CLASSES

PYTHON

OLD VS NEW

No real official Python terminology for Old-Style and New-style Classes

- ”new-style”
- ”old-style”, ”classic”

Avoid mixing new-style and old-style classes in your code !

SYNTAX

Minimal syntactic difference:

”Old-style” (Was the only choice until Python 2.1)

```
class old1:
        xxxxxxx

class old2(old1):
        yyyyyy
```

SYNTAX

”New-style” (from Python 2.2)

```
class new1(object):
        xxxxxxx

class new2(new1):
        yyyyyy
```

New-style always derive from the object ”object”.

New-style classes can use descriptors.

OUR INSTANCES

```
>>> type(old_instance)
>>> <type 'instance'>

>>> type(new_instance)
>>> <class '__main__.new1'>
```

New-style classes were introduced to unify the concepts of classes and types.

A new-style class is now a user-defined type.

SPECIAL METHODS

For new-style classes:

Special methods only works if implemented on the ”type” objects.

(Not directly in the instance dictionary)

```
class myOldStyleClass:
    pass

a = myOldStyleClass()
a.__len__ = lambda: 27
len(a)
>>> 27
```

For a new-style class:

```
>>> TypeError: object of type 'myNewStyleClass' has no len()
```

A BASIC CLASS

Informator
A SOFRANO COMPANY

```
class Hello(object):
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def writeMyClassName(self):
        return self.__class__.__name__

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age
```

Heading/tailing underscores are used for different purposes.

THE UNDERSCORE

A single leading underscore: _myInternal

Just a simple convention "Stay away from this. Internal use only."

But "from moduleX import *" skips objects with a leading underscore.

Double leading underscore: __myPrivateVar

Changes the name of a class attribute/method to a mangled name like: __class__myPrivateVar

Two classes in a hierarchy could now have the same class attribute/methods without collisions.

Also works on instance attributes/methods.

CALLING METHODS

```
a = Hello('Jonas', 14)

print a.get_name()
```

The syntax for calling a method is:

```
instance.method(args)
```

The Python engine transforms this into:

```
class.method(instance, args)
```

INSTANCE ATTRIBUTES

The basic syntax for grabbing attributes using an instance is:

```
instance.attribute
```

The Python engine is transforming this into:

```
instance.__getattribute__('attribute')
```

That is equal to:

```
class.__getattribute__(instance,'attribute')
```

The __getattribute__ method is basically doing:

```
instance.__dict__['attribute']
```

CLASS ATTRIBUTES

Informator
A SOFRAND COMPANY

```
class myClass(object):

    y = 2

    def __init__(self, n):
        self.n = n
        myClass.x = 0
```

n = instance attribute

y = class attribute

x = class attribute

LOOKUP CHAIN

Instance.x has a lookup chain starting with:

instance.__dict__['x']

then moves on and look into:

instance.__class__.__dict__['x']

Then it continues through the base classes of instance.__class__

DEFAULT VALUES

You could use **class attributes** to give **default values**.

```
class myClass(object):
    maxNr = 15

    def __init__(self):
        self.storage = []

    def item(self, n):
        return self.storage[n]

    def add(self, x):
        if len(self.data >= self.maxNr):
            raise Exception("Storage full")
        self.storage.append(x)
```

EXERCISE

Create a nodeConfig-object that has the following instance attributes:

- ip
- netmask
- gw
- numberOfTestsPerformed

Add the following magic special method. It should return the ip/netmask/gw info.

- __str__

Extra: Add the __repr__ method. What info should it return?

EMULATE BUILT-INS

PYTHON

EMULATING BUILT-INS

Emulating other built-in types is very useful for your own objects

- functions
- iterators
- containers

EMUL. FUNCTIONS

To get a callable object we could just add the following method to our class:

```
def __call__(self, x):
    return self.__age + x
```

```
hello1 = Hello(7)
print hello1(3)
```

or we can skip using a named instance and do like below

```
Hello(7)(3)
```

EMUL. FUNCTIONS

The standard ”callable” syntax is just a shortcut:

```
hello1 = Hello(7)
hello1(3)
```

The last call is actually the same as writing:

```
hello1.__call__(3)
```

EMUL. CONTAINERS

Containers are usually sequences or mappings.

