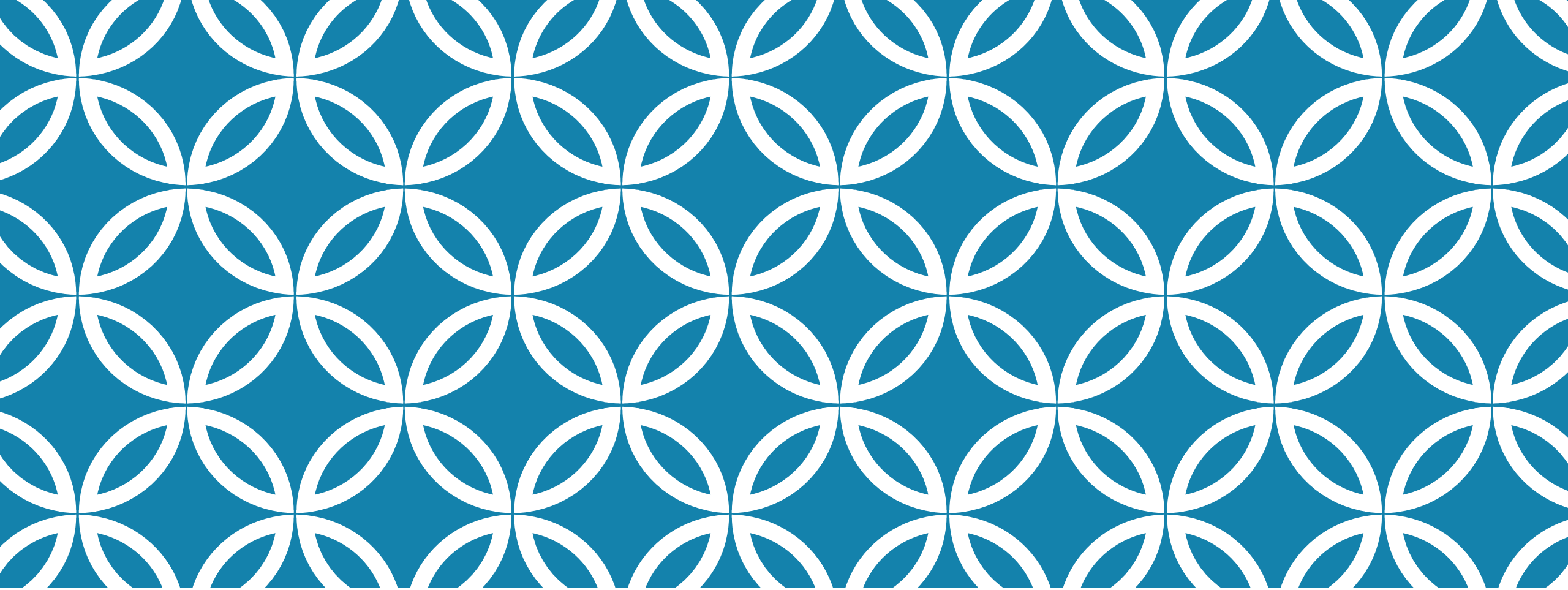


PROGRAMMING IN PYTHON

Gavrilut Dragos

Course 6

EXCEPTIONS

Exceptions in Python have the following form:

Python 3.x	Python 3.x

`try:`
`    #code`
`except:`
`    #code that will be executed in`
`    #case of any exception`
	`try:`
	`        x = 5 / 0`
	`except:`
	`        print("Exception")`

Output
Exception

EXCEPTIONS

Exceptions in Python have the following form:

Python 3.x

```
try:
    #code
except:
    #code that will be executed in
    #case of any exception
else:
    #code that will be executed if
    #there is no exception
```

Python 3.x

```
try:
        x = 5 / 1
except:
        print("Exception")
else:
        print("All ok")
```

Output

```
All ok
```

EXCEPTIONS

All exceptions in python are derived from **BaseException** class. There are multiple types of exceptions including: **ArithmeticError, BufferError, AttributeError, FloatingPointError, IndexError, KeyboardInterrupt, NotImplementedError, OverflowError, IndentationError,** and many more.

A list of all the exceptions can be found on:

- https://docs.python.org/3.8/library/exceptions.html#Exception
- https://docs.python.org/3.9/library/exceptions.html#Exception

A custom (user-defined) exception type can also used (more on this topic at "Classes").

EXCEPTIONS

Exceptions in Python have the following form:

Python 3.x

```
try:
    #code
except ExceptionType1:
    #code for exception of type 1
except ExceptionType2:
    #code for exception of type 1
except:
    #code for general exception
else:
    #code that will be executed if
    #there is no exception
```

Python 3.x

```
def Test (y):
    try:
        x = 5 / y
    except ArithmeticError:
        print("ArithmeticError")
    except:
        print("Generic exception")
    else:
        print("All ok")

Test(0)
Test("aaa")
Test(1)
```

Output

```
ArithmeticError
Generic exception
All ok
```

EXCEPTIONS

Exceptions in Python have the following form:

Python 3.x

```
try:
    #code
except ExceptionType1:
    #code for exception of type 1
except ExceptionType2:
    #code for exception of type 1
except:
    #code for general exception
else:
    #code that will be executed if
    #there is no exception
```

Python 3.x

```
def Test (y):
    try:
        x = 5 / y
    except:
        print("Generic exception")
    except ArithmeticError:
        print("ArithmeticError")
    else:
        print("All ok")

Test(0)
Test("aaa")
Test(1)
```

Generic exception must be the last one. Code will not compile.

EXCEPTIONS

Python also have a finally keyword that can be used to executed something at the end of the try block.

Python 3.x

```
try:
    #code
except:
    #code for general exception
else:
    #code that will be executed
    #if there is no exception
finally:
    #code that will be executed
    #after the try block execution
    #is completed
```

Python 3.x

```
def Test (y):
    try:
        x = 5 / y
    except:
        print("Error")
    else:
        print("All ok")
    finally:
        print("Final")
Test(0)
Test(1)
```

Output

Test(0):
Error
Final

Test(1):
All ok
Final

EXCEPTIONS

Python also have a finally keyword that can be used to executed something at the end of the try block.

Python 3.x

```
try:
    #code
except:
    #code for general exception
else:
    #code that will be executed
    #if there is no exception
finally:
    #code that will be executed
    #after the try block execution
    #is completed
```

Python 3.x

```
def Test (y):
    try:
        x = 5 / y
    except:
        print("Error")
    finally:
        print("Final")
    else:
        print("All ok")
Test(0)
Test(1)
```

Finally must be the last statement

EXCEPTIONS

Exceptions in Python have the following form:

Python 3.x

```
try:
    #code
except (Type1,Type2,…Typen):
    #code for exception of type
    #1,2,…
except:
    #code for general exception
else:
    #code that will be executed
    #if there is no exception
```

Python 3.x

```
def Test (y):
    try:
        x = 5 / y
    except (ArithmeticError,TypeError):
        print("ArithmeticError")
    except:
        print("Generic exception")
    else:
        print("All ok")

Test(0)
Test("aaa")
Test(1)
```

Output

```
ArithmeticError
ArithmeticError
All ok
```

EXCEPTIONS

Exceptions in Python have the following form:

Python 3.x

```
try:
    #code
except Type1 as <var_name>:
    #code for exception of type
    #1.
except:
    #code for general exception
else:
    #code that will be executed
    #if there is no exception
```

Python 3.x

```
try:
    x = 5 / 0
except Exception as e:
    print( str(e) )
```

Output

```
division by 0
```

EXCEPTIONS

Exceptions in Python have the following form:

Python 3.x

```
try:
    #code
except (Type1,Type2,…Typen) as <var>:
    #code for exception of type 1,2,… n
```

```
try:
    x = 5 / 0
except (Exception,ArithmeticError,TypeError) as e:
    print( str(e), type(e) )
```

Output

Python3: division by zero <class 'ZeroDivisionError'>

EXCEPTIONS

Python also has another keyword (**raise**) that can be used to create / throw an exception:

Python 3.x

```
raise ExceptionType (parameters)
raise ExceptionType (parameters) from <exception_var>
```

```
try:
    raise Exception("Testing raise command")
except Exception as e:
    print(e)
```

Output

```
Testing raise command
```

EXCEPTIONS

Each exception has a list of arguments (parameter ***args***)

Python 3.x

```
try:
    raise Exception("Param1",10,"Param3")
except Exception as e:
    params = e.args
    print (len(params))
    print (params[0])
```

Output

```
3
Param1
```

EXCEPTIONS

raise keyword can be used without parameters. In this case it indicates that the current exception should be re-raised.

Python 3.x

```
try:
    try:
        x = 5 / 0
    except Exception as e:
        print(e)
        raise
except Exception as e:
    print("Return from raise -> ",e)
```

Output (Python 3.x)

```
division by zero
Return from raise -> division by zero
```

EXCEPTIONS

Python 3.x supports chaining exception via **from** keyword.

Python 3.x

```
1  try:
2      x = 5 / 0
3  except Exception as e:
4      raise Exception("Error") from e
```

Output

```
Traceback (most recent call last):
  File "a.py", line 2, in <module>
    x = 5 / 0
ZeroDivisionError: division by zero
The above exception was the direct cause of the following exception:
Traceback (most recent call last):
  File "a.py", line 4, in <module>
    raise Exception("Error") from e
Exception: Error
```

EXCEPTIONS

Python has a special keyword (**assert**) that can be used to raise an exception based on the evaluation of a condition:

Python 3.x

```
age = -1
try:
        assert (age>0),"Age should be a positive number"
except Exception as e:
        print (e)
```

Output

```
Age should be a positive number
```

EXCEPTIONS

pass keyword is usually used if you want to catch an exception but do not want to process it.

Python 3.x

```
try:
        x = 10 / 0
except:
        pass
```

Some exceptions (if not handled) can be used for various purposes.

Python 3.x

```
print("Test")
raise SystemExit
print("Test2")
```

This exception (SystemExit) if not handle will imediatelly terminate your program

Output

```
Test
```

MODULES

Modules are python's libraries and extends python functionality. Python has a special keyword (**import**) that can be used to import modules.

Format (Python 3.x)

```
import module1,[module2,module3, … modulen]
```

Classes or items from a module can be imported separately using the following syntax.

Format (Python 3.x)

```
from module import object1,[object2,object3, … objectn]
from module import *
```

When importing a module aliases can also be made using "**as**" keyword

Format (Python 3.x)

```
import module1 as alias1,[module2 as alias2, … modulen as aliasn]
```

MODULES

Python has a lot of default modules (**os, sys, re, math,** etc).

There is also a keyword (**dir**) that can be used to obtain a list of all the functions and objects that a module exports.

Format (Python 3.x)

```
import math
print ( dir(math) )
```

Output (Python 3.x)

['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

The list of functions/items from a module may vary from Python 2.x to Python 3.x and from version to version, or from different versions of Python.

MODULES

Python distribution modules:

- Python 3.x ➔ https://docs.python.org/3/py-modindex.html

Module	Purpose
collections	Implementation of different containers
ctype	Packing and unpacking bytes into c-like structures
datetime	Date and Time operators
email	Support for working with emails
hashlib	Implementation of different hashes (MD5, SHA, …)
json	JSON encoder and decoder
math	Mathematical functions
os	Different functions OS specific (make dir, delete files, rename files, paths, …)

Module	Purpose
re	Regular expression implementation
random	Random numbers
socket	Low-level network interface
subprocess	Processes
sys	System specific functions (stdin,stdout, arguments, loaded modules, …)
traceback	Exception traceback
urllib	Handling URLs / URL requests, etc
xml	XML file parser

MODULES - SYS

Python documentation page:

- Python 3.x ➔ https://docs.python.org/3/library/sys.html#sys.modules

object	Purpose
sys.argv	A list of all parameters send to the python script
sys.platform	Current platform (Windows / Linux / MAC OSX)
sys.stdin sys.stdout, sys.stderrr	Handlers for default I/O operations
sys.path	A list of strings that represent paths from where module will be loaded
sys.modules	A dictionary of modules that have been loaded

MODULES - SYS

sys.argv provides a list of all parameters that have been send from the command line to a python script. The first parameter is the name/path of the script.

File 'test.py' (Python 3.x)

```
import sys
print ("First parameter is",sys.argv[0])
```

Output

```
>>> python.exe C:\test.py
First parameter is C:\test.py
```

MODULES - SYS

Python 3.x (File: sum.py)

```
import sys
suma = 0
try:
    for val in sys.argv[1:]:
        suma += int(val)
    print("Sum=",suma)
except:
    print("Invalid parameters")
```

Output

```
>>> python.exe C:\sum.py 1 2 3 4
Sum = 10

>>> python.exe C:\sum.py 1 2 3 test
Invalid parameters
```

MODULES - OS

Python documentation page:

- Python 3.x ➔ https://docs.python.org/3/library/os.html

Includes functions for:

- Environment
- Processes (PID, Groups, etc)
- File system (change dir, enumerate files, delete files or directories, etc)
- File descriptor functions
- Terminal informations
- Process management (spawn processes, fork, etc)
- Working with file paths

MODULES - OS

Listing the contents of a folder (os.listdir → returns a list of child files and folders).

Python 3.x

```
import os
print (os.listdir(".."))
```

Output

['$Recycle.Bin', 'Android', 'Documents and Settings', 'Drivers', 'hiberfil.sys', 'Program Files', 'Program Files (x86)', 'ProgramData', 'Python27', 'Python38', 'System Volume Information', 'Users', 'Windows', ...]

File and folder operations:

- os.mkdir / os.mkdirs → to create folders
- os.chdir → to change current path
- os.rmdir / os.removedirs → to delete a folder
- os.remove / os.unlink → to delete a file
- os.rename / os.renames → rename/move operations

MODULES - OS

os has a submodule (**path**) that can be used to perform different operations with file/directories paths.

Python 3.x

```
import os
print (os.path.join ("C:","Windows","System32"))
print (os.path.dirname ("C:\\Windows\\abc.txt"))
print (os.path.basename ("C:\\Windows\\abc.txt"))
print (os.path.splitext ("C:\\Windows\\abc.txt"))
print (os.path.exists ("C:\\Windows\\abc.txt"))
print (os.path.exists ("C:\\Windows\\abc.txt"))
print (os.path.isdir ("C:\\Windows"))
print (os.path.isfile ("C:\\Windows"))
print (os.path.isfile ("C:\\Windows\\abc.txt"))
```

Output

```
C:\Windows\System32
C:\Windows
abc.txt
["C:\Windows\abc", ".txt"]
False
True
False
False
```

MODULES - OS

Listing the contents of a folder recursively.

