

# Algorithmique Appliquée

BTS SIO SISR

## Tests, exceptions, assertions



CHAMBRE DE COMMERCE  
ET D'INDUSTRIE

1<sup>er</sup> ACCÉLÉRATEUR DES ENTREPRISES



Loïc Yvonnet



# Plan

- Gestion d'erreurs avec des codes de retour
- Notion d'exception
- Gestion d'exceptions et classes d'exception
- Programmation offensive et défensive
- Invariants
- Assertions
- Tests en boîte opaque
- Tests en boîte transparente
- Automatisation des tests
- Tests unitaires
- Tests pilotant le développement
- Pyramide de tests

# Gestion d'erreurs avec des codes de retour

# Sources d'erreurs (1/2)

- Première source d'erreur : **entrées utilisateur.**
  - Utilisateur **distract.**
  - Utilisateur **malveillant.**

# Sources d'erreurs (2/2)

- Autres sources d'erreur :
  - Limites de l'ordinateur :
    - `int`  $\neq \mathbb{Z}$ .
    - `float`  $\neq \mathbb{R}$ .
    - Mémoire de la machine limitée.
    - Espace disque limité.
  - Pannes réseau.
  - Bug dans une bibliothèque utilisée.

# Problématiques

- Comment gérer efficacement les erreurs sans impacter les performances ni la clarté du code ?
- Quand doit-on gérer les erreurs ?
- Comment documenter les erreurs et informer de manière claire, accessible et complète ?

## ! Mauvaise idée

```
def divide(a, b):  
    """Renvoie la division de a par b ou une erreur."""  
    if b == 0:  
        return "Erreur : division par 0"  
  
    return a / b
```

➡ Pourquoi est-ce une mauvaise idée ?

## ! Mauvaise idée

```
def divide(a, b):  
    """Renvoie la division de a par b."""  
    if b == 0:  
        return float("inf")  
  
    return a / b
```

➡ Pourquoi est-ce une mauvaise idée ?





# Mauvaise idée

```
def niveau_gris(rouge, vert, bleu):  
    """Renvoie un niveau de gris à partir d'une couleur RVB.  
  
    rouge, vert et bleu sont des entiers dans [0 ; 255].  
    """  
    if rouge < 0 or rouge > 255:  
        return None  
    if vert < 0 or vert > 255:  
        return None  
    if bleu < 0 or bleu > 255:  
        return None  
  
    return (rouge + vert + bleu) // 3
```



Pourquoi est-ce une mauvaise idée ?



# Solution *partielle*

```
def divise(a, b):  
    """Renvoie la division de a par b et un statut."""  
    if b == 0:  
        return 0, False  
  
    return a / b, True  
  
resultat, succes = divise(15, 0)  
if succes:  
    print(resultat)
```



# Solution *partielle*

```
def niveau_gris(rouge, vert, bleu):  
    """Renvoie un niveau de gris à partir d'une couleur RVB.  
  
    rouge, vert et bleu sont des entiers dans [0 ; 255].  
    """  
    if rouge < 0 or rouge > 255:  
        return 0, 1  
    if vert < 0 or vert > 255:  
        return 0, 2  
    if bleu < 0 or bleu > 255:  
        return 0, 3  
  
    return (rouge + vert + bleu) // 3, 0  
  
gris, statut = niveau_gris(255, 0, 0)  
if statut == 0:  
    print(f"ok : {gris}")  
else:  
    print(f"Erreur : l'argument n°{statut} n'est pas dans [0 ; 255]")
```

# Amélioration : quelques constantes

```
SUCCES = 0
ERREUR_ROUGE = 1
ERREUR_VERT = 2
ERREUR_BLEU = 3

def niveau_gris(rouge, vert, bleu):
    """Renvoie un niveau de gris à partir d'une couleur RVB.

    rouge, vert et bleu sont des entiers dans [0 ; 255].
    """
    if rouge < 0 or rouge > 255:
        return 0, ERREUR_ROUGE
    if vert < 0 or vert > 255:
        return 0, ERREUR_VERT
    if bleu < 0 or bleu > 255:
        return 0, ERREUR_BLEU

    return (rouge + vert + bleu) // 3, SUCCES
```