If you need support for addition/concatenation, multiplication/repetition you implement:
__add__(), __radd__(), __iadd__(), __mul__(), __rmul__(), __imul__()

Mutable sequences is recommended to have:
append(), count(), index(), extend(), insert(), pop(), remove(), reverse() and sort()

EMUL. CONTAINERS

Your classes with ”mapping”-behaviour should contain:

keys(), values(), items(), has_key(), get(), clear(), setdefault(), iterkeys(), itervalues(), iteritems(), pop(), popitem(), copy(), update()

All your sequences and mappings should have:

```
__iter__()

__contains__()                    # to implement the IN-opretaror
```

EMUL. CONTAINERS

There is a bunch more generic container methods that you want:

```
__len__(self)                    # Length of the object
__getitem__(self, key)           # self[key]
__missing__(self, key)           # For dict when key not found
__setitem__(self, key, value)    # self[key] = xx
__delitem__(self, key)           # For mappings to remove a pair
```

EMUL. CONTAINERS

Informator
A SOFRAND COMPANY

Sequences sometimes implement these deprecated methods:

```
__setslice__(self, i, j)

__delslice__(self, i, j)

__getslice__(self, i ,j)
```

Nowadays a slice-object is sent to the **__setitem__**, **__delitem__**, **__getitem__** methods instead. That is also the methods you should implement for your sequences.

PYTHON

EXERCISE

Create a nodeConfigList-object.

Attributes:

- nodeList (a listObj-instance that hold nodeConfig-instances)
- maxNrOfNodes

Methods:

- append(*nodeConfigs) (to add nodeConfig-instances to your nodeList)
- __len__ len(nodeList-instance) should nr of nodeConfigs

The nodeConfigList-object should be an iterable and return each nodeConfig.

EXTRA EXERCISE

Make your nodeConfigList-object act like a sequence "container" object.

Implement the following magic methods:

__getitem__

__contains__

Add a checking functionality to your append-method so that you never store more nodeConfig-instances than the maxNrOfNodes allows.

SPECIAL METHODS

PYTHON

@CLASSMETHOD

Informator
A SOFRAND COMPANY

Built-in decorator that gives a method some new features.

@classmethods are often used as a factory for creating instances from a sequence.

Classmethods is automatically called with the class object as the first argument.

@CLASSMETHOD

```
class Book(object):
        def __init__(self, title, author):
                self.title = title
                self.author = author

        @classmethod
        def create_books(cls, myLibrary):
                for title,author in myLibrary:
                        yield cls(title,author)
```

@CLASSMETHOD

Informator
A SOFRAND COMPANY

```
myLibrary = (('BookA', 'Author A'),('BookB', 'Author B'),
('BookC', 'Author C'),)

for book in Book.create_books(myLibrary):
    print book
```

PYTHON

EXERCISE

Add a classmethod called ”bulkInsertNewInstancesToNodeList” to your nodeConfig-object.

This factory-method should create and add nodeConfig-instances to the nodeConfigList using a tuple with nodeConfig data.

The method should take a nodeConfig-tuple and a nodeConfigList-instance as arguments.

Example of nodeConfig-tuple:

```
nodeConfig = (
  ('213.133.66.7', '255.255.255.224','213.133.66.1'),
  ('213.133.66.8', '255.255.255.224','213.133.66.1'),
  ('213.133.66.9', '255.255.255.224','213.133.66.1')
  )
```

@STATICMETHOD

Informator
A SOFRAND COMPANY

Static methods are VERY similar to class methods.

No ”cls” argument is needed.

```
@staticmethod
def create_books(myLibrary):
        …….
```

SPECIAL METHODS

__new__(cls,) *Called to create a new instance of a class cls. Intended to allow subclasses of immutable types (int, str, tuple) to customize instance creation.*

__init__(self,) *Called after instance is created. (__new__) but before it is returned to caller.*

__del__(self) *Destructor. Called when instance is about to be destroyed.*

__str__(self) *Should return "informal" description of object. (A str-obj)*

__repr__(self) *Called by repr() and string conversions (reverse quotes).The "official" string representation of an object.Should look like a Python expression that could be used to recreate an object with the same value.*

ABSTRACT CLASSES

```
from abc import ABCMeta, abstractmethod

class ClassName():
    __metaclass__ = ABCMeta                         # descriptor

    def __init__(self, other_args):
        self.attribute1 = …
        self.attribute2 = …

    @abstractmethod                                 # decorator
    def myMethod(self):
        raise NotImplementedError
        # Forces subclasses to define a method named myMethod
```

@PROPERTY METHODS

Informator
A SOFRAND COMPANY

```
class Bird(object):
    def __init__(self, color):
        self.__color = color

    @property
    def paint(self):
        return self.__color


a = Bird('blue')

print a.paint
```

@PROPERTY SETTERS

```
class Bird(object):
    def __init__(self, color):
        self.__color = color

    @property
    def paint(self):
        return self.__color

    @paint.setter
    def paint(self, n):
       self.__color = n

a = Bird('blue')

a.paint = 'green'
```

STATIC VARIABLES

We could use the @property decorator on class attributes to get ”static variables”.

