Programmation avec Python

Module 06 : La programmation orientée objets

OBJECTIFS

Savoir décrire un objet en Python

Savoir créer et manipuler des objets

Découvrir les méthodes spécifiques à Python

Comprendre l'héritage et le polymorphisme

SOMMAIRE

Créer une classe

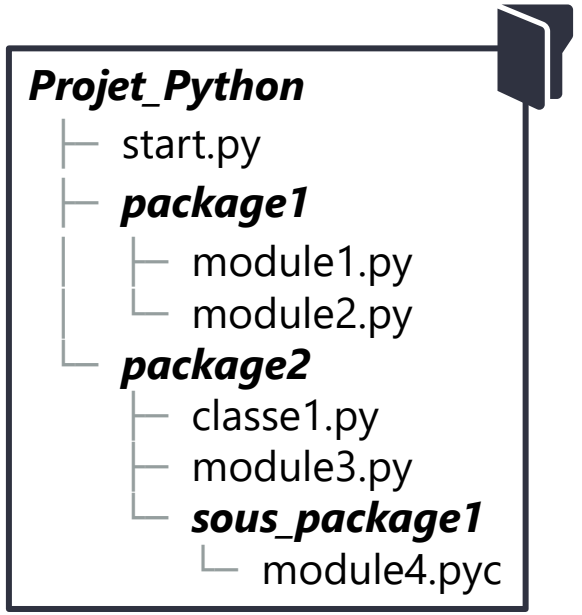
Créer des instances

Les méthodes spécifiques

Héritage

Polymorphisme

- Définition :
  - Un ***module*** contient des définitions et des instructions ;
    Le nom du fichier correspond au nom du module (suffixé de l'extension **.py** ou **.pyc**)
    (son propre nom implicite est accessible par la variable `__name__`)
  - Un ***package*** correspond à un dossier (ou répertoire), pouvant contenir des ***sous-packages*** (ou sous-dossiers) mais aussi des *modules*.
- En résumé :
  - Un ***module*** est un fichier (Python)
  - Un ***package*** est un dossier (ou répertoire)
  - Un ***sous-package*** est un sous-dossier (ou sous-répertoire)
  - Une ***librairie*** est un dossier contenant plusieurs *packages*
  - Un ***framework*** est un ensemble de *librairies* structurées cohérentes



- Définition :
  - Espace de nommage permet de séparer les appels des ***méthodes*** importées depuis les ***modules*** (ou fichiers).
  - Résultat d'une importation d'un ***module*** ou d'une ***librairie*** mais aussi d'une ***classe***, afin de structurer un programme dans l'appel des différentes ***méthodes***.
- Syntaxe :
  - `import un_fichier  # Import d'un fichier par son nom (sans l'extension)`
  - `import nom_module as alias  # Importation d'un module avec alias`
- Exemple :

```
import math as lib_math  # Import de la librairie 'math' avec alias 'lib_math'

print("Nombre Pi =", lib_math.pi)  # Affiche le nombre pi

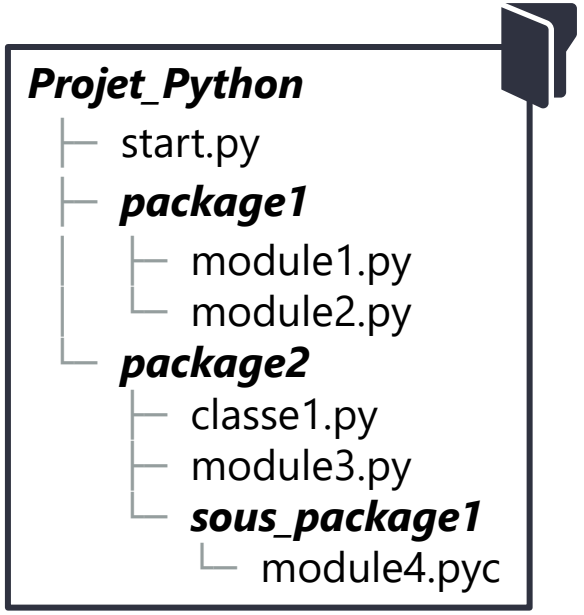
une_racine_carree = lib_math.sqrt(un_nombre := 16)  # Calcul de la racine carrée
print("La racine carrée de {} est le nombre {} !".format(un_nombre, une_racine_carree))
# Affiche le résultat du calcul (4)
```