Python 3.x

```
import os

for (root,directories,files) in os.walk("."):
        for fileName in files:
                full_fileName = os.path.join(root,fileName)
                print (full_fileName)
```

Output

```
.\a
.\a.py
.\all.csv
.\run.bat
.\Folder1\version.1.6.0.0.txt
.\Folder1\version.1.6.0.1.txt
.\Folder1\Folder2\version.1.5.0.8.txt
```

os module can also be used to execute a system command or run an application via **system** function

Python 3.x

```
import os
os.system("dir *.* /a")
```

INPUT/OUTPUT

Python has 3 implicit ways to work with I/O:

A) **IN**: via keyboard (with **input** or **raw_input** keywords)

- There are several differences between python 2.x and python 3.x regarding reading from stdin

B) **OUT**: via **print** keyword

C) **IN/OUT**: via **open** keyword (to access files)

INPUT/OUTPUT

In Python 3.x, the content read from the input is considered to be a string and returned

Format (Python 3.x)

```
input ()
input (message)
```

Python 3.x

```
x = input("Enter: ")
print (x,type(x))
```

Python 3.x

```
>>> Enter: 10
10 <class 'str'>
```

```
>>> Enter: 1+2*3.0
1+2*3.0 <class 'str'>
```

```
>>> Enter: ”123”
"123" <class 'str'>
```

```
>>> Enter: test
test <class 'str'>
```

INPUT/OUTPUT

print can be used to print a string or an object/variable that can be converted into a string.

Format (Python 3.x)
`print` (*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Python 3.x	
>>> print ("test") test	>>> print ("test",10) test 10
>>> print ("test",10,sep="---") test---10	>>> print ("test");print("test2") test test2
>>> print ("test",end="***");print("test2") test***test2	

FILE MANAGEMENT

A file can be open in python using the keyword **open.**

Format (Python 3.x)

```
FileObject = open (filePath, mode='r', buffering=-1, encoding=None,
                   errors=None, newline=None, closefd=True, opener=None)
```

Where mode is a combination of the following:

- "r" – read (default)
- "w" – write
- "x" – exclusive creation (fail if file exists)
- "a" – append
- "b" – binary mode
- "t" – text mode
- "+" – update (read and write)

FILE MANAGEMENT

Python 3 also supports some extra parameters such as:

- **encoding** → if the file is open in text mode and you need translation from different encodings (UTF, etc)
- **error** → specify the way conversion errors for different encodings should be processed
- **newline** → also for text mode, specifies what should be consider a new line. If this value is set to None the character that is specific for the current operating system will be used

Documentation for **open** function:

- Python 3.x ➔ https://docs.python.org/3/library/functions.html#open

FILE MANAGEMENT

A file object has the following methods:

- f.close → closes current file
- f.tell → returns the current file position
- f.seek → sets the current file position
- f.read → reads a number of bytes from the file
- f.write → write a number of bytes into the file
- f.readline → reads a line from the file

Also – the file object is iterable and returns all text lines from a file.

Python 3.x

```
for line in open("a.py"):
        print (line.strip())
```

Lines read using this method contain the line-feed terminator. To remove it, use **strip** or **rstrip** methods.

FILE MANAGEMENT

Functional programming can also be used:

Python 3.x

```
x = [line for line in open("file.txt") if "Gen" in line.strip()]
print (len(x))
```

To read the entire content of the file in a buffer:

Python 3.x

```
data = open("file.txt","rb").read()
print (len(data))
print (data[0])
```

read method returns a string in Python 2.x and a buffer or string depending on how the file is opened ("rt" vs "rb") in Python 3.x → The output of the previous code will be a character (in Python 2.x) and a number representing the ascii code of that character in Python 3.x

To obtain a string in Python 3.x use "**rt**" instead of "**rb**"

FILE MANAGEMENT

To create a file and write content in it:

Python 3.x

```
open("file.txt","wt").write("A new file ...")
```

It is a good policy to embed file operation in a try block

Python 3.x

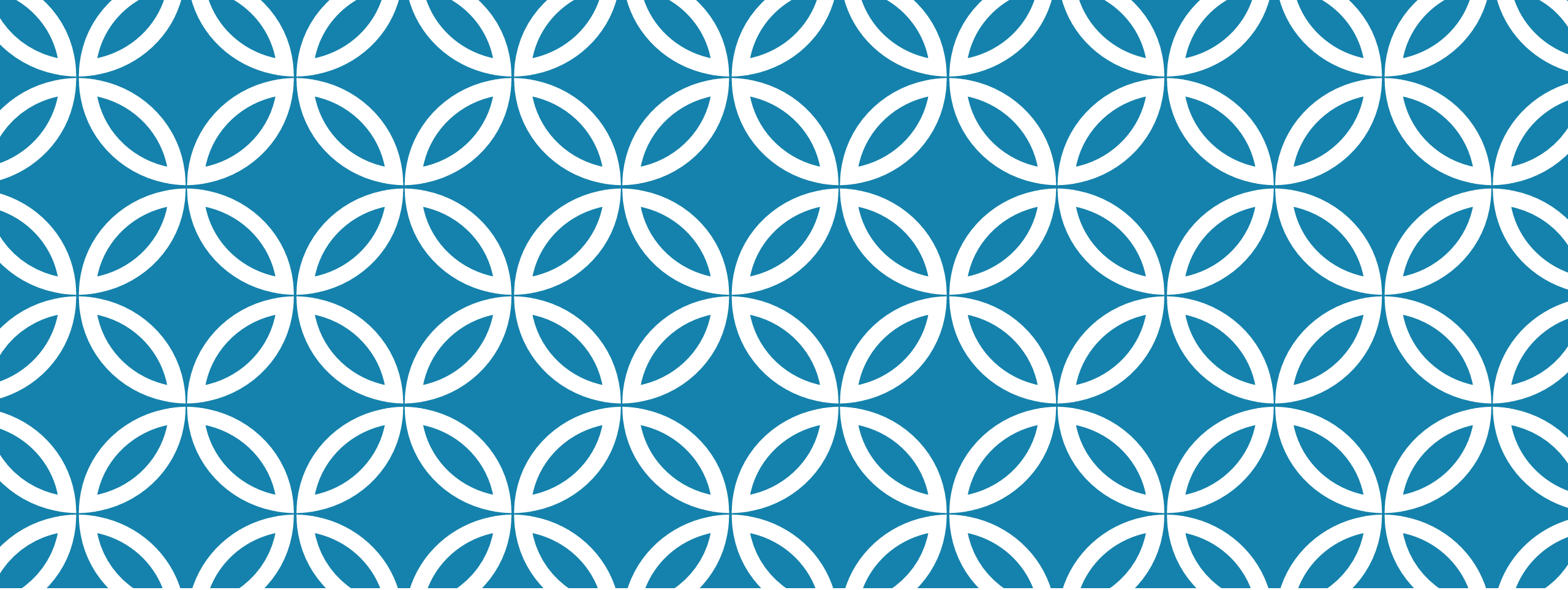
```
try:
    f = open("abc.txt")
    for line in f:
        print(line.strip())
    f.close()
except:
    print("Unable to open file abc.txt")
```

FILE MANAGEMENT

Once a file is open, the file object handle can be used to retrieve different information regarding that file:

Python 3.x

```
f = open("a.py","rb")
print ("File name      : ", f.name)
print ("File open mode : ", f.mode)
print ("Is it closed ? : ", f.closed)
```



PROGRAMMING IN PYTHON

Gavrilut Dragos

Course 5

CLASSES (INHERITANCE)

Python classes supports both simple and multiple inheritance.

Python 3.x (simple inheritance)

```
class <name>(Base):
      <statement1>
       …
      <statementn>
```

Where **statement$_i$** is usually a declaration of a method or data member.

Python 3.x (multiple inheritance)

```
class <name>(Base1,Base2,… Basem):
      <statement1>
       …
      <statementn>
```

CLASSES (INHERITANCE)

Python has two keywords (**issubclass** and **isinstance**) that can be used to check if an object is a subclass of an instance of a specific type.

Python 3.x (simple inheritance)

```
class Base:
        x = 10
class Derived(Base):
        y = 20

d = Derived()
print ("d.X = ",d.x)
print ("d.Y = ",d.y)
print ("Instance of Derived:",isinstance(d,Derived))
print ("Instance of Base:",isinstance(d,Base))
print ("Derived is a subclass of Base:",issubclass(Derived,Base))
print ("Base is a subclass of Derived:",issubclass(Base,Derived))
```

Output

```
d.X =  10
d.Y =  20
Instance of Derived: True
Instance of Base: True
Derived is a subclass of Base: True
Base is a subclass of Derived: False
```

CLASSES (INHERITANCE)

Inheritances does not assume that the __init__ function is automatically called for the base when the derived object is created.

Python 3.x (simple inheritance)

```
class Base:
        def __init__(self):
                self.x = 10

class Derived(Base):
        def __init__(self):
                self.y = 20

d = Derived()
print ("d.X = ",d.x)
print ("d.Y = ",d.y)
```

Execution error – d.X does not exists because base.__init__ was never called

CLASSES (INHERITANCE)

Inheritances does not assume that the __init__ function is automatically called for the base when the derived object is created.

Python 3.x (simple inheritance)

```
class Base:
        def __init__(self):
                self.x = 10

class Derived(Base):
        def __init__(self):
                Base.__init__(self)
                self.y = 20

d = Derived()
print ("d.X = ",d.x)
print ("d.Y = ",d.y)
```

In Python 3 you can also write super().__init__()

Output

```
d.X =  10
d.Y =  20
```

CLASSES (INHERITANCE)

Inheriting from a class will overwrite all base class members (methods or data members).

Python 3.x (simple inheritance)

```
class Base:
        def Print(self):
                print("Base class")

class Derived(Base):
        def Print(self):
                print("Derived class")

d = Derived()
d.Print()
```

Output

```
Derived class
```

CLASSES (INHERITANCE)

Inheriting from a class will overwrite all base class members (methods or data members).

Python 3.x (simple inheritance)

```
class Base:
        def Print(self,value):
                print("Base class",value)

class Derived(Base):
        def Print(self):
                print("Deri

d = Derived()
d.Print()
d.Print(100)
```

Print function from Base class was completely overwritten by **Print** function from the derived class. The code will produce a runtime error.

CLASSES (INHERITANCE)

Inheriting from a class will overwrite all base class members (methods or data members).

In this case member “x” from Base class will be overwritten by member “x” from the derived class.

Python 3.x (simple inheritance)

```
class Base:
        x = 10
class Derived(Base):
        x = 20

d = Derived()
print (d.x)
```

Output

```
20
```

CLASSES (INHERITANCE)

Polymorphism works in a similar way. In reality the inheritance is not necessary to accomplish polymorphism in Python.

Python 3.x (simple inheritance)

```
class Forma:
        def PrintName(self): pass
class Square(Forma):
        def PrintName(self): print("Square")
class Circle(Forma):
        def PrintName(self): print("Circle")
class Rectangle(Forma):
        def PrintName(self): print("Rectangle")

for form in [Square(),Circle(),Rectangle()]:
        form.PrintName()
```

Output

```
Square
Circle
Rectangle
```

CLASSES (INHERITANCE)

Polymorphism works in a similar way. In reality the inheritance is not necessary to accomplish polymorphism in Python.

Python 3.x (simple inheritance)

```
class Square:
        def PrintName(self): print("Square")
class Circle:
        def PrintName(self): print("Circle")
class Rectangle:
        def PrintName(self): print("Rectangle")

for form in [Square(),Circle(),Rectangle()]:
        form.PrintName()
```

Output

```
Square
Circle
Rectangle
```

CLASSES (INHERITANCE)

In case of multiple inheritance, Python derives from the right most class to the left most class from the inheritance list.

Python 3.x (multiple inheritance)

```
class BaseA:
        def MyFunction(self):
                print ("Base A")
class BaseB:
        def MyFunction(self):
                print ("Base B")
class Derived(BaseA, BaseB):
        pass

d = Derived()
d.MyFunction()
```

Output

```
Base A
```

CLASSES (INHERITANCE)

In case of multiple inheritance, Python derives from the right most class to the left most class from the inheritance list.

Python 3.x (multiple inheritance)

```
class BaseA:
        def MyFunction(self):
                print ("Base A")
class BaseB:
        def MyFunction(self):
                print ("Base B")
class Derived(BaseA, BaseB):
        pass

d = Derived()
d.MyFunction()
```

CLASSES (INHERITANCE)

In case of multiple inheritance, Python derives from the right most class to the left most class from the inheritance list.

Python 3.x (multiple inheritance)

```
class BaseA:
        def MyFunction(self):
                print ("Base A")
class BaseB:
        def MyFunction(self):
                print ("Base B")
class Derived(BaseA, BaseB):
        pass

d = Derived()
d.MyFunction()
```

Then **MyFunction** from class **BaseA** will overwrite **MyFunction** from **BaseB**

CLASSES (SPECIAL METHODS)

If we reverse the order (BaseB will be first and BaseA wil be the last one), MyFunction will print “Base B” instead of “Base A”

Python 3.x (multiple inheritance)

```
class BaseA:
        def MyFunction(self):
                print ("Base A")
class BaseB:
        def MyFunction(self):
                print ("Base B")

class Derived(BaseB, BaseA):
        pass

d = Derived()
d.MyFunction()
```

Output

```
Base B
```

CLASSES (SPECIAL METHODS)

Python defines a special set of functions that can be use do add additional properties to a class. Just like the initialization function (__init__) , these functions start and end with "__".

Function	Purpose
__repr__, __str__	Called when the object needs to be converted into string
__lt__, __le__, __eq__, __ne__, __gt__, __ge__	Operators used to compare instances of the same class.
__bool__	To evaluate the truth value of an object (instance of a class)
__getattr__, __getattribute__	For attribute look-ups
__setattr__, __delattr__ __set__, __get__	For attribute operations
__len__, __del__,	For len / del operators
__setitem__, __getitem__, __contains__, __reversed__, __iter__, __next__	Iterator operators

CLASSES (SPECIAL METHODS)

Python also defines a set of mathematical functions that can be used for the same purpose:

- __add__, __sub__, __mul__, __matmul__, __truediv__, __floordiv__, __mod__, __divmod__, __pow__, __lshift__, __rshift__, __and__, __xor__, __or__

- __radd__, __rsub__, __rmul__, __rmatmul__, __rtruediv__, __rfloordiv__, __rmod__, __rdivmod__, __rpow__, __rlshift__, __rrshift__, __rand__, __rxor__, __ror__,

- __iadd__, __isub__, __imul__, __imatmul__, __itruediv__, __ifloordiv__, __imod__, __ipow__, __ilshift__, __irshift__, __iand__, __ixor__, __ior__

- __neg__, __int__, __float__, __round__

CLASSES (SPECIAL METHODS)

Converting a class to a string. It is recommended to overwrite both __str__ and __repr__

Python 3.x

```
class Test:
        x = 10

class Test2:
        x = 10
        def __str__(self): return "Test2 with X = "+str(self.x)

t = Test()
t2 = Test2()
print (t,":",str(t))
print (t2, ":", str(t2))
```

Output (Python 3)

```
<__main__.Test object at 0x..> : <__main__.Test object at 0x..>
Test2 with X = 10 : Test2 with X = 10
```

CLASSES (SPECIAL METHODS)

Converting to an integer value.