# Amélioration : dictionnaire de messages

```
SUCCES = 0
ERREUR_ROUGE = 1
ERREUR_VERT = 2
ERREUR_BLEU = 3

MESSAGES = {
    SUCCES : "OK",
    ERREUR_ROUGE : "Rouge à l'extérieur de l'intervalle [0 ; 255]",
    ERREUR_VERT : "Vert à l'extérieur de l'intervalle [0 ; 255]",
    ERREUR_BLEU : "Bleu à l'extérieur de l'intervalle [0 ; 255]"
}

def niveau_gris(rouge, vert, bleu):
    # [...]

gris, statut = niveau_gris(255, 0, 0)
if statut == 0:
    print(f"ok : {gris}")
else:
    print(f"Erreur : {MESSAGES[statut]}")
```



# Notion d'exception

# Introduction aux exceptions

```
liste = [0, 1, 2]  
liste[3]
```



```
IndexError: list index out of range
```

# Autre exemple

```
resultat = 1 / 0
```



```
ZeroDivisionError: division by zero
```



# Quelques exceptions classiques

- `IndexError`
- `NameError`
- `TypeError`
- `ValueError`
- `ZeroDivisionError`

# Bug ou erreur ?

- Une exception peut survenir à cause d'un bug dans le programme.
- Dans ce cas, il faut simplement corriger le code.
- Une exception peut survenir à cause d'une mauvaise entrée d'un utilisateur.
- Dans ce cas, le programme doit réagir de manière appropriée.

# Exception non gérée

- Jusqu'à présent, les exceptions ont été traitées comme événements terminaux.
- Lorsqu'une exception survient, le programme s'arrête.
- On parle dans ce cas d'**exception non gérée** (*unhandled exception* ).

# Gestion des exceptions

- Lorsqu'une exception est **levée** (*raised* ) , il est possible de la **gérer**.
- Cela signifie : exécuter du code spécifique au lieu de terminer le programme.
- On dit que l'on **attrape** (*catch* ) l'exception pour la traiter.

# Syntaxe

```
try:
    # Bloc1 (bloc de code n°1)
except Erreur:
    # Bloc2
except (AutreErreur, PasDeBol):
    # Bloc3
except:
    # Bloc4
else:
    # Bloc5
finally:
    # Bloc6
```



# Retour sur la division

```
def divise(a, b):  
    return a / b  
  
try:  
    resultat = divise(15, 0)  
except ZeroDivisionError:  
    print("Error: division par zéro")  
else:  
    print(f"Le résultat est {resultat}")
```

# Arrêt du flot de contrôle

```
def divise(a, b):  
    return a / b  
  
try:  
    resultat = divise(15, 0)  
    print(f"Le résultat est {resultat}")  
except ZeroDivisionError:  
    print("Error: division par zéro")
```

# Attraper les toutes 🐰

```
def divise(a, b):  
    return a / b  
  
try:  
    resultat = divise(15, 0)  
    print(f"Le résultat est {resultat}")  
except:  
    print("Error: division par zéro")
```



# Finalement (1/2)

```
def divide(a, b):  
    return a / b  
  
try:  
    resultat = divide(15, 0)  
except ZeroDivisionError:  
    print("Error: division par zéro")  
else:  
    print(f"Le résultat est {resultat}")  
finally:  
    print("On passe ici")
```



```
Error: division par zéro  
On passe ici
```

# Finalement (2/2)


```
def divide(a, b):  
    return a / b  
  
try:  
    resultat = divide(15, 1)  
except ZeroDivisionError:  
    print("Error: division par zéro")  
else:  
    print(f"Le résultat est {resultat}")  
finally:  
    print("On passe ici")
```



```
Le résultat est 15  
On passe ici
```

# Gestion d'exceptions et classes d'exception

# Levée d'exception

- Il est possible de **lever explicitement** des exceptions.
- Cela permet de stopper le flot de contrôle pour rentrer dans un mode de gestion d'erreur.
- La pile d'appels de fonction est déroulée (*unwind* ) jusqu'à trouver un **except** adapté.