- Optimisation :
  - Il est possible d'importer quelques ***méthodes*** spécifiques, directement accessibles depuis l'espace de noms principal.
  - Ainsi, il n'est plus nécessaire de préfixer par le nom de la ***librairie*** lors de l'appel. (fusion de l'espace de noms importé avec l'espace de nom principal)
- Syntaxe :
  - `from un_fichier import *  # Import d'un module complet (*)`
  - `from nom_module import une_methode, une_classe, …  # Import spécifique`
  - `from un_module import une_methode as m, …  # Importation unitaire avec alias`
- Exemple :

```
from math import fabs  # Importation que d'une seule méthode (par fusion d'espace de noms)

valeur_absolue = fabs(8 * -4)  # Appel à la méthode sans préfixer la librairie
print("Valeur absolue =", valeur_absolue)  # Affichage du résultat (32)
```

- Importation absolus/projet et relatif
- Exemple :
  - ```
    # références absolues / projet depuis start.py
    from package1 import module1
    from package1.module2 import methode1
    from package2 import classe1
    from package2.sous_package1.module4 import methode2
    ```
  - ```
    # référence relatives : package1/module1.py
    from .module2 import function1
    ```
  - ```
    # package2/module3.py (1e . = dossier courant)
    from . import classe1
    from .sous_package1.module4 import classe2
    ```
  - ```
    # package1/module2.py (.. = dossier parent)
    from ..package2.module3 import methode5
    ```



- Principe de base d'une classe objet :
  - Utilisation du mot clé **`class`**
  - Le constructeur est indiqué par la méthode **`__init__(self [, arg1, arg2, …])`**
  - **`self`** est une sorte d'équivalent à ***`this`*** en java : il représente l'instance
  - Les attributs d'instance sont indiqués par **`self.nom_attribut`**

- Exemple :

```
class Chat:
    def __init__(self, nom):
        self.nom = nom
```

- Pour utiliser une classe:
  - On créé une instance en utilisant le nom de la classe
  - C'est le *constructeur* **`__init__(self)`** qui est appelée
  - On ajoute les éventuels paramètres nécessaires dans **`__init__(self)`**

- Exemple :

```
class Chat:
    def __init__(self, nom):
        self.nom = nom


ella = Chat("Ella")
print(ella.nom)
```

- On peut avoir :
  - Des attributs spécifiques à chaque instance (indiqués par `self`)
  - Ou communs à toute la classe

- Exemple :

```
class Chat:
    espece = "Félin" # Attribut de classe
    def __init__(self, nom):
        self.nom = nom # Attribut d'instance


ella = Chat("Ella")
print(ella.nom) # Ella
print(ella.espece) # Félin
print(Chat.espece) # Félin
print(Chat.nom) #Erreur car c'est un attribut d'instance
```

- Il existe trois types de méthodes :
  - Méthode d'instance : avec self en premier paramètre
  - Méthode de classe : avec la classe en premier paramètre
  - Méthode statique : n'accède à aucun élément de la classe

```
class Chat:
    espece="Félin"

    def __init__(self, nom):
        self.nom = nom

    def afficher_nom(self):
        print(self.nom)

    @classmethod
    def afficher_espece(cls):
        print("Espece :", cls.espece)

    @staticmethod
    def hello():
        print('Hello World')
```

