TRAVAUX DIRIGÉS
GESTION DES ERREURS ET TESTS UNITAIRES

Ce TD va aborder les erreurs en python (les *exceptions*), leur génération et la façon de les gérer dans des codes qui utilisent vos méthodes. La seconde partie de cd TD va aborder les tests unitaires, tests permettant de valider chaque méthode que vous allez créer.

1. LA GESTION DES ERREURS

1.1. **introduction.** Tout programme rencontrant des éléments perturbants sa bonne exécution va produire une erreur qui entraine l'arrêt du programme. Ces erreurs peuvent être de nature très différentes en fonction du problème qui les génère, telle que des erreurs de Type, des erreurs d'entrée/sortie, ... Nous allons voir dans un premier temps comment permettre à notre programme de capturer ces erreurs et de les traiter afin de ne pas arrêter son exécution.

1.2. **Syntaxe de traitement.** La première syntaxe possible ressemble à celle du test simple. Le mot clé **try** permet de définir à partir de quelle ligne de code nous serons attentif à récupérer les exceptions produites. Le second mot clé **except** permet d'indiquer ce que notre programme doit faire lorsqu'une erreur est émise.

```
try :
   # code pouvant generer une erreur
except :
    # que doit on faire si une erreur quelconque a ete levee
```

Il est parfois utile de pouvoir distinguer quelle erreur a été générée et d'adapter la gestion en fonction de cette erreur. Dans cette optique, il est possible d'avoir plusieurs directives **except** suivie chacune de l'erreur qui sera capturée. Toutes les erreurs définies en Python sont des classes héritant de la classe **Exception**. Il existe une arborescence de classe d'exception

```
try :
    # code pouvant generer une erreur
except TypeError:
    # que doit on faire si erreur TypeError a ete levee
 except ValueError:
   # que doit on faire si erreur ValueError a ete levee
```

Pour être sur de capturer toute les erreurs, on peut ajouter une dernière directive avec **Exception**, classe de base de toutes les exceptions.

Il est bien sur possible d'ajouter un alias pour nommer l'erreur, à l'aide du mot clé **as** et ainsi la réutiliser dans le code de traitement.

La directive **else** permet de définir une partie de code qui ne s'exécute que si le code testé ne présente aucune erreur.

```
try :
    # code pouvant generer une erreur
except TypeError as terr:
    # que doit on faire si erreur TypeError a ete levee
 except ValueError as verr:
    # que doit on faire si erreur ValueError a ete levee
 except Exception as e:
    # traitement de toutes les autres exceptions
 else:
    # que fait on si aucune exception n'est generee
```

Enfin, nous pouvons vouloir exécuter un ensemble de code quelquesoit le statut d'exécution du code. C'est la directive **finally** qui va permettre de l'effectuer. Par exemple, dans la gestion d'un fichier ouvert en écriture, il est nécessaire de fermer le fichier que l'on ai rencontré une erreur d'entrée/sortie ou non.

```
try :
    # code pouvant generer une erreur
except TypeError as terr:
    # que doit on faire si erreur TypeError a ete levee
```

```
 except ValueError as verr:
    # que doit on faire si erreur ValueError a ete levee
 except Exception as e:
    # traitement de toutes les autres exceptions
 else:
    # que fait on si aucune exception n'est generee
finally:
    # code execute quelquesoit le resultat du try
# suite du fil d'execution
```

1.3. **Génération d'erreur.** Dans l'exécution de votre propre code vous pouvez avoir besoin d'informer un usager du mauvais usage de votre code. Dans ce cas nous pourrions fournir un message d'erreur indiquant l'erreur qui a été commise, mais encore faut il que l'utilisateur affiche ces messages, en tienne compte et soit en capacité de les traiter. Le fait de générer une exception permet de rendre obligatoire le traitement sous peine de stopper net l'exécution du code en erreur.

Pour cela, nous utilisons l'appel **raise** suivi de l'exception à générer. Cette exception peut être accompagnée d'un message d'erreur à destination de l'utilisateur. Par exemple, si la valeur affectée à la variable *alea* n'est pas de type entier, la suite de votre code ne pouvant pas s'exécuter correctement, il est préférable de générer une erreur indiquant que le type de valeur fournie n'est pas entier

```
if isinstance(alea,int):
    raise TypeError("la valeur fournie doit etre de type entier")
```

1.4. **exercice1.** Vous aller implémenter un petit programme permettant d'effectuer la division de 2 nombres réels passés en argument d'une fonction. Gérer toutes les exceptions qui peuvent être générée par ce code. Vous penserez aussi à traiter le cas ou l'un des deux arguments n'est pas de type réel.

1.5. **exercice2.** implementer des générations d'exceptions dans les constructeurs et les accesseurs des classes du projet Formes et gérer ces exceptions dans le code principal.

2. les Tests unitaires

En développement informatique, il est important de vérifier la validité de l'ensemble des fonctions, méthodes que vous concevez. Cela pourrais passer par l'utilisation dans le fil d'exécution de toutes les méthodes avec une comparaison entre le résultat obtenu et celui attendu. Mais cette méthode est fastidieuse et ne permet pas une automatisation du traitement de votre code. De plus, l'usage de fonctionnalité interne pourrais permettre de séparer la partie test de la partie développement et de confier ces taches à des personnes différentes.

2.1. **environnement pytest.** pour réaliser les tests de façon plus automatisée, vous allez installer le package **pytest** à l'aide de la commande *pip*. Nous allons ensuite travailler avec pytest sur le projet Formes. vous allez déplacer tous vos fichiers sources dans un répertoire src (ce qui doit normalement déjà avoir été fait pour isoler le code source de le documentation générée par sphinx). Ajouter dans ce répertoire ainsi qu'à la racine de votre projet un fichier vide nommé **__init__.py**. Celui ci définit que le répertoire dans lequel il se trouve est un package ou un sous package. Créez alors au même niveau que le répertoire **src** un répertoire **Tests** (avec un s, important car sinon, il peut rentrer en conflit avec un autre package test) A l'intérieur de ce répertoire, vous pourrez créer tous les fichiers de test qui seront nécessaire, normalement un par fichier de classe permettant de valider l'ensemble des méthodes de la classe.

2.2. **premier test.** Dans le fichier *test_point.py*, vous allez créer la méthode **test_distance_coordonnees**. Elle porte le même nom que la méthode précédé *test_* ou suivi de *_test*, façon pour l'exécutable pytest de savoir quels tests il doit exécuter.

```
def test_distance_coordonnee():
    p1 = Point(2,3)
    assert p1.distance_coordonnees(2,3) == 0
```

Un test python utilise le mot clé **assert** suivi de l'égalité à tester entre le retour d'une méthode et le résultat escompté, ici la méthode distance_coordonnees appliquée au point p1, et le jeu de coordonnée (2,3)

Pour exécuter les différents tests, vous allez lancer la commande **pytest** dans le terminal au niveau du répertoire de test. Cette commande va parcourir tous les modules présents et exécuter les fonctions

débutant et terminant par **test** et dressera le compte rendu d'exécution, précisant quels sont les tests qui ne sont pas réussis.

2.3. **Exercice.** réaliser l'ensemble des tests des classes du projet Formes.