Exercice 02

Révisions;

1. Définir des classes et des sous-classes, des classes abstraites
2. Quand définir un constructeur dans une sous-classe?
3. Pourquoi typer son code?
4. Qu'est-ce qu'une importation circulaire?
5. Utiliser une collection d'objets

Quand surcharger le constructeur d'une sous-classe?

1. Pas nécessaire si pas d'attribut spécifique (ImageFile et Moderator)
2. Nécessaire si…
    1. Attribut spécifique (FilePost.file)
    2. Argument supplémentaire (file)
    3. Initialisation spécifique

Pourquoi typer son code?

1. Documentez son code
2. Détectez les erreurs de « duck typing » (avertissement PyCharm)

Quel code doit être typé?

1. Les variables (leur 1ère affectation)
2. Les paramètres des méthodes/fonctions
3. La valeur de retour d'une méthode/fonction (si return explicite)

Quel code doit être typé?

```python
# dictionnaire
filesDic: Dict[str, File] = {"f1": unFichier, "f2": unFichierImage, "f3": unFichierGIF, "f4": unFichierPNG,
                             "test": SimpleFile("test.txt", 100)}
unAutreFichier: File = filesDic["test"]
print(unAutreFichier)
filesDic["test"] = SimpleFile("test2.txt", 200)
```

```python
class File(ABC):
    """Fichier."""

    def __init__(self, name: str, size: int):
        """Initialise le nom et la taille."""
        self.name = name
        self.size = size

    @abstractmethod
    def display(self):
        """Affiche le fichier."""
        pass

    def __str__(self) -> str:
        """Représentation d'un fichier."""
        return f"'{self.name}' est un fichier de taille {self.size}"
```

Importation de module circulaire

```
Traceback (most recent call last):
  File "C:\Users\lv027851\Desktop\Ahuntsic\313\Cours04\Exercice\main.py", line 3, in <module>
    from moderator import Moderator
  File "C:\Users\lv027851\Desktop\Ahuntsic\313\Cours04\Exercice\moderator.py", line 6, in <module>
    from user import User
  File "C:\Users\lv027851\Desktop\Ahuntsic\313\Cours04\Exercice\user.py", line 4, in <module>
    from thread import Thread
  File "C:\Users\lv027851\Desktop\Ahuntsic\313\Cours04\Exercice\thread.py", line 3, in <module>
    from post import Post
  File "C:\Users\lv027851\Desktop\Ahuntsic\313\Cours04\Exercice\post.py", line 2, in <module>
    from user import User
ImportError: cannot import name 'User' from partially initialized module 'user' (most likely due to a circular import) (C:\Users\lv027851\Desktop\Ahuntsic\313\Cours04\Exercice\user.py)
```

Import circulaire (User -> Post et Post _> User)

Il y a 3 solutions pour corriger une importation circulaire;

1. Utiliser un type documenté qui ne sera pas vérifié par PEP8 (‘User’ au lieu de User)
2. Regrouper les classes dans le même module (user_post.py)
3. Revoir la conception des classes (mauvais design)

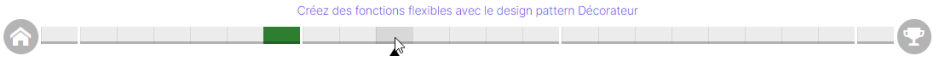
Objectifs – Cours 03

Écrire du code Python maintenable

1. Conformer son code à la norme PEP8

2. Utiliser le patron constant (“no magic string”)

3. Utiliser un décorateur



Écrivez du code Python maintenable

Écrivez du code Python maintenable

Découvrez les conventions Python

Résumé

- D’autres développeurs auront fréquemment besoin d’utiliser votre code, et donc de le comprendre.
- Les conventions Python se trouvent sur le site de Python dans des documents qui s'appellent les PEP.
- La PEP 8 contient de nombreuses recommandations pour aider les développeurs Python à écrire du code compréhensible par les autres.