If you want immutable static variables remove the @birdCounter-setter method.

```
class Bird(object):
    __birdCounter = 0

    def __init__(self, color):
        self.__birdCounter += 1

    @property
    def birdCounter(self):
        return self.__birdCounter

    @birdCounter.setter
    def birdCounter(self, n):
        self.__birdCounter = n
```

RICH COMPARISON

Rich comparison methods

```
__lt__(self, other)          x < y          Eg:  x.__lt__(y)
__le__(self, other)
__eq__(self, other)
__ne__(self, other)
__gt__(self, other)
__ge__(self, other)

Used for comparisons if rich comparison methods not implemented
__cmp__(self, other)
```

PYTHON

EXERCISE

Add a @maxNodesInList-property for the attribute ”maxNrOfNodes”.

If you implements the @maxNodesInList.setter don’t forget to check that the value isn’t lower than the current number of nodeConfigs in your nodeList.

PYTHON

EXERCISE

Add some rich comparison methods so you could use the numberOfTestsPerformed attribute of your nodeConfig-objects in comparison situations.

Add a method called ”testDone” to the nodeConfig-object so you could increase the numberOfTestsPerformed-attribute.

Example:

```
if nodeConfig1 < nodeConfig2:
    print ”node1 has less tests performed than node2”
```

__SLOTS__

PYTHON

SIDE EFFECTS

Informator
A SOFRAND COMPANY

```
class spam(object):
    __slots__ = ['x']
    def __init__(self, n):
        self.x = n
```

First side effect - Slots gives a documentation of all arguments for a class.

Second side effect - It prevents dynamically created attributes

Third side effect - It could boost performance

Remember! This is just side effects !!

WHY USE SLOTS?

Using slots is a **memory optimization** tool.

You define exactly which attributes you want and removes the dynamic attribute dictionaries. (__dict__ & __weakref__)

This could save enormous amount of memory !!

WHY USE SLOTS?

Slots automatically creates **descriptors** for each attribute binded to __slots__.

Drawback:

You need to specify each attribute twice

Don't use __slots__ for attribute management.

It could "breaks" important features like static serialization. (pickle, json etc)

DESCRIPTORS

A descriptor is an object attribute ”binding” behavior.

They are used for implementing static methods, class methods and properties.

Only works for New-style classes.

Access to attributes has been overridden by methods in the

”object descriptor protocol”

E.g. The sequence of things that happens in the background under normal condition is changed to achieve something.

OUR DESCRIPTORS

```
class mySlotObj(object):
    __slots__ = ("a", "b")

mySpam = mySlotObj()

mySpam.a = "Testing"

print mySlotObj.a.__get__(mySpam, mySlotObj)


mySlotObj.a.__set__(mySpam, "World")
print mySlotObj.a.__get__(mySpam, mySlotObj)

print mySpam.a
```

IMPLEMENTING

__get__(self, instance, owner) *Called to get attribute of the owner class. (class attribute access) or attribute from an instance of that class.(instance attribute access).*

__set__(self , instance, value) *Called to set attribute on an instance.*

__del__(self, instance) *Called to delete the attribute on an instance.*

INSTANCE.X

How arguments are assembled depends on the class **A** and the instance **a**.

Let’s see how arguments works on different situations.

Instance binding

```
type(a).__dict__['x'].__get__(a,type(a))
```

Class binding

```
A.__dict__['x'].__get__(None, A)
```

DATA VS NON-DATA

@staticmethods and @classmethods are non-data descriptors.

Instances can redefine and override methods.

Different instances could have different behaviors.

DATA VS NON-DATA

A ”data descriptor” has __set__ and/or __delete__ methods implemented.

Normally __get__ and __set__ is defined for a data descriptors.

Non-data descriptors implements neither of __set__/__delete__.

Normally just the __get__ method.

Data descriptors always override a redefinition in an instance dictionary.

Non-data descriptors can be overridden by instances.