- Pour appeler les méthodes
  - Méthode d'instance : toujours sur une instance
  - Méthode de classe : sur la classe ou une instance
  - Méthode statique : sur la classe ou une instance

```
class Chat:
    espece="Félin"
    def __init__(self, nom):
        self.nom = nom
    def afficher_nom(self):
        print(self.nom)

    @classmethod
    def afficher_espece(cls):
        print("Espece :", cls.espece)

    @staticmethod
    def hello():
        print('Hello World')
```

```
ella=Chat("Ella")
ella.afficher_nom()
ella.afficher_espece()
Chat.afficher_espece()
ella.hello()
Chat.hello()
```

Ella
Espece : Félin
Espece : Félin
Hello World
Hello World

- Python ne peut pas assurer strictement l'encapsulation :
  - Rappel : « *Tout est modifiable dans Python* », soit tout est de visibilité ***publique***.
  - Mise en place d'une alternative conventionnelle de nommage responsable par le concepteur·trice : préfixer par _ devant les membres ***privés***
    - `_attribut_prive = "Attribut privé" # simple convention`
    - `def _methode_privee(self):  # méthode privée`
    - `__attribut_invisible = "Non-visible" # mais reste accessible`
  - Le développeur·euse a la responsabilité de **ne pas accéder** aux membres ***privés***, lorsque ceux-ci commencent par un ou deux caractères _, mais de respecter l'accès depuis les membres publics uniquement.
  - L'import (`import`) d'un module ne va pas importer les méthodes invisibles (préfixées par double souligné __).

```
class Chat:
    espece="Félin"

    def __init__(self, nom):
        self._nom = nom
```

- Conséquence de l'encapsulation :
  - On ajoute les getters pour autoriser à consulter la valeur d'un attribut
  - On ajoute les setters pour autoriser à modifier

- Exemple :

```
class Chat:
    def __init__(self, nom):
        self._nom = nom

    def get_nom(self):
        return self._nom

    def set_nom(self, nom):
        self._nom = nom
```

- Définition :
  - Mécanisme technique permettant l'utilisation d'une ***méthode*** comme d'un ***attribut***.
- Mise en place :
  - Création depuis le corps de la ***classe*** (pas dans le constructeur)
  - Renommage en visibilité privée des méthodes accesseurs (préfixer par _)
  - La ***propriété*** est liée à des méthodes (privées) en relation avec l'attribut :
    1. Méthode d'accès à l'attribut : ***accesseur*** ou *getter* (obligatoire)
    2. Modification de l'attribut : ***mutateur*** ou *setter* (facultatif)
    3. Suppression de l'attribut : « ***suppresseur*** » ou *deleter* (facultatif)
    4. Aide sur l'attribut : aide en ligne ou *docstring* (facultative)
- Syntaxe :

```
nom_propriete = property(methode_accesseur, methode_mutateur, methode_suppression,
                         """docstring : aide en ligne""")
```

```
class PythonEncapsulation:
    """Principe d'encapsulation v1.1 par convention de nommage responsable"""

    def __init__(self, valeur_initiale="Valeur à protéger"):
        set_mutateur(self, valeur_initiale) # setter/constructeur
        self._lecture_seule = "Lecture SEULE" # attribut

    def  get_accesseur(self):  # (get_attribut)
        return self._attribut.upper()  # Retourné en MAJUSCULE

    def  set_mutateur(self, value):  # (set_attribut)
        self._attribut = value.strip().lower()  # Définition de l'attribut

    def  get_lecture_seule(self):  # accesseur seul sans mutateur
        return self._lecture_seule.upper() # Retourné en MAJUSCULE
```

```
class PythonEncapsulation:
    """Principe d'encapsulation v2.0 avec l'utilisation de propriétés     """

    def __init__(self, valeur_initiale="Valeur à protéger"):
        self.propriete = valeur_initiale # Utilisation de la propriété
        self.lecture_seule = "Lecture SEULE" # Utilisation de la propriété

    def _get_accesseur(self):  # (get_attribut)
        return self._attribut.upper()  # Retourné en MAJUSCULE

    def _set_mutateur(self, value):  # (set_attribut)
        self._attribut = value.strip().lower()  # Définition de l'attribut

    def _get_lecture_seule(self):  # accesseur seul sans mutateur
        return self._lecture_seule.upper() # Retourné en MAJUSCULE

    # Mise en place des propriétés (property) liées aux accesseurs privés
    propriete = property(_get_accesseur, _set_mutateur)
    lecture_seule = property(_get_lecture_seule)
```