Écrivez du code Python maintenable

Écrivez du code qui s’explique lui-même

Convention de nommage PEP8

- Écrivez les noms de variables en minuscules, avec des *underscores* (tirets bas). C'est ce qu'on appelle la convention « *snake_case* » :

python

```python
name = "Jeanne"
fuel_level = 100
famous_singers = ["Céline Dion", "Michael Jackson", "Edith Piaf"]
```

- Écrivez les variables constantes en majuscules, avec des *underscores* :

python

```python
DAYS_PER_WEEK = 7
PERSONAL_EMAIL = "myemail@email.com"
```

- Écrivez les noms de classe avec une majuscule au début de chaque mot, sans ponctuation (la convention « *CapitalizedCase* ») :

python

```python
class JukeBox:
    pass

class SpaceShip:
    pass
```

- Écrivez les noms de modules en minuscules, en évitant au maximum l'utilisation des *underscores* :

python

```python
# Exemples pris de la bibliothèque standard de Python
import os
import sys
import shutil  # contraction de “sh” (langage bash) et “util”
import pathlib  # contraction de “path” et “lib”
```

Résumé

- Les variables, classes et modules doivent porter des noms utiles et écrits selon le style suggéré.
- Les commentaires doivent être utilisés avec modération, maintenus à jour, et indentés de la même façon que la ligne de code suivante.
- Les docstrings sont des commentaires spéciaux, auxquels on peut accéder avec `__doc__` .
- À l'inverse des commentaires, usez et abusez des docstrings ! 😊

Écrivez du code Python maintenable

Écrivez du code facile à lire

Résumé

- Pour que le code Python soit cohérent et facile à lire, la PEP a différentes recommandations sur la présentation du code.
- Le code doit être indenté avec 4 espaces (et non des tabulations).
- Le code doit utiliser des espaces et sauts de lignes adaptés pour aérer le code sans trop le gonfler.
- Les lignes de code ne doivent pas excéder 79 caractères, et il faut être prudent si l'on découpe de longues expressions en plusieurs lignes.
- Les fichiers doivent être écrits en commençant par les commentaires qui concernent le fichier, puis les imports, puis les constantes, et ensuite tout le reste du code.

Écrivez du code Python maintenable

Écrivez du code antibug

Recommandations de programmation

A Foolish Consistency is the Hobgoblin of Little Minds

One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 says, "Readability counts".

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.

However, know when to be inconsistent -- sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

In particular: do not break backwards compatibility just to comply with this PEP!

Some other good reasons to ignore a particular guideline:

1. When applying the guideline would make the code less readable, even for someone who is used to reading code that follows this PEP.
2. To be consistent with surrounding code that also breaks it (maybe for historic reasons) -- although this is also an opportunity to clean up someone else's mess (in true XP style).
3. Because the code in question predates the introduction of the guideline and there is no other reason to be modifying that code.
4. When the code needs to remain compatible with older versions of Python that don't support the feature recommended by the style guide.

