Fonctions, assertions et docstring en logo Python

1. Premiers pas avec les fonctions

1.1. Pourquoi des fonctions ?

Dans la partie précédente, nous avons terminé par un petit programme qu demande à un·e utilisateur·trice de donner un nombre entier entre 1 et 10, et qui vérifie la saisie jusqu'à ce que l'utilisateur·trice ait effectuer une saisie correcte. Voici un exemple simple d'implémentation de ce programme :

```
while True :
    n = input("Entrez un nombre entier ent
    if n.isdigit() : # regarde si la chair
        n = int(n)
        if n>=1 and n<= 10 :
            print("Le nombre est correct"
            break # L'instruction break s
        else :
            print("Votre nombre est entier
    else :
        print("Vous n'avez pas saisi un er
print(f"Vous avez saisi {n}")
```

Un blocage ?

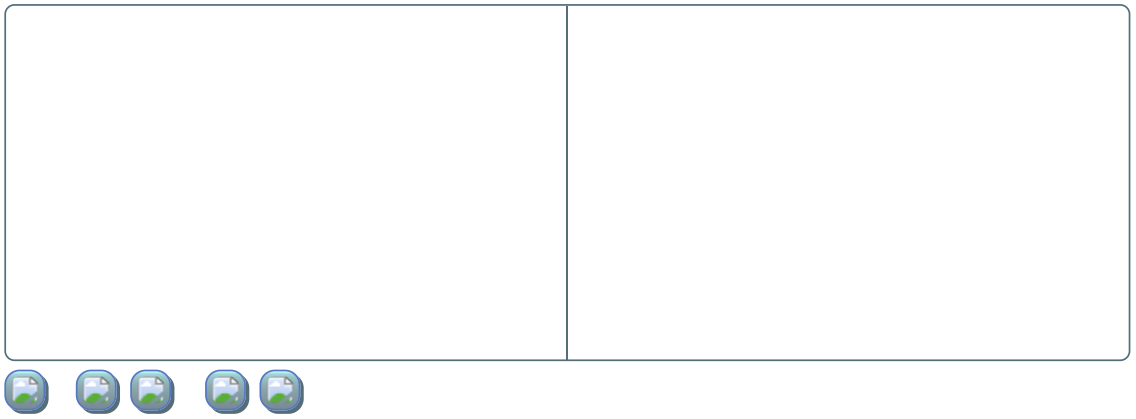
Enoncé

Essayez de saisir différentes valeurs **de différents types**. Quel est le problème rencontré ?

Solution

Si 'utilisateur·trice saisi un nombre négatif, même entier, celui-ci n'est pas reconnu comme un entier. Ce peut-être parfois complètement bloquant pour le fonctionnement réel d'une application.

La version ci-dessous est un peu plus *dumbproof*, mais je ne vous demande pas encore de comprendre les blocs `try/except/else`.



Application 1 : Multiplier les codes

Enoncé

Imaginons maintenant un programme qui demande **trois fois** à l'utilisateur d'entrer un tel nombre, puis qui vérifie si les trois nombres correspondent à une configuration de Pythagore.

Vous devez vous inspirer du programme précédent pour compléter le programme ci-dessous afin d'obtenir trois nombres entiers `nb1`, `nb2` et `nb3` saisis par l'utilisateur :