Python 3.x

```
class Test:
        x = 10

class Test2:
        x = 10
        def __int__(self): return self.x

t = Test()
t2 = Test2()
Value = int(t)
```

This code will produce a runtime error because Python don't know how to translate an object of type Test to an integer

CLASSES (SPECIAL METHODS)

Converting to an integer value.

Python 3.x

```
class Test:
        x = 10

class Test2:
        x = 10
        def __int__(self): return self.x

t = Test()
t2 = Test2()
Value = int(t2)
```

This code works, **Value** will be 10

CLASSES (SPECIAL METHODS)

Iterating through a class instance

Python 3.x

```
class CarList:
        cars = ["Dacia","BMW","Toyota"]
        def __iter__(self):
                self.pos = -1
                return self
        def __next__(self):
                self.pos += 1
                if self.pos==len(self.cars): raise StopIteration
                return self.cars[self.pos]

c = CarList()
for i in c:
        print (i)
```

Output (Python 3)

```
Dacia
BMW
Toyota
```

CLASSES (SPECIAL METHODS)

Using class operators. In this case we overwrite __eq__ (==) operator.

Python 3.x

```
class Number:
        def __init__(self, value):
                self.x = value
        def __eq__(self, obj):
                return self.x+obj.x == 0

n1 = Number(-5)
n2 = Number(5)
n3 = Number(6)
print (n1==n2)
print (n1==n3)
```

Output

```
True
False
```

CLASSES (SPECIAL METHODS)

Overwriting the "**in**" opertator (__contains__).

Python 3.x

```
class Number:
        def __init__(self, value):
                self.x = value
        def __contains__(self, value):
                return str(value) in str(self.x)

n = Number(123)
print (12 in n)
print (5 in n)
print (3 in n)
```

Output

```
True
False
True
```

CLASSES (SPECIAL METHODS)

Overwriting the "**len**" opertator (__len__).

Python 3.x

```
class Number:
        def __init__(self, value):
                self.x = value
        def __len__(self):
                return len(str(self.x))

n1 = Number(123)
n2 = Number(99999)
n3 = Number(2)
print (len(n1),len(n2),len(n3))
```

Output

```
3 5 1
```

CLASSES (SPECIAL METHODS)

Building your own dictionary (overwrite __setitem__ and __getitem__)

Python 3.x

```
class MyDict:
        def __init__(self): self.data = []
        def __setitem__(self,key,value): self.data += [(key,str(value))]
        def __getitem__(self,key):
                for i in self.data:
                        if i[0]==key:
                                return i[1]

d = MyDict()
d["test"] = "python"
d["numar"] = 123
print (d["test"],d["numar"])
```

Output

```
python 123
```

CLASSES (SPECIAL METHODS)

Building a bit set (overloading operator [])

Python 3.x

```
class BitSet:
        def __init__(self): self.value = 0
        def __setitem__(self,index,value):
                if value: self.value |= (1 << (index & 31))
                else: self.value -= (self.value & (1 << (index & 31))
        def __getitem__(self,key):
                return (self.value & (1 << (index & 31)))!=0
b = BitSet()
b[0] = True
b[2] = True
b[4] = True
for i in range(0,8):
        print("Bit ",i," is ",b[i])
```

Output

```
Bit  0  is  True
Bit  1  is  False
Bit  2  is  True
Bit  3  is  False
Bit  4  is  True
Bit  5  is  False
Bit  6  is  False
Bit  7  is  False
```

CONTEXT MANAGER

A context manager is a mechanism where an object is created an notification about the moment that object is being access and the moment that object is being terminated.

Context managers are used along with **with** keyword. The objects that available in a context manager should implement __enter__ and __exit__ methods.

with $item_1$ **as** $alias_1$, [$item_2$ **as** $alias_2$, … $item_n$ **as** $alias_n$]:
    <statement 1>
    <statement 2>
      ….
    <statement n>

with $item_1$, [$item_2$, … $item_n$]:
    <statement 1>
    <statement 2>
      ….
    <statement n>

CONTEXT MANAGER

Whenever a **with** command is encounter, the following steps happen:

1. All items are evaluated
2. For all items __enter__ is called
3. If aliases are provided, the result of the __enter__ method is store into the alias
4. The block within the **with** is executed
5. If an exception appears, __exit__ is called and information related to the exception (type, value and traceback) are provided as parameters. If the __exit__ method returns false, the exception is re-raised. If the __exit__ method returns true, the exception is ignored.
6. If no exception appear, __exit__ is called with None parameters for (type, value and traceback). The result from the __exit__ method will be ignored.

CONTEXT MANAGER

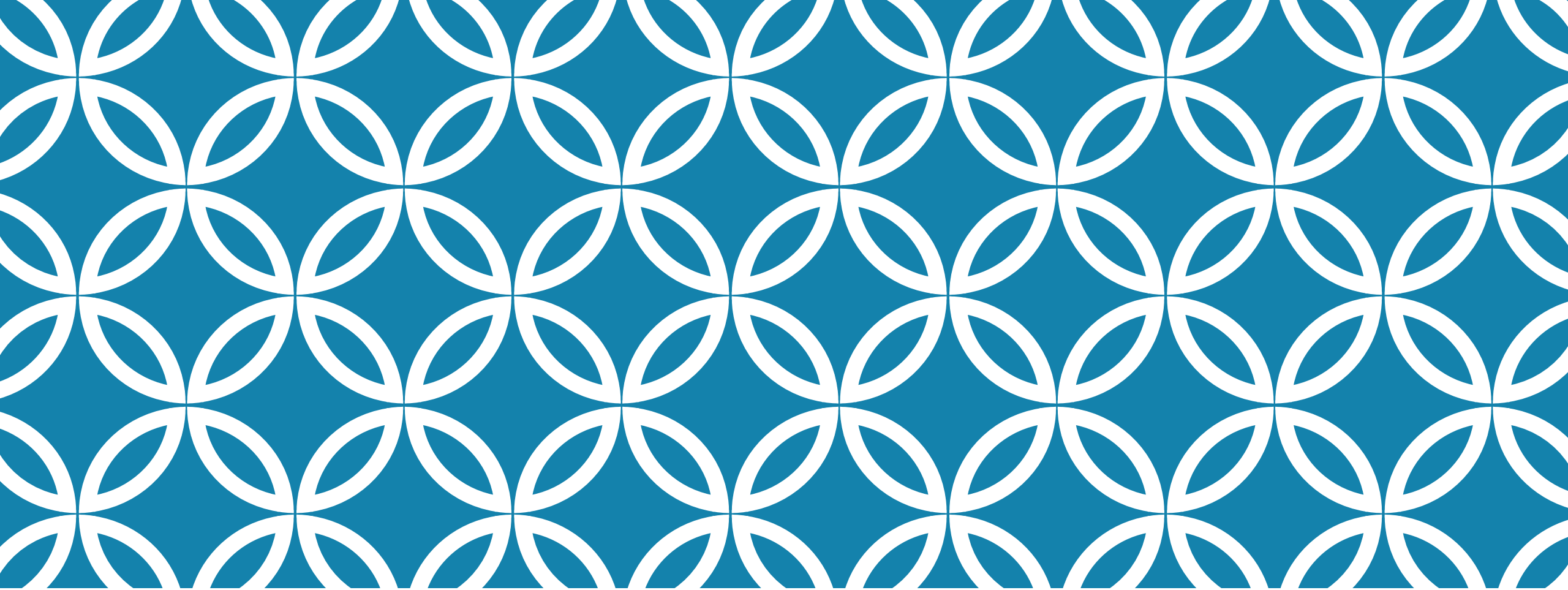
File context manager

Python 3.x

```
class CachedFile:
        def __init__(self,fileName):
                self.data = ""
                self.fileName = fileName
        def __enter__(self):
                print("__enter__ is called")
                return self
        def __exit__(self, exc_type, exc_value, traceback):
                print("__exit__ is called")
                open(self.fileName,"wt").write(self.data)
                return False
with CachedFile("Test.txt") as f:
        f.data = "Python course"
```

Output

```
__enter__ is called
__exit__ is called
```



PROGRAMMING IN PYTHON

Gavrilut Dragos

Course 4

CLASSES

Classes exists in Python but have a different understanding about their functionality than the way classes are defined in C-like languages. Classes can be defined using a special keyword: **class**

Python 3.x

```
class <name>:
      <statement_1>
       …
      <statement_n>
```

Where **statement**$_i$ is usually a declaration of a method or data member.

Documentation for Python classes can be found on:

- Python 3: https://docs.python.org/3/tutorial/classes.html

CLASSES

Classes have a special keyword (**self**) that resembles the keyword **this** from c-like languages.

Whenever you reference a data member (variable that belongs to a class) within the class definition the **self** keyword must be used.

Constructors can be defined by creating a "**__init__**" function. "__init__" function must have the first parameter **self**.

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p = Point()
print (p.x,p.y)
```

Output

```
0 0
```

Class Point has two members (x and y)

CLASSES

For a function defined within a class to be a method of that class it has to have the first parameter **self.**

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
        def GetX(self):
                return self.x

p = Point()
print (p.GetX())
```

Output

```
0
```

CLASSES

Defining a function within a class without having the first parameter **self** means that that function is a static function for that class.

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
        def GetY():
                return self.y

p = Point()
print (p.GetY())
```

Execution error (GetY is static)

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
        def GetY():
                print("Test")

Point.GetY()
```

Output

```
Python 3: will print "Test" on the screen
```

CLASSES

A data member can also be defined directly in the class definition. However, if mutable object are used the behavior is different (similar in terms of behavior to a static

Python 3.x

```
class Point:
        x = 0
        y = 0

p1 = Point()
p2 = Point()
p1.x = 10
p2.x = 20
print (p1.x,p2.x)
```

Output

```
10 20
```

Python 3.x

```
class Point:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Point()
p2 = Point()
p1.AddNumber(4)
p2.AddNumber(5)
print (p1.numbers)
print (p2.numbers)
```

Output

```
[1,2,3,4,5]
[1,2,3,4,5]
```

CLASSES

To avoid problems with mutable objects it is better to defined them in a constructor (__init__) function:

Python 3.x

```
class Point:
        def __init__(self):
                self.numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Point()
p2 = Point()
p1.AddNumber(4)
p2.AddNumber(5)
print (p1.numbers)
print (p2.numbers)
```

Output

```
[1,2,3,4]
[1,2,3,5]
```

CLASSES

It is not required for two instances of the same class to have the same members. A class instance is more like a dictionary where each key represent either a member function or a data member

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
print (p1.x,p1.y,p1.z)
```

Output

```
0 0 10
```

CLASSES

It is not required for two instances of the same class to have the same members. A class instance is more like a dictionary where each key represent either a member function or a data member

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
print (p1.x,p1.y,p2.z)
```

Error during runtine. “p2” does not have a data member “z” (only “p1” has a data member “z”)

CLASSES

It is not required for two instances of the same class to have the same members. A class instance is more like a dictionary where each key represent either a member function or a data member

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
print ("x" in dir(p1))
print ("z" in dir(p1))
print ("z" in dir(p2))
```

Output

```
True
True
False
```

CLASSES

We can write an equivalent representation of the functionality done by classes by using dictionaries:

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
```

Python 3.x (dictionary representation)

```
def PointClass__init__(obj):
        obj["x"] = 0
        obj["y"] = 0

Point = { "__init__":PointClass__init__ }
p1 = dict(Point)
p1["__init__"](p1)
p2 = dict(Point)
p2["__init__"](p2)
p1["z"] = 10
```

CLASSES

We can write an equivalent representation of the functionality done by classes by using dictionaries:

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0

p1 = Point()
p2 = Point()
p1.z = 10
```

Python 3.x (dictionary representation)

```
def PointClass__init__(obj):
        obj["x"] = 0
        obj["y"] = 0

Point = { "__init__":PointClass__init__ }
p1 = dict(Point)
p1["__init__"](p1)
p2 = dict(Point)
p2["__init__"](p2)
p1["z"] = 10
```

CLASSES

We can write an equivalent representation of the functionality done by classes by using dictionaries:

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
```

Python 3.x (dictionary representation)

```
def PointClass__init__(obj):
        obj["x"] = 0
        obj["y"] = 0

Point = { "__init__":PointClass__init__ }
p1 = dict(Point)
p1["__init__"](p1)
p2 = dict(Point)
p2["__init__"](p2)
p1["z"] = 10
```

CLASSES

We can write an equivalent representation of the functionality done by classes by using dictionaries:

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
```

Python 3.x (dictionary representation)

```
def PointClass__init__(obj):
        obj["x"] = 0
        obj["y"] = 0

Point = { "__init__":PointClass__init__ }
p1 = dict(Point)
p1["__init__"](p1)
p2 = dict(Point)
p2["__init__"](p2)
p1["z"] = 10
```

CLASSES

What happens if a class has some objects defined directly in class ?

Python 3.x

```
class Test:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Test()
p2 = Test()
p1.AddNumber(4)
p2.AddNumber(5)
```

As both **p1.numbers** and **p2.numbers** refer to the same vector (**numbers_vector**) they will both modify the same object thus creating the illusion of a static variable.

Python 3.x (dictionary representation)

```
numbers_vector = [1,2,3]
def TestClass_AddNumber(obj,n):
        obj["numbers"]+=[n]

TestClass = {
    "AddNumber":TestClass_AddNumber,
    "numbers":numbers_vector
}

p1 = dict(TestClass)
p2 = dict(TestClass)
p1["AddNumber"](p1,4)
p2["AddNumber"](p2,5)
```

CLASSES

What happens if a class has some objects defined directly in class ?

Python 3.x

```
class Test:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Test()
p2 = Test()
p1.AddNumber(4)
p2.AddNumber(5)
```

As both **p1.numbers** and **p2.numbers** refer to the same vector (**numbers_vector**) they will both modify the same object thus creating the illusion of a static variable.

Python 3.x (dictionary representation)

```
numbers_vector = [1,2,3]
def TestClass_AddNumber(obj,n):
        obj["numbers"]+=[n]

TestClass = {
    "AddNumber":TestClass_AddNumber,
    "numbers":numbers_vector
}

p1 = dict(TestClass)
p2 = dict(TestClass)
p1["AddNumber"](p1,4)
p2["AddNumber"](p2,5)
```

CLASSES

What happens if a class has some objects defined directly in class ?

Python 3.x

```
class Test:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Test()
p2 = Test()
p1.AddNumber(4)
p2.AddNumber(5)
```

As both **p1.numbers** and **p2.numbers** refer to the same vector (**numbers_vector**) they will both modify the same object thus creating the illusion of a static variable.

Python 3.x (dictionary representation)

```
numbers_vector = [1,2,3]
def TestClass_AddNumber(obj,n):
        obj["numbers"]+=[n]

TestClass = {
    "AddNumber":TestClass_AddNumber,
    "numbers":numbers_vector
}

p1 = dict(TestClass)
p2 = dict(TestClass)
p1["AddNumber"](p1,4)
p2["AddNumber"](p2,5)
```

CLASSES

What happens if a class has some objects defined directly in class ?

Python 3.x

```
class Test:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Test()
p2 = Test()
p1.AddNumber(4)
p2.AddNumber(5)
```

As both **p1.numbers** and **p2.numbers** refer to the same vector (**numbers_vector**) they will both modify the same object thus creating the illusion of a static variable.

Python 3.x (dictionary representation)

```
numbers_vector = [1,2,3]
def TestClass_AddNumber(obj,n):
        obj["numbers"]+=[n]

TestClass = {
    "AddNumber":TestClass_AddNumber,
    "numbers":numbers_vector
}

p1 = dict(TestClass)
p2 = dict(TestClass)
p1["AddNumber"](p1,4)
p2["AddNumber"](p2,5)
```

CLASSES

What happens if a class has some objects defined directly in class ?

Python 3.x

```
class Test:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Test()
p2 = Test()
p1.AddNumber(4)
p2.AddNumber(5)
```

As both **p1.numbers** and **p2.numbers** refer to the same vector (**numbers_vector**) they will both modify the same object thus creating the illusion of a static variable.

Python 3.x (dictionary representation)

```
numbers_vector = [1,2,3]
def TestClass_AddNumber(obj,n):
        obj["numbers"]+=[n]

TestClass = {
    "AddNumber":TestClass_AddNumber,
    "numbers":numbers_vector
}

p1 = dict(TestClass)
p2 = dict(TestClass)
p1["AddNumber"](p1,4)
p2["AddNumber"](p2,5)
```

CLASSES

You can also delete a member of a class instance by using the keyword **del.**

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p = Point()
print (p.x,p.y)
p.x = 10
print (p.x,p.y)
del p.x
print (p.x,p.y)
```

"x" is no longer a member of p. Code will produce a runtine error.

