Programmation orientée objets [C++ & **Python**]

Dr. Pape Abdoulaye BARRO

La POO

- Généralités
- Données et manipulations
- Structures conditionnelles et itératives
- Autres types (conteneurs standard)
- **Les fonctions**
- Classes et héritages
- Les modules
- Les entrées/sorties
- La gestion des exceptions

Les fonctions
Définitions

- Les fonctions permettent d'éviter des répétitions de code. Un programme subdivisé en plusieurs petites portions de code est beaucoup plus aisé à relire et à maintenir.
  - La définition d'une fonction se fait par le biais du mot-clé def suivi du nom de la fonction. Suivent des parenthèses qui contiennent les éventuels paramètres de la fonction puis le caractère : qui délimite le début d'une séquence de code.
  - **Exemple**:

```
def stl(prenom):
    print('Bonjour %s' % prenom)
```

Les fonctions
Définitions

- Lorsque des variables sont définies dans le code, elles sont placées par l'interpréteur soit dans le contexte local(locals()) ou dans le contexte global(globals()).
  - Une variable est insérée dans le contexte local si elle est définie dans un bloc (boucle, fonction, …).
  - Elle est insérée dans le contexte global si elle est définie en dehors de tout bloc. Ce qui veut dire qu'il est impossible d'affecter directement les variables du contexte global depuis un bloc.

- **Exemple**:

```
prenom = 'Massamba'
def slt(prenom):
    print(locals())
    print('Bonjour %s' % prenom)
```

```
>>> slt('Mademba')
{'prenom': ' Mademba '}
Bonjour Mademba
```

```
>>> print(globals())
{'__builtins__': <module '__builtin__' (built-in)>,
'__name__':'__main__',
'slt': <function slt at oxb7fedfoc>,
'__doc__': None, 'prenom': 'Massamba'}
```

Les fonctions
Directives

- Pour pouvoir contourner cette limitation il est nécessaire d'utiliser la directive global qui permet de spécifier que la variable est dans le contexte global.

- Exemple:

```
identite = 'Soundiata KeÏta'
def slt(prenom, nom):
    global identite
    identite = '%s %s' %(prenom, nom)
    print(locals())
    print(identite)
```

```
>>> slt('Samory', 'Touré')
{'prenom': 'Samory', 'nom': 'Touré'}
Samory Touré
>>> print(identite)
Samory Touré
```

Les fonctions
Directives

- Lorsqu'une fonction doit renvoyer un résultat explicite, la directive return est utilisée. À précisé qu'en Python, il n'y a pas de distinction entre les fonctions et les procédures, contrairement à certains langages fortement typés. Les procédures sont tout simplement des fonctions qui ne renvoient pas de résultat comme en C.
  - Exemple:

```
def carre(nombre):
    return nombre*nombre
```

```
>>> carre(2)
4
```

  - Il est possible de retourner plusieurs résultats en les séparant par des virgules. Dans ce cas, l'interpréteur renvoie ces éléments dans un tuple.
  - Exemple:

```
def trois_valeur():
    return 1, 2, 3
```

```
>>> trois_valeur()
(1, 2, 3)
```

Les fonctions

Paramètres d'une fonction

- Un paramètre est une variable définie dans la fonction qui recevra une valeur lors de chaque appel. Cette valeur pourra être de tout type, suivant ce qui est fourni en argument.
- En Python, Il existe trois types de paramètres :
  - les paramètres explicites sont définit par des noms et sont séparés par des virgules. Certains paramètres peuvent prendre des valeurs par défaut et devenir optionnel ;
    - Exemple:

```
def slt(prenom, nom='Diop'):
    print('Bonjour %s %s' % (prenom, nom))
```

```
>>> slt('Elhadji')
Bonjour Elhadji Diop
>>> slt('Elhadji', 'Kaly')
Bonjour Elhadji Kaly
```

  - Il est cependant nécessaire de regrouper tous les paramètres optionnels à la fin de la liste des paramètres.

