

Formation Python : Orienté objet et plus

par Bernard CHAMBON

Subatech, Nantes - France

23, 24 janvier 2018

- Orienté objet
 - Classe, donnée et méthode, variable et méthode de classe
 - Héritage, surcharge de méthode
 - Syntaxe par l'exemple
- Décorateurs
- Gestion des exceptions
 - Objectif et solution
 - Syntaxe par l'exemple
- Tests unitaires
 - Objectif et solution
 - Syntaxe par l'exemple

■ Classe

- `class`
mot clé qui débute une classe, fin de classe selon indentation !
- `__init__(self)`
constructeur, Il n'est pas possible de définir plusieurs constructeurs
- `self`
mot clé (et réservé) qui définit les fonctions comme méthodes et les variables comme données membres
`self.value` fait de `value` une donnée membre
`get_value(self)` fait de `get_value` une méthode (↔ toutes les méthodes prennent `self` en premier paramètre)
- Variable de classe
Variable qui existe en dehors de toute instanciation de la classe
(donc non préfixée par `self` mais par le nom de la classe)
- Méthode de classe
Fonction qui existe en dehors de toute instanciation de la classe.
Utilisée en préfixant par le nom de la classe
Caractérisée par l'annotation `@staticmethod` ou `@classmethod`
(`@classmethod` impose que la fonction ait un paramètre qui représente la classe)

■ Syntaxe par l'exemple

```
class Person:
    def __init__(self): # The constructor
        print("Calling constructor of class Person")
        self.name = "" # name will be an instance variable due to 'self' keyword
        self.email_address = ""

    language = "French" # language will be a class variable (no self.)

    @staticmethod # get_language will be a class method using @staticmethod annotation
    def get_language():
        return(Person.language)

    @classmethod # set_language will be a class method (using alternative @classmethod annotation)
    def set_language(cls, language):
        cls.language = language

    def get_name(self):
        return(self.name)

    def get_email_address(self):
        return(self.email_address)

    def set_name(self, name):
        self.name = name

    def set_email_address(self, email_address):
        self.email_address = email_address

    def display(self):
        print("Email for {} is {}".format(self.get_name(),
                                          self.get_email_address()))

# class ends here

# Usage of class Person
def main():
    person = Person()
    person.set_name("Pierre Dupond")
    person.set_email_address("pierre.dupond@xyz.fr")
    person.display()
    # here object person will be destroy automatically
    print("Native spoken language is {}".format(Person.get_language()))

# Exécution
Calling constructor of class Person
Email for Pierre Dupond is pierre.dupond@xyz.fr
Native spoken language is French
```

■ Être ou ne pas être 'pythonic' ?

- Les getters et setters ne sont pas justifiés, car les données membres sont forcément accessibles (pas de notion de contrôle d'accès)
- Le style de code 'pythonic' propose d'accéder directement aux données membres depuis l'objet au lieu d'écrire ceci

```
person = Person()
person.set_name("Pierre Dupond")
```

préférer écrire cela

```
person = Person()
person.name = "Pierre Dupond"
```

- Mon avis
Le style de code dit 'pythonic' ne préserve pas l'encapsulation.
Avec un historique en C++ et Java, je n'y suis pas complètement favorable
- @property
Python propose un mécanisme dit de 'properties' pour traiter ce point là - NON étudié

■ Héritage, surcharge de méthode

- Héritage

Définition d'une classe à partir d'une classe dite de base, pour étendre (et spécialiser) les fonctionnalités

Héritage multiple possible en python

- Surcharge de méthode, surcharge d'opérateur

Redéfinition de méthode | opérateur existant dans une classe de base

Possibilité d'invoquer la méthode | l'opérateur de la classe de base, en préfixant par le nom de la classe ou par `super()` en Python 3

■ Virtuelle

- Classe virtuelle

Classe qui définit mais n'implémente pas toutes les méthodes, donc non instanciable (sans implémentation de ce qui manque)

- Méthode virtuelle

Méthode seulement définie (sans implémentation).