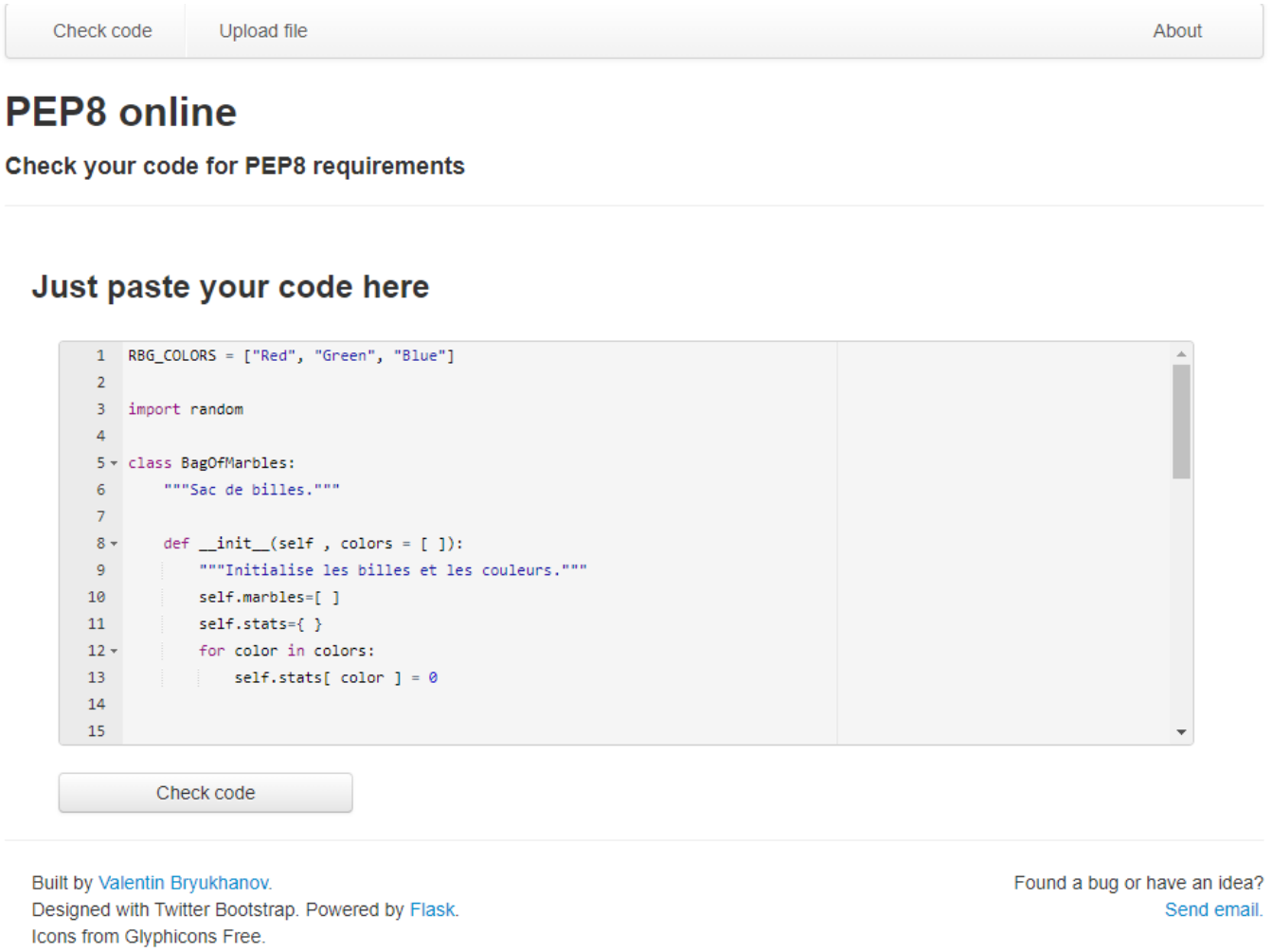
Résumé

- La PEP 8 comprend différentes recommandations de programmation pour vous aider à écrire du code antibug.
- Les fonctions doivent retourner un seul type (entier, chaîne, booléen, etc.) et/ou None.
- On préfère `str.startswith` et `str.endswith` au tranchage de chaînes, car vous n'avez qu'un élément de votre code à changer lorsque vous les modifiez.
- Les blocs try doivent être placés de façon à couvrir le moins de code possible.
- Les clauses except doivent toujours attraper un type spécifique d'exception.