# Retour sur le niveau de gris

```
def niveau_gris(rouge, vert, bleu):  
    """Renvoie un niveau de gris à partir d'une couleur RVB.  
  
    rouge, vert et bleu sont des entiers dans [0 ; 255].  
    """  
    if rouge < 0 or rouge > 255:  
        raise ValueError("Rouge en dehors de [0 ; 255]")  
    if vert < 0 or vert > 255:  
        raise ValueError("Vert en dehors de [0 ; 255]")  
    if bleu < 0 or bleu > 255:  
        raise ValueError("Bleu en dehors de [0 ; 255]")  
  
    return (rouge + vert + bleu) // 3  
  
try:  
    gris = niveau_gris(255, -1, 0)  
    print(gris)  
except ValueError as erreur:  
    print(erreur)
```

# Déroulement de pile d'appels - stack unwinding 🇬🇧 (1/2)

```
def f():  
    print("Entrée dans f")  
    raise ValueError("peu importe...")  
    print("Sortie de f")  
  
def g():  
    print("Entrée dans g")  
    f()  
    print("Sortie de g")  
  
def h():  
    print("Entrée dans h")  
    g()  
    print("Sortie de h")
```

## Déroulement de pile d'appels - stack unwinding 🇬🇧 (2/2)

```
try:  
    h()  
except ValueError:  
    print("Fin")
```



```
Entrée dans h  
Entrée dans g  
Entrée dans f  
Fin
```

# Retourner une valeur depuis un try

```
def lire_valeur(cast, message, erreur):  
    valeur = input(f"{message} : ")  
    try:  
        return cast(valeur)  
    except ValueError:  
        print(f"{valeur} : {erreur}")  
  
valeur = lire_valeur(int, "Entrer un entier", "n'est pas un entier")  
print(valeur)
```

↓ si on entre "chocolat"

```
chocolat : n'est pas un entier  
None
```



# Chaîner les exceptions

```
def divide(a, b):  
    try:  
        return a / b  
    except ZeroDivisionError as erreur:  
        raise ValueError("Dénominateur nul") from erreur  
  
divide(15, 0)
```



```
ZeroDivisionError: division by zero  
The above exception was the direct cause of the following exception:  
ValueError: Dénominateur nul
```

# Définir vos propres exceptions (1/3)

```
class RougeErreur(Exception):  
    pass  
  
class VertErreur(Exception):  
    pass  
  
class BleuErreur(Exception):  
    pass
```

# Définir vos propres exceptions (2/3)

```
def niveau_gris(rouge, vert, bleu):  
    """Renvoie un niveau de gris à partir d'une couleur RVB.  
  
    rouge, vert et bleu sont des entiers dans [0 ; 255].  
    """  
    if rouge < 0 or rouge > 255:  
        raise RougeErreur("Rouge en dehors de [0 ; 255]")  
    if vert < 0 or vert > 255:  
        raise VertErreur("Vert en dehors de [0 ; 255]")  
    if bleu < 0 or bleu > 255:  
        raise BleuErreur("Bleu en dehors de [0 ; 255]")  
  
    return (rouge + vert + bleu) // 3
```

# Définir vos propres exceptions (3/3)

```
try:
    gris = niveau_gris(255, -1, 0)
except RougeErreur as e:
    print(f"Ecarlate : {e}")
except VertErreur as e:
    print(f"Trop vert : {e}")
except BleuErreur as e:
    print(f"Schtroumpf : {e}")
```



Trop vert : Vert en dehors de [0 ; 255]



# Invariants

Préconditions et post-conditions

# Contrat d'une fonction

- Une fonction a un **contrat**.
- Ce contrat est un ensemble de :
  - **préconditions** : contraintes sur les valeurs d'entrée.
  - **invariants** : garantie sur les valeurs d'entrée.
  - **post-conditions** : contraintes sur les valeurs de sortie.

# Contrat de la fonction `racine_carree`

- **Préconditions :**

- La variable `x` est un nombre flottant positif ou nul.
- La variable `epsilon` est un nombre flottant strictement positif.

- **Invariants :** `x` et `epsilon` sont inchangés.

- **Post-conditions :** la valeur retournée est proche de la racine carrée de `x` , à plus ou moins `epsilon` .