Ceci n'est pas possible en Python, nécessité d'afficher un message ou de lancer l'exception

`NotImplementedError`

■ Autres

- Control d'accès

Pas de contrôle d'accès (cf. `public`, `private`, `protected` de Java)

(mais par convention une variable préfixée par un simple underscore `_` est privé ce n'est qu'une convention, à voir ce qu'il en est avec la directive `import`)

- Annotation

Méta-données pour ajouter des caractéristiques (ex. `@classmethod`)

- Classe et héritage : syntaxe par l'exemple avec le compte bancaire
 - Compte basique avec les opérations déposer et retirer un montant, connaître le solde
 - Compte courant, dérivé du compte basique
 - avec un découvert possible spécifique à chaque compte créé
 - avec re-définition de l'opération 'retirer un montant', tenant compte du découvert possible

Classe de base `BankAccount`

```
class BankAccount:
    def __init__(self, owner):
        # create account with a unique id, no amount
        self.id = uuid.uuid4()
        self.amount = 0
        self.owner = owner
        return # for readability

    def deposit(self, amount):
        self.amount += amount
        return

    def withdraw(self, amount):
        self.amount -= amount
        return

    def get_id(self):
        return(self.id)

    def get_amount(self):
        return(self.amount)

    def get_owner(self):
        return(self.owner)

# end of class BankAccount()
```

Classe dérivée `CurrentAccount`

```
class CurrentAccount(BankAccount):
    """ CurrentAccount extends BankAccount
        new withdraw implementation according to min_amount allowed
    """

    # this is the constructor
    def __init__(self, owner):
        # invoke constructor from super class
        BankAccount.__init__(self, owner)
        # a new field specific to CurrentAccount
        self.min_amount = 0
        return

    def set_min_amount(self, min_amount):
        self.min_amount = min_amount

    def get_min_amount(self):
        return(self.min_amount)

    def withdraw(self, amount):
        """ withdraw allowed only if min amount is not reached """
        # just to show : min_amount has a local scope
        min_amount = self.amount - amount
        if (min_amount < self.min_amount):
            print("withdraw operation of {} Euros is denied since"
                  " the low threshold of {} Euros "
                  "would be reached !"
                  .format(amount, self.get_min_amount()))
        else:
            # case ok, invoke super class's method
            BankAccount.withdraw(self, amount)
        return

# end of class CurrentAccount()
```


Code utilisant la classe `CurrentAccount`

```
def main():
    bank_account = CurrentAccount("Gaston Lagaffe")
    bank_account.set_min_amount(-20)
    print("Bank account for {} is created with a low threshold of {} Euros" \
          .format(bank_account.get_owner(), bank_account.get_min_amount()))

    bank_account.deposit(123)
    print("Current amount is {} Euros".format(bank_account.get_amount()))

    bank_account.withdraw(100)
    print("Current amount is {} Euros".format(bank_account.get_amount()))

    bank_account.withdraw(50)
    print("Current amount is {} Euros".format(bank_account.get_amount()))
```

Exécution

```
Bank account for Gaston Lagaffe is created with a low threshold of -20 Euros
Current amount is 123 Euros
Current amount is 23 Euros
withdraw operation of 50 Euros is denied since the low threshold of -20 Euros would be reached !
Current amount is 23 Euros
```

Exercice 3 - étapes 1 et 2
prévisionnel de 50 mn

■ Méthodes cachées, inspection d'un objet

- `__del__(self)` : Destructeur, invoqué automatiquement lors du recyclage de la mémoire
- `__str__(self)` : invoqué lors d'un `print(object)` (\Leftrightarrow `toString()` de Java)
- `__doc__(self)` : Pour les docstrings