Écrivez du code Python maintenable

Utilisez des linters pour que votre code reste propre

PEP8 en ligne



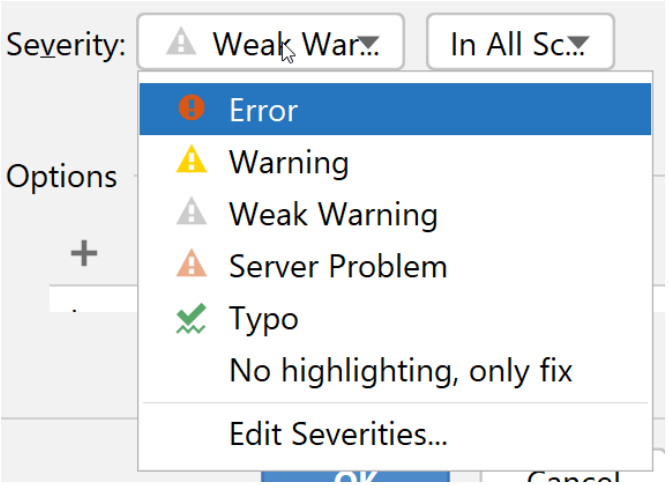
Résumé

- Les linters sont des outils automatisés qui vous préviennent lorsque votre code n'est pas conforme à la PEP 8.
- Les linters ne trouveront pas toutes vos erreurs de programmation, mais uniquement les principales recommandations de style de la PEP 8.

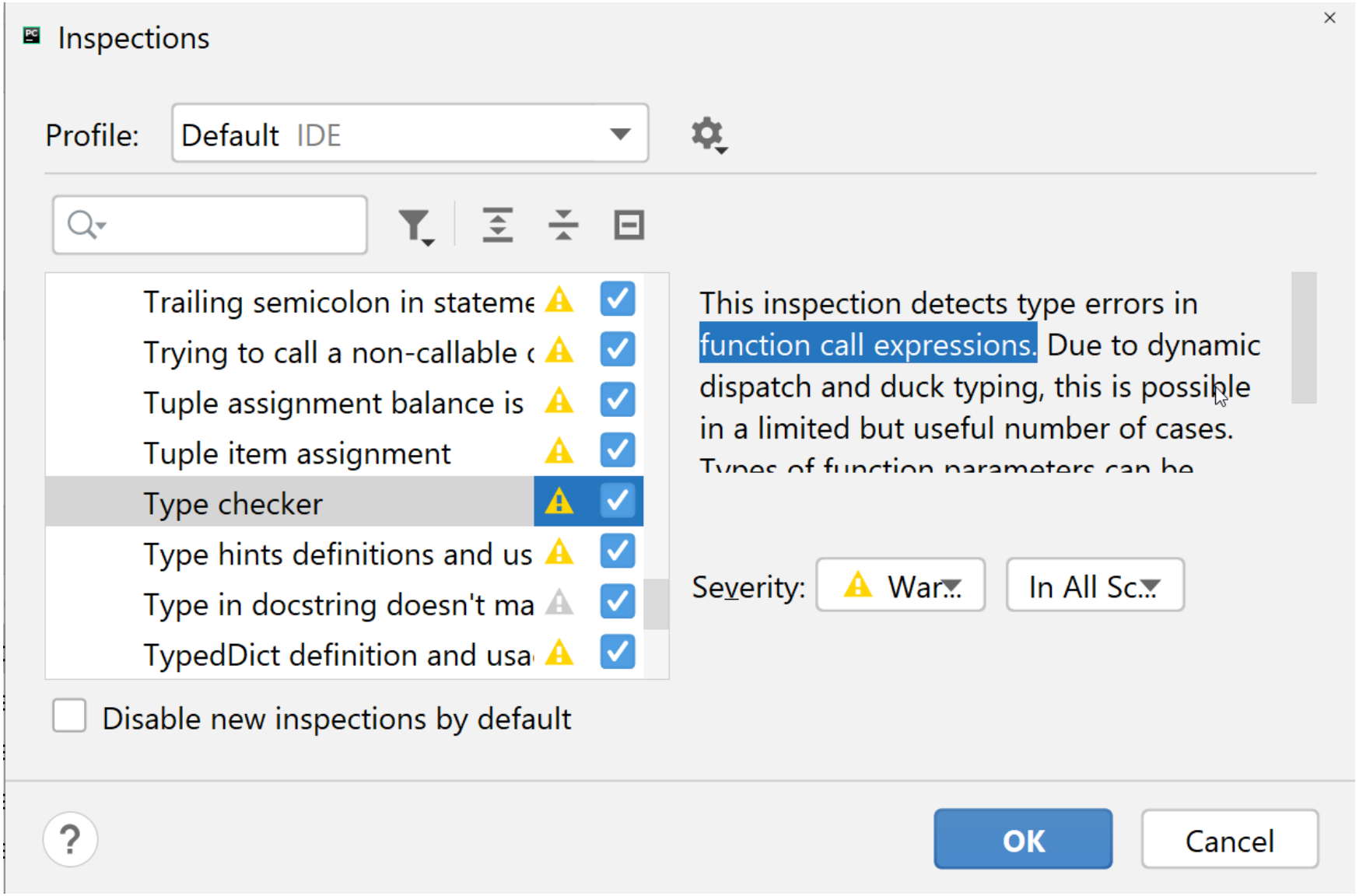
Comment conformer son code à la norme PEP8?