# Attention à la sur-spécification

- On pourrait également préciser que `x` et `epsilon` doivent être différents de NAN (Not A Number) et de l'infinie.
- On pourrait également préciser que `x` et `epsilon` peuvent également être des entiers.
- Certaines choses sont implicites et n'ont pas besoin d'être spécifiées.
- C'est l'expérience qui dicte ce qui est explicite et implicite.
- Il vaut mieux commencer par être trop explicite et réduire progressivement.





# Programmation offensive et défensive

# Vérification des préconditions

- Il existe 2 approches :
  - Programmation **offensive** : les préconditions sont décrites en commentaires mais non vérifiées.
    - Avantages : performance et simplicité.
    - Inconvénients : robustesse.
  - Programmation **défensive** : les préconditions sont vérifiées et on renvoie une erreur si nécessaire.
    - Avantages : robustesse.
    - Inconvénients : lenteur et complexité.

# Division offensive

```
def divide(a, b):  
    """Divise a par b.  
  
    a - nombre flottant.  
    b - nombre flottant non nul.  
    Retourne la division a / b.  
    """  
    return a / b
```

# Division défensive

```
def divise(a, b, epsilon=0.000001):  
    """Divise a par b.  
  
    a - nombre flottant.  
    b - nombre flottant non nul.  
    epsilon - valeur autour de laquelle b est considérée nulle.  
    Retourne la division a / b.  
    """  
    if abs(b) < epsilon:  
        raise ValueError("b est trop proche de 0")  
  
    return a / b
```

# Division défensive extrême

```
def divide(a, b, epsilon=0.000001):  
    """Divise a par b.  
  
    a - nombre flottant.  
    b - nombre flottant non nul.  
    epsilon - valeur autour de laquelle b est considérée nulle.  
    Retourne la division a / b si a et b sont corrects.  
    """  
    if type(a) != float and type(a) != int:  
        raise TypeError("a n'est ni int, ni float")  
    if type(b) != float and type(b) != int:  
        raise TypeError("b n'est ni int, ni float")  
    if abs(b) < epsilon:  
        raise ValueError("b est trop proche de 0")  
  
    return a / b
```

# Approche pragmatique

- Souvent, en Python, on privilégie l'**approche offensive** avec une bonne documentation.
- Dans d'autres langages de programmation, ou certains contextes industriels, d'autres approches peuvent être favorisées.
- Il faut **se renseigner** sur les bonnes pratiques dans votre environnement, et suivre ces bonnes pratiques.



# Assertions

# Assert

- Une **assertion** permet de confirmer que l'état d'un calcul est celui attendu.
- On utilise pour cela le mot clé `assert` .
- Une expression Booléenne est attendue.
- Si cette expression vaut `True` , le programme continue son exécution.
- Dans le cas contraire, une exception `AssertionError` est levée.



# Exemple

```
assert 3 % 2 == 0  
print("3 est divisible par 2")
```



AssertionError

# Exemple

```
assert 3 % 2 == 0, "Si 3 était divisible par 2, on le saurait"  
print("3 est divisible par 2")
```



AssertionError: Si 3 était divisible par 2, on le saurait

# Exemple

```
assert 3 % 2 == 1  
print("Le reste de la division de 3 par 2 est 1")
```



```
Le reste de la division de 3 par 2 est 1
```

# Intérêts

- Les assertions peuvent être utilisées dans la **programmation défensive**.
- Elles peuvent également être utilisées dans le cadre de **tests unitaires**.

# TP : Exceptions dans une calculatrice

# TP : Exceptions dans une calculatrice

[Lien vers le sujet de TP.](#)



# Tests en boîte opaque

# Introduction

- **Albert Einstein** : "Aucune expérience ne peut jamais prouver que j'ai raison ; une seule expérience peut prouver que j'ai tort."
- **Edsger Dijkstra** : "Le test de programmes peut montrer la présence de bugs, mais ne peut jamais montrer leur absence."
- Les tests constituent un **filet de sécurité**.
- Les bugs peuvent malgré tout passer à travers les mailles du filet.



# Combinatoire

- Même le programme le plus simple a une **forte combinatoire**.
- Par exemple, si on doit écrire la fonction `min`, on a 2 entiers en entrée.
- On ne peut pas tester chaque combinaison de paires d'entiers.
- Cela représenterait  $2^{64} \times 2^{64} = 2^{128} \approx 3.4 \cdot 10^{38}$  opérations environ.