■ Exemple pour fixer les idées :

```
import inspect
```

```
class Person:
    # next is a docstring, that will be displayed
    # via __doc__ magic method
    """Basic class for modelisation"""

    def __init__(self): # The constructor
        print("Calling constructor of class Person")
        self.name = "Nobody"

    # The destructor :
    # just to show, should never be re-defined !
    def __del__(self):
        print("Calling destructor of class Person")

    def __str__(self):
        s = "My name is {}".format(self.name)
        return(s)

    def get_name(self):
        return(self.name)
# class ends here
```

```
# Usage of class Person
def main():
    person = Person()
    print(person)
    # just to show __doc__ usage
    print("Class description : {}".format(person.__doc__))

    print("Playing with inspect")
    for x in inspect.getmembers(person):
        print(x)
```

```
# Résultat d'exécution
Calling constructor of class Person
My name is Nobody
Class description : Basic class for modelisation
Playing with inspect
('__del__', <bound method Person.__del__ of <__main__.Person instance>)
('__doc__', 'Basic class for a modelisation')
('__init__', <bound method Person.__init__ of <__main__.Person instance>)
('__module__', '__main__')
('__str__', <bound method Person.__str__ of <__main__.Person instance>)
('get_name', <bound method Person.get_name of <__main__.Person instance>)
('name', 'Nobody')
Calling destructor of class Person
```

■ Rappel à propos des fonctions

- Le nom d'une fonction `f()` peut être affecté à une variable `p` comme ceci `p = f` (attention pas `f()` car on ne veut pas l'invoquer)
- L'invocation de la fonction peut se faire à travers la variable `p` comme ceci `p()`

Code

```
def do_that_task():  
    print("Executing that task")  
  
p = do_that_task  
p()
```

Exécution

```
Executing that task
```

- Une fonction peut prendre en paramètre une autre fonction comme ceci `def f(f1):`

Code

```
def vi_editor():  
    print(" Basic tool, all is in my head !")  
  
def pycham_ide():  
    print(" Advanced tools, focusing on business code")  
  
def do_coding(tool):  
    print("do coding with ...")  
    tool()  
  
do_coding(vi_editor)  
do_coding(pycham_ide)
```

Exécution

```
do coding with ...  
Basic tool, all is in my head !  
do coding with ...  
Advanced tools, helping to focus on business code
```

■ Rappel à propos des fonctions (suite)

- Une fonction peut retourner une fonction (qui en plus pourra être créée à la volée)

Code

```
def get_inner_function():
    def inner_function():
        print("Executing an inner function")
        return

    return(inner_function)

# invoke get_inner_function, result will be accessible via p
p = get_inner_function()

# invoke p function
p()
```

Exécution

```
Executing an inner function
```

■ Décorateurs vue coté utilisateur

- dans une "entité" donnée
 - si je passe en paramètre une fonction
 - si je définie une fonction à la volée (et utilisant la fonction passée en paramètre)
 - si retourne cette fonction nouvellement crée
 - je peux ensuite invoquer ce qui est retournée

La fonction passée en paramètre pourra ainsi être modifiée (on dira décorée)
L'entité pré-citée est un décorateur

• Exemple

Code

```
def do_decorate_function(f):  
    def g():  
        print("Start wrapping function")  
        f()  
        print("End wrapping function")  
        return  
    return(g)  
  
d = do_decorate_function(do_that_task)  
d()
```

Exécution

```
Start wrapping function  
Executing that task  
End wrapping function
```

Attention : Je me suis placé coté utilisateur, qui a enrichi une fonction existante mais je peux vouloir enrichir une fonction, sans changer les appels fait par les utilisateurs ...

■ Décorateurs vue coté développeur

- Exécution de la fonction `do_compute()`

Code

```
def do_compute(p1, p2, p3):  
    print("Running a tricky software with "  
          "parameters {}, {}, {} "  
          .format(p1, p2, p3))  
    return
```

Utilisation et exécution

```
# invoke method  
do_compute(1, 17, 45)  
  
# getting results  
Running a tricky software with parameters 1, 17, 45
```

- Décoration de la fonction `do_compute()`
puis exécution

Code

```
def do_decorate(f, *args):  
    def g(*args):  
        print("Prepare env. to run new "  
              "version with parameters {}".format(args))  
        f(*args)  
        print("Restore environnement after run")  
        return  
    return(g)
```

```
@do_decorate  
def do_compute(p1, p2, p3):  
    print("Running a smarter version of software ")  
    return
```

