Mémo Python

Table des matières

Remarque Ne sont pas reprises les opérations « basiques » (boucles, manipulation des listes). On s'intéresse directement à la programmation orientée objet. Ensuite sont détaillés quelques fonctionnalités apportées par des modules.

Première partie

Programmation orientée objet en Python

1 Constantes et types natifs

Quelques constantes sont définies par Python comme `True` et `False`. De la même manière, quelques classes sont définies par défaut, appeléss types natifs, comme :

1. Types booléens
   (a) Booléen (`bool`)
   (b) Opérations booléennes (`and, or, not`)
2. Types numériques
   (a) Entier (`int`)
   (b) Flottant (`float`)
   (c) Complexe (`complex`)
3. Types séquentiels
   (a) Liste (`list`)
   (b) Tuple (`tuple`)
   (c) Range (`range`)
4. Chaîne de caractères (`str`)
5. Séquences binaires (`bytes, bytearray, memoryview`)
6. Ensemble (`set` (muable), `frozenset` (non muable))
7. Dictionnaire (`dict`)

La liste n'est pas exhaustive, des compléments sur les types natifs sont disponibles dans la documentation.

Documentation Documentation Python 3 : constantes natives, types natifs

2 Variables

3 Classes

Les classes permettent des créer des objets appelés instances) qui partagent des caractéristiques de leur classe. Une classe correspond donc à un type, ou plutô à un gabarit d'objet.

Documentation Introduction OpenClassrooms, Documentation Python 3, Wikilivres

3.1 Structure d'une classe

Les objets d'une classe partagent des caractéristiques communes à la classe : des attributs et des méthodes (des fonctions qui agissent sur leur attributs). Les objets sont créés gâce à une méthode spéciale appelée constructeur.

3.1.1 Création

Pour créer une classe, la syntaxe est la suivante :

```
class MaClasse:
    <contenu>
```

Si la classe hérite d'une classe mère, alors il faut l'ajouter en argument :

```
class MaClasse(ClasseMere):
    <contenu>
```

3.1.2 Initialiseur

L'initialiseur est une méthode spéciale appelée `__init__`, il prend en argument `self` (toutes les méthodes de la classe prennent en argument `self` qui est en fait l'instance en question) et tous les paramètres nécessaires à l'initialisation de l'instance. Il ne s'agit pas du constructeur (même si on l'appelle parfois ainsi par abus de langage) : il ne crée pas à proprement parler l'instance, mais agit sur celle-ci lorsqu'elle vient d'être créée. Le « véritable » constructeur est la méthode `__new__`, il n'est en général pas nécessaire de l'implémenter, sauf par exemple lorsque l'on crée des métaclasses ou si l'on veut créer des classes qui héritent des types natifs. L'initialiseur est appelé automatiquement lors que l'on crée l'objet (après le constructeur).

```
class MaClasse:
    CONSTANTE = ...
    def __init__(self, att1, att2):
        """Initialiseur"""
        self.attribut_1 = att1
        self.attribut_2 = att2
        self.attribut_3 = CONSTANTE
```

Ici, les deux premiers attributs sont personnalisables lors de la création des objets alors que le dernier est commun à tous. `CONSTANTE` est une variable de classe. Pour créer un objet on écrit simplement :

```
objet = MaClasse(att1, att2)
```

3.1.3 Méthodes

Les méthodes se définissent comme des fonctions, elles agissent en général sur les instances de la classe. Elles doivent prendre `self` en argument :

```
class MaClasse:
    def __init__(self):
         <contenu>

    def methode(self, arg1, arg2):
         <contenu>
```

Ensuite on les appelle de la manière suivante :

```
objet.methode(arg1, arg2)
```

3.2 Héritage

L'héritage est un moyen de créer des classes dérivées (classes filles) d'une classe source (classe mère). Une classe fille hérite de toutes les méthodes et variables de sa classe mère. Pour créer une classe fille, on utilise la syntaxe suivante.

```
class Mere:
    <contenu>

class Fille(Mere):
    <contenu>
```

Il est possible d'écraser une méthode héritée en la redéfinissant dans la classe fille. Si on veut accéder à une méthode héritée alors qu'on l'a redéfinie dans la classe fille, on utilise la fonction `super()` qui permet d'appeler la méthode de la classe mère de la classe présente (sans l'argument self).