- Définition :
  - Amélioration syntaxique du mécanisme technique permettant l'utilisation d'une ***méthode*** comme d'un attribut à l'aide de ***décorateurs*** (annotation préfixée de **@**), directement intégré à la méthode de type ***accesseur/mutateur***.
- Syntaxe :
  - ```
    @property # getter ou accesseur implicite (lecture)
    def nom_propriete(self):
        """docstring""" # aide en ligne de la propriété
        return self._attribut_prive
    ```
  - ```
    @nom_propriete.setter  # setter ou mutateur (modification)
    def nom_propriete(self, value):
        self._attribut_prive = value
    ```
  - ```
    @nom_propriete.deleter # deleter (suppression)
    def nom_propriete(self):
        self._attribut_prive = None # suppression logique
    ```
    ```

```
class PythonEncapsulation:
    """Principe d'encapsulation v2.1 avec décorateurs de propriétés   """

    def __init__(self, valeur_initiale="Valeur à protéger"):
        self.propriete = valeur_initiale # Utilisation de la propriété

    @property # .getter implicite
    def propriete(self):  # Définition du nom de la propriété
        """Documentation de l'attribut protégé par la propriété"""
        return self._attribut.upper()  # Définition de l'attribut

    @propriete.setter # Rappel de la propriété avec mutateur (modification)
    def propriete(self, value):  # Rappel du nom de la propriété
        self._attribut = value.strip().lower()  # Rappel de l'attribut

    @propriete.deleter # Propriété pour contrôler la suppression d'attribut
    def propriete(self):  # correspondant à la décoration .deleter
        del self._attribut  # Suppression physique de l'attribut
```

- Parmi les fonctionnalités utiles sur une classe, on peut:
  - Ajouter une description avec un commentaire
  - Ajouter une méthode d'affichage de l'instance avec `__str__(self)`, appelée automatiquement
- Exemple :

```
class Chat:
    """Description d'un Chat"""
    def __init__(self, nom):
        self._nom = nom

    def __str__(self):
            return "Le chat " + self._nom


ella = Chat("Ella")
print(ella.__doc__)
print(ella)
print(ella.__str__())
```

```
Description d'un Chat
Le chat Ella
Le chat Ella
```

- Pour chaque objet, il existe des méthodes spéciales partagées par tous les objets de Python :

```
['__class__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

- ⚠ Il est possible de les utiliser, mais aussi de les modifier !

MÉTHODES SPÉCIALES DE COMPARAISON

Nom de la méthode	Opérateur représenté
`__eq__`	`==`
`__ge__`	`>=`
`__gt__`	`>`
`__le__`	`<=`
`__lt__`	`<`
`__ne__`	`!=`

AUTRES MÉTHODES SPÉCIALES

Nom de la méthode	Signification
`__class__`	Pour connaître la classe d'une instance
`__new__`	Créer une instance
`__init__`	Initialiser une instance
`__del__`	Supprimer une instance
`__repr__`	Représentation exacte d'une instance (dans la console)
`__str__`	Représentation informelle d'une instance

- On peut utiliser `__dict__` pour accéder à l'ensemble des attributs d'une instance sous forme de dictionnaire

- Exemple :

```
ella=Chat("Ella")
print(ella.__dict__)
```

```
{'nom': 'Ella'}
```

