Out[5]: 9
```

Ref: https://docs.python.org/3/library/operator.html

48

SEE YOU NEXT WEEK!!!

DR. ORHAN GÖKÇÖL
gokcol@gmail.com
orhan.gokcol@ozyegin.edu.tr

49