Exemple

```
class Meuble:
    def __init__(self, couleur, materiau):
        self.couleur = couleur
        self.materiau = materiau

class Bibliotheque(Meuble):
    def __init__(self, couleur, materiau, n):
        super().__init__(couleur, materiau)
        self.nb_livres = n
```

On peut utiliser deux fonctions pour vérifier l'héritage : `isinstance` renvoie `True` si l'objet est une instance de la classe ou de ses classes filles ; `issubclass` permet de voir si une classe est fille d'une autre.

```
>>> bibli = Bibliotheque('blanc', 'vert', 150)
>>> bibli.__dict__
{'couleur': 'blanc', 'materiau': 'vert', 'nb_livres': 150}
>>> isinstance(bibli, Meuble)
True
>>> isinstance(bibli, Bibliotheque)
True
>>> issubclass(Bibliotheque, Meuble)
True
>>> issubclass(Meuble, Bibliotheque)
False
>>> isinstance(bibli, int)
False
>>> isinstance(bibli, object)
True
```

Documentation OpenClassrooms, Documentation Python 3, Programiz

3.3 Méthodes spéciales

Les méthodes spéciales sont déjà définies par défaut dans Python mais on peut les personnaliser. Elles sont reconnaissables par leur typographies : leur nom commence et se termine par deux soulignés.

Documentation Documentation Python 3, OpenClassrooms

3.3.1 Construction, initialisation et destruction

Le constructeur est la méthode `__new__`. C'est une méthode de classe qui prend en argument `cls` et les autres arguments qui seront en paramètres de l'initialiseur ; il doit retourner un objet (l'instance à créer). `__init__` a déjà été décrit précédemment (contrairement au constructeur, cette méthode ne retourne rien). En pratique, on n'implémente pas la méthode `__new__` sauf dans certains cas.

Exemple On veut définir une classe « singleton » qui ne peut créer qu'une instance.

```
class Singleton:
    """Classe qui ne peut instancier qu'une fois."""

    instance = None

    def __new__(cls, *args, **kwargs):
        if instance is None:
            cls.instance = super().__new__(cls, *args, **kwargs)
            return cls.instance
        else:
            raise TypeError("Cette classe singleton possède déjà une instance")

    def __init__(self, *args, **kwargs):
        pass
```

Pour détruire un objet, on définit la méthode `__del__`. On l'appelle comme ceci :

```
del objet
```

3.3.2 Représentation et chaine de caractère d'un objet

Il existe deux méthodes spéciales nommées `__repr__` et `__str__` qui sont appelées lorsque l'on exécute `repr(objet)` ou `return objet`, et quand on exécute `str(objet)` ou bien `print(objet)`. La fonction `__repr__` est donc utilisée lorsque l'on veut avoir accès à la représentation d'un objet, tandis que `__str__` permet de présenter l'objet de manière plus élégante en chaine de caractères. Ces deux fonctions prennent en argument `self`. Lorsque la méthode `__str__` n'est pas définie, Python appelle la fonction de représentation à la place.

Exemple L'exemple suivant

```
class MaClasse:
    def __init__(self):
         self.attribut = 'Exemple'

    def __repr__(self):
        return "MaClasse({})".format(self.attribut)

    def __str__(self):
        return "Instance de MaClasse ayant comme attribut {}"
              .format(self.attribut)
```

permet de faire :

```
>>> obj = MaClasse()
>>> obj
MaClasse(Exemple)
>>> print(obj)
Instance de MaClasse ayant comme attribut Exemple.
```

3.3.3 Accesseur et mutateur

Lorsque Python essaie d'accéder à un attribut, il appelle en premier la méthode spéciale `__getattribute__`, puis il appelle les descripteurs s'il sont définis. Lorsque l'on veut modifier un attribut, c'est la méthode spéciale `__setattr__` puis les descripteurs qui sont appelés. Si on essaie d'accéder à un attribut non défini, Python appelle en guise de dernière chance la méthode `__getattr__`. On peut personnaliser cette fonction de manière à ce qu'elle envoie une erreur, ou bien à ce qu'elle redirige vers un autre attribut ou effectue un calcul.