- Grâce à ce dictionnaire on peut utiliser :
  - `getattr(instance, 'attribut')` pour accéder à la valeur d'un attribut
  - `setattr(instance, 'attribut', 'valeur')` pour modifier la valeur d'un attribut
  - `delattr(instance, 'attribut', 'valeur')` pour supprimer un attribut
  - `hasattr(instance, 'attribut', 'valeur')` pour vérifier l'existence d'un attribut

- En Python, la construction d'une instance en deux étapes est claire :
  - `__new__` pour créer l'espace mémoire
  - `__init__` pour initialiser les attributs

- A l'aide des signatures génériques, on peut avoir une gestion automatisée des attributs :

```
class Chat:
    espece="Félin"

    def __init__(self, **kwargs):
        for k, v in kwargs.items():
            setattr(self, k, v)
```

➔ pratique… mais dangereux !

- Principe :
  - Lorsqu'un ***objet*** est détruit (manuellement ou automatiquement via le ***garbage collector***), la méthode d'instance (spéciale) **`__del__`** est appelée implicitement.
  - La redéfinition de cette ***méthode*** permet (en général) de mettre à jour les attributs de classe de l'objet en cours de suppression de la mémoire.
- Syntaxe :
  - `def __del__(self): # redéfinition du destructeur`
  - `del un_objet # Entraine la destruction (manuelle) de l'objet`

- Définition :
  - Méthodes portant le même nom dans une ***classe***,
    mais pas le même nombre de paramètres, ou un des paramètres de type différent.
- Rappel : Python remplace la dernière méthode traitée par l'interpréteur :
  - Le nom de ***méthode*** doit rester unique dans l'objet,
    et utiliser la possibilité de mettre en place de paramètres formels
    (obligatoires, optionnels, nommés, non nommés, obligatoirement nommés)
- Syntaxe :

```
def une_methode_unique(self, contexte1="Defaut1", contexte2="Defaut2"):
```

- Exemple :

```
def accelerer(self, duree=0, vitesse_cible=0): # version 1.0
    if duree > 0: # paramètre 'duree' priorisé / 'limite'
        for nb_secondes in range(duree + 1):
            self.vitesse += 1
    elif self.vitesse < vitesse_cible:
        self.vitesse = vitesse_cible
    return self.vitesse
```

DÉMONSTRATION

Créer et manipuler des objets

TRAVAUX PRATIQUES

Créer et manipuler des objets

- Python est un langage qui propose les principales fonctionnalités de l'héritage et du polymorphisme

- Syntaxe de base :

```
class Animal:
    # Code classe parent

class Chat(Animal):
    # Code classe enfant
```

APPELER LA CLASSE MÈRE

- Pour faire appel à une méthode de la classe mère, on peut :
  - Utiliser `Classe_Mère.méthode()` (appel statique)
  - Ou `super.méthode()`

```
class Animal:
    def __init__(self, nom):
        self.nom = nom


class Chat(Animal):
    def __init__(self, nom, taille):
        Animal.__init__(self, nom)
        self.taille = taille
```

```
class Animal:
    def __init__(self, nom):
        self.nom = nom


class Chat(Animal):
    def __init__(self, nom, taille):
        super.__init__(self, nom)
        self.taille = taille
```

- ⚠ Particularité Python : on peut faire de l'héritage multiple !
- Plus souple qu'avec des interfaces
- On peut hériter d'autant de classes que l'on veut

```
class Cheval:
    def __init__(self, vitesse_galop):
        self._vitesse_galop = vitesse_galop


class Humain:
    def __init__(self, metier):
        self._metier = metier


class Centaure(Cheval, Humain): # super == Cheval, mais dangereux !
    def __int__(self, vitesse, metier):
        Cheval.__init__(self, vitesse)
        Humain.__init__(self, metier)
```

```
class Cheval:
    def __init__(self, vitesse_galop):
        self._vitesse_galop = vitesse_galop

    def galoper(self)
        print("Je galope à", self._vitesse_galop, "km/h")

class Humain:
    def __init__(self, metier):
        self._metier = metier

    def travailler(self):
        print("J'exerce mon métier de", self._metier)

class Centaure(Cheval, Humain):
    def __int__(self, vitesse, metier):
        Cheval.__init__(self, vitesse)
        Humain.__init__(self, metier)

firenze = Centaure(30, 'Enseignant')
firenze.galoper()
firenze.travailler()
```