Utilisation et exécution

```
# invoke method (the same way as above)  
do_compute(1, 17, 45)  
  
# getting results  
Prepare env. to run new version with parameters (1, 17, 45)  
Running a smarter version of software  
Restore environnement after run
```

Gestions des exceptions

■ Objectif

Disposer d'un moyen élégant et sûr de gestion des cas d'erreurs lors de l'exécution

Élégant : Eviter un trop grand nombres de `if ... else ...`

Sûr : Etre certain que l'erreur sera vue ('trappée') et qu'une action y sera associée

Pouvoir hiérarchiser les cas d'erreurs

■ Solution

Mécanisme de gestion des exceptions, via les mots clés `try, except, finally, raise`

■ Syntaxe par l'exemple

```
class LearningException:
    def __init__(self):
        return
    def read_file(self):
        try:
            afile = open("/abc/input.txt", "r");
            print("File exist and is readable")
            return("Good choice for filename ! ")
        except Exception as ex:
            print("Exception occurred, message {}".format(ex))
            return("Bad choice for filename ! ")
        finally:
            print("Finally bloc always executed (even in case of return)")

# class ends here

def main():
    learning_exception = LearningException()
    msg = learning_exception.read_file()
    print(msg)
```

Exception occurred, message [Errno 2] No such file or directory:
Finally bloc always executed (even in case of return)
Bad choice for filename !

■ Génération des exceptions

- `raise` pour lancer une exception

```
raise Exception("An error occurred ...") # Python 3
```

```
raise Exception, "An error occurred ..." # Python 2
```

- Aiguillage

Il est possible de catcher plusieurs exception (un `try`, plusieurs `except`)

La correspondance entre l'objet "lancé" (via `raise`) et "attrapé" (via `except`) se fait sur le type (`BaseException` dans l'exemple suivant)

```
def do_raise(self):
    try:
        afile = open("/tmp/input.txt", "r");
        print("File exists, but ... ")
        # raise BaseException, "raising an exception for test" # with Python 2
        raise BaseException("raising an exception for test") # with Python 2 and 3
    return
    except Exception as ex:
        print("Exception occurred, message: {}".format(ex))
    except BaseException as ex:
        print("BaseException occurred, message: {}".format(ex))
```

File exists, but ...
BaseException occurred, message: raising an exception for test

- Diagramme d'héritage des exceptions

Il existe par défaut un grand nombre d'exceptions qui obéissent à un diagramme d'héritage <https://docs.python.org/3.6/library/exceptions.html#exception-hierarchy>

Il est possible de définir ses propres exceptions (éventuellement en dérivant une classe d'exception existante)

- Objectif : Modélisation d'un compte bancaire (suite) :
 - Compte d'épargne, dérivé du compte basique
 - Avec des montants plancher et plafond et redéfinition des méthodes associées
 - Avec un taux de rendement et les nouvelles méthodes associées
 - Ce code utilisera les exceptions

Gestions des exceptions > Syntaxe par l'exemple

Classe dérivée SavingAccount

```
class SavingAccount(BankAccount):
    """ new implementation for withdraw and deposit according to MIN_AMOUNT, MAX_AMOUNT
        new implementation for get_amount according to INTEREST_RATE
        implementation using Exception """
    ...

    # these values are constant for all instance of SavingAccount
    # => defined as class variables and constant (CAPITAL_CASE)
    MIN_AMOUNT = 100 # capitals letters as convention
    MAX_AMOUNT = 123456

    def __init__(self, owner):
        BankAccount.__init__(self, owner)
        return

    # this value is the same for all instance of SavingAccount
    # => defined as class variable, not constant
    interest_rate = 0.02

    # providing a class method for getting|setting interest_rate
    @classmethod
    def set_interest_rate(cls, interest_rate):
        SavingAccount.interest_rate = interest_rate
        return

    @classmethod
    def get_interest_rate(cls):
        return(SavingAccount.interest_rate)

    ...

    def withdraw(self, amount):
        """ withdraw allowed only if min amount is not reached """
        min_amount = self.amount - amount
        if (min_amount < SavingAccount.MIN_AMOUNT):
            raise Exception("withdraw operation of {} Euros is denied"
                             " since the low threshold of {} Euros "
                             " would be reached !")
            .format(amount, SavingAccount.MIN_AMOUNT))
        BankAccount.withdraw(self, amount)
        return

    def deposit(self, amount):
        """ raise exception if max_amount deposit is reached """
        if ((self.amount + amount) > SavingAccount.MAX_AMOUNT):
            raise Exception("deposit operation of {} Euros is denied "
                             " since the high threshold of {} Euros "
                             " would be reached !")
            .format(amount, SavingAccount.MAX_AMOUNT))
        BankAccount.deposit(self, amount)
        return

    def get_amount_with_interests(self):
        """ amount should include interest_rate over the ad-hoc period """
        new_amount = self.compute_interests()
        return(new_amount)

    def compute_interests(self):
        raise NotImplementedError("Not implemented yet")
```