Exemple

```
class MaClasse:
    def __init__(self)
        self.a = int()

    def __getattribute__(self, attribut):
        print("J'accède à l'attribut {}...".format(attribut))
        return object.__getattribute__(self, attribut)

    def __getattr__(self, attribut):
        print("L'attribut {} est inaccessible !".format(attribut))

    def __setattr__(self, attribut, valeur):
        object.__setattr__(self, attribut, valeur)
        print("L'attribut a été changé !")
        # Il est nécessaire d'appeler la méthode par défaut, car appeler self.__setattr__
        # donnerait une récursivité infinie. En fait, on ne sait à ce stade pas comment
        # Python change concrètement la valeur de l'attribut.
```

On note que l'on utilise les méthodes spéciales de la classe `object` (méthodes par défaut) car appeler `self.__getattribute__` ou `self.__setattr__` donnent une récursivité sans fin ! Cela permet de faire :

```
>>> objet = MaClasse()
L'attribut a été changé !
>>> objet.b
L'attribut b est inaccessible !
>>> objet.a
J'accède à l'attribut a...
0
>>> objet.attribut = 1
L'attribut a été changé !
>>> objet.a
J'accède à l'attribut a...
1
```

Il existe aussi `__delattr__` qui prend en arguments `self` et le nom de l'attribut. Cette méthode est appelée lorsque l'on effectue `del objet.attribut`. Lors de l'écriture de la méthode, il faut utiliser `object.__delattr__` de la même manière que l'on utilise `object.__setattr__` pour `__setattr__` ou `object.__getattribute__` pour `__getattribute__`.

3.3.4 Méthodes de conteneur

Il existe trois méthodes (accesseur, mutateur, destructeur) qui permettent d'agir sur l'objet avec l'opérateur `[]` (utilisé pour les listes par exemple). Dans ce cas, l'objet peut être un conteneur qui contient d'autres objets. Le fonctionnement de ses méthodes est similaires aux précédentes. Sont définies en outre `__contains__` qui permet de déterminer si un élément est présent ou non dans le conteneur (retourne un booléen) et `__len__` qui retourne la longueur du conteneur. Tableau récapitulatif :

Méthode	Arguments	Appel
`__getitem__`	`self, index`	`conteneur[index]`
`__setitem__`	`self, index, valeur`	`conteneur[index] = valeur`
`__delitem__`	`self, index`	`del conteneur[index]`
`__contains__`	`self, element`	`element in conteneur`
`__len__`	`self`	`len(conteneur)`

3.3.5 Surcharges d'opérateur

Les surcharges d'opérateur permettent de faire des opérations arithmétiques avec des objets, c'est-à-dire d'indiquer à Python ce qu'il faut faire lorsque l'on exécute `objet1 + objet2`. Ces méthodes prennent en arguments `self` (l'objet 1) et l'objet 2.

Méthode	Appel
`__add__`	`objet1 + objet2`
`__sub__`	`objet1 - objet2`
`__mul__`	`objet1 * objet2`
`__truediv__`	`objet1 / objet2`
`__floordiv__`	`objet1 // objet2`
`__mod__`	`objet1 % objet2`

Les deux objets ne sont pas nécessairement du même type ! Cependant, cette opération n'est pas symétrique : le code `objet + 5` par exemple exécute `objet.__add__(5)`, alors que `5 + objet` exécute `int.__add__(5)`. Pour que l'opération soit symétrique, il faut aussi définir ces fonctions avec le préfixe `r` (par exemple `__radd__`).

3.4 Méthodes statiques et méthodes de classes

3.4.1 Méthode statique

Les méthodes que l'on a vues jusqu'à maintenant agissent sur les instances des classes : elles prennent toujours en premier argument le mot clé `self` qui renvoie à l'instance elle même. Lorsque l'on appelle une telle méthode sur une instance comme ceci : `instance.methode(<arguments>)`, Python exécute en fait `Classe.methode(instance, <arguments>)`.