Je galope à 30 km/h
J'exerce mon métier de Enseignant

- Héritage multiple :
  - Et si plusieurs parents définissent la même méthode ?
    ➔ ⚠ C'est l'ordre d'héritage qui définit la priorité d'une même méthode !

```python
class Cheval:
    def __init__(self, vitesse_galop):
        self._vitesse_galop = vitesse_galop

    def dormir(self):
        print("Je dors comme un cheval")

class Humain:
    def __init__(self, metier):
        self._metier = metier

    def dormir(self):
        print("Je dors comme un humain")
```

```python
# Héritage multiple en fonction des priorités
class Centaure(Cheval, Humain):
    def __int__(self, vitesse, metier):
        Cheval.__init__(self, vitesse)
        Humain.__init__(self, metier)

eurytion = Centaure(30, 'Enseignant')
eurytion.dormir() # Je dors comme un cheval

# Annule et remplace la définition de classe précédente
class Centaure(Humain, Cheval):
    def __int__(self, vitesse, metier):
        Cheval.__init__(self, vitesse)
        Humain.__init__(self, metier)

hyleos = Centaure(30, 'Enseignant')
hyleos.dormir() # Je dors comme un humain
```

LES MÉTHODES DE VÉRIFICATION

- Principe :
  - La méthode **isinstance** vérifie qu'un objet a été créé à partir d'une classe spécifiée.
    (peut être sa classe de construction, mais aussi une classe parente suivant l'héritage)
  - La méthode **issubclass** vérifie qu'une classe est une sous-classe ou une classe dérivée.
    (soit avec un lien de parenté avec une classe spécifiée par rapport à un héritage mis en place.)
- Syntaxe :
  - `isinstance(un_objet, UneClasse) # un_objet a lien de parenté avec UneClasse ?`
  - `issubclass(UneClasse, AutreClasse) # UneClasse est une sous-classe de AutreClasse ?`
- Exemple :

```
isinstance(une_voiture, Voiture) # une_voiture est une instance de Voiture ? True
isinstance(un_bateau, Vehicule) # un_bateau est une instance de Vehicule ? True
isinstance(une_personne, Vehicule) # une_personne est une instance de Vehicule ?
False

issubclass(Voiture, Vehicule) # Voiture est une sous-classe de Vehicule ? True
issubclass(Moto, Vehicule) # Moto est une sous-classe de Vehicule ? True
issubclass(Personne, Agence) # Personne est une sous-classe de Agence ? False
```

LA CLASSE FINALE

- Définition :
  - L'objectif consiste d'empêcher l'héritage de classe (ou stopper les dérivations)
  - Obligatoirement instanciables, il n'est pas possible d'en hériter.
  - Impossible de définir des membres abstraits (puisque non héritable, donc non-surchargeable)
- Syntaxe :

```
from typing import final
@final # Décorateur de classe pour préciser la finalisation
class Class_Finalized:
```

- Exemple :

```
from typing import final

@final # Empêche l'héritage à partir de cette classe
class Class_Finalized:
    pass

class Class_Derivative(Class_Finalized):
    # Error: Cannot inherit from final class "Class_Finalized"
```

TRAVAUX PRATIQUES

Utiliser l'héritage

- Tentative d'instruction avec Python : *Forme minimale (2 blocs consécutifs)*

```
try:  # Bloc de tentative
    pass  # Tentative d'exécution d'instructions
except:  # Interception des erreurs
    """Exécution d'instructions en cas d'erreur entre try et except"""
```