# Stratégie de test

- Au mieux, on peut **tester quelques combinaisons** qui ont de fortes chances de produire une réponse fausse s'il y a un bug dans le programme.
- Cette collection d'entrées à tester s'appelle une **suite de tests**.

# Partitionnement

- On va partitionner l'espace de valeurs en sous-ensembles qui doivent produire des résultats similaires.

	$a < 0$	$a == 0$	$a > 0$
$b < 0$	3 tests ( $a < b$ , $a == b$ , $a > b$ )	Test 6	Test 9
$b == 0$	Test 4 $a < 0$	Test 7 $a == 0$	Test 10 $a > 0$
$b > 0$	Test 5	Test 8	3 tests

# Contre-exemple

```
def min(a, b):  
    """Renvoie le minimum entre a et b.  
  
    a - entier.  
    b - entier.  
    Renvoie a s'il est plus petit que b et b sinon.  
    """  
    if b == 424242: # bug ou backdoor  
        return a  
    return a if a < b else b
```

# Familles de stratégies

- **Boîte opaque** : tests effectués par des personnes ne connaissant pas l'implémentation du programme.
- **Boîte transparente** : tests effectués par les implémenteurs du programme.

# Equipe Qualité

- De nombreuses entreprises ont une **équipe Qualité** séparée de l'équipe de développement du logiciel.
- Cette équipe est **indépendante** de l'équipe de développement.
- L'objectif de cette équipe est de **trouver un maximum de bugs** avant que le logiciel arrive en production chez des clients.

# Vérification par des tiers

- Il est même possible de faire appel à des **entreprises tierces**.
- Ces entreprises indépendantes vont faire un **audit**.
- C'est notamment le cas dans le domaine de la cybersécurité.

# Intérêt (1/2)

- Les développeurs peuvent **mal comprendre** les spécifications.
- Les développeurs peuvent **reproduire un bug** dans leurs tests.
- Dans ce cas, le test réalisé par le développeur innocent son code de manière injustifiée.



# Intérêt (2/2)

- **Biais psychologique :**
  - Un développeur a intérêt à dire que son programme fonctionne dans tous les cas.
  - Un testeur a intérêt à montrer qu'il trouve des bugs dans le code du développeur.
- **Concurrence bénéfique :** cette concurrence entre développeur et testeur crée une émulation et booste les projets.

# Processus

- Le testeur repart des **spécifications**.
- Le testeur établit les conditions d'usage classiques (fil rouge).
- Le testeur détermine les **conditions limites**.
- Le testeur crée un **plan de tests** qui comporte une suite de tests.
- Le testeur **exécute** régulièrement ce plan de tests.

# Tests en boîte transparente

# Bugs cachés dans le code

- Certains bugs sont cachés dans le code.
- Pour le trouver, il faut regarder le code.
- Avec la connaissance de ce code, on sait que l'on doit tester la valeur 424242 :

```
def min(a, b):  
    if b == 424242:  
        return a  
    return a if a < b else b
```

# Chemins d'exécution

- On cherche à emprunter chaque **chemin d'exécution** possible.
- On souhaite passer dans chaque branche de chaque condition.
- On souhaite rentrer dans chaque exception.
- On souhaite rentrer dans chaque boucle.

# Cas des boucles `for`

- Il faut tester les cas où :
  - on ne rentre pas dans la boucle.
  - le corps de la boucle est exécuté une fois.
  - le corps de la boucle est exécuté plus d'une fois.
- Il faut passer dans tous les `break` , `continue` , `return` , `yield` , etc.

# Cas des boucles `while`

- Il faut tester les mêmes cas qu'une boucle `for`.
- Par ailleurs, il faut exercer toutes les conditions de fin de boucle.
- Dans cet exemple, les 3 conditions de fin doivent être testées indépendemment.

```
while len(L) > 0 and (L[i] == "ok" or est_vrai):  
    # [...]
```

# Cas des fonctions récursives

- Il faut tester les cas où :
  - il n'y a pas d'appel récursif.
  - il y a exactement un appel récursif.
  - il y a plus qu'un appel récursif.