Solution

Un code pouvant être inséré (avec notre niveau actuel en Python et sans utiliser de structures de listes) est le suivant :

```
while True :
    nb1 = input("Entrez un nombre entier entre 1 et 10 : ")
    try :
        nb1 = int(nb1)
    except ValueError :
        print("Vous n'avez pas saisi un entier. Veuillez recommencer !")
    else :
        if 1<= nb1= 10 :
            break
        else :
            print("Votre nombre n'est pas compris entre 1 et 10. Veuillez recommencer !")
while True :
    nb2 = input("Entrez un nombre entier entre 1 et 10 : ")
    try :
        nb2 = int(nb2)
    except ValueError :
        print("Vous n'avez pas saisi un entier. Veuillez recommencer !")
    else :
        if 1<= nb2 <= 10 :
            break
        else :
            print("Votre nombre n'est pas compris entre 1 et 10. Veuillez recommencer !")
while True :
    nb3 = input("Entrez un nombre entier entre 1 et 10 : ")
    try :
        nb3 = int(nb3)
    except ValueError :
        print("Vous n'avez pas saisi un entier. Veuillez recommencer !")
    else :
        if 1<= nb3 <= 10 :
            break
        else :
            print("Votre nombre n'est pas compris entre 1 et 10. Veuillez recommencer !")
```

	###

Mauvaise pratique

Si ce code fait le travail demandé, il n'en est pas moins de *mauvaise qualité* pour plusieurs raisons :

- Il est long, avec des répétitions. Plus un code est long, plus il prend de place en mémoire, alors qu'on cherche plutôt à minimiser ce facteur.
- Le copié-collé force quand même à repasser sur chacune des parties pour modifier le nom des variables (sinon les valeurs seraient écrasées). Ainsi toute modification doit être **réécrite à chaque fois** que le morceau de code est appelé. Dans notre exemple, ce n'est que trois fois, le risque d'erreurs est minime, mais pour un plus grand nombre de répétitions, il est très facile d'oublier une modification.
- Le code n'est plus lisible. Dans le cas d'un travail en équipe, il est impératif d'avoir un code clairement écrit, avec des fonctionnalités clairement définies.

1.2. Première fonction

Définition d'une fonction

On va donc améliorer non seulement la lisibilité de notre code, mais aussi son efficacité et sa simplicité, en utilisant une **fonction**.

Définition : fonction

Une **fonction** est un bloc de code nommé (c'est-à-dire possédant un nom dans l'espace de noms, comme toute autre variable). L'appel par l'interpréteur du nom de la fonction **suivi d'une paire de parenthèses** exécutera alors l'intégralité du code et renverra une **valeur de retour**, c'est-à-dire un objet qui pourra être utilisé normalement dans la partie de code ayant appelé la fonction.

Pour notre exemple :

```
###
```

En regardant la première ligne :

```
def ask_user_int() -> int:
```

- La fonction est introduite par le mot clé `def`, suivi du **nom de la fonction** puis d'un couple de parenthèses `()`, ce qui rend l'objet **callable** ("appelable").
- la notation `-> int` est un **type hint**, autreùment dit un *indice de type*, qui indique que la valeur renvoyée par la fonction sera de type `int`. Les **type hints** sont *facultatifs en Python*, mais strictement nécessaires dans d'autres langages (`Java`, `C`, `C++`, etc).
- Les deux points définissent un bloc de code qui est repéré par **une indentation**, tout comme on définit des blocs dans des structures conditionnelles ou des boucles.

On fait appel à cette fonction en appelant le nom `ask_user_int()`, ce qui déclenche le bloc de code, puis crée un objet de retour correspondant à la valeur saisie par l'utilisateur.

Oubli des parenthèses

Dans le cas d'un oubli des parenthèses lorsqu'on appelle une fonction, on obtient dans le terminal le message suivant :

```
>>> ask_user_int
<function ask_user_int at 0xe8c1a0>
```

Qui signifie simplement que le nom `ask_user_int` fait référence à une fonction dont l'adresse mémoire est donnée sous forme hexadécimale.

Notez que `ask_user_int` est juste un nom, l'objet correspondant est stocké dans l'espace des objets. On peut donc écrire les choses suivantes :

```
>>> demande_entier=ask_user_int
>>> demande_entier()
```

Utilisation de la valeur de retour

Comme tout objet, la valeur de retour d'une fonction doit elle-même être stockée dans une variable afin de ne pas être ramassée par le *garbage collector*.