Code utilisant la classe `SavingAccount`

```
def main():
    rc=-1
    try:
        saving_account = SavingAccount("Gaston Lagaffe")

        saving_account.deposit(123)
        print("Current amount is {} Euros".format(saving_account.get_amount()))

        saving_account.withdraw(50)
        print("Current amount is {} Euros".format(saving_account.get_amount()))
        rc=0
    except Exception as ex:
        print("Exception occurred with the message '{}'.format(ex))
    finally:
        return(rc) # status code will always be returned

if __name__ == '__main__':
    rc = main()
    sys.exit(rc)
```

Exécution

Current amount is 123 Euros

Exception occurred with the message 'withdraw operation of 50 Euros is denied since the low threshold of 100 Euros would be reached !'

Process finished with exit code 255

■ Objectif

- Tester le bon fonctionnement du code (niveau méthode via tests unitaires, plus globalement via les tests d'intégrations)
- Définir ce qu'on veut (passer le test) puis écrire le code à tester (TDD : Test Driven Development ou développement piloté par les tests)
- Vérifier la non régression du logiciel, lorsqu'on ré-implémente une méthode ou une classe

■ Comment

- Via du code et des assertions générant un résultat pass/fail
- Des facilités pour positionner des séquences d'initialisation/nettoyage par méthode, par classe, par module
- Des facilités pour inhiber un test et avec des conditions
- Des facilités pour regrouper des tests en suite de tests

■ Outils (Framework)

- Nombreux : `unittest`, `doctest`, `pytest`, etc.
A exécuter par exemple via `python -m pytest module_a_tester`
- A propos de ...
`unittest`, `doctest` disponible par défaut avec python
`pytest` à installer (ex via `pip install pytest`)

■ Syntaxe par l'exemple

Objectif : test de la méthode `withdraw()` de `SavingAccount`, avec `unittest` puis avec `pytest`

■ Code de test unitaire

```
import unittest

def setUpModule():
    SavingAccountTest.MY_LOGGER.info("Before running test module")
def tearDownModule():
    SavingAccountTest.MY_LOGGER.info("After running test module")

class SavingAccountTest(unittest.TestCase):
    MY_FORMAT = "%(asctime)-32s %(levelname)-6s %(message)s"
    logging.basicConfig(format=MY_FORMAT, level=logging.INFO)
    MY_LOGGER = logging.getLogger()

    @classmethod
    def setUpClass(cls):
        SavingAccountTest.MY_LOGGER.info("Before running test class {}".format(cls.__name__))
    @classmethod
    def tearDownClass(cls):
        SavingAccountTest.MY_LOGGER.info("After running test class {}".format(cls.__name__))
    def setUp(self):
        SavingAccountTest.MY_LOGGER.info("Before running test function")
    def tearDown(self):
        SavingAccountTest.MY_LOGGER.info("After running test function")

    def test_withdraw(self) :
        SavingAccountTest.MY_LOGGER.info("Running test on withdraw method")
        sa = SavingAccount("Gaston Lagaffe")
        sa.deposit(123)

        SavingAccount.MIN_AMOUNT=100
        sa.withdraw(23)
        self.assertEqual(sa.get_amount(), 100)
        with self.assertRaises(Exception) as ex:
            sa.withdraw(2)
        self.assertIsNotNone(ex , msg="Exception expected !")
        ...

        # please notice how to invoke test class
        if __name__ == '__main__':
            unittest.main()
```