Vérifier la section « Problems » de chaque fichier de code python et Corriger tous les items quelques soit le niveau de sévérité (configurable);

1. Erreurs
2. Avertissements
3. Avertissement léger
4. Typos



Configuration PEP8 – Type Checker



Exemple

main.py C:\Users\lv027851\Desktop\Ahuntsic\313\Cours04\Exerci

- Expected type 'int', got 'Moderator' instead :11
- Unresolved attribute reference 'post' for class 'int' :16
- Unresolved attribute reference 'edit' for class 'int' :22
- Unresolved attribute reference 'delete' for class 'int' :28
- Unresolved attribute reference 'post' for class 'int' :33

```
7   from user import User
8
9   """Lance le code principal."""
10  user: User = User("John", "superpassword")
11  moderator: int = Moderator("Lucie", "helloworld")
12
13  cake_thread: Thread = user.make_thread("Gâteau à la
14  cake_thread.display()
15
16  moderator.post(cake_thread, content="Oui j'aime bea
17  cake_thread.display()
18
```

Exemple

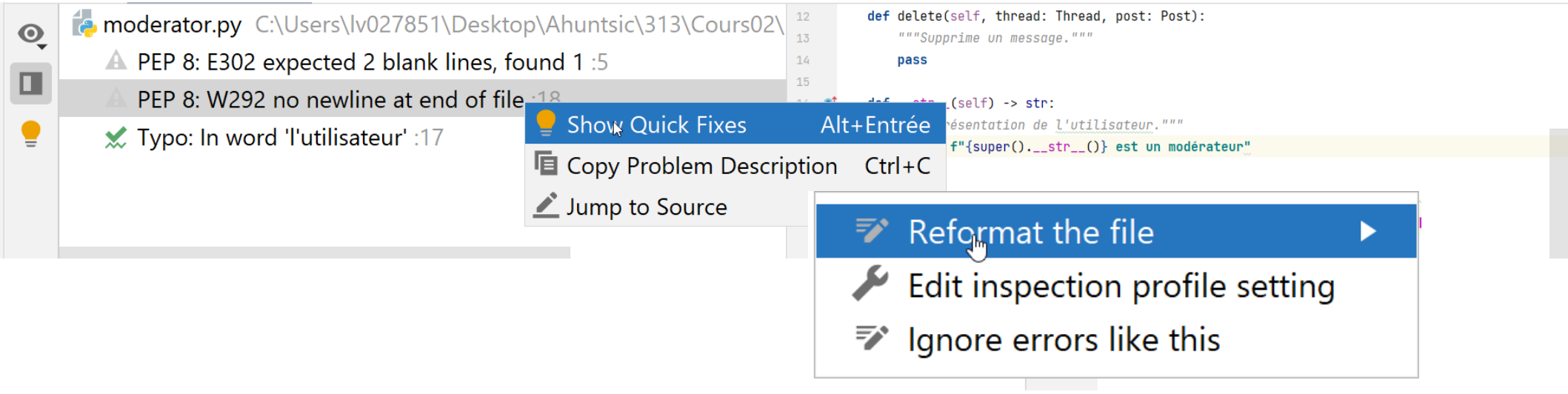
main.py C:\Users\lv027851\Desktop\Ahuntsic\313\Cours02\Exerci