Exemple

```
>>> entier = ask_user_int()
>>> print(f'La racine carrée du nombre {entier} est {entier**(1/2)}')
```

L'interpréteur Python évalue la ligne `entier = ask_user_int()` :

- Il commence par appeler le code correspondant à La fonction `ask_user_int()`.
- Le code de la fonction demande une saisie à l'utilisateur, et une fois l'instruction `return` atteinte, l'instruction `ask_user_int()` est remplacée à l'endroit de l'appel par la valeur saisie par l'utilisateur, converti en objet de type int.
- L'objet est ensuite stocké dans une variable nommée `entier`, et peut alors être utilisé en dehors de la fonction.

Factorisation du code de Pythagore

Le code du programme de vérification de Pythagore peut alors être **factorisé**, et devient alors :

	###

Ce qui a l'avantage d'être vraiment vraiment plus clair.

1.3. Exercices

Application 2

Enoncé

Créer une fonction nommée `table7` qui **renvoie** la table de multiplication de 7 avec un multiplicateur allant de 0 à 10, sous la forme d'une chaîne de caractères comme ci-dessous :

```
"7x0=0 \n 7x1=7 \n 7x2=14 ..."
```

Indication : le symbole `\n`, insère un saut de ligne dans une chaîne de caractères.

Solution

```
def table7() -> str:
    table = ''
    for i in range(11) :
        table += f'7x{i} = {7*i} \n'
    return table
```

1.4. Augmenter la capacité des fonctions : les arguments obligatoires

L'exemple de la fonction `ask_user_int` est assez limité. Dans l'absolu, on pourrait souhaiter que la fonction demande un nombre entier entre 2 valeurs variables, par exemple entre 1 et 100 ou bien entre -10 et 10.

Pour ce faire, il faut, dans la définition de la fonction, préciser des **arguments** qui seront des objets transmis et nommés **initialisés à certaines valeurs lors de l'appel à la fonction** :

Exemple :

	###

La fonction `ask_user_int` utilise maintenant deux arguments `borne_min` et `borne_max`, qui sont deux entiers, et renverra toujours un entier

Une fois la fonction définie, on peut l'appeler en précisant les valeurs des deux arguments :

```
>>> ask_user_int(1,100)
>>> ask_user_int(-10,10)
```

Erreurs d'appels et notion de précondition

Avec le code actuel de la fonction `ask_user_int`, on peut rencontrer un certain nombre de difficultés. Testez les commandes suivantes dans la console

Test 1

Code

```
ask_user_int()
```

Problème

L'erreur signalée est de type `TypeError`, et le message d'erreur nous dit qu'il manque 2 arguments lors de l'appel, et que ces arguments sont `borne_min` et `borne_max`.

Test 2

Code

```
ask_user_int(45)
```

Problème

L'erreur signalée est de type `TypeError`, et le message d'erreur nous dit qu'il manque 1 argument lors de l'appel, et que cet argument est `borne_max`. Cela signifie aussi que le nom `borne_min` a bien reçu la valeur `45`.

Test 3

Code

```
ask_user_int(0.5,2.5)
```

Problème

Aucun véritable erreur n'est levée par Python, bien que les *type hints* demandent des arguments de type `int`. Comme dit précédemment, ces indications sont facultatives (à destination des programmeur·euse·s), et ici les transgresser ne pose pas vraiment de problème.

Test 4

Code

```
ask_user_int(30,10)
```

Problème

Ici, la fonction tourne en boucle et il est impossible de saisir une réponse qui amènera à l'instruction `return`. En effet, lors de l'appel à la fonction, on a :

- `borne_inf` initialisé à `30` ;
- `borne_sup` initialisé à `10` ;
- pour atteindre l'instruction `return`, il faudrait que l'utilisateur·trice ait saisi·e un nombre `nb` à la fois supérieur à `30` et inférieur à `10`, ce qui est impossible.