■ Exécution du code de test

```
via unittest (notice 'discover' command which looks for tests in test* module)
> python -m unittest discover -s /Users/bchambon/Documents/PycharmProjects/ObjectOriented/
ou
via pytest (yes, pytest allow running unittest code !)
> python -m pytest -s /Users/bchambon/Documents/PycharmProjects/ObjectOriented/test_with_unittest.py
ou
via PyCharm
```

■ Résultat du test

```
Testing started at 15:46 ...
Launching py.test with arguments /Users/bchambon/Documents/PycharmProjects/ObjectOriented/test_with_unittest.py in \
/Users/bchambon/Documents/PycharmProjects/ObjectOriented

===== test session starts =====
platform darwin -- Python 2.7.10, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
rootdir: /Users/bchambon/Documents/PycharmProjects/ObjectOriented, inifile:
collected 1 items

test_with_unittest.py

===== 1 passed in 0.02 seconds =====
2017-05-15 15:46:17,527      INFO  Before running test module
2017-05-15 15:46:17,527      INFO  Before running test class SavingAccountTest
2017-05-15 15:46:17,528      INFO  Before running test function
2017-05-15 15:46:17,528      INFO  Running test on withdraw method
2017-05-15 15:46:17,528      INFO  After running test function
2017-05-15 15:46:17,528      INFO  After running test class SavingAccountTest
2017-05-15 15:46:17,528      INFO  After running test module

Process finished with exit code 0
```

Test de code avec pytest

■ Code de test unitaire

```
import pytest

MY_FORMAT = "%(asctime)-32s %(levelname)-6s %(message)s"
logging.basicConfig(format=MY_FORMAT, level=logging.INFO )
my_logger=logging.getLogger()

def setup_module(m):
    my_logger.info("Before running test module {}".format(m.__name__))

def teardown_module(m):
    my_logger.info("After running test module {}".format(m.__name__))

@classmethod
def setup_class(cls):
    my_logger.info("Before running test class {}".format(cls.__name__))

@classmethod
def teardown_class(cls):
    my_logger.info("After running test class {}".format(cls.__name__))

def setup_function(f):
    my_logger.info("Before running test function {}".format(f.__name__))

def teardown_function(f):
    my_logger.info("After running test function {}".format(f.__name__))

def test_withdraw() :
    my_logger.info("Running test on withdraw method")
    sa = SavingAccount("Gaston Lagaffe")
    sa.deposit(123)

    SavingAccount.MIN_AMOUNT=100
    sa.withdraw(23)
    assert(sa.get_amount() == 100)

    with pytest.raises(Exception) as ex:
        sa.withdraw(2)
    assert(ex is not None)
```


■ Exécution du code de test

```
> python -m pytest -s /Users/bchambon/Documents/PycharmProjects/ObjectOriented/test_with_pytest.py
ou
> pytest -s /Users/bchambon/Documents/PycharmProjects/ObjectOriented/test_with_pytest.py
ou
via PyCharm
```

■ Résultat du test

```
Testing started at 15:49 ...
Launching py.test with arguments test_with_pytest.py in /Users/bchambon/Documents/PycharmProjects/ObjectOriented

===== test session starts =====
platform darwin -- Python 2.7.10, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
rootdir: /Users/bchambon/Documents/PycharmProjects/ObjectOriented, inifile:
collected 1 items

test_with_pytest.py
2017-05-15 15:49:13,022          INFO  Before running test module test_with_pytest
2017-05-15 15:49:13,022          INFO  Before running test function test_withdraw
2017-05-15 15:49:13,023          INFO  Running test on withdraw method
2017-05-15 15:49:13,024          INFO  After running test function test_withdraw
2017-05-15 15:49:13,024          INFO  After running test module test_with_pytest

===== 1 passed in 0.02 seconds =====

Process finished with exit code 0
```

Exercice 3 - étape 3
prévisionnel de 30 mn