Les fonctions

Paramètres d'une fonction

- Lorsqu'il y a plusieurs paramètres optionnels, le code appelant peut définir ou non la valeur de chacun sans avoir à respecter un ordre précis, en utilisant la notation nom=valeur pour ce paramètre. On parle alors de nommage des paramètres.
- Exemple:

```
def somme(a, b=3, c=4):
    return a + b + c
```

```
>>> somme(5)
12
>>> somme(5, 2, 8)
15
>>> somme(5, c=2)
10
>>> somme(c=5, a=2, b=6)
11
```

Les fonctions
Paramètres d'une fonction

- Les paramètres non explicites sont laissés à l'appréciation de l'utilisateur. Il peut mettre autant de valeurs nommées qu'il le souhaite sans qu'il soit nécessaire de les définir dans la liste des arguments.
- Ces paramètres sont fournis sous la forme nom=valeur à la fonction. L'interpréteur place ces valeurs dans un dictionnaire qu'il faut au préalable définir en fin de liste par son nom précédé de deux étoiles.
- Exemple:

```
def phrase(**mots):
    print ('phrase de %d mot(s)' % len(mots))
    print ('Liste des mots: %s' % ' '.join(mots.values()))
    print ('Nom des paramètres: %s' % ' '.join(mots.keys()))
```

```
>>> phrase(mot1='devoir', mot2='annulé')
phrase de 2 mot(s)
Liste des mots:  devoir annulé
Nom des paramètres: mot1 mot2
>>> phrase()
phrase de 0 mot(s)
Liste des mots:
Nom des paramètres:
```

Les fonctions
Paramètres d'une fonction

- Les paramètres arbitraires sont équivalents aux paramètres non explicites sauf qu'ils ne sont pas nommés. L'interpréteur les regroupe dans un tuple nommé qu'il passe à la fonction. Le nom du tuple est fourni préfixé cette fois-ci d'une seule étoile.
- Exemple:

```
def planning(phrase, *vals):
    print (phrase % vals)
```

```
>>> planning('L\'examen de Python est fixé pour le %d du mois de %s', 25, 'Juin')
L'examen de Python est fixé pour le 25 du mois de Juin
```

- Lorsque des paramètres arbitraires sont combinés avec des paramètres explicites ou non explicites, la déclaration du nom du tuple qui contiendra les valeurs se place toujours après les paramètres explicites et avant les paramètres non explicites.
  - def nom_fonction(a, b, c, ..., *arbitraires, **explicites):

Les fonctions
Applications

Application 1:

Nombres parfaits et nombres chanceux.

- On appelle nombre premier tout entier naturel supérieur à 1 qui possède exactement deux diviseurs, lui-même et l'unité.
- On appelle diviseur propre de n, un diviseur quelconque de n, n exclu.
- Un entier naturel est dit parfait s'il est égal à la somme de tous ses diviseurs propres.
- Un entier n tel que : (n+i+i2) est premier pour tout i dans [0,n−2] est dit chanceux.

Écrire un module (parfait_chanceux_m.py) définissant quatre fonctions : somDiv, estParfait, estPremier, estChanceux et un autotest.

- La fonction somDiv retourne la somme des diviseurs propres de son argument.
- Les trois autres fonctions vérifient la propriété donnée par leur définition et retourne un booléen. Si par exemple la fonction estPremier vérifie que son argument est premier, elle retourne True, sinon elle retourne False.

La partie de test doit comporter quatre appels permettant de tester somDiv(12), estParfait(6), estPremier(31) et estChanceux(11).