On essayera donc le plus possible de créer des tests dans les fonctions permettant de tester des **préconditions**, c'est-à-dire des tests vérifiant les propriétés nécessaires concernant les arguments, par exemple ici que les arguments sont bien de type entier, et que `borne_inf` est bien inférieur ou égal à `borne_sup`. Ces tests seront vu dans la partie concernant les **assertions**.

Application 3

Enoncé

Créer une fonction `table_multi(m : int)-> str` qui prend un argument entier `m` et écrit la table de multiplication de ce nombre, avec un multiplicateur allant de 1 à 10.

Solution

```
def table_multi(m : int ) -> str :
    table = ''
    for i in range(11) :
        table += f'{nb}}x{i} = {nb*i} \n'
    return table
```

Application 4 : motif dans une chaine

Enoncé

Créer une fonction `trouve_chaine(motif : str, texte : str) -> bool` qui prend deux arguments, un `motif` (une chaine de caractères) et un `texte` (une autre chaine de caractères) et qui renvoie `True` si le `motif` est présent dans la `chaine`, quel que soit la casse du motif ou celle de la chaine, et `False` sinon. Vous pouvez tester avec les lignes suivantes :

```
assert trouve_chaine('Toto', 'Toto va à la plage')==True, 'Meme casse pas trouvée'
assert trouve_chaine('Totos', 'Toto va à la plage')==False, 'Chaine non présente trouvée'
assert trouve_chaine('TOTO', 'Toto va à la plage')==True, 'Problème de majuscules dans le
motif'
assert trouve_chaine('toto', 'TOTO va à la plage')==True, 'Problème de minuscules dans le
motif'
assert trouve_chaine('ToTo', 'OtOtO va à la plage')==True, 'Que dire ?'
```

Solution

```
def trouve_chaine(motif : str, texte: str) -> bool:
    return motif.lower() in texte.lower()
```

1.5. Augmenter la capacité des fonctions : les arguments optionnels

Notre fonction `ask_user_int` commence à être intéressante. Mais nous pourrions souhaiter personnaliser le message de la question, sans pour autant avoir envie de le changer systématiquement. C'est tout à fait possible en Python, grâce aux **arguments optionnels**. Il s'agit d'arguments dont le nom est donné dans la fonction, mais avec **une valeur par défaut**. Ainsi :

###

Ainsi, la fonction ci-dessus possède trois arguments :

- deux arguments **obligatoires**, `borne_min` et `borne_max`, de type `int` ;
- un argument **optionnel**, `prenom`, de type `str`, dont la **valeur par défaut est la chaine `Inconnu`.

Il est à noter qu'impérativement les **arguments obligatoires doivent être placés avant les arguments optionnels**.

On peut alors appeler la fonction des différentes manières suivantes (à tester) :

```
ask_user_int(0, 10)
ask_user_int(0, 10, prenom='Toto')
ask_user_int(0, prenom='foo', 10)
```

```
>>>
```

Application 5 : arguments optionnels

Enoncé

Compléter la fonction `table_multi` afin qu'elle utilise deux arguments optionnels, la valeur de départ, fixée à 0 initialement, et la valeur d'arrivée du multiplicateur, fixée à 10 initialement.

Solution

```
def table_multi(nb : int, depart : int = 0, fin : int =10) -> str :
    table = ''
    for i in range(depart, fin+1) :
        table += f'{nb}x{i} = {nb*i} \n'
    return table
```