CLASSES

If a class member is like a dictionary – what does this means in terms of POO concepts:

A. method overloading is NOT possible (it would mean to have multiple functions with the same key in a dictionary). You can however create one method with a lot of parameters with default values that can be used in the same way.

B. There are no private/protected attributes for data members in Python. This is not directly related to the similarity to a dictionary, but it is easier this way as all keys from a dictionary are accessible.

C. CAST-ing does not work in the same way as expected. Up-cast / Down-cast are usually done with specialized functions that create a new object

D. Polymorphism is implicit (basically all you need to have is some classes with some functions with the same name). Even if this supersedes the concept of polymorphism, you don't actually need to have classes that are derived from the same class to simulate a polymorphism mechanism.

CLASSES

Just like normal variables in Python, data members can also have their type changed dynamically.

Python 3.x

```
class MyClass:
        x = 10
        y = 20

m = MyClass()
print (m.x,"=>",type(m.x))
m.x = "a string"
print (m.x,"=>",type(m.x))
```

Output

```
10 => <class 'int'>
a string => <class 'str'>
```

CLASSES

The same can be applied for class methods – however in this case there are some restrictions related to the **self** keyword.

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - "+str(self.x)+","+str(self.y)
m = MyClass()
print (m.Test(15),m.Test(16))
m.Test = m.MyFunction
print (m.Test(1,2))
```

Output

```
True False
3 - 10,20
```

CLASSES

The same can be applied for class methods – however in this case there are some restrictions related to the **self** keyword.

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
        def MyFunction(self,v1,v2):
                return str(v1+v2
                                                    lf.y)
m = MyClass()
print (m.Test(15),m.Test(
m.Test = MyClass.MyFunction
print (m.Test(1,2))
```

Runtime error because “MyFunction” is a method that needs to be bound to an object instance !

CLASSES

The same can be applied for class methods – however in this case there are some restrictions related to the **self** keyword.

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - "+str(self.x)+","+str(self.y)
m = MyClass()
print (m.Test(15),m.Test(16))
m.Test = MyClass().MyFunction
print (m.Test(1,2))
```

Output

```
True False
3 - 10,20
```

CLASSES

The same can be applied for class methods – however in this case there are some restrictions related to the **self** keyword.

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - "+str(self.x)+","+str(self.y)
m = MyClass()
m2 = MyClass()
print (m.Test(15),m.Test(16))
m.Test = m2.MyFunction
print (m.Test(1,2))
```

Output

```
True False
3 - 10,20
```

CLASSES

Methods are bound to the **self** object of the class they were initialized in. Even if you associate a method from a different class to a new method, the **self** will belong to the original class.

Python 3.x

```
class MyClass:
        x = 10
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - "+str(self.x)
m = MyClass()
m2 = MyClass()
m2.x = 100
m.Test = m2.MyFunction
print (m.Test(1,2))
print (m.MyFunction(1,2))
```

m.Test actually refers to **m2.MyFunction**

Output

```
3 - 100
3 - 10
```

CLASSES

A method from another class can also be used, but it will refer to the self from the original class.

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
class AnotherClass:
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - "+str(self.x)+","+str(self.y)
m = MyClass()
print (m.Test(15),m.Test(16))
m.Test = AnotherClass().MyFunction
print (m.Test(1,2))
```

The code will produce a runtime error because the **self** object from AnotherClass does not have “x” and “y” members.

CLASSES

Normal functions can also be used. However, in this case, the **self** object will not be send when calling them and it will not be accessible.

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
def MyFunction(self,v1,v2):
        return str(v1+v2)
m = MyClass()
print (m.Test(15),m.Test(16))
m.Test = MyFunction
print (m.Test(1,2))
```

Output

```
True False
3
```

CLASSES

Similarly a class method can be associated (linked) to a normal variable and used as such. It will be able to use the **self** and it will be affected if **self** members are changed.

Python 3.x

```
class MyClass:
        x = 10
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - self.x:"+str(self.x)
m = MyClass()
fnc = m.MyFunction
print (fnc(15,35))
m.x = 123
print (fnc(15,35))
```

Output

```
50 - self.x: 10
50 - self.x: 123
```

CLASSES

self object is assign during the construction of an object. This means that a function can be defined outside the class and used within the class if it is set during the construction phase.

Python 3.x

```
def MyFunction(self,v1,v2):
        return str(v1+v2)+" - X = "+str(self.x)

class MyClass:
        x = 10
        Test = MyFunction

m = MyClass()
m2 = MyClass()
m2.x = 15
print (m.Test(1,2))
print (m2.Test(10,20))
```

Output

```
3 - X = 10
30 - X = 15
```

CLASSES

This type of assignment can not be done within the constructor method (__init__), it must be done through direct declaration in the class body.

Python 3.x

```
def MyFunction(self,v1,v2):
        return str(v1+v2)+" - X = "+str(self.x)

class MyClass:
        x = 10
        def __init__(self):
                self.Test = MyFunction
m = MyClass()
m2 = MyClass()
m2.x = 15
print (m.Test(1,2))
print (m2.Test(10,20))
```

The code will produce a runtime error because MyFunction is not bound to any **self** at this point

CLASSES

The same error will appear if we try to link a method from a class using it's instance with a non-class function.

Python 3.x

```
def MyFunction(self,v1,v2):
        return str(v1+v2)+" - X = "+str(self.x)

class MyClass:
        x = 10

m = MyClass()
m.Test = MyFunction

print (m.Test(1,2))
```

The code will produce a runtime error because MyFunction is not bound to any **self** at this point

CLASSES

A class can be used like a container of data (a sort of name dictionary). It's closest resemblance is to a **struct** in C-like languages. For this an empty class need to be create (using keyword **pass**)

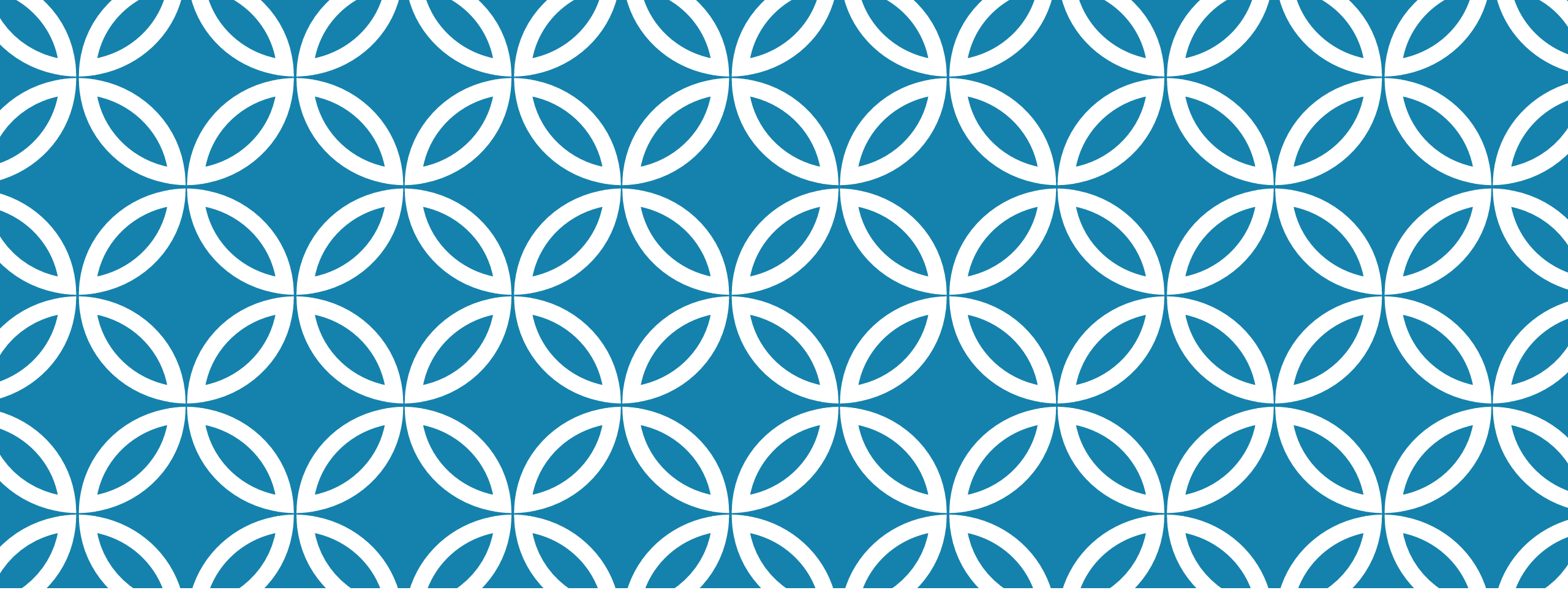
Python 3.x

```
class Point:
        pass

p = Point()
p.x = 100
p.y = 200
p_3d = Point()
p_3d.x = 10
p_3d.y = 20
p_3d.z = 30
print ("P = ",p.x,p.y)
print ("3D= ",p_3d.x,p_3d.y,p_3d.z)
```

Output

```
P =  100 200
3D=  10 20 30
```



PROGRAMMING IN PYTHON

Gavrilut Dragos

Course 3

(rev 1)

SETS

A list of unique data (two elements **a** and **b** are considered unique if **a** is different than **b** ➔ this translates as **a** is of a different type than **b** or if **a** and **b** are of the same type, that ***a != b***)

A special keyword **set** can be used to create a set. The **{** and **}** can also be used to build a set. Set keyword can be used to initialize a set from tuples ,lists or strings.

Sets supports some special mathematical operations like:

- Intersection
- Union
- Difference
- Symmetric difference

SETS

Python 3.x

```
x = set()                  #x is an empty set

x = {1,2,3}                #x is a set containing 3 elements: 1,2 and 3
x = {1,2,2,3,1,1}          #x is a set containing 3 elements: 1,2 and 3
x = {1,2,"AB","ab"}        #x is a set containing 4 elements: 1,2,"AB" and "ab"
x = set((1,2,3,2))         #x is a set containing 3 elements: 1,2 and 3
x = set([1,2,3,2])         #x is a set containing 3 elements: 1,2 and 3
x = set("Hello")           #x is a set containing 4 characters: H,e,l and o
```

SETS

Elements from a set can NOT be accessed (they are unordered collections):

Python 3.x

```
x = {'A', 'B', 2, 3, 'C'}
x[0], x[1], x[1:2], … ➔ all this expression will produce an error
```

Similarly – there is no addition operation defined between two sets:

Python 3.x

```
x = {'A', 'B', 2, 3, 'C'}
y = {'D', 'E', 1}
z = y + z                    #!!!ERROR !!
```

SETS

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Add a new element in the set (either use the member function(method) **add**)

Python 3.x

```
x = {1,2,3}          #x = {1, 2, 3}
x.add(4)             #x = {1, 2, 3, 4}
x.add(1)             #x = {1, 2, 3, 4}
```

- Remove an element from the set (methods **remove** or **discard**). Remove throws an error if the set does not contain that element. Use **clear** method to empty an entire set.

Python 3.x

```
x = {1,2,3}          #x = {1, 2, 3}
x.remove(1)          #x = {2, 3}
x.discard(2)         #x = {3}
x.discard(2)         #x = {3}
```

```
x = {1,2,3}          #x = {1, 2, 3}
x.clear()            #x = {}
```

SETS

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Several elements can be added to a set by either use the member function(method) **update** or by using the operator **|=**

Python 3.x

```
x = {1,2,3}                 #x = {1, 2, 3}
x |= {3,4,5}                #x = {1, 2, 3, 4, 5}
x.update({5,6})             #x = {1, 2, 3, 4, 5, 6}
x.update({5,6},{6,7})       #x = {1, 2, 3, 4, 5, 6, 7}
x.update({8},{6},{9})       #x = {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

- **update** method can be called with multiple parameters (sets)

SETS

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Union operation can be performed by using the operator | or the method **union**

Python 3.x

```
x = {1,2,3}
y = {3,4,5}
t = {2,4,6}
z = x | y | t              #z = {1, 2, 3, 4, 5, 6}
s = {7,8}
w = x.union(s)             #w = {1, 2, 3, 7, 8}
w = x.union(s, y, t)       #w = {1, 2, 3, 4, 5, 6, 7, 8}
```

- **union** method can be called with multiple parameters (sets)

SETS

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Intersection operation can be performed by using the operator & or the method **intersection**

Python 3.x

```
x = {1,2,3,4}
y = {2,3,4,5}
t = {3,4,5,6}
z = x & y & t                #z = {3, 4}
w = x.intersection(y)        #w = {2, 3, 4}
w = x.intersection(y, t)#w = {3, 4}
```

- **intersection** method can be called with multiple parameters (sets)

SETS

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Difference operation can be performed by using the operator - or the method **difference**

Python 3.x

```
x = {1,2,3,4}
y = {2,3,4,5}
z = x - y                  #z = {1}
z = y - x                  #z = {5}
w = x.difference(y)        #w = {1}
s = {1,2,3}
w = x.difference(y,s)      #w = {} → empty set
```

- **difference** method can be called with multiple parameters (sets)

SETS

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Symmetric difference operation can be performed by using the operator ^ or the method **symmetric_difference**

Python 3.x