La POO

- Généralités
- Données et manipulations
- Structures conditionnelles et itératives
- Autres types (conteneurs standard)
- Les fonctions
- **Classes et héritages**
- Les modules
- Les entrées/sorties
- La gestion des exceptions

Classes et héritages

Définitions

- Une classe peut être vue comme un regroupement logique de fonctions et de variables permettant de définir un comportement et un état du programme. Rappelons que Python est Orienté Objet, ce qui veut dire que tous les éléments qu'il manipule sont considérés comme étant des objets.
- Une classe définit un modèle d'objet que l'on peut ensuite instancier autant de fois que nécessaire.
  - Une instance devient un objet indépendant qui contient les fonctions et les variables définies dans le modèle.
- Le mot réservé class sert à définir un modèle en associant un certain nombre de variables et de fonctions à un nom.
  - Exemple:

```
class Voiture:
    couleur = 'Rouge'
```

  - Les éléments (variables et fonctions membres) de la classe sont nommés attributs et on parle plus précisément de méthodes pour les fonctions et d'attributs de données pour les variables.
  - La classe Voiture pourra donc être utiliser pour instancier des objets en l'appelant comme une fonction:
    - v=Voiture()
  - On pourra donc accéder à couleur en faisant:
    - v.couleur

Classes et héritages

Paramètre self

- De la même manière que pour une fonction, l'interpréteur met à jour les variables locales et globales lors de l'exécution des méthodes. Le code exécuté a donc une visibilité locale aux éléments définis dans la méthode et globale aux éléments en dehors de l'instance.
- Pour atteindre les éléments définis dans l'espace de noms de l'instance de la classe, il est donc nécessaire d'avoir un lien qui permette de s'y référer. L'interpréteur répond à ce besoin en fournissant l'objet instancié en premier paramètre de toutes les méthodes de la classe.
- Par convention, et même si ce nom n'est pas un mot-clé du langage, ce premier paramètre prend toujours le nom self.
  - Exemple:

```
class C:
    x = 8
    y = x + 5
    def affiche(self):
        self.z = 20
        print(C.y)
        print(self.z)
```

```
>>> obj = C()
>>> obj.affiche()
13
20
```

  - Les méthodes définies dans les classes ont donc toujours un premier paramètre fourni de manière transparente par l'interpréteur, obj.affiche() étant remplacé au moment de l'exécution par obj.affiche(obj).
  - self représente l'objet sur lequel la méthode sera appliquée.

Classes et héritages

Héritage simple - Héritage multiple

- L'héritage est la faculté d'une classe B de s'approprier les fonctionnalités d'une classe A. On dit que B hérite de A ou encore que B dérive de A. Python permet de définir des classes dérivées très simplement :

```
class B(A):
    pass
```

- Au moment de l'instanciation de la classe dérivée, l'interpréteur mémorise le nom de la classe parente afin de l'utiliser lorsque des attributs de données ou des méthodes sont utilisés :
  - si l'attribut en question n'est pas trouvé dans la classe, l'interpréteur le recherche dans la classe parente.
  - Si l'attribut n'est pas trouvé dans la classe parente, l'interpréteur remonte l'arbre de dérivation à la recherche d'une méthode portant la même signature avant de provoquer une exception AttributeError.

```
class Voiture:
    type = 'voiture'
    def affiche(self):
        print(self.type)

class Mercedes(Voiture):
    pass
class MercedesTurbo(Mercedes):
    pass
```

```
>>> v=MercedesTurbo()
>>> v.affiche()
voiture
```

- Python supporte l'héritage multiple en laissant la possibilité de lister plusieurs classes parentes dans la définition.

```
class C(A, B):
    pass
```

Classes et héritages
Surcharge des attributs - Polymorphisme

- Toutes les méthodes et attributs de données peuvent être surchargés, en utilisant la même signature.

```
class Voiture:
    type = 'voiture'
    def affiche(self):
        print(self.type)
    def utilise(self):
        self.affiche()

class Mercedes(Voiture):
    def utilise(self):
        print('Mercedes est une %s' % self.type)
```

```
>>> v=Mercedes()
>>> v.affiche()
Voiture
>>> v.utilise()
Mercedes est une voiture
```

Classes et héritages
Surcharge des attributs - Polymorphisme

- Si une méthode doit spécifiquement utiliser un attribut que la règle de surcharge ne lui renvoie pas, il est possible de préciser à l'interpréteur de quelle classe il s'agit, en utilisant un préfixe de la forme:

  ClasseDeBase.methode(self, parametres).