Dictionary contains duplicate keys 'test' :20

Dictionary contains duplicate keys 'test' :20

```
18  # dictionnaire
19  filesDic: Dict[str, File] = {"f1": unFichier, "f2": unFichierImage, "f3": unFichierGIF, "f4": un
20                               "test": SimpleFile("test.txt", 100), "test": SimpleFile("test.txt",
21  unAutreFichier: File = filesDic["test"]
22  print(unAutreFichier)
23  filesDic["test"] = SimpleFile("test2.txt", 200)
24
25  print(len(filesDic))
26  del filesDic["test"]
27  print(len(filesDic))
28  for key in filesDic:
29      print(filesDic[key])
```

Exemple


moderator.py C:\Users\lv027851\Desktop\Ahuntsic\313\Cours02\
PEP 8: E302 expected 2 blank lines, found 1 :5
PEP 8: W292 no newline at end of file :18
Typo: In word 'l'utilisateur' :17
Show Quick Fixes Alt+Entrée
Copy Problem Description Ctrl+C
Jump to Source
Reformat the file
Edit inspection profile setting
Ignore errors like this
def delete(self, thread: Thread, post: Post):
"""Supprime un message."""
pass
(self) -> str:
ésentation de l'utilisateur."""
f"{super().__str__()} est un modérateur"

Écrivez du code Python maintenable

Construisez des systèmes complexes à l'aide de patterns

« Design Pattern »

En 1994, le **« Gang des Quatre » (Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides)** sortait le livre ***Design Patterns : Catalogue de modèles de conception réutilisables***. Ces quatre auteurs ont observé plus de 300 projets sur lesquels d'autres développeurs travaillaient. Ils ont constaté que de nombreux **problèmes** récurrents étaient **identiques**.

Patrons (« pattern») de conception

1. Le design pattern Constant : facilite la mise à jour et donne un sens **aux valeurs** dans le code.

2. Le design pattern Décorateur (« Decorator ») : facilite la création de nombreuses fonctions qui accomplissent des choses similaires.

Écrivez du code Python maintenable

Évitez les confusions avec le pattern constant

« Design pattern » constant

- Identifiez tout nombre (ou autre variable) qui est utilisé à **plusieurs emplacements**, ou dont la signification n'est **pas claire**.
- Déclarez sa valeur en tant que variable ***globale*** (dans la portée du module), avec un nom clair – même si nous n'avons pas l'intention de la changer. Par exemple : `number_of_guests = 2` .
- Ces valeurs constantes seront souvent utilisées par **plusieurs fonctions, classes et fichiers** au sein d'un projet, donc d'après la PEP 8, elles devraient être définies vers le haut du fichier en majuscules, par exemple `NUMBER_OF_GUESTS = 2` .
- **Reformulez** toutes les déclarations qui reposent sur ce nombre en utilisant la nouvelle valeur constante.
- Ensuite, si un développeur a un jour besoin de modifier ce comportement, il n'a plus qu'à **changer la définition de la variable**, tout simplement ! Par exemple : `NUMBER_OF_GUESTS = 15` .

Démo

```python
from film import Film

class FilmCassette(Film):
    """Un film en cassette !"""

    def __init__(self, name):
        """Initialise le nom et la bande magnetique."""
        #self.name = name
        super().__init__(name)
        self.magnetic_tape = True

    def rewind(self):
        """Rembobine le film."""
        print(f"Patience..., {self.name} est assez long à rembobiner!")

    def watch(self, player):
        if player.type != "cassette":
            print(f"Le lecteur n'est pas un lecteur de cassettes, le fim {self.name} ne peut pas être lu!")
            return
        else:
            self.rewind()
            super().watch(player)
```

Quelle valeur serait candidate?

Quand utiliser le patron Constant?

- Le design pattern Constant est aussi une forme de documentation du code : il met des mots sur des valeurs «brutes».
- NUMBER_OF_MONTHS sera toujours plus clair que 12
- On appelle les valeurs mal documentées des «nombres magiques».