```
x = {1,2,3,4}
y = {2,3,4,5}
z = x ^ y                          #z = {1, 5}
z = y ^ x                          #z = {1, 5}
w = x.symmetric_difference(y)      #w = {1, 5}
s = {1,2,3}
w = x.symmetric_difference(y,s)    #!!! ERROR !!!
```

- **symmetric_difference** method can **NOT** be called with multiple parameters (sets)

SETS

All sets operations also support some operations that apply to one variable such as:

- Intersection
  - **intersection_update**
  - **&=**
- Difference
  - **difference_update**
  - **-=**
- Symmetric difference
  - **symmetric_difference_update**
  - **^=**

SETS

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- To test if an element exists in a set, we can use the **in** operator

Python 3.x

```
x = {1,2,3,4}
y = 2 in x                    #y = True
z = 5 not in x                #z = True
```

- Total number of elements from a set can be found out using the **len** keyword

Python 3.x

```
x = {10,20,30,40}
y = len (x)                   #y = 4
```

SETS

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Use method **isdisjoint** to test if a set has no common elements with another one

Python 3.x

```
x = {1,2,3,4}
y = {10,20,30,40}
z = x.isdisjoint(y)                    #z = True
```

- Use method **issubset** or operator **<=** to test if a set is included in another one

Python 3.x

```
x = {1,2,3,4}
y = {1,2,3,4,5,6}
z = x.issubset(y)                      #z = True
t = x <= y                             #t = True
```

SETS

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Use method **issuperset** or operator **>=** to test if a set is included in another one

Python 3.x

```
x = {1,2,3,4}
y = {1,2,3,4,5,6}
z = y.issuperset(x)          #z = True
t = y >= x                   #t = True
```

- Operator **>** can also be used → it checks if a set is included in another **BUT** is not identical to it. Operator **<** can be used in the same way.

Python 3.x

```
x = {1,2,3,4}
y = {1,2,3,4,5,6}
t = y > x                    #t = True
```

```
x = {1,2,3,4}
y = {1,2,3,4}
t = y > x                    #t = False
```

SETS

Sets support a set of functions that can be used to modify its content. Some of these functionalities can also be achieved by using some operators.

- Use method **pop** to remove one element from the set. The remove element is different from Python 2.x to Python 3.x in terms of the order the element are kept in memory. Even if sets are unordered collection, in order to have quick access to different elements of the set these elements must be kept in memory in a certain way.

Python 3.x

```
x = {"A","a","B","b",1,2,3}
print (x)
print (x.pop())
```

Output (Python 3)

```
{1, 2, 3, 'b', 'B', 'A', 'a'}
1
```

- Use **copy** method to make a shallow copy of a set.

SETS AND FUNCTIONAL PROGRAMMING

A set can also be built using functional programming

- The main difference is that instead of operator [...] to build a set one need to use {...}

Python 3.x

```
x = {i for i in range(1,9)}                      #x = {1,2,3,4,5,6,7,8}
x = {i for i in range(1,100) if i % 23 == 0}     #x = {23, 46, 69, 92}
x = {i*i for i in range(1,6)}                    #x = {1, 4, 9, 16, 25}
```

- The condition of the set (all elements are unique) still applies. In the next case, only the first elements that meet the required criteria will be added.

Python 3.x

```
x = {i%5 for i in range(0,100)}                  #x = {0, 1, 2, 3, 4}
```

SETS AND BUILT-IN FUNCTIONS

The default build-in functions for list can also be used with sets and lambdas.

- Use **map** to create a new set where each element is obtained based on the lambda expression provided.

Python 3.x

```
x = {1,2,3,4,5}
y = set(map(lambda element: element*element,x))    #y = {1,4,9,16,25}
```

```
x = [1,2,3]
y = [4,5,6]
z = set(map(lambda e1,e2: e1+e2,x,y))              #z = {5,7,9}
```

SETS AND BUILT-IN FUNCTIONS

The default build-in functions for list can also be used with sets and lambdas.

- Use **filter** to create a new set where each element is filtered based on the lambda expression provided.

Python 3.x

```
x = [1,2,3,4,5]
y = set(filter(lambda element: element%2==0,x))      #y = {2,4}
```

- Both **filter** and **map** are used to create a set (usually in conjunction with **range** keyword)

Python 3.x

```
x = set(map(lambda x: x*x, range(1,10)))
#x = {1, 4, 9, 16, 25, 36, 49, 64, 81}
x = set(filter(lambda x: x%7==1,range(1,100)))
#x = {1, 8, 15, 22, 29, 36, 43, 50, 57, 64, 71, 78, 85, 92, 99}
```

SETS AND BUILT-IN FUNCTIONS

The default build-in functions for list can also be used with sets and lambdas.

- Other functions that work in a similar way as the build-in functions for list are **min, max, sum, any, all, sorted, reversed**

- **for** statement can also be used to enumerate between elements of a set

Python 3.x

```
for i in {1,2,3,4,5}:
        print(i)
```

- Python language also has another type → ***frozenset***. A frozen set has all the characteristics of a normal set, but it can not be modified. To create a frozen set use the **frozenset** keyword.

Python 3.x

```
x = frozenset ({1,2,3})
x.add(10)                                    #!!!ERROR!!!
```

DICTIONARIES

A dictionary is python implementation of a hash-map container. Design as a (key – value pair) where Key is a unique element within the dictionary.

A special keyword **dict** can be used to create a dictionary. The **{** and **}** can also be used to build a dictionary – much like in the case of sets.

Python 3.x

```
x = dict()                                  #x is an empty dictionary
x = {}                                      #x is an empty dict (typeof(x)="dict")
x = {"A":1, "B":2}                          #x is a dictionary with 2 keys
                                            #("A" and "B")
x = dict(abc=1,aaa=2)                       #equivalent to x= {"abc":1,  "aaa":2}
x = dict({"abc":1,"aaa":2})                 #equivalent to x= {"abc":1,  "aaa":2}
x = dict([("abc",1) ,("aaa",2)])            #equivalent to x= {"abc":1,  "aaa":2}
x = dict((("abc",1) ,("aaa",2)))            #equivalent to x= {"abc":1,  "aaa":2}
x = dict(zip(["abc","aaa"],[1,2]))          #equivalent to x= {"abc":1,  "aaa":2}
```

DICTIONARIES

To set a value in a dictionary use [] operator. The same operator can be used to read an existing value. If a value does not exist, an exception will be thrown.

Python 3.x

```
x = {}                          #x is an empty dictionary
x["ABC"] = 2                    #x is a dictionary with one key (ABC)
y = x["ABC"]                    #y = 2
y = x["test"]                   #!!! ERROR !!!
```

To check if a key exists in a dictionary, use **in** operator; **len** can also be used to find out how may keys a dictionary has.

Python 3.x

```
x = {"A":1, "B":2}              #x is a dictionary with 2 keys
"A" in x                        #True
len (x)                         #2
```

DICTIONARIES

Values from a dictionary can also be manipulated with **setdefault** member.

Python 3.x

```
x = {"A":1, "B":2}          #x = {"A":1,"B":2}
y = x.setdefault("C",3)     #x = {"A":1,"B":2,"C:3"},              y=3
y = x.setdefault("D")       #x = {"A":1,"B":2,"C:3","D":None},     y=None
y = x.setdefault("A")       #x = {"A":1,"B":2,"C:3","D":None},     y=1
y = x.setdefault("B",20)    #x = {"A":1,"B":2,"C:3","D":None},     y=2
```

Method **update** can also be used to change the value associated with a key.

Python 3.x

```
x = {"A":1, "B":2}          #x = {"A":1,"B":2}
x.update({"A":10})          #x = {"A":10,"B":2}
x.update({"A":100,"B":5})   #x = {"A":100,"B":5}
x.update({"C":3})           #x = {"A":100,"B":5,"C":3}
x.update(D=123,E=111)       #x = {"A":100,"B":5,"C":3,"D":123,"E":111}
```

DICTIONARIES

To delete an element from a dictionary use **del** keyword or **clear** method

Python 3.x

```
x = {"A":1, "B":2}                  #x = {"A":1,"B":2}
del x["A"]                          #x = {"B":2}
x.clear()                           #x is an empty dictionary
del x["C"]                          #!!! ERROR !!! "C" is not a key in x
```

To create a new dictionary you can use **copy** or static method **fromkeys**

Python 3.x

```
x = {"A":1, "B":2}                  #x={"A":1,"B":2}
y = x.copy()                        #makes a shallow copy of x
y["C"]=3                            #x={"A":1,"B":2},y={"A":1,"B":2,"C":3}
```

```
x = dict.fromkeys(["A","B"])        #x = {"A":None,"B":None}
x = dict.fromkeys(["A","B"],2)      #x = {"A":2,"B":2}
```

DICTIONARIES

Elements from the dictionary can also be accessed with method **get**

Python 3.x

```
x = {"A":1, "B":2}                  #x = {"A":1,"B":2}
y = x.get("A")                      #y = 1
y = x.get("C")                      #y = None
y = x.get("C",123)                  #y = 123
```

An element can also be extracted using pop method.

Python 3.x

```
x = {"A":1, "B":2}                  #x={"A":1,"B":2}
y = x.pop("A")                      #x={"B":2}, y = 1
y = x.pop("C",123)                  #x={"B":2}, y = 123
y = x.pop("D")                      #!!! ERROR !!! Key "D" does not exist
                                    #and no default value was provided
```

DICTIONARIES AND FUNCTIONAL PROGRAMMING

A dictionary can also be built using functional programming

Python 3.x

```
x = {i:i for i in range(1,9)}
#x = {1:1,2:2,3:3,4:4,5:5,6:6,7:7,8:8}
```

```
x = {i:chr(64+i) for i in range(1,9)}
#x = {1:"A",2:"B",3:"C",4:"D",5:"E",6:"F",7:"G",8:"H"}
```

```
x = {i%3:i for i in range(1,9)}
#x = {0:6,1:7,2:8} → last values that were updated
```

```
x = {i:chr(64+i) for i in range(1,9) if i%2==0}
#x = {2:"B", 4:"D", 6:"F", 8:"H"}
```

```
x = {i%3:chr(64+i) for i in range(1,9) if i<7}
#x = {1:"D", 2:"E", 0:"F"}
```

DICTIONARIES

Keys from the dictionary can be obtained with method keys

Python 3.x

```
x = {"A":1, "B":2}                    #x = {"A":1,"B":2}
y = x.keys()                          #y = ["A","B"] → an iterable object
```

To iterate all keys from a dictionary:

Python 3.x

```
x = {"A":1, "B":2}
for i in x:
        print (i)
```

```
x = {"A":1, "B":2}
for i in x.keys():
        print (i)
```

Output

```
A
B
```

DICTIONARIES

Values from the dictionary can be obtained with method **values**

Python 3.x

```
x = {"A":1, "B":2}                  #x = {"A":1,"B":2}
y = x.values()                      #y = ["1","2"] → an iterable object
```

To iterate all values from a dictionary:

Python 3.x

```
x = {"A":1, "B":2}
for i in x.values():
        print (i)
```

Output

```
1
2
```

Output order may be different for different versions of python depending on how data is stored/ordered in memory.

DICTIONARIES

All pairs from a dictionary can be obtained using the method **items**

Python 3.x

```
x = {"A":1, "B":2}                  #x = {"A":1,"B":2}
y = x.items()                       #y = an iterable object (Python 3) or
                                    #a list of tuples for Python 2.
                                    #[ ("A":1) , ("B":2) ]
```

To iterate all keys from a dictionary:

Python 3.x

```
x = {"A":1, "B":2}
for i in x.items():
        print (i)
```

Output

("A", 1)
("B", 2)

DICTIONARIES

Using the **items** method elements from a dictionary can be sorted according to their value.

Python 3.x

```
x = {
        "Dacia"  : 120,
        "BMW"    : 160,
        "Toyota" : 140
    }
for i in sorted(x.items(),key = lambda element : element[1]):
       print (i)
```

Output

("Dacia", 120)
("Toyota", 140)
("BMW", 160)

DICTIONARIES

Operator ** can be used in a function to specify that the list of parameters of that function should be treated as a dictionary.

Python 3.x

```
def GetFastestCar(**cars):
        min_speed = 0
        name = None
        for car_name in cars:
                if cars[car_name] > min_speed:
                        name = car_name
                        min_speed = cars[car_name]
        return name
fastest_car = GetFastestCar(Dacia=120,BMW=160,Toyota=140)
print (fastest_car)
#fastest_car = "BMW"
```

DICTIONARIES

Build-in functions such as **filter** can also be used with dictionaries.

Python 3.x

```
x = {
        "Dacia"  : 120,
        "BMW"    : 160,
        "Toyota" : 140
    }
y = dict(filter(lambda element : element[1]>=140,x.items()))
#y = {"Toyota":140, "BMW":160}
```

To delete an entire dictionary use **del** keyword.

DICTIONARIES

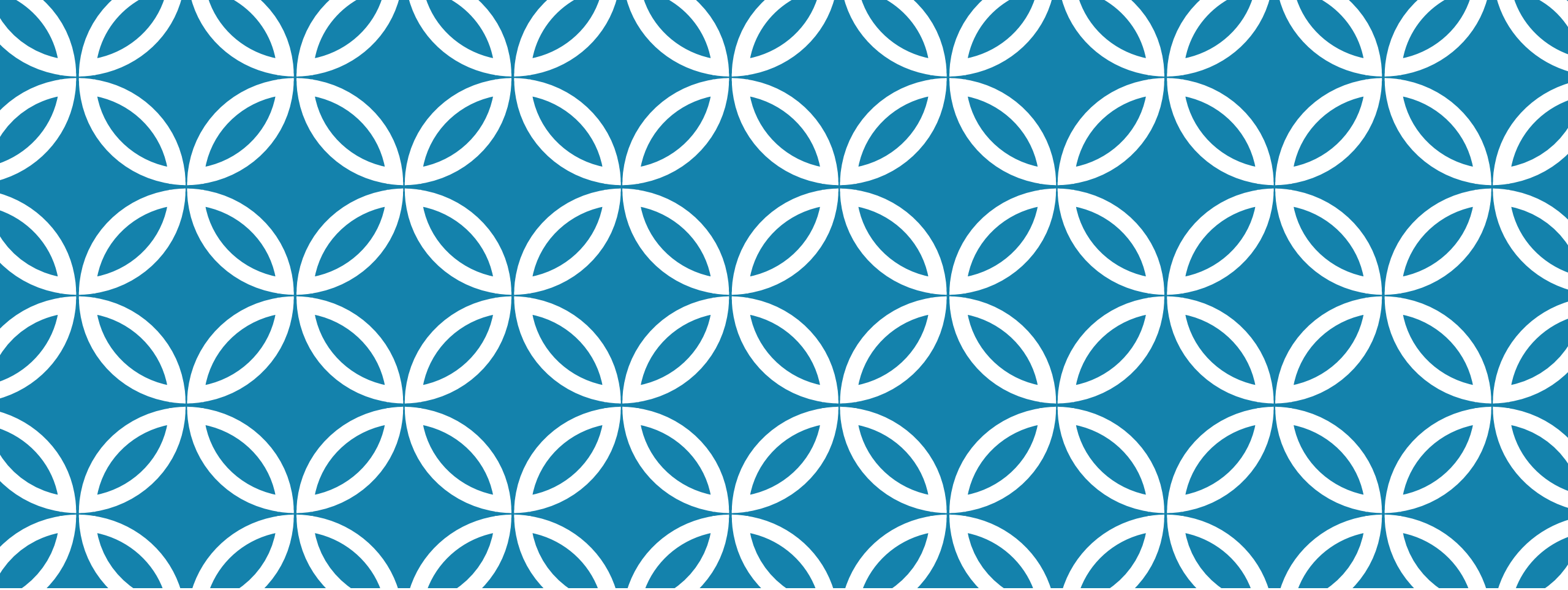
enumerate keyword can also be used with dictionaries.

Python 3.x	Output

`x = {`
`"Dacia" : 120,`
`"BMW" : 160,`
`"Toyota" : 140,`
`"Volvo" : 115,`
`"Renault" : 120,`
`}`
`for a in enumerate (x):`
`print (a)`
	`(0, 'Dacia')`
	`(1, 'BMW')`
	`(2, 'Toyota')`
	`(3, 'Volvo')`
	`(4, 'Renault')`

In this case, the resulted tuple contains the index and the key !

Just like in the case of lists (sequences), enumerate can receive a secondary parameter that states the initial index ➔ "enumerate (x,2)" will start with the index 2.



PROGRAMMING IN PYTHON

Gavrilut Dragos

Course 2

(rev. 1)

LAMBDA FUNCTIONS

A lambda function is a function without any name. It has multiple roles (for example it is often use as a pointer to function equivalent when dealing with other functions that expect a callback).

Lambdas are useful to implement closures.

A lambda function is defined in the following way:

lambda <*list_of_parameters*> : *return_value*

The following example uses lambda to define a simple addition function

Python 3.x(without lambda)	Python 3.x(with lambda)

`def addition (x,y):`
`  return x+y`
`print (addition (3,5))`
	`addition = lambda x,y: x+y`
	`print (addition(3,5))`

LAMBDA FUNCTIONS

Lambdas are bind during the run-time. This mean that a lambda with a specific behavior can be build at the run-time using the data dynamically generated.

Python 3.x