Exemple:

```
class Mercedes(Voiture):
    def utilise(self):
        print('Mercedes est une %s' % self.type)
    def affiche(self):
        Voiture.affiche(self)
```

```
>>> v=Mercedes()
>>> v.affiche()
Voiture
```

Classes et héritages
Constructeur et destructeur

- Lorsqu'une classe est instanciée, la méthode spéciale __init__() est invoquée avec en premier paramètre l'objet nouvellement instancié par l'interpréteur. Ce fonctionnement permet de procéder à un certain nombre d'initialisations lorsque l'on crée une instance de classe.

```
class Voiture:
    def __init__(self):
        print("Nouvelle voiture n°%s" % id(self))
        self.immatriculation = 'TH %s' % id(self)
```

```
>>> v=Voiture()
Nouvelle voiture n°322569512
>>> v. immatriculation
'TH 322569512'
```

- Il est d'usage de déclarer les attributs de données directement dans le constructeur lorsque ceux-ci ne sont pas partagés par toutes les instances:
  - Ils sont attachés à l'objet au moment de leur initialisation comme c'est le cas dans notre exemple pour immatriculation.

Classes et héritages

Constructeur et destructeur

- Comme pour une méthode classique, le constructeur peut recevoir des paramètres supplémentaires, qui sont directement passés au moment de l'instanciation.

```
class Voiture:
    def __init__(self, type):
        self.type = type
```

```
>>> v=Voiture('Mercedes')
>>> v.type
'Mercedes'
```

- Un destructeur peut également être défini grâce à la méthode spéciale __del__() lorsque du code doit être appelé au moment de la destruction de l'instance. Le code contenu dans cette méthode doit explicitement appeler la méthode __del__() des classes parentes, si elles existent.

```
class Voiture:
    def __del__(self):
        print('destruction')
```

```
>>> v=Voiture()
>>> del v
destruction
```

Classes et héritages

Attributs privés

- En ce qui concerne la protection des attributs, il est possible de définir des attributs privés à la classe en préfixant le nom de deux espaces soulignés. Si l'attribut se termine aussi par des espaces soulignés, ils ne doivent pas être plus de deux pour qu'il reste considéré comme privé.
- L'interpréteur repère ces attributs et modifie leurs noms dans le contexte d'exécution. Pour un attribut __a de la classe Class, le nom devient _Class__a.
- Le mapping étend alors la recherche à cette notation lorsque les appels se font depuis le code de la classe, de manière à ce que les appelants extérieurs n'aient plus d'accès à l'attribut par son nom direct.

```
class Voiture:
    __defaults=['silencieuse']
    qualites=['rapide', 'economique']
    def  caracteristiques(self):
        print(self.__defaults)
        print(self.qualites)
```

```
>>> v=Voiture()
>>> v.caracteristiques()
['silencieuse']
['rapide', 'economique']
>>> v.qualites
['rapide', 'economique']
>>> v.__defaults
Traceback (most recent call last):
File "<stdin>", line 1, in ?
AttributeError: Voiture instance has no attribute '__defaut'
```

Classes et héritages
Méthodes spéciales

- Il est possible en Python de définir d'autres méthodes spéciales que __init__() et __del__(), qui déterminent un fonctionnement spécifique pour une classe lorsqu'elle est utilisée dans certaines opérations.
- Ces méthodes permettent de faire varier le comportement des objets et sont regroupées en fonction des cas d'utilisation:
  - représentation et comparaison de l'objet:
    - __str__() doit renvoyer une représentation sous forme de chaine de caractères d'un objet.
    - Exemple:

```
class Voiture:
    __defaults=['silencieuse']
    qualites=['rapide', 'economique']
    def caracteristiques(self):
        print(self.__defaults)
        print(self.qualites)
    def __str__(self):
        return 'je suis un objet de type Voiture'
```

```
>>> o = Voiture()
>>> str(o)
'je suis un objet de type Voiture'
```