Résumé

- Le design pattern Constant est un modèle simple qui n'affecte qu'une seule valeur.
- Les valeurs qui se répètent peuvent être définies une seule fois dans l'application.
- Les futurs développeurs pourront facilement comprendre la signification de la valeur.
- Les futurs développeurs pourront facilement modifier la valeur si les prérequis sont modifiés.
- De nombreux bugs étonnants peuvent être évités en utilisant le design pattern Constant.

Écrivez du code Python maintenable

Créez des fonctions flexible avec le pattern décorateur

Pourquoi?

- Séparer les responsabilités d'une fonction/méthode
- Éviter de répéter du code

Comment l'utiliser?

Créez une fonction décorateur qui

- Attend une autre fonction en paramètre,
- Retourne une variation décorée de cette fonction en retour.

Le décorateur n'est qu'un modificateur de fonction. Il va la transformer pour rajouter des choses avant, et après.

Pseudo code du décorateur

```
def decorate_function(function):
    """Cette fonction va générer le décorateur."""

    def wrapper():
        """Voici le "vrai" décorateur.

        C'est ici que l'on change la fonction de base
        en rajoutant des choses avant et après.
        """
        print("Do something at the start")

        result = function()

        print("Do something at the end")

        return result

    return wrapper

```

Appel explicite d’un décorateur

```
20
21 def travelling_through_the_stars():
22     """Voyage à travers les étoiles."""
23     print("C'est parti pour un long voyage !")
24
25
26 # ici, nous allons récupérer le retour de "decorate_function",
27 # qui n'est autre que la fonction "wrapper" !
28 # Notez que nous pouvons très bien renommer une fonction en
29 # l'assignant dans une nouvelle variable (ici "wrapper" devient "decorated").
30 decorated = decorate_function(travelling_through_the_stars)
31 decorated()  # nous executons la fonction "wrapper"
```

Appel implicite par annotation

```
14 @decorate_function  # c'est ici que ça se passe !
15 def travelling_through_the_stars():
16     """Voyage à travers les étoiles."""
17     print("C'est parti pour un long voyage !")
18 
19 
20 # la fonction est directement décorée, et s'utilise comme telle, comme si rien
21 # comme si rien n'avait changé ! ;)
22 travelling_through_the_stars()
```