```
def CreateDivizibleCheckFunction(n):
        return lambda x: x%n==0
fnDiv2 = CreateDivizibleCheckFunction (2)
fnDiv7 = CreateDivizibleCheckFunction (7)
x = 14
print ( x, fnDiv2(x), fnDiv7(x) )
```

In this case fnDiv2 and fnDiv7 are dynamically generated.

This programming paradigm is called **closure**.

Output

```
14 True True
```

SEQUENCES

A sequence in python is a data structure represented by a vector of elements that don't need to be of the same type.

Lists have two representation in python:

- **list** ➔mutable vector (elements from that list can be added, deleted, etc). List can be defined using […] operator or the **list** keyword
- **tuple** ➔ immutable vector (the closest equivalent is a constant list) ➔ addition, deletion, etc operation can not be used on this type of object. A tuple is usually defined using (…) or by using the **tuple** keyword

list and **tuple** keywords can also be used to initialized a tuple or list from another list of tuple

SEQUENCES

Python 3.x

```
x = list ()              #x is an empty list
x = []                   #x is an empty list
x = [10,20,"test"]       #x is list
x = [10,]                #x is list containing [10]
x = [1,2] * 5            #x is list containing [1,2,  1,2,  1,2,  1,2,  1,2]
x,y = [1,2]              #x is 1 and y is 2
```

```
x = tuple ()             #x is an empty tuple
x = ()                   #x is an empty tuple
x = (10,20,"test")       #x is a tuple
x = 10,20,"test"         #x is a tuple
x = (10,)                #x is tuple containing (10)
x = (1,2) * 5            #x is tuple containing (1,2,  1,2,  1,2,  1,2,  1,2)
x = 1,2 * 5              #x is tuple containing (1,10)
x,y = (1,2)              #x is 1 and y is 2 (the same happens for x,y = 1,2 )
```

SEQUENCES

Elements from a list can be accessed in the following way

Python 3.x

```
x = ['A', 'B', 2, 3, 'C']

x[0]      #Result is A
x[-1]     #Result is C
x[-2]     #Result is 3
x[:3]     #Result is ['A', 'B', 2]
x[3:]     #Result is [3, 'C']
x[1:3]    #Result is ['B', 2]
x[1:-3]   #Result is ['B']
```

SEQUENCES

Elements from a tuple can be accessed in the same way

Python 3.x

```
x = ('A', 'B', 2, 3, 'C')

x[0]      #Result is A
x[-1]     #Result is C
x[-2]     #Result is 3
x[:3]     #Result is ('A', 'B', 2)
x[3:]     #Result is (3, 'C')
x[1:3]    #Result is ('B', 2)
x[1:-3]   #Result is ('B')
```

SEQUENCES

tuple and **list** keywords can also be used to convert a tuple to a list and vice-versa.

Python 3.x

```
x = ('A', 'B', 2, 3, 'C')
y = list (x) #y = ['A', 'B', 2, 3, 'C']
```

```
x = ['A', 'B', 2, 3, 'C']
y = tuple (x) #y = ('A', 'B', 2, 3, 'C')
```

Both lists and tuples can be concatenated, **but not with each other**.

Python 3.x

```
x = ('A', 2)
y = ('B', 3)
z = x + y
#z = ('A', 2, 'B', 3)
```

```
x = ['A', 2]
y = ['B', 3]
z = x + y
#z = ['A', 2, 'B', 3]
```

```
x = ('A', 2)
y = ['B', 3]
z = x + y
#!!! Error !!!
```

SEQUENCES

Tuples are also used to return multiple values from a function.

The following example computes both the sum and product of a sequence of numbers

Python 3.x

```
def ComputeSumAndProduct(*list_of_numbers):
        s = 0
        p = 1
        for i in list_of_numbers:
                s += i
                p *= i
        return (s,p)

suma,produs = ComputeSumAndProduct(1,2,3,4,5)
#suma =15, produs = 120
```

SEQUENCES

tuple and **list** can also be organized in matrixes:

Python 3.x

```
x = ((1,2,3), (4,5,6))
x = ([1,2,3], (4,5,6)) #matrix subcomponents don't have to be of the
                       #same type
x = ( ((1,2,3), (4,5,6)), ((7,8), (9,10,11, 12)) )
#a matrix does not have to have the same number of elements on each
#dimension
```

```
#the same rules from tuples apply to lists as well
x = [[1,2,3], [4,5,6]]
x = [[1,2,3], (4,5,6)]
```

SEQUENCES

Both **tuples** and **lists** can be enumerated with a **for** keyword:

Python 3.x

```
for i in [1,2,3,4,5]:
        print(i)
```

Python 3.x

```
for i in (1,2,3,4,5):
        print(i)
```

Output

```
1
2
3
4
5
```

Lists and tuples have a special keyword (**len**) that can be used to find out the size of a list/tuple:

Python 3.x

```
x = [1,2,3,4,5]
y = (10,20,300)
print (len(x), len(y))
```

Output 3.x

```
5 3
```

SEQUENCES

One can also use the **enumerate** keyword to enumerate a list and get the index of the item at the same time:

Python 3.x

```
for index,name in enumerate(["Dragos","Mihai","Nicu","Vlad"]):
        print("Index:%d => %s"%(index,name))
```

Or use an external variable:

Python 3.x

```
index = 0
for name in ["Dragos","Mihai","Nicu","Vlad"]):
        print("Index:%d => %s"%(index,name))
        index += 1
```

Output

```
Index:0 => Dragos
Index:1 => Mihai
Index:2 => Nicu
Index:3 => Vlad
```

SEQUENCES

enumerate functions also allows a second parameter to specify the index base (default is 0 → just like in C-like languages).

Python 3.x

```
for index,name in enumerate(["Dragos","Mihai","Nicu","Vlad"], 2 ):
        print("Index:%d => %s"%(index,name))
```

In this example, the index base will be 2:

- Dragos (the first name) will have index 2
- Mihai (the second name) will have index 3
- And so on …

Output

```
Index:2 => Dragos
Index:3 => Mihai
Index:4 => Nicu
Index:5 => Vlad
```

LISTS AND FUNCTIONAL PROGRAMMING

A list can also be build using functional programming.

- A list of numbers from 1 to 9

Python 3.x

```
x = [i for i in range(1,10)] #x = [1,2,3,4,5,6,7,8,9]
```

- A list of all divisor of 23 smaller than 100

Python 3.x

```
x = [i for i in range(1,100) if i % 23 == 0] #x = [23, 46, 69, 92]
```

- A list of all square values for number from 1 to 5

Python 3.x

```
x = [i*i for i in range(1,6)] #x = [1, 4, 9, 16, 25]
```

LISTS AND FUNCTIONAL PROGRAMMING

A list can also be build using functional programming.

- A list of pairs of numbers from 1 to 10 that summed up produce a number that divides with 7

Python 3.x

```
x=[[x, y] for x in range(1,10) for y in range(1,10) if (x+y)%7==0]
#x = [[1, 6], [2, 5], [3, 4], [4, 3], [5, 2], [5, 9], [6, 1],
#     [6, 8], [7, 7], [8, 6], [9, 5]]
```

- A list of tuples of numbers from 1 to 10 that summed up produce a number that divides with 7

Python 3.x

```
x=[(x, y) for x in range(1,10) for y in range(1,10) if (x+y)%7==0]
#x = [(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (5, 9), (6, 1),
#     (6, 8), (7, 7), (8, 6), (9, 5)]
```

LISTS AND FUNCTIONAL PROGRAMMING

A list can also be build using functional programming.

- A list of prime numbers that a smaller than 100

Python 3.x

```
x=[x for x in range(2,100) if len([y for y in range(2,x//2+1) if x % y==0])==0]
#x = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
      59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Using functional programming in Python drastically reduces the size of code. However, depending on how large the expression is to build a list, functional programming may not be advisable if the program purpose is readability.

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Add a new element in the list (either use the member function(method) **append** or the operator **+=**). To add lists or tuples use **extend** method

Python 3.x

```
x = [1,2,3]           #x = [1, 2, 3]
x.append(4)           #x = [1, 2, 3, 4]
x+=[5]                #x = [1, 2, 3, 4, 5]
x+=[6,7]              #x = [1, 2, 3, 4, 5, 6, 7]
x+=(8,9,10)           #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x[len(x):] = [11]     #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
x.extend([12,13])     #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
x.extend((14,15))     #x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                      #     14,15]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieved by using some operators.

- Insert a new element in the list using member function(method) **insert**

Python 3.x

```
x = [1,2,3]                 #x = [1, 2, 3]
x.insert(1,"A")             #x = [1, "A", 2, 3]
x.insert(-1,"B")            #x = [1, "A", 2, "B", 3]
x.insert(len(x),"C")        #x = [1, "A", 2, "B", 3, "C"]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Insert a new element or multiple elements can be done using [:] operator. Similarly [] operator can be used to change the value of one element

Python 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
x[2] = 20                #x = [1, 2, 20, 4, 5]
x[3:] = ["A","B","C"] #x = [1, 2, 20, "A", "B", "C"]
x[:4] = [10]             #x = [10, "B", "C"]
x[1:3] = ['x','y','z'] #x = [10, "x", "y", "z"]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Remove an element in the list ➔ using member function(method) **remove**. This method removes the first element with a given value

Python 3.x

```
x = [1,2,3]                 #x = [1, 2, 3]
x.remove(1)                 #x = [2, 3]
x.remove(100)               #!!! ERROR !!! - 100 is not a value from x
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- To remove an element from a specific position the **del** keyword can be used.

Python 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[2]                 #x = [1, 2, 4, 5]
del x[-1]                #x = [1, 2, 4]
del x[0]                 #x = [2, 4]
del x[1000]              #!!! ERROR !!! - 1000 is not a valid index
```

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[4:]                #x = [1, 2, 3, 4]
del x[:2]                #x = [3, 4]
```

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[2:4]               #x = [1, 2, 5]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- To **pop** method can be used to remove an element from a desire position an return it. This method can be use without any parameter (and in this case it refers to the last element)

Python 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
y = x.pop(2)             #x = [1, 2, 4, 5]      y = 3
y = x.pop(0)             #x = [2, 4, 5]         y = 1
y = x.pop(-1)            #x = [2, 4]            y = 5
y = x.pop()              #x = [2]               y = 4
y = x.pop(1000)          #!!! ERROR !!! - 1000 is not a valid index
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- To clear the entire list the **del** command can be used

Python 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
del x[:]                 #x = []
```

- Python 3.x also has a method **clear** that can be used to clear an entire list

Python 3.x

```
x = [1,2,3,4,5]          #x = [1, 2, 3, 4, 5]
x.clear()                #x = []
```

LISTS

Be aware that using the operator (=) does not make a copy but only a reference of a list.

Python 3.x

```
x = [1,2,3]
y = x
y.append(10)
#x = [1,2,3,10]
#y = [1,2,3,10]
```

If you want to make a copy of a list, use the **list** keyword:

Python 3.x

```
x = [1,2,3]
y = list (x)
y.append(10)
#x = [1,2,3]
#y = [1,2,3,10]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Python 3.x also has a method **copy** that can be used to create a shallow copy of a list

Python 3.x

```
x = [1,2,3]                 #x = [1, 2, 3]
b = x.copy()                #x = [1, 2, 3]    b = [1, 2, 3]
b += [4]                    #x = [1, 2, 3]    b = [1, 2, 3, 4]
```

- The operator [:] can also be use to achieve the same result

Python 3.x

```
x = [1,2,3]                 #x = [1, 2, 3]
b = x[:]                    #x = [1, 2, 3]    b = [1, 2, 3]
b += [4]                    #x = [1, 2, 3]    b = [1, 2, 3, 4]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Use **index** method to find out the position of a specific element in a list

Python 3.x

```
x = ["A","B","C","D"] #x = [”A”, ”B”, ”C”, ”D”, ”E”]
y = x.index("C")      #y = 2
y = x.index("Y")      #!!! ERROR !!! - ”Y” is not part of list x
```

- The operator **in** can be used to check if an element exists in the list

Python 3.x

```
x = ["A","B","C","D"] #x = [”A”, ”B”, ”C”, ”D”, ”E”]
y = "C" in x          #y = True
y = "Y" in x          #y = False
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Use **count** method to find out how many elements of a specific value exists in a list

Python 3.x

```
x = [1,2,3,2,5,3,1,2,4,2]  #x = [1,2,3,2,5,3,1,2,4,2]
y = x.count(2)             #y = 4 [1,2,3,2,5,3,1,2,4,2]
y = x.count(0)             #y = 0
```

- The **reverse** method can be used to reverse the elements order from a list

Python 3.x

```
x = [1,2,3]                #x = [1,2,3]
x.reverse ()               #x = [3,2,1]
```

LISTS

Lists support a set of functions that can be used to modify and access elements and modify the list of elements. Some of these functionalities can also be achieve by using some operators.

- Use **sort** method to sort elements from the list

sort (*key=None, reverse=False*)

Python 3.x (version 3.7.4 → sort algorithm might be different from one version to another)

```
x = [2,1,4,3,5]
x.sort()                                        #x = [1,2,3,4,5]
x.sort(reverse=True)                            #x = [5,4,3,2,1]
x.sort(key = lambda i: i%3)                     #x = [3,4,1,2,5]
x.sort(key = lambda i: i%3,reverse=True)        #x = [5,2,4,1,3]
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **map** to create a new list where each element is obtained based on the lambda expression provided.

map (*function, iterableElement*$_1$*, [iterableElement*$_2$*,... iterableElement*$_n$*]*)

Python 3.x

```
x = [1,2,3,4,5]
y = list(map(lambda element: element*element,x))   #y = [1,4,9,16,25]
```

```
x = [1,2,3]
y = [4,5,6]
z = list(map(lambda e1,e2: e1+e2,x,y))             #z = [5,7,9]
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- **map** function returns an iterable objetc in Python 3.x

Python

```
x = [1,2,3]
y = map(lambda element: element*element,x)
#y = iterable object  →Python 3.x
```

- to create a list from an iterable object, use the **list** keyword

Python

```
x = [1,2,3]
y = [4,5,6,7]
z = list(map(lambda e1,e2: e1+e2,x,y))   #z = [5,7,9] → Python 3.x
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **filter** to create a new list where each element is filtered based on the lambda expression provided.