Classes et héritages
Méthodes spéciales

- représentation et comparaison de l'objet ;
  - __repr__() similaire à la méthode __str__() sauf qu'elle renvoie une chaîne plus riche en informations qui peut être utilisée pour recréer l'objet (v1=eval(repr(o))).
  - Généralement, la chaîne __str__() est destinée aux utilisateurs et la chaîne __repr__() est destinée aux développeurs.
  - Pour implémenter cette méthode, il suffit de la surcharger à l'intérieur d'une classe :
  - Exemple:

```
class User:
    def __init(self, prenom, nom):
        self.prenom = prenom
        self.nom = nom
    def __repr__(self):
        return "Utilisateur(prenom='{}', nom='{}')".format(self.prenom, self.nom)
```

```
>>> u = User(prenom="Thomas", nom="SANKARA")
>>> print(repr(u))
Utilisateur(prenom='Thomas', nom='SANKARA')
```

Classes et héritages
Méthodes spéciales

- utilisation de l'objet comme fonction:
  - __call__() qui permet d'écrire des classes dont les instances se comportent comme des fonctions et peuvent être appelées comme une fonction.

  - Exemple:

```
class Somme:
    def __init__(self):
        print("Instance Créé")

    def __call__(self, a, b):
        print(a + b)
```

```
>>> som = somme()
Instance Créé
>>> som(3, 5)
8
```

Classes et héritages
Méthodes spéciales

- accès aux attributs de l'objet :
  - Lorsque l'interpréteur rencontre une écriture de type objet.attribut, il utilise le dictionnaire interne __dict__ pour rechercher cet attribut, et remonte dans les dictionnaires des classes dérivées si nécessaire.
  - L'utilisation des trois méthodes suivantes permet d'influer sur ce fonctionnement.
    - __setattr__() qui est utilisée lorsqu'une valeur est assignée, en lieu et place d'une modification classique de l'attribut __dict__ de l'objet.
      - objet.attribut = 'valeur' devient équivalent à objet.__setattr__('attribut','valeur')
      - Le code contenu dans __setattr__() ne doit pas appeler directement l'attribut à mettre à jour, au risque de s'appeler lui-même récursivement. Il faut utiliser un accès à __dict__.
      - Exemple:

```
class Personne:
    def __init__(self, prenom):
        self.prenom = prenom
-----------------------------------------------
class Majuscule:
    def __setattr__(self, nom, valeur):
        self.__dict__[nom] = valeur.upper()
```

```
>>> p = Personne('Kinta')
>>> setattr(p, 'nom', 'KINTE')
>>> print(p.__dict__)
{'prenom':'Kinta', 'nom':'KINTE'}
```

```
>>> m = Majuscule()
>>> m.nom = 'kinté'
>>> m.nom
'KINTE'
```

Classes et héritages
Méthodes spéciales

- __getattr__() est appelée en dernier recours lorsqu'un attribut est recherché dans un objet. Cette méthode ne surcharge pas le fonctionnement normal afin de permettre à __setattr__(), lorsqu'elle est surchargée, d'accéder aux attributs normalement.
  - Exemple:

```
class Personne:
    def __init__(self, prenom):
        self.prenom = prenom
    def __getattr__(self, nom):
        return 'pas de`{}` dans les attributs'.format(str(nom))
```

```
>>> p = Personne('Kinta')
>>> p.nom
'pas de`nom` dans les attributs'
>>> p.prenom
'Kinta'
```

- getattr(object, name) ou getattr(object, name, default) renvoie la valeur de l'attribut nommé de l'objet. name doit être une chaîne de caractères.
  - Si la chaîne est le nom d'un des attributs de l'objet, le résultat est la valeur de cet attribut. Par exemple, getattr(o,'nom') est équivalent à o.nom.
  - Si l'attribut nommé n'existe pas, la valeur par défaut est renvoyée si elle est fournie, sinon AttributeError est levée.