- Exemple :

```
a_number = input("Saisir quelque chose : ")  # Saisie utilisateur
try:  # Ouverture du bloc de tentative
    a_number = int(a_number)  # Tentative de conversion d'une chaîne en nombre
    print("Le nombre saisi est :", a_number)
except:  # Interception en cas d'erreur
    print("Une erreur s'est produite durant la conversion numérique !")
```

- Principe :
  - Si aucune erreur ne survient à l'intérieur du bloc, alors on passe directement dans le bloc « autre » (**else**).
- Syntaxe : *Forme intermédiaire (3 blocs consécutifs)*

```
try:  # Bloc de tentative
    pass  # Tentative d'exécution d'instructions
except: # Interception des erreurs
    """Exécution d'instructions en cas d'erreur entre try et except"""
else: # Si aucune erreur ne survient dans le bloc
    """Lorsque tout s'est bien déroulé sans erreur."""
```

- Exemple :

```
a_number = input("Saisir un nombre : ")  # Saisie utilisateur
try: # Ouverture du bloc de tentative
    a_number = int(a_number) # Tentative de conversion
except: # Interception en cas d'erreur
    print("Une erreur s'est produite durant la conversion numérique !")
else: # Si aucune erreur n'a été rencontrée
    print("Le nombre saisi est :", a_number)
```

- Distinguer les types d'erreurs :
  - Les erreurs sont identifiées en fonction de leur nature (ou d'un type d'erreur)
  - Liste (non exhaustive) de types d'erreur :
    - ***NameError*** : une variable est non définie.
    - ***TypeError*** : le type d'une variable est incohérent empêchant un calcul.
    - ***ZeroDivisionError*** : une erreur de calcul lors d'une division par zéro inattendue.
    - ***ValueError*** : une erreur dans la tentative de conversion d'un type de donnée
    - ***KeyError*** : une erreur sur une clef de dictionnaire inconnue
    - ...
- Syntaxe :

```
try:  # Bloc de tentative
    pass  # Tentative d'exécution d'instructions
except UnTypeDErreur: # Interception de type spécifié
    """Traitement pour ce type d'exception"""
except: # Interception des autres erreurs
    """Traitement pour tous les autres types d'exception."""
```

```
numerateur = input("Saisir un numérateur : ")  # Saisie d'un nombre sans conversion
denominateur = int(input("Saisir un dénominateur : "))  # … avec conversion
try: # Ouverture du bloc de tentative
    resultat = numerateur / denominateur # Tentative de calcul
except NameError: # Interception uniquement les erreurs portant sur le nom d'une variable
    print("Une des variables n'a pas été définie !")
except TypeError: # Interception en cas d'erreur sur le type incohérent d'une variable
    print("Un type de variable n'est pas compatible avec l'opération mathématique !")
except ZeroDivisionError: # Interception en cas de division par zéro
    print("Tentative de division par zéro (impossible) ! ")
except: # Interception si aucune autre de type d'erreur ci-dessus n'a été interceptée
    print("Une erreur non gérée s'est produite ?")
else: # Si aucune erreur n’a été rencontrée
    print("Le résultat calculé : {} / {} = {}".format(numerateur, denominateur, resultat))
```

- Regrouper les types d'erreurs :
  - On peut traiter un ensemble de types d'erreur, afin de mutualiser la gestion des exceptions.
- Syntaxe :

```
try:  # Bloc de tentative
    pass  # Tentative d'exécution d'instructions
except (UnTypeErreur, UnTypeErreur2, …) as types_erreur: # Interception groupée
    """Traitement pour cet ensemble de types d'exception"""
except Exception as autre_erreur: # Interception des autres erreurs
    """Traitement pour toutes les autres exceptions."""
```

- Exemple :

```
try: # Ouverture du bloc de tentative
    resultat = numerateur / denominateur # Tentative de calcul
except (NameError, TypeError, ValueError) as erreur_variable:
    print("Il y a un problème avec une des variables !\n", erreur_variable)
except Exception as autre_erreur: # Interception en cas d'autres types d'erreur
    print("Impossible de réaliser le calcul (erreur inattendue) !\n", autre_erreur)
else: # Si aucune erreur n'a été rencontrée
    print("Le résultat calculé : {} / {} = {}".format(numerateur, denominateur, resultat)
```