Filter (*function, iterableElement*)

Python 3.x

```
x = [1,2,3,4,5]
y = list(filter(lambda element: element%2==0,x))   #y = [2,4]
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Both **filter** and **map** can also be used to create a list (usually in conjunction with **range** keyword)

Python 3.x

```
x = list(map(lambda x: x*x, range(1,10)))
#x = [1, 4, 9, 16, 25, 36, 49, 64, 81]

x = list(filter(lambda x: x%7==1,range(1,100)))
#x = [1, 8, 15, 22, 29, 36, 43, 50, 57, 64, 71, 78, 85, 92, 99]
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **min** and **max** functions to find out the biggest/smallest element from an iterable list based on the lambda expression provided.

max (*iterableElement, [key]*) **max** (el_1, el_2, … *[key]*)	**min** (*iterableElement, [key]*) **min** (el_1, el_2, … *[key]*)

Python 3.x

```
x = [1,2,3,4,5]
y = max (x)                          #y = 5
y = max (1,3,2,7,9,3,5)              #y = 9
y = max (x,key = lambda i: i % 3)    #y = 2
```

- If you want to use a **key** for max and/or min function, be sure that you added with the parameter name decoration: key = <function>, and not just the key_function or a lambda.

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **sum** to add all elements from an iterable object. Elements from the iterable objects should allow the possibility of addition with other elements.

sum (iterableElement, *[startValue]*)

- *startValue* represent the value from where to start summing the elements. Default is 0

Python 3.x

```
x = [1,2,3,4,5]
y = sum (x)                    #y = 15
y = sum (x,100)                #y = 115 (100+15)
```

```
x = [1,2,"3",4,5]
y = sum (x)                    #ERROR→ Can't add int and string
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **sorted** to sort the element from a list (iterable object). The key in this case represents a compare function between two elements of the iterable object.

sorted (iterableElement, *[key],[reverse]*)

- The *reverse* parameter if not specified is considered to be False

Python 3.x

```
x = [2,1,4,3,5]
y = sorted (x)                                  #y = [1,2,3,4,5]
y = sorted (x,reverse=True)                     #y = [5,4,3,2,1]
y = sorted (x,key = lambda i: i%3)              #y = [3,1,4,2,5]
y = sorted (x,key = lambda i: i%3,reverse=True) #y = [2,5,1,4,3]
```

- Just like in the precedent case, you must use the optional parameter with their name

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **reversed** to reverse the element from a list (iterable object).

Python 3.x

```
x = [2,1,4,3,5]
y = list (reversed(x))                                    #y = [5,3,4,1,2]
```

- Use **any** and **all** to check if at least one or all elements from a list (iterable objects) can be evaluated to true.

Python 3.x

```
x = [2,1,0,3,5]
y = any(x)     #y = True, all numbers except 0 are evaluated to True
y = all(x)     #y = False, 0 is evaluated to False
```

BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **zip** to group 2 or more iterable objects into one iterable object

Python 3.x

```
x = [1,2,3]
y = [10,20,30]
z = list(zip(x,y))   #z = [(1,10) , (2,20) , (3,30)]
```

- Use **zip** with * character to unzip such a list. The unzip variables are tuples

Python 3.x

```
x = [(1,2) , (3,4) , (5,6)]
a,b = zip(*x)          #a = (1,3,5) and b = (2,4,6)
```

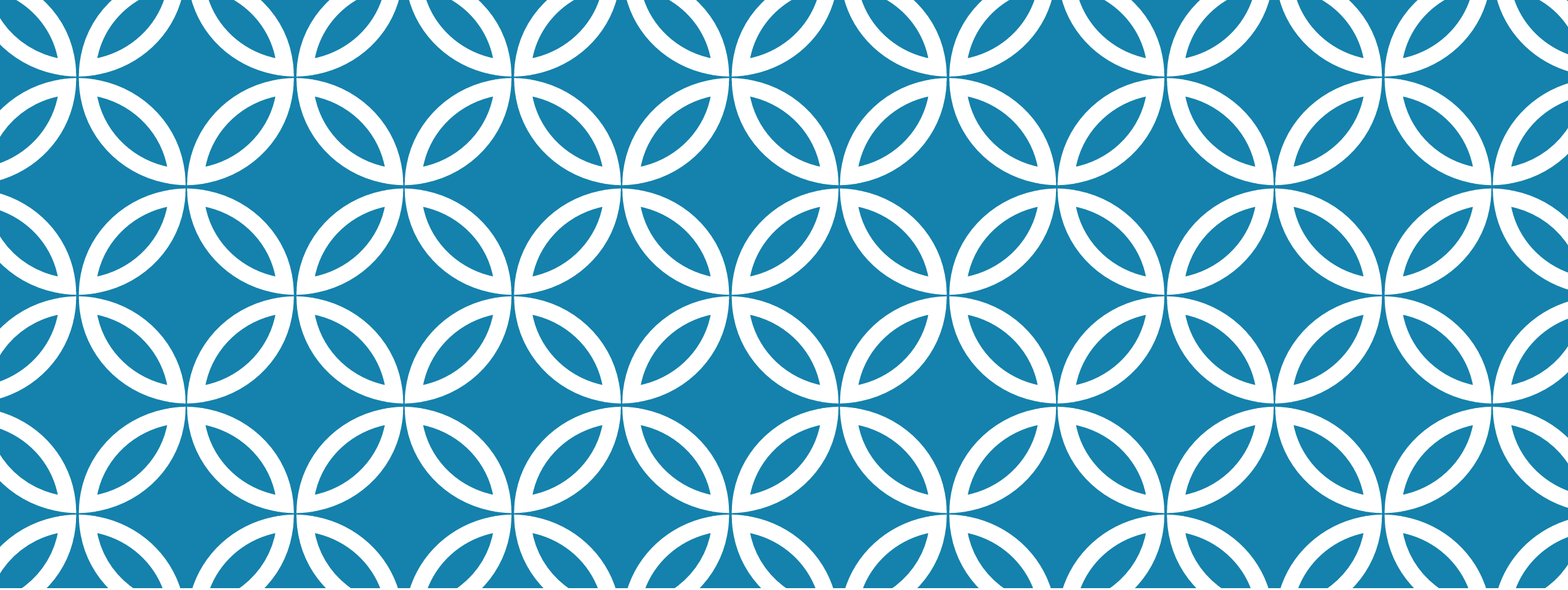
BUILT-IN FUNCTIONS FOR LIST

Python has several build-in functions design to work with list (iterators). These functions rely heavily on lambda expressions:

- Use **del** to delete a list or a tuple

Python 3.x

```
x = [1,2,3]
del x
print (x)      #!!!ERROR!!! x no longer exists
```



PROGRAMMING IN PYTHON

Gavrilut Dragos

Course 1

(rev.2)

ADMINISTRATIVE

Final grade for the Python course is computed using Gauss over the total points accumulated.

One can accumulate a maximum of 300 of points:

- A lab project (developed between week 8 and week 14) - up to 100 points. Projects selection will be decided in week 8
- Lab activity - maximum of 8 points / lab (starting with lab2) => 8 x 6 = 48 points
- Lab test (week 8) - up to 52 points
- Maximum 100 points at the final examination (course)

The minimum number of points that one needs to pass this exam:

- 120 points summed up from all tests
- 30 points minimum for each category (course, project and lab activity + lab test)

Course page: https://gdt050579.github.io/python-course-fii/

HISTORY

1980 – first design of Python language by Guido van Rossum

1989 – implementation of Python language started

2000 – Python 2.0 (garbage collector, Unicode support, etc)

2008 – Python 3.0

2020 – Python 2 is discontinued

Current Versions:

- 2.x → 2.7.18 (20.Apr.2020)
- 3.x → 3.12 (02.Oct.2023)

Download python from: https://www.python.org

Help available at : https://docs.python.org/3/

Python coding style: https://www.python.org/dev/peps/pep-0008/#id32

GENERAL INFORMATION

- Companies that are using Python: Google, Reddit, Yahoo, NASA, Red Hat, Nokia, IBM, etc
- TIOBE Index for September 2021 → **Python** is ranked no. 2 (Sep. 2021) → https://www.tiobe.com/tiobe-index/python/
- Default for Linux and Mac OSX distribution (both 2.x and 3.x versions)
- Open source
- Support for almost everything: web development, mathematical complex computations, graphical interfaces, etc.
- .Net implementation → IronPython (http://ironpython.net) for 2.x version

CHARACTERISTICS

- Un-named type variable
- Duck typing → type constrains are not checked during compilation phase
- Anonymous functions (lambda expressions)
- Design for readability (white-space indentation)
- Object-oriented programming support
- Reflection
- Metaprogramming → the ability to modify itself and create new types during execution

ZEN OF PYTHON

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Unless explicitly silenced.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

PYTHON EDITORS

Notepad++	➔ https://notepad-plus-plus.org/downloads/v7.7.1/
Komodo IDE	➔ http://komodoide.com
PyCharm	➔ https://www.jetbrains.com/pycharm/
VSCode	➔ https://marketplace.visualstudio.com/items?itemName=donjayamanne.python
Eclipse	➔ http://www.liclipse.com
PyDev	➔ https://wiki.python.org/moin/PyDev
WingWare	➔ http://wingware.com
PyZO	➔ http://www.pyzo.org
Thonny	➔ http://thonny.cs.ut.ee

……..

FIRST PYTHON PROGRAM

C/C++

```
void main(void)
{
        printf("Hello world");
}
```

Python 3.x

```
print ("Hello world")
```

VARIABLES

Variable are defined and use as you need them.

Python 3.x

```
x = 10                  #x is a number
s = "a string"          #s is a string
b = True                #b is a Boolean value
```

Variables don't have a fixed type – during the execution of a program, a variable can have multiple types.

Python 3.x

```
x = 10
#do some operations with x
x = "a string"
#x is now a string
```

BASIC TYPES

Python 3.x

```
x = 10  #x is an integer (32 bit precision)
x = 99999999999999999999999999999999 #x is a long (unlimited precision)
x = 1.123 #x is an float
x = 1.2j #x is a complex number
x = True #x is a bool ( a special case of integer )
x = None #x is a NoneType(the closest C/C++ equivalent is NULL/nullptr)
```

Python 3.x

```
x = 10
print (x, type (x))
```

Output

```
10 <class 'int'>
```

NUMERICAL OPERATIONS

Arithmetic operators (+, - , *, /, %) – similar to C like languages

Python 3.x

```
x = 10+20*3          #x will be an integer with value 70
x = 10+20*3.0        #x will be a float with value 70.0
```

Operator ** is equivalent with the pow function from C like languages

Python 3.x

```
x = 2**8             #x will be an integer with value 256
x = 2**8.1           #x will be a float with value 274.374
```

A number can be casted to a specific type using **int** or **float** method

Python 3.x

```
x = int(10.123)      #x will be an integer with value 10
x = float(10)        #x will be a float with value 10.0
```

NUMERICAL OPERATIONS

Division operator has a different behavior in Python 2.x and Python 3.x

Python 3.x

```
x = 10.0/3          #x will be a float with value 3.3333
x = 10.0%3          #x will be a float with value 1.0
```

Division between integers is interpreted differently

Python 2.x

```
x = 10/3
#x is 3 (int)
```

Python 3.x

```
x = 10/3
#x is 3.33333 (float)
```

A special operator exists // that means integer division (for integer operators)

Python 3.x

```
x = 10.0//3         #x will be a float with value 3.0
x = 11.9//3         #x will be a float with value 3.0
```

NUMERICAL OPERATIONS

Bit-wise operators (& , | , ^ , <<, >>). In particular & operator can be used to make sure that a behavior specific to a C/C++ operation can be achieve

C/C++

```
void main(void)
{
        unsigned int x;
        x = 0xFFFFFFFE;
        x = x + x;
        unsigned char y;
        y = 123;
        y = y + y;
}
```

Python 3.x

```
x = 0xFFFFFFFE
x = (x + x) & 0xFFFFFFFF
y = 123
y = (y + y) & 0xFF
```

NUMERICAL OPERATIONS

Compare operators (>, < , >=, <=, ==, !=). C/C++ like operators && and || are replaced with **and** and **OR.** Similar "**! operator**" is replaced with **not** keyword. However, unlike C/C++ languages Python supports a more mathematical like evaluation.

Python 3.x

```
x = 10 < 20 > 15          #x is True
                          #identical to (10<20) and (20>15)
```

All of these operators produce a bool result. There are two special values (keywords) defined in Python for constant bool values:

- True
- False

STRING TYPES

Python 3.x

```
s = "a string\nwith lines"
s = 'a string\nwith lines'
s = r"a string\nwithout any line"
s = r'a string\nwithout any line'
```

Python 3.x

```
s = """multi-line
string
"""
```

Python 3.x

```
s = '''multi-line
string
'''
```

STRING TYPES

Strings in python have support for different types of formatting – much like in C/C++ language.

Python 3.x

```
s = "Name: %8s Grade: %d"%("Ion",10)
```

If only one parameter has to be replaced, the same expression can be written in a simplified form:

Python 3.x