Classes et héritages
Méthodes spéciales

- __getattribute__() est appelé inconditionnellement pour mettre en œuvre les accès aux attributs pour les instances de la classe. Cette méthode doit renvoyer la valeur de l'attribut ou lever une exception AttributeError.
  - Si la classe définit également __getattr__(), cette dernière ne sera pas appelée à moins que __getattribute__() ne l'appelle explicitement ou ne lève une AttributeError.
  - Afin d'éviter une récursivité infinie dans cette méthode, son implémentation doit toujours appeler la méthode de la classe de base portant le même nom pour accéder aux attributs dont elle a besoin, par exemple, object.__getattribute__(self, name).
  - Exemple:

```
class Personne (object):
    def __getattribute__(self, nom):
        print(f"obtenir `{str(nom)}`")
        return  object.__getattribute__(self, nom)
```

```
>>> p = Personne()
>>> p.nom = 'Kounte'
>>> p.nom
obtenir `nom` Kounte
```

Classes et héritages
Méthodes spéciales

- __delattr__() est le complément des deux méthodes précédentes, objet.__delattr__('attribut') est équivalent à del objet.attribut.
  - Exemple:

```
class Personne:
    def __getattr__(self, name):
        print('getattr %s' % name)
        if name in self.__dict__:
            return self.__dict__[name]
        else:
            print("attribut '%s' inexistant" % name)
    def __setattr__(self, name, valeur):
        print('set %s: %s' % (name, str(valeur)))
        self.__dict__[name] = valeur
    def __delattr__(self, name):
        print('del %s' % name)
        if name in self.__dict__:
            del self.__dict__[name]
        else:
            print("attribut '%s' inexistant" % name)
```

```
>>> p= Personne()
>>> p.age = 20
set age: 20
>>> p.first_name
getattr first_name
attribut 'first_name' inexistant
>>> p.first_name = 'Kinta'
set first_name: Kinta
>>> del p.first_name
del first_name
>>> p.first_name
getattr first_name
attribut 'first_name' inexistant
```

Classes et héritages
Méthodes spéciales

- Utilisation de l'objet comme conteneur :
  - Les mappings et les séquences sont tous des objets de type conteneurs, qui implémentent un tronc commun de méthodes. Ces méthodes sont présentées ci-dessous et peuvent être définies dans toute classe.
    - __getitem__(key) est utilisée lorsqu'une évaluation de type objet[key] est effectuée.
    - Pour les objets de type séquences, key doit être un entier positif ou un objet de type slice. Les mappings, quant à eux, utilisent des clés de tout type non modifiable.
    - Exemple:

```
class Personne:
    def __init__(self):
        self._data = {}
    def __getitem__(self, key):
        if key in self._data:
            return self._data[key]
        else:
            print("pas de %s" % key)
```

```
>>> p = Personne()
>>> p['nom']
Pas de nom
>>> p['penom']= 'kinta'
>>> p['penom']
kinta
```

    - Si la clé fournie n'est pas d'un type compatible, une erreur TypeError est retournée.
    - Si la clé est en dehors des valeurs autorisées, une erreur de type IndexError est retournée.

Classes et héritages

Méthodes spéciales

- __setitem__(key, value) utilisée lorsqu'une assignation de type objet[key] = valeur est effectuée.
  - Les mêmes erreurs peuvent être utilisées que celles de __getitem__. Les mappings ajoutent automatiquement la clé lorsqu'elle n'existe pas, contrairement aux séquences qui retournent une erreur si la clé n'existe pas.
- __delitem__(key) permet de supprimer une entrée du conteneur.
- __len__() est appelée par la primitive len(), et permet de renvoyer le nombre d'éléments du conteneur.
- __iter__() est appelée par la primitive iter(), et doit renvoyer un iterator capable de parcourir les éléments.
- __contains__(item) renvoie vrai si item se trouve parmi les éléments.
- Exemple:

```
class Personne:
    ....
    def __setitem__(self, key, value):
        self._data[key] = value
    def __delitem__(self, key):
        print('objet supprime')
    def __len__(self):
        return len(self._data)
    def __contains__(self, item):
        return item in self._data.values()
```

```
>>> p = Personne()
>>> p['nom']
Pas de nom
>>> p['penom']= 'kinta'
>>> p['penom']
Kinta
>>> len(p)
1
>>> del p['penom']
```