En fait, ces deux objets sont différents. `Classe.methode` est une simple fonction, alors que `instance.methode` est une méthode évaluée sur l'instance (en anglais « bound method »), c'est-à-dire que l'instance est mise en premier argument. On considère cet exemple :

Exemple

```
class Maths:

    def addition(x, y):
        return x + y

    def multiplication(x, y):
        return x * y

    def division(x, y):
        return x / y
```

On choisit ici de grouper trois fonctions car elles sont logiquement liées. Elles n'influent pas les instances donc elles ne prennent pas `self` en argument. Si l'on appelle ces méthodes sur une instance, une exception sera levée car Python entrera automatiquement l'argument `self` (donc en tout trois arguments) alors que les méthodes n'en prennent que deux. Pour remédier à cela, on les décore avec `@staticmethod`. On peut maintenant les appeler indifféremment sur la classe ou sur des instances.

3.4.2 Méthode de classe

Lorsque l'on veut manipuler des variables de classe et non des attributs d'instances, on crée des méthodes de classe. Celles-ci prennent la classe en premier argument, par convention on le note `cls` ; elles ne prennent logiquement pas `self` comme argument. Cette méthode est donc évaluée sur la classe. Sans autre modification, on ne peut peut appeler cette méthode que sur les instances car Python attend l'argument `cls`. Pour pouvoir appeler cette méthode sur la classe (logique car c'est une méthode de classe), on la décore avec `@classmethod`.

3.4.3 Cas de l'héritage

En résumé :

1. Les méthodes statiques sont des fonctions reliées à des classes, mais qui n'agissent pas sur celles-ci.
2. Les méthodes de classe sont des fonctions qui prennent la classe en paramètre.

Une classe qui hérite d'une classe mère hérite de toutes ses méthodes. Les méthodes statiques restent donc inchangées, tandis que les méthodes de classe s'adaptent à la nouvelle classe, car elles la prennent en premier argument.

Exemple Un exemple d'utilisation de méthodes statiques et de classe sont la création de constructeurs alternatifs. On s'aperçoit de la différence des deux notions.

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

    @staticmethod
    def par_date_de_naissance(nom, date):
        return Personne(nom, 2018-date)

    @classmethod
    def par_date_de_naissance2(cls, nom, date):
        return cls(nom, 2018-date)

class Homme(Personne):
    sexe = 'homme'
```

```
>>> homme1 = Homme.par_date_de_naissance('Jean', 1997)
>>> homme2 = Homme.par_date_de_naissance2('Jean', 1997)
>>> type(homme1)
<class '__main__.Personne'>
>>> type(homme2)
<class '__main__.Homme'>
```

Pour avoir `homme1` de type `Homme`, il faut redéfinir la méthode statique dans la classe fille.

Documentation Méthode statique sur Programiz, Méthode de classe sur Programiz, StackOverflow

3.5 Propriétés

Les propriétés représentent en Python le principe d'encapsulation. Elles sont utiles si on souhaite contrôler l'accès à un attribut ou si on veut que le changement d'une valeur d'un attribut engendre des modifications sur d'autres attributs. Les propriétés sont un cas particulier des descripteurs.

3.5.1 Définition d'une propriété

On crée les propriétés en utilisant des décorateurs. Elles contiennent un accesseur, un mutateur, un destructeur et une aide (docstring de l'accesseur).

Les propriétés sont aussi un moyen de simuler des attributs privés : pour simuler un attribut privé, on précède son nom d'un souligné. Ainsi, on appelle cet attribut sans le souligné dans le code gâce aux propriétés. Par convention, on n'agit pas sur les attributs qui commencent par un souligné en Python.

Exemple

```
class MaClasse:
    def __init__(self):
        self._attribut = 'Je suis un attribut'

    @property
    def attribut(self):
        """Propriété 'attribut'."""
        print("Accès à l'attribut")
        return self._attribut

    @attribut.setter
    def attribut(self, valeur):
        print("Modification de l'attribut")
        self._attribut = valeur

    @attribut.deleter
    def attribut(self):
        print('Adieu :(')
        del self._attribut
```

On utilise la propriété de la manière suivante :

```
>>> instance = MaClasse()
>>> instance.attribut
Accès à l'attribut
'Je suis un attribut'
>>> instance.attribut = 'Ah bon ?'
Modification de l'attribut
>>> del instance.attribut
Adieu :(
>>> help(MaClasse.attribut)
Help on property:

    Propriété 'attribut'.
```

Documentation Documentation Python 3, Priorités entre propriété et méthodes spéciales

3.5.2 Généralisation : les descripteurs

On dit qu'un objet est un descripteur s'il possède au moins une méthode `__get__` (accesseur), `__set__` (mutateur), ou `__delete__` (destructeur).

Exemple Exemple d'implémentation

```
class Attribut:
    def __get__(self, inst, insttype):
        print("Accès à l'attribut")
        return inst._attribut

    def __set__(self, inst, valeur):
        print("Modification de l'attribut")
        inst._attribut = valeur

class MaClasse:
    def __init__(self):
        self._attribut = 'Je suis un attribut'

    attribut = Attribut()
```

Documentation Documentation Python 3

4 Itérateurs

5 Générateurs

6 Décorateurs

Les décorateurs sont des fonctions ou des classes qui permettent de modifier le comportement d'une autre fonction (ou classe). Les décorateurs sont utiles lorsque l'on souhaite qu'un certain nombre de fonctions effectuent des tâches communes comme par exemple donner leur temps d'exécution. On appelle un décorateur de la manière suivante.

```
@decorateur
def fonction():
    pass
```

Le code précédent a le même comportement que le code suivant.

```
def fonction():
    pass

fonction = decorateur(fonction)
```

Ainsi, `fonction` devient l'objet retournée par `decorateur(fonction)`. Le décorateur doit donc retourner un objet que l'on peut appeler en écrivant `objet()` (avec d'éventuels arguments), on appelle ce type d'objet un « callable ». Le décorateur est bien sûr lui même un callable. Si on le définit comme une classe, on doit définir la méthode `__call__` qui permet de rendre ses instances callable.

Documentation Stack Overflow

6.1 En tant que classe

Une façon d'implémenter un décorateur est d'utiliser les classes. La fonction décorée deviendra alors une instance de la classe de ce décorateur. Il faut obligatoirement définir la méthode `__call__` pour pouvoir rendre cette instance callable.

Exemple On considère ici un décorateur qui compte le nombre d'appels de la fonction décorée.

```
class Compteur:
    def __init__(self, f):
        self.call = 0
        self.f = f

    def __call__(self, *args, **kwargs):
        self.call += 1
        print("La fonction {} a été appelée {} fois.".format(self.f.__name__, self.call))
        return self.f(*args, **kwargs)
```

6.2 En tant que fonction

Comme un décorateur est un objet callable qui n'a d'autre utilité que d'être appelé, il est aussi logique de le définir en tant que fonction.

Exemple Même décorateur que précédemment mais en l'implémentant en tant que fonction.

```
def compteur(f):
    def wrapper(*args, **kwargs):
        wrapper.call += 1
        print("La fonction {} a été appelée {} fois.".format(f.__name__, wrapper.call))
        return f(*args, **kwargs)
    wrapper.call = 0
    return wrapper
```

Remarques On voit dans cet exemple que l'on peut définir des fonctions dans les définitions de fonctions. La mention `*args` fait référence à tous les arguments non nommés que l'on a entrés (c'est un tuple, par exemple `(arg1, arg2)`). La mention `**kwargs` fait référence aux arguments nommés (c'est un dictionnaire). Ainsi on est sûr de récupérer tous les arguments.

Dans cet exemple, on assigne à `wrapper` un attribut de fonction (on peut le faire, puisqu'une fonction est un objet – de la classe `function`). On le définit après avoir défini cette fonction.

6.3 Décorateurs à paramètres

On peut faire en sorte que le décorateur prenne un ou plusieurs paramètres. Dans ce cas, il faut définir le décorateur à l'intérieur d'une clôture qui prend en argument ces différents paramètres.

Exemple On veut retourner une erreur quand la fonction retourne une valeur trop élevée.

```
def depasse_max(max):
    def deco(f):
        def wrapper(*args, **kwargs):
            n = f(*args, **kwargs)
            if n > max:
                print("Maximum {} dépassé.".format(max))
                return
            return n
        return wrapper
    return deco
```

Ces deux syntaxes sont équivalentes :

```
@depasse_max(10)
def demande_nombre():
    n = int(input("Entrer un nombre : "))
    return n

def demande_nombre():
    n = int(input("Entrer un nombre : "))
    return n

demande_nombre = depasse_max(10)(demande