```
s = "Grade: %d"%10
```

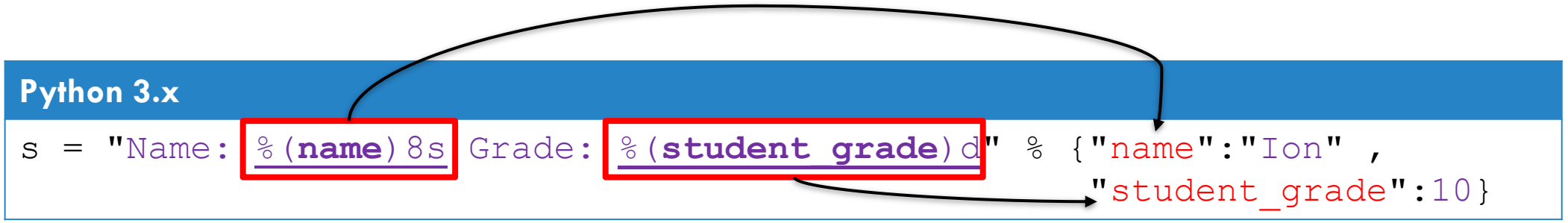
Two special keywords **str** and **repr** can be used to convert variables from any type to string.

Python 3.x

```
s = str (10)                    #s is ”10”
s = repr (10.25)                #s is ”10.25”
```

STRING TYPES

Formatting can be extended by adding naming to formatting variables.



Python 3.x

```
s = "Name: %(name)8s Grade: %(student_grade)d" % {"name":"Ion" ,
                                                  "student_grade":10}
```

A special character "\" can be place at the end of the string to concatenate it with another one from the next line.

Python 3.x

```
s = "Python"\
"Exam"
#s is "PythonExam"
```

STRING TYPES

Starting with version 3.6, Python also supports formatted string literals. These are strings preceded by an “f” or “F” character

Python 3.6+

```
a = 100
s = f"A = {a}"              #s will be 'A = 100'
s = f"A = {a+10}"           #s will be 'A = 110'
s = f"A = {float(a)}"       #s will be 'A = 100.0'
s = f"A = {float(a):10}"    #s will be 'A =       100.0' (preceded by spaces)
s = f"A = {a#:0x}"          #s will be 'A = 0x64'
```

There are some special characters that can be used to trigger a string representation for an object: !s (means **str**), !r (means **repr**), !a (means **ascii**)

More on this topic: https://docs.python.org/3/tutorial/inputoutput.html

STRING TYPES

Strings also support different ways to access characters or substrings

Python 3.x

```
s = "PythonExam"     #s is "PythonExam"

s[1]                 #Result is "y" (second character, first index is 0)
s[-1]                #Result is "m" → "PythonExam"(last character)
s[-2]                #Result is "a" → "PythonExam"
s[:3]                #Result is "Pyt" → "PythonExam"(first 3 characters)
s[4:]                #Result is "onExam" → "PythonExam"
                     #(all the characters starting from the 5th character
                     #of the string until the end of the string)
s[3:5]               #Result is "ho" → "PythonExam" (a substring that
                     #starts from the 3rd character until the 5th one)
s[2:-4]              #Result is "thon" → "PythonExam"
```

STRING TYPES

Strings also support a variety of operators

Python 3.x

```
s = "Python"+"Exam"#s is “PythonExam”
s = "A"+"12"*3     #s is “A121212” ➔ ”12” is multiplied 3 times
"A" in "Python"    #Result is False (”A” string does not exists in
                   #                 ”Python” string)
"A" not in "ABC"   #Result is False (”A” string exists in ”ABC”)
len (s)            #Result is 10 (10 characters in “PythonExam” string)
```

And slicing:

Python 3.x

```
s = "PythonExam"   #s is “PythonExam”
s[1:7:2]           #Result is ”yhn” (Going from index 1, to index 7
                   #with step 2 (1,3,5) ➔ PythonExam
```

STRING TYPES

Every string is considered a class and has member functions associated with it.
These methods are accessible through "." operator.

- Str.**startswith**("…") ➔ checks if a string starts with another one
- Str.**endswith**("…") ➔ checks if a string ends with another one
- Str.**replace**(toFind,replace,[count]) ➔ returns a string where the substring <*toFind*> is replaced by substring <*replace*>. Count is a optional parameter, if given only the firs <*count*> occurrences are replaced
- Str.**index**(toFind) ➔ returns the index of <*toFind*> in current string
- Str.**rindex**(toFind) ➔ returns the right most index of <*toFind*> in current string
- Other functions: **lower**(), **upper**(), **strip**(), **rstrip**(), **lstrip**(), **format**(), **isalpha**(), **isupper**(), **islower**(), **find**(…), **count**(…), etc

STRING TYPES

Strings splitting via **.split** function

Python 3.x

```
s = "AB||CD||EF||GH"
s.split("||")[2]    #Result is ”EF”. Split produces an array of 4
                    #elements AB,CD,EF and GH. The second element is EF
s.split("||")[-1]   #Result is ”GH”.
s.split("||",1)[0]  #Result is ”AB”. In this case the second parameter
                    #tells the function to stop after <count> (in this
                    #case 1) splits. Split produces an array of 2
                    #elements AB and CD||EF||GH. The fist element is AB
s.split("||",2)[2]  #Result is ”EF||GH”. Split produces an array of 3
                    #elements AB, CD and EF||GH.
```

Strings also support another function **.rsplit** that is similar to **.split** function with the only difference that the splitting starts from the end and not from the beginning.

BUILT-IN FUNCTIONS FOR STRINGS

Python has several build-in functions design to work characters and strings:

- **chr** (*charCode*) ➔ returns the string formed from one character corresponding to the code *charCode*. *charCode* is a Unicode code value.
- **ord** (character) ➔ returns the Unicode code corresponding to that specific character
- **hex** (number) ➔ converts a number to a lower-case hex representation
- **oct** (number) ➔ converts a number to a base-8 representation
- **format** ➔ to format a string with different values

STATEMENTS

Python is heavily based on indentation to express a complex instruction

C/C++

```
if (a>b)
{
        a = a + b
        b = b + a
}
```

Python 3.x

```
if a>b:
        a = a + b
        b = b + a
```

Python 3.x

```
if a>b:
        a = a + b
        b = b + a
```

Complex instruction

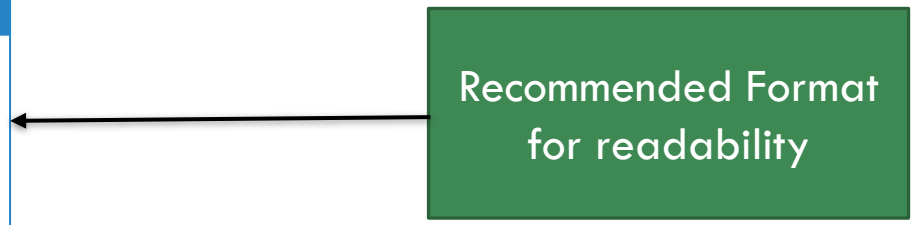
STATEMENTS

While python coding style recommends using indentation, complex instruction can be written in a different way as well by using a semicolon and add simple expression on the same line:

For example, the following expression:

Python 3.x

```
if a>b:
        a = a + b
        b = b + a
        b = a * b
```



Can also be written as follows:

Python 3.x

```
if a>b: a = a + b ; b = b + a ; b = a * b
```

IF-STATEMENT

Python 3.x

```
if expression:
        complex or simple statement
```

Python 3.x

```
if expression:
        complex or simple statement
else:
        complex or simple statement
```

Python 3.x

```
if expression:
        complex or simple statement
elif expression:
        complex or simple statement
```

Python 3.x

```
if expression:
        complex or simple statement
elif expression:
        complex or simple statement
elif expression:
        complex or simple statement
elif expression:
        complex or simple statement
…
else:
        complex or simple statement
```

SWITCH/CASE - STATEMENTS

Python (**until 3.10 version**) **does not have** a special keyword to express a switch statement. However, if-elif-else keywords can be used to describe the same behavior.

C/C++

```
switch (var) {
        case value_1:
                statements;
                break;
        case value_2:
                statements;
                break;
...
        default:
                statements;
                break;
}
```

Python 3.x

```
if var == value_1:
        complex or simple statement
elif var == value_2:
        complex or simple statement
elif var == value_3:
        complex or simple statement
…
else:   #default branch from switch
        complex or simple statement
```

Python 3.10 match…case statements will be discussed in course no. 2

WHILE - STATEMENT

C/C++

```
while (expression) {
        statements;
}
```

Python 3.x

```
while expression:
        complex or simple statement
```

Python 3.x

```
while expression:
        complex or simple statement
else:
        complex or simple statement
```

Python 3.x

```
a = 3
while a > 0:
        a = a - 1
        print (a)
else:
        print ("Done")
```


Output

```
2
1
0
Done
```

WHILE - STATEMENT

The **break** keyword can be used to exit the while loop. Using the **break** keyword will not move the execution to the **else** statement if present !

Python 3.x

```
a = 3
while a > 0:
        a = a - 1
        print (a)
        if a==2: break
else:
        print ("Done")
```

Output

```
2
```

WHILE - STATEMENT

Similarly, the **continue** keyword can be used to switch the execution from the while loop to the point where the while condition is tested.

Python 3.x

```
a = 10
while a > 0:
        a = a - 1
        if a % 2 == 0: continue
        print (a)
else:
        print ("Done")
```

Output

```
9
7
5
3
1
Done
```

DO...WHILE - STATEMENT

Python **does not have** a special keyword to express a do … while statement. However, using the **while…else** statement a similar behavior can be achieved.

C/C++

```
do {
        statements;
}
while (test_condition);
```

Python 3.x

```
while test_condition:
        statements
else:
        statements
```

Example:

C/C++

```
do {
        x = x - 1;
}
while (x > 10);
```

Python 3.x

```
while x > 10:
        x = x - 1
else:
        x = x - 1
```

FOR- STATEMENT

For statement is different in Python that the one mostly used in C/C++ like languages.
It resembles more a foreach statement (in terms that it only iterates through a list of objects, values, etc). Besides this, all of the other known keywords associated with a for (**break** and **continue**) work in a similar way.

Python 3.x

```
for <list_of_iterators_variables> in <list>:
        complex or simple statement
```

Python 3.x

```
for <list_of_iterators_variables> in <list>:
        complex or simple statement
else:
        complex or simple statement
```

FOR- STATEMENT

A special keyword **range** that can be used to simulate a C/C++ like behavior.

Python 3.x

```
for index in range (0,3):
        print (index)
```

Output

```
0
1
2
```

Python 3.x

```
for index in range (0,3):
        print (index)
else:
        print ("Done")
```

Output

```
0
1
2
Done
```

FOR- STATEMENT

range operator in Python 3.x returns an itereable object

range is declared as follows **range** (*start*, *end*, [*step*])

Python 3.x

```
for index in range (0,8,3):
        print (index)
```

Output

```
0
3
6
```

for statement will be further discuss in the course no. 2 after the concept of list is presented.

FUNCTIONS

Functions in Python are defined using **def** keyword

Python 3.x

```
def function_name (param1,param2,… paramn ):
        complex or simple statement
```

Parameters can have a default value.

Python 3.x

```
def function_name (param1,param2 [= defaultVal],… paramn[= defaultVal] ):
        complex or simple statement
```

And finally, **return** keyword can be used to return values from a function. There is no notion of void function (similar cu C/C++ language) → however, this behavior can be duplicated by NOT using the **return** keyword.

FUNCTIONS

Example of a function that performs a simple arithmetic operation

Python 3.x

```
def myFunc (x, y, z):
        return x * 100 + y * 10 + z
print ( myFunc (1,2,3) )                                    #Output:123
```

Parameters can be explicitly called

Python 3.x

```
def sum (x, y, z):
        return x * 100 + y * 10 + z
print ( sum (z=1,y=2,x=3) )                                 #Output:321
```

FUNCTIONS

Function parameters can have default values. Once a parameter is defined using a default value, every parameter that is declared after it should have default values.

Python 3.x

```
def myFunc (x, y=6, z=7):
        return x * 100 + y * 10 + z
print (myFunc (1) )                          #Output:167
print (myFunc (2,9) )                        #Output:297
print (myFunc (z=5,x=3) )                    #Output:365
print (myFunc (4,z=3) )                      #Output:463
print (myFunc (z=5) )                        #ERROR: missing x
```

Python 3.x

```
def myFunc (x=2, y, z=7):
        return x * 100 + y * 10 + z
```


Code will not compile as x has a default value, but Y does not !

FUNCTIONS

A function can return multiple values at once. This will also be discussed in course no. 2 along with the concept of tuple.

Python also uses **global** keyword to specify within a function that a specific variable is in fact a global variable.

Python 3.x

```
x = 10
def ModifyX ():
        x = 100
ModifyX ()
print ( x ) #Output:10
```

Python 3.x

```
x = 10
def ModifyX ():
        global x
        x = 100
ModifyX ()
print ( x ) #Output:100
```

FUNCTIONS

Functions can have a variable – length parameter (similar to the … from C/C++).
It is preceded by "*" operator.

Python 3.x

```
def multi_sum (*list_of_numbers):
        s = 0
        for number in list_of_numbers:
                s += number
        return s

print ( multi_sum (1,2,3) )                    #Output:6
print ( multi_sum (1,2) )                      #Output:3
print ( multi_sum (1) )                        #Output:1
print ( multi_sum () )                         #Output:0
```

FUNCTIONS

Functions can return values of different types. In this case you should check the type before using the return value.

Python 3.x

```
def myFunction(x):
        if x>0:
                return "Positive"
        elif x<0:
                return "Negative"
        else:
                return 0
result = myFunction (0)
if type(result) is int:
        print("Zero")
else:
        print(result)
```

FUNCTIONS

Functions can also contain another function embedded into their body.

That function can be used to compute results needed in the first function.

Python 3.x

```
def myFunction(x):
        def add (x,y):
                return x+y
        def sub(x,y):
                return x-y

        return add(x,x+1) + sub(x,2):
print (myFunction (5))
```

The previous code will print 14 into the screen.

FUNCTIONS

Functions can also be recursive (see the following implementation for computing a Fibonacci number)

Python 3.x

```
def Fibonacci (n):
        if n == 1:
                return 1
        elif n == 2:
                return 1
        else:
                return Fibonacci (n-1) + Fibonacci (n-2)

print ( Fibonacci (10) )
```

The previous code will print 55 into the screen.

FUNCTIONS

It is recommended to add a short explanation for every defined function by adding a multi-line string immediately after the function definition
https://www.python.org/dev/peps/pep-0257/#id15

Python 3.x

```
def Fibonacci (n):
        """
        Computes the n-th Fibonaci number using recursive calls
        """
        if n == 1:
                return 1
        elif n == 2:
                return 1
        else:
                return Fibonacci (n-1) + Fibonacci (n-2)
```

HOW TO CREATE A PYTHON FILE

- Create a file with the extension .py
- If you run on a Linux/OSX operation system, you can add the following line at the beginning of the file (the first line of the file):
  - `#!/usr/bin/python3` ➔ for python 3
  - `#!/usr/bin/python` ➔ for python (current version – usually 2)
- These lines can be added for windows as well ("#" character means comment in python so they don't affect the execution of the file too much
- Write the python code into the file
- Execute the file.
  - You can use the python interpreter directly (usually C:\Python27\python.exe or C:\Python310\python.exe for Windows) and pass the file as a parameter
  - Current distributions of python make some associations between .py files and their interpreter. In this cases you should be able to run the file directly without using the python executable.