Classes et héritages

Méthodes spéciales

- Utilisation de l'objet comme type numérique:
  - Ces méthodes peuvent être utilisées pour définir le fonctionnement de l'objet lorsqu'il est employé dans toute opération numérique, que ce soit une addition, un décalage de bits vers la gauche, ou encore une inversion.

Méthode	Opération
__add__(other)	objet + other
__sub__(other)	Objet - other
__mul__(other)	objet * other
__floordiv__(other)	objet // other
__mod__(other)	objet % other
__divmod__(other)	divmod(objet, other)
__pow__(other[, modulo])	objet ** other
__lshift__(other)	objet << other
__rshift__(other)	objet >> other
__and__(other)	objet & other
__xor__(other)	objet ^ other
__or__(other)	objet | other
__div__(other)	objet / other

__truediv__(other)	objet / other
__neg__()	- objet
__pos__()	+ objet
__abs__()	abs(objet)
__invert__()	~ objet
__complex__()	complex(objet)
__int__()	int(objet)
__long__()	long(objet)
__float__()	float(objet)
__oct__()	oct(objet)
__hex__()	hex(objet)
__coerce__(other)	coerce(objet, other)

- Pour toutes ces méthodes, un appel à objet opérateur other déclenche un appel à objet.methode(other).

Classes et héritages
Méthodes spéciales

- Exemple: Surcharge de l'addition

```
class Addition:
    def __init__(self, value):
        self.value = value
    def __add__(self, other):
        return Addition(self.value + other.value )
    def __iadd__(self, other):
        return self.__add__(other)
    def __str__(self):
        return str(self.value)
```

```
>>> a1= Addition(2)
>>> a2= Addition(3)
>>> a3= a1 + a2
>>> str(a3)
'5'
```

Classes et héritages

Applications

- Application2
  1. Créez une classe Python appelée Voiture qui a les attributs suivants:
     - Marque
     - Modèle
     - Vitesse
     - moteur (essence ou diesel)
  2. Ajoutez un constructeur à la classe Voiture qui initialise les attributs ci-dessus lorsque l'on crée un nouvel objet de la classe.
  3. Ajoutez une méthode à la classe Voiture appelée accélérer qui augmente la vitesse de 10 km/h à chaque fois que cette méthode est appelée.
  4. Ajoutez une méthode à la classe Voiture appelée afficher_vitesse qui affiche la vitesse actuelle de la voiture.
  5. Créez un objet de la classe Voiture et utilisez les méthodes accélérer et afficher_vitesse pour vérifier que tout fonctionne correctement.
- Application3
  1. Ecrire une classe Rectangle permettant de construire un rectangle dotée d'attributs longueur et largeur.
  2. Créer une méthode Perimetre() permettant de calculer le périmètre du rectangle et une méthode Surface() permettant de calculer la surface du rectangle
  3. Créer les getters et setters.
  4. Créer une classe fille Parallelepipede héritant de la classe Rectangle et dotée en plus d'un attribut hauteur et d'une autre méthode Volume() permettant de calculer le volume du Parallélépipède.
  5. Dans le programme principal, créer une instance de Rectangle et une instance de Parallelepipede.
     a) Afficher le périmètre du rectangle
     b) Afficher la surface du rectange
     c) Afficher le volume du parallélépipède

À suivre

Feedback sur:

pape.abdoulaye.barro@gmail.com