- Exécuter même en cas d'erreur :
  - Quel que soit le résultat obtenu, les instructions du bloc **`finally`** seront exécutées.
- Syntaxe : *à placer après le dernier bloc `except` ou le bloc `else` par rapport au bloc `try` correspondant*
  - **`finally`**: *# quel que soit le résultat obtenu*
    `"""Qu'il y ait des erreurs ou non : execution des instructions…"""`
- Exemple :

```
try:
    resultat = numerateur / denominateur
except (NameError, TypeError) : # Regroupement d'exceptions
    print("Variables non définie, ou problème de type !")
except ZeroDivisionError : # Division par zéro
    print("Division impossible : le dénominateur est égal à zéro !")
except: # Gestion des autres erreurs (inattendues)
    print("Erreur non gérée !?")
else: # Lorsque tout s'est bien passé (sans rencontrer aucune erreur)
    print("Le résultat :", resultat)
finally: # Instruction(s) exécutée(s) systématiquement
    print("Fin du calcul")
```

- Personnaliser une exception :
  - Définition d'une erreur spécifique à une application,
    par héritage depuis une classe existante depuis la hiérarchie des exceptions.
- Syntaxe :
  - `class MonException(Exception): # Hérite d'une classe déjà présente`
    `"""Description de l'exception personnalisée"""`
- Exemple :

```
class AgeException(Exception):
    """Contrôle de l'âge limite d'une personne"""

try:
    age = int(input("Saisir un âge : ")) # Saisie avec conversion
    if not (18 <= age <= 65):
        raise AgeException("L'âge saisi n'est pas autorisé !")
except ValueError as ve:
    print("La valeur saisie est invalide :", ve)
except AgeException as ae:
    print("Problème avec l'âge :", ae)
else:
    print("La valeur saisie est correcte.")
```

- Principe de **raise** :
  - Permet de propager une exception ou de forcer une levée d'erreur, par la construction d'une exception.
  - Si l'instruction **raise** n'est pas contenu dans un bloc **try** … **except**, alors l'exception est propagée au-delà du contenant (méthode appelante, modules, …)
- Syntaxe :
  - `raise UneException()  # Propagation d'une erreur sans justification`
  - `raise UneException("motif de l'erreur") # Déclenchement d'une erreur avec motif`
- Exemple :

```
year = input("Saisir une année : ") # L'utilisateur saisit l'année (str)
try:
    year = int(year) # Tentative de conversion l'année en nombre
    if year <= 0: # Vérification dans le cas où la valeur est négative
        raise ValueError("l'année saisie est négative ou nulle")
except ValueError as e:
    print("La valeur saisie est invalide :", e)
else:
    print("La valeur saisie est correcte.")
```

- Principe :
  - Permet de vérifier qu'une condition est respectée (renvoi **True**), avant de poursuivre l'interprétation des instructions suivantes.
  - Si la condition n'est pas respectée (renvoi **False**), alors une exception (***AssertionError***) est propagée.
- Syntaxe : équivalent à l'instruction *if not condition_a_respecter: raise AssertionError()*
  - `assert condition_a_verifier # Doit être respectée pour continuer`
  - `assert condition_a_respecter, message_d_erreur # avec personnalisation`
- Exemple :

```
try:
    annee = int(input("Saisir une année : ")) # Saisie avec conversion
    assert annee > 0, "l'année saisie est négative ou nulle"
except ValueError as ve:
    print("La valeur saisie est invalide :", ve)
except AssertionError as ae:
    print("Condition non respectée :", ae)
else:
    print("La valeur saisie est correcte.")
```

TRAVAUX PRATIQUES

Les exceptions