Démo

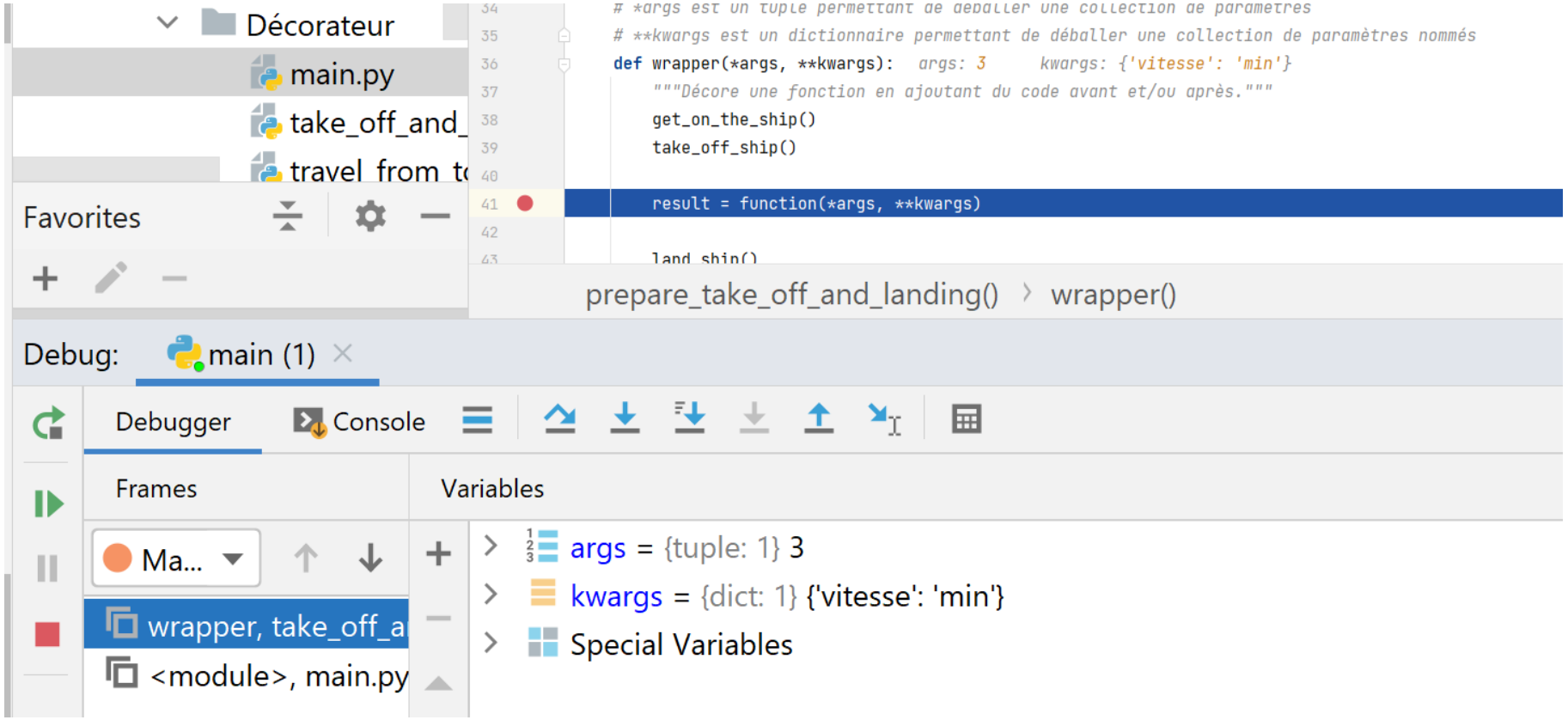
python

```python
def travel_from_to_earth_to_ganymede():
    """Commence l'aventure, de la terre à Ganymède."""
    get_on_the_ship()
    take_off_ship()

    # code pour voyager dans l'espace

    land_ship()
    exit_the_ship()
    open_the_hold()

def travel_from_ganymede_to_kepler():
    """Voyage de Ganymède à Kepler 438 b..."""
    get_on_the_ship()
    take_off_ship()

    # code pour voyager dans l'espace

    land_ship()
    exit_the_ship()
    open_the_hold()
```

Paramètres dynamiques

- *args et **kwargs sont des paramètres dynamiques
- « unpacking » ; l'opérateur * « défait » une liste et ** un dictionnaire
- args est un tuple permettant de passer les paramètres d’une fonction
- kwargs est un dictionnaire utilisé lorsque les paramètres de la fonction sont **nommés**

Démo


Décorateur
main.py
take_off_and_
travel_from_t
Favorites
*args est un tuple permettant de déballer une collection de paramètres
**kwargs est un dictionnaire permettant de déballer une collection de paramètres nommés
def wrapper(*args, **kwargs): args: 3 kwargs: {'vitesse': 'min'}
"""Décore une fonction en ajoutant du code avant et/ou après."""
get_on_the_ship()
take_off_ship()
result = function(*args, **kwargs)
land_ship()
prepare_take_off_and_landing() › wrapper()
Debug:
main (1)
Debugger
Console
Frames
Variables
Ma...
wrapper, take_off_a
<module>, main.py
args = {tuple: 1} 3
kwargs = {dict: 1} {'vitesse': 'min'}
Special Variables

Résumé

- En Python, les fonctions sont des objets comme les autres : elles peuvent donc être passées à d'autres fonctions et en sortir comme n'importe quelle autre valeur.
- Le design pattern Décorateur fournit une façon de modifier une fonction, souvent en ajoutant des fonctionnalités avant et après son exécution.
- Il peut être utile lorsque plusieurs fonctions similaires ont des fonctionnalités centrales différentes, mais des fonctionnalités partagées significatives.
- Il peut aussi être utile lorsque vous ne souhaitez pas modifier le code interne d'une fonction, pour pouvoir la réutiliser de différentes manières.
- La syntaxe `@decorate_function` simplifie l'écriture de code impliquant des décorateurs.