Application 6 : arguments optionnels

Enoncé

Réécrire la fonction `trouve_chaine` afin qu'elle utilise un *argument booléen optionnel* `verifCasse`, afin de déterminer si le `motif` est présent dans le `texte` en vérifiant la casse ou non. Par défaut l'argument sera `False`. Vous pouvez utiliser les tests ci-dessous :

```
### Cette cellule est une cellule vous permettant de tester votre fonction
##les assertions suivantes sont les même que précédemment
assert trouve_chaine('Toto', 'Toto va à la plage')==True, 'Meme casse pas trouvée'
assert trouve_chaine('Totos', 'Toto va à la plage')==False, 'Chaine non présente trouvée'
assert trouve_chaine('TOTO', 'Toto va à la plage')==True, 'Problème de majuscules dans le
motif'
assert trouve_chaine('toto', 'TOTO va à la plage')==True, 'Problème de minuscules dans le
motif'
assert trouve_chaine('ToTo', 'OtOtO va à la plage')==True, 'Que dire ?'
# Mais on rajoute celles-ci :
assert trouve_chaine('Toto', 'Toto va à la plage',verifCasse = True )==True, 'Meme casse
pas trouvée'
assert trouve_chaine('TOTO', 'TOTO va à la plage',verifCasse = True )==True, 'Meme casse
pas trouvée'
assert trouve_chaine('Totos', 'Toto va à la plage',verifCasse = True)==False, 'Chaine non
présente trouvée'
assert trouve_chaine('TOTO', 'Toto va à la plage',verifCasse = True)==False, 'Problème de
majuscules dans le motif'
assert trouve_chaine('toto', 'TOTO va à la plage',verifCasse = True)==False, 'Problème de
minuscules dans le motif'
assert trouve_chaine('ToTo', 'OtOtO va à la plage',verifCasse = True)==False, 'Que dire ?'
```

Solution

```
def trouve_chaine(motif : str,texte : str, verifCasse : bool=False) -> bool :
    if verifCasse==True :
        return motif in texte
    else :
        return  motif.lower() in texte.lower() :
```

1.6. Commenter son code : les docstrings

Commenter son code

Une bonne habitude, à prendre immédiatement, est celle de commenter son code, c'est-à-dire d'expliquer l'implémentation de votre code sous la forme de commentaires, écrits en français (ou mieux, en anglais). Les commentaires sont **des lignes non-lues par l'interpréteur Python**, commençant par le symbole *dièse* `#`.

Ces explications sont importantes, car vous vous rendrez vite compte que vous serez parfois incapable de comprendre un code que vous avez écrit vous-même quelques semaines voir quelques jours auparavant !

Alors imaginez celui qui doit lire votre code, mais qui ne l'a pas écrit...

Les concepteur·trice·s de Python ont créé une fonction spécifique permettant d'obtenir des informations sur les autres objets : la fonction `help`.

Testez par exemple la commande `help(print)` dans la console suivante, puis essayez avec d'autres objets de Python.

```
>>>
```

La fonction `help` va chercher dans l'objet passé en argument sa **docstring**, littéralement *chaine de documentation*, qui est une chaine de caractères crée par le ou la codeur·euse présentant l'utilisation de la fonction, ses paramètres obligatoires, ses paramètres optionnels, etc... Une **docstring** est construite comme une chaine de caractères *non nommée* présentée immédiatement après la déclaration de la fonction, comme dans l'exemple ci-dessous :

```
                                                    ###
```

La fonction `somme` contient donc une **docstring** - introduite par trois guillemets (pour permettre les sauts de lignes). Celle-ci décrit l'effet de la fonction, de manière exacte.

On peut alors accéder à la **docstring** d'une fonction en utilisant la fonction *built-in* `help` :

Testez par exemple `help(somme)`.

Les docstrings

Une docstring est essentielle pour comprendre l'utilité d'une fonction ! Vous devrez en utiliser le plus souvent possible !

2. Notion de portée des variables

Au sein d'un même programme, les variables définies n'ont pas systématiquement la même **portée**. La **portée d'une variable**, c'est l'espace des objets/noms (on parle aussi de *monde*) dans lequel est défini cette variable.

En utilisant le site Python Tutor, nous allons essayer de comprendre cette notion de portée des variables.

2.1. Variables globales

Dans l'exemple ci-dessus, la variable `gvar` est définie dans **l'espace de nom global** (*global frame*). Elle est accessible en lecture depuis l'intérieur de la fonction `f`. On parlera alors de **variable globale**.

2.2. Variables locales

Dans l'exemple ci-dessus, la variable `gvar2` est définie dans l'espace des noms associé à la fonction `f`, et qui est créé **au moment de l'appel à cette fonction** (*frame f*). Cet espace est détruit par le *garbage collector* dès que l'exécution de la fonction est terminé (une fois la valeur de `return` renvoyée dans l'espace appelant). Il devient donc impossible d'utiliser la variable `gvar2` puisqu'elle a disparue. `gvar2` est une **variable locale** à la fonction `f`.

Changer la valeur d'une variable globale

Regardons maintenant le code suivant :

Le code précédent déclenche une erreur `UnboundLocalError` en ligne 3. Cela signifie que l'interpréteur Python ne peut pas effectuer la ligne `gvar=gvar+2` car il cherche une variable `gvar` qu'il peut modifier. Or **une variable définie hors d'une fonction ne peut pas être modifiée par celle-ci**.

Il est par contre possible de travailler sur **une copie** de la variable souhaitée (*uniquement dans le cas des types primitifs* `int`, `float`, `str`...*), en utilisant une fonction ayant un argument. Il faudra par conséquent retourner la valeur changée** pour qu'elle soit effective. Par exemple comparez les deux scripts suivants :

Code incorrect

Ici on constate qu'en fait il y a deux variables `a` :

- une en dehors de la fonction, qui n'est pas modifiée. C'est une variable **globale**.
- une à l'intérieur de la fonction, qui peut être modifiée, mais qui ne change pas la variable globale. C'est une variable **locale** à la fonction.

La fonction `f` ne renvoyant aucune donnée, la variable locale `a` est détruite après la fin de la fonction `f`.

Code correct

Ici on a rajouté deux lignes :

- `return a` qui permet à la fonction de renvoyer la valeur modifiée ;
- `a = f(a)` La valeur renvoyée par l'appel `f(a)` est affecté au nom de variable `a`. L'opération effectuée à l'intérieur de la fonction se retrouve répercutée sur la variable globale `a`.

3. Tests, assertions et module Doctest

3.1. Préconditions

Comme nous l'avons vu pour la fonction `ask_user_int`, il est souvent nécessaire de tester des **préconditions** sur les arguments d'une fonction, pour s'assurer que celle-ci fonctionnera bien selon le schéma voulu.

Une méthode pour tester les préconditions est d'utiliser des **assertions**.

L'instruction `assert` de Python fonctionne de la manière suivante :

```
assert trouve_chaine('Toto', 'Toto va à la plage')==True, 'Meme casse pas trouvée'
```

L'instruction `assert` teste un booléen, ici `trouve_chaine('Toto', 'Toto va à la plage')==True`. Il peut alors se produire deux cas :

- soit le booléen est `True`, auquel cas l'interpréteur passe à la ligne suivante (il ne se passe rien de visible) ;
- soit le booléen est `False`, auquel cas l'interpréteur **arrête le code** en levant une erreur de type `AssertionError` et affiche la chaine de caractère passée en second argument, ici `'Meme casse pas trouvée'`.

Correction de la fonction `ask_user_int`

```
def ask_user_int(borne_min : int, borne_max : int, prenom : str ='Inconnu' ) -> int :
    """ Fonction demandant un entier compris entre borne_nf et borne_sup.
    personnalisation du message avec l'argument optionnel prenom.
    renvoir un objet de type entier.
    """
    assert type(borne_inf) == int and type(borne_sup) == int, "Erreur de type sur
borne_inf et borne_sup"
    assert type(prenom) == str, "Erreur de type sur prenom"
    assert borne_inf <= borne_sup , "borne_inf n'est pas inférieure à borne_sup"

    while True :
        ...# Le code ici n'est pas changé
```

Testez maintenant cette fonction avec de mauvaises valeurs pour les arguments.

3.2. Réfléchir avant d'agir : écrire les tests avant la fonction

Lorsqu'on écrit une fonction, il est très important d'avoir une idée précise de ce que la fonction doit renvoyer, y compris dans les *cas extrêmes* ou *cas spécifiques*.

Par exemple, on pourrait considérer une fonction `coefficient_directeur` qui donne le coefficient directeur d'une droite quand on lui passe en argument les coordonnées de deux points A et B. Je rappelle que le calcul du coefficient directeur de la droite (AB) se fait par l'intermédiaire de la formule :

$$m = \frac{y_B - y_A}{x_B - x_A}$$

Ainsi nous aimerions que la fonction travaille avec 4 arguments de type entiers `xA, yA, xB, yB`, qu'elle renvoie le coefficient directeur de (AB) sous la forme d'un flottant si ce coefficient existe, et qu'elle renvoie `None` quand il n'existe pas. On, définit ici ce qu'on appelle une **interface** de la fonction :

```
def coefficient_directeur(xA : int, yA : int, xB : int, yB : int) -> float :
    """ Renvoie le coefficient directeur de la droite (AB) telle que :
        A (xA ; yA) et B(xB ; yB), et les coordonnées de A ety B sont entières.
        Dans le cas ou le coefficient directeur n'existe pas, renvoie None
    """
    ...
```

On souhaite donc **prévoir en avance** différents cas de fonctionnement de la fonction, et on aimerait vérifier que les calculs effectués par la fonction correspondent bien à ces cas.

Par exemple, nous aimerions que la fonction vérifie le test suivant :

```
>>> coefficient_directeur(2, 4, 3, 7)
3.0
```

Écrire des tests

Enoncé

1. Ecrire 5 tests prenant en compte tous les cas possibles d'utilisation de la fonction, en supposant que les types fournis en argument soient bien des entiers.
2. Compléter la fonction `coefficient_directeur` afin qu'elle remplisse le rôle qui lui est demandé.

Solution

A venir !

Tester avec le module doctest

Pyhton étant *user friendly*, il permet au programmeur de tester automatiquement, grâce au module `doctest`.

Un module Python est un fichier (ou un ensemble de fichiers) qui comporte(nt) des objets et des fonctions qui peuvent être ajoutés aux fonctionnalité de base de Python. Il en existe un très grand nombre, tous étant spécialisés dans un domaine. On trouve par exemple :

- le module *math*, qui contient beaucoup de fonctions mathématiques ;
- le module *turtle*, qui permet de dessiner géométriquement ;
- le module *pygame*, qui est un module permettant de gérer les différents éléments d'un jeu vidéo ;
- le module *panda*, qui est utilisé pour faire du traitement de données ;
- le module *flask*, qui permet de créer une application web
- ...

Un module doit être chargé en mémoire une fois (de préférence au début du code), par l'intermédiaire de la commande :

```
import nom_du_module
```

Mais non, ca ne se fait pas que comme ça !

Le module *doctest*, lui, permet d'intégrer à la **docstring un ensemble de tests** qui sont vérifiés par l'appel de la fonction **testmod** du module *doctest*.

Par exemple, copiez-collez le code suivant dans un fichier :

```
import doctest # On charge en mémoire le module doctest

# Zone de déclaration des fonctions

def times2(n : object) -> object:
    """
    Fonction qui multiplie par 2, selon le type d'objet

>>> times2(4)
8
>>> times2(5.4)
10.8
>>> times2('a')
'aa'
"""
    return 2*n


# Zone du script global

doctest.testmod() # Lance les tests des fonctions déclarées
```

L'appel à la fonction `doctest.testmod()` déclenche les trois tests présents dans la doctsring, et vérifie que les résultats de la fonctions correspondent à ceux fournit dans la docstring.

Application 7 : Ajouter une fonction et des tests

Enoncé

Créez une fonction `times3` qui multiplie un objet par 3, en écrivant les tests correspondants.

Solution

A venir !

Je me servirai de doctests ou bien d'assertions pour estimer la justesse de vos fonctions et de vos programmes. Une bonne idée serait de TOUJOURS fournir un jeu d'exemple avant de se lancer dans la construction d'une fonction.

4. Exercices

Dans tous les exercices suivants, on supposera que l'utilisateur·trice de la fonction fournit des arguments du bon type

Pour tester vos fonctions avec les jeux fournis, n'oubliez pas :

- d'importer le module avec `import doctest` ;
- de lancer les tests avec `doctest.testmod()` ;

Vous pourrez créer un seul fichier contenant l'ensemble des fonctions ci-dessous.

1. Ecrire une fonction qui renvoie le maximum de deux nombres `int` donnés :

```
def maxi(a : int,b :int) -> int :
    """
    Fonction qui renvoie le maximum de deux nombres de type int.
>>> maxi(12,3)
12
>>> maxi(-5,9)
9
>>> maxi(6,6)
6
    """
```

2. Écrire une fonction qui renvoie le minimum de deux nombres `int` donnés :

```
def mini(a : int, b : int ) -> int :
    """
    Fonction qui renvoie le minimum de deux nombres
>>> mini(12,3)
3
>>> mini(-5,9)
-5
>>> mini(6,6)
6
    """
```

3. Écrire une fonction qui renvoie le maximum de trois nombres `int` donnés :

```
def maxi3(a : int, b : int, c : int) -> int :
    """
    Fonction qui renvoie le maximum de trois nombres
>>> maxi3(5,12,3)
12
>>> maxi3(-5,-7,2)
2
>>> maxi3(6,6,6)
6
>>> maxi3(5,7,7)
7
    """
```

4. Écrire une fonction qui renvoie le nombre intermédiaire dans trois nombres `int` donnés

```
def intermediaire(a : int, b : int, c : int) -> int :
    """
    Fonction qui renvoie le nombre intermédiaire deux nombres
>>> intermediaire(12,8,3)
8
>>> intermediaire(-5,9,0)
0
>>> intermediaire(7,7,7)
7
>>> intermediaire(7,4,7)
7
>>> intermediaire(4,4,7)
4
    """
```

5. Erire une fonction qui supprime tous les caractères qui ne sont pas des lettres (majuscules ou minuscules, sans accents) d'une chaine de caractères donnée.

```
def rienQueDesLettres(chaine) :
    """
>>> rienQueDesLettres('  toto  ')
'toto'
>>> rienQueDesLettres('123Toto456')
'Toto'
>>> rienQueDesLettres("Et!C'est Toto ?")
'EtCestToto'
    """
```

6. Ecrire la fonction coefficient_directeur qui vérifie les conditions ci-dessous

```
def coefficient_directeur(xA,yA,xB,yB) :
    """Fonction renvoyant le coefficient directeur de la droite (AB)
en connaissant les coordonnées des points A et B, ou None si c'est impossible
>>> coefficient_directeur(0,0,1,5)
5
>>> coefficient_directeur(0,0,2,10)
5
>>> coefficient_directeur(3,4,4,6)
2
>>> coefficient_directeur(3,4,4,6)
2
>>> coefficient_directeur(3,4,4,4)
0
>>> coefficient_directeur(3,4,4,3)
-1
>>> coefficient_directeur(3,4,3,7)==None
True
>>> coefficient_directeur(3,4,3,4)==None
True
>>> coefficient_directeur(4,4,3,4)==None
False
    """
###VOTRE CODE ICI
```

7. Ecrire une fonction - et le jeu de test correspondant, qui calcule l'ordonnée à l'origine d'une droite (AB), en prenant en argument les coordonnées des points A et B comme la fonction précédente, et qui renvoie None si c'est impossible.
8. Ecrire une fonction - et le jeu de test qui va avec, qui renvoie l'équation réduite de la droite (AB), en prenant en argument les coordonnées des points A et B comme dans les fonctions précédentes.
9. Ecrire une fonction qui donne le discriminant d'un trinôme du second degré $ax^2 +bx+c $, en fournissant un jeu d'exemples complets.