# Couverture de code

- La couverture de code est le pourcentage de lignes de code couvertes par les tests sur le nombre de lignes de code totales du programme.
- C'est un **indicateur** de la qualité logicielle.
- Une couverture supérieure à 80% est souhaitable.
- Une couverture à 100% est difficile et souvent trop coûteuse.

# Automatisation des tests

# Automatiser

- Le travail d'un informaticien est d'**automatiser** des tâches.
- Il est possible d'**automatiser les tests**.
- On écrit des programmes qui testent d'autres programmes.
- On les appelle des **programmes de tests**.

# Intérêt

- **Gagner du temps** en évitant de tester manuellement.
- **Eviter des régressions** pendant des phases de maintenance.
- Rejouer les tests dans **différents environnements** (ex : machine plus lente).
- Calculer des **indicateurs** automatiquement (ex : couverture de code).

# Fonctionnement

- L'environnement d'exécution se lance (via potentiellement de la virtualisation).
- Les programmes de test sont invoqués avec un jeu de données prédéfinies et/ou générées aléatoirement.
- Le résultat des invocations est sauvegardé.
- L'acceptabilité des résultats est vérifiée.
- Un rapport de test est généré.

# Pyramide de tests





# Tests unitaires

# Tests unitaires

- Les tests unitaires sont à la **base** de la pyramide de tests.
- Ils sont également à la base de la stratégie d'automatisation des tests.





# Concept

- 1 test unitaire teste **1 fonction** et 1 seule.
- 1 fonction est couverte par **plusieurs tests unitaires**.

# Effets de bord et composants externes

- Un test unitaire est **indépendant**.
- Un test unitaire est **pur** :
  - Il ne dépend pas de variable globale.
  - Il n'utilise pas le réseau.
  - Il n'utilise pas la base de données.
  - Il n'utilise pas de composant tier.

# AAA

- Un test unitaire suit les 3 étapes suivantes :
  - **Arrange** : initialise le test.
  - **Agit** (*act* 🇬🇧) : appelle la fonction à tester.
  - **Affirme** (*assert* 🇬🇧) : vérifie le résultat de la fonction.

## Exemple : tests unitaires pour la racine carrée (1/6)

```
def racine_carree(x, epsilon=0.000001):  
    """Renvoie la racine carrée de x à epsilon près."""  
    if x < 0:  
        raise ValueError("x est négatif")  
  
    s = x / 2  
    while abs(s ** 2 - x) >= epsilon:  
        P = s ** 2 - x  
        P_prime = 2 * s  
        s = s - P / P_prime  
  
    return s
```

## Exemple : tests unitaires pour la racine carrée (2/6)

```
def test_racine_carree_25():  
    # Arrange  
    x = 25  
    epsilon = 0.00001  
    attendu = 5  
  
    # Agit  
    resultat = racine_carree(x, epsilon)  
  
    # Affirme  
    assert abs(resultat - attendu) <= epsilon
```

## Exemple : tests unitaires pour la racine carrée (3/6)

```
def test_racine_carree_25_grand_epsilon():  
    # Arrange  
    x = 25  
    epsilon = 0.1  
    attendu = 5  
  
    # Agit  
    resultat = racine_carree(x, epsilon)  
  
    # Affirme  
    assert abs(resultat - attendu) <= epsilon
```

## Exemple : tests unitaires pour la racine carrée (4/6)

```
def test_racine_carree_0():  
    # Arrange  
    x = 0  
    epsilon = 0.00001  
    attendu = 0  
  
    # Agit  
    resultat = racine_carree(x, epsilon)  
  
    # Affirme  
    assert abs(resultat - attendu) <= epsilon
```

## Exemple : tests unitaires pour la racine carrée (5/6)

```
def test_racine_carree_1():  
    # Arrange  
    x = 1  
    epsilon = 0.00001  
    attendu = 1  
  
    # Agit  
    resultat = racine_carree(x, epsilon)  
  
    # Affirme  
    assert abs(resultat - attendu) <= epsilon
```



## Exemple : tests unitaires pour la racine carrée (6/6)

```
def test_racine_carree_negatif():  
    # Arrange  
    x = -1  
    exception_attrappee = False  
  
    # Agit  
    try:  
        racine_carree(x)  
    except:  
        exception_attrappee = True  
  
    # Affirme  
    assert exception_attrappee
```

# Notes pratiques

- Pensez bien à écrire des tests unitaires **lors de l'examen.**
- Si vous n'avez plus le temps, écrivez au moins sur votre copie que des tests unitaires devraient être ajoutés.

# Tests pilotant le développement

Test Driven Development 

# TDD

- Le développement piloté par les tests (ou TDD pour Test-Driven Development) est une **méthodologie**.
- Cette méthodologie vise à **garantir** que tout le code est **couvert par des tests**.

# Principe

- Ecrire la déclaration de la fonction.
- Ecrire un test unitaire.
- Exécuter le test unitaire et vérifier qu'il échoue.
- Ecrire le minimum de code pour que ce test réussisse.
- Ecrire un test unitaire.
- Etc.

# Exemple de TDD avec la racine carrée (1/10)

```
def racine_carree(x, epsilon=0.000001):  
    """Renvoie la racine carrée de x à epsilon près."""  
    pass
```

# Exemple de TDD avec la racine carrée (2/10)

```
def test_racine_carree_25():  
    # Arrange  
    x = 25  
    epsilon = 0.00001  
    attendu = 5  
  
    # Agit  
    resultat = racine_carree(x, epsilon)  
  
    # Affirme  
    assert abs(resultat - attendu) <= epsilon
```

# Exemple de TDD avec la racine carrée (3/10)

```
def racine_carree(x, epsilon=0.000001):  
    """Renvoie la racine carrée de x à epsilon près."""  
    return 5
```



# Exemple de TDD avec la racine carrée (4/10)

```
def test_racine_carree_25_grand_epsilon():  
    # Arrange  
    x = 25  
    epsilon = 0.1  
    attendu = 5  
  
    # Agit  
    resultat = racine_carree(x, epsilon)  
  
    # Affirme  
    assert abs(resultat - attendu) <= epsilon
```

# Exemple de TDD avec la racine carrée (5/10)

```
def test_racine_carree_0():  
    # Arrange  
    x = 0  
    epsilon = 0.00001  
    attendu = 0  
  
    # Agit  
    resultat = racine_carree(x, epsilon)  
  
    # Affirme  
    assert abs(resultat - attendu) <= epsilon
```

# Exemple de TDD avec la racine carrée (6/10)

```
def racine_carree(x, epsilon=0.000001):  
    """Renvoie la racine carrée de x à epsilon près."""  
    return 5 if x == 25 else 0
```

# Exemple de TDD avec la racine carrée (7/10)

```
def test_racine_carree_1():  
    # Arrange  
    x = 1  
    epsilon = 0.00001  
    attendu = 1  
  
    # Agit  
    resultat = racine_carree(x, epsilon)  
  
    # Affirme  
    assert abs(resultat - attendu) <= epsilon
```

# Exemple de TDD avec la racine carrée (8/10)

```
def racine_carree(x, epsilon=0.000001):  
    """Renvoie la racine carrée de x à epsilon près."""  
    if x == 25:  
        return 5  
    elif x == 0:  
        return 0  
    elif x == 1:  
        return 1
```

# Exemple de TDD avec la racine carrée (9/10)

```
def test_racine_carree_negatif():  
    # Arrange  
    x = -1  
    exception_attrappee = False  
  
    # Agit  
    try:  
        racine_carree(x)  
    except:  
        exception_attrappee = True  
  
    # Affirme  
    assert exception_attrappee
```

# Exemple de TDD avec la racine carrée (10/10)

```
def racine_carree(x, epsilon=0.000001):  
    """Renvoie la racine carrée de x à epsilon près."""  
    if x < 0:  
        raise ValueError("x est négatif")  
    elif x == 25:  
        return 5  
    elif x == 0:  
        return 0  
    elif x == 1:  
        return 1
```

# TP : Ecriture de tests unitaires





# **TP : Ecriture de tests unitaires**

[Lien vers le sujet de TP.](#)

# Devoir à la Maison 04



# **DM : Retour sur la complexité et les tests**

**Lien vers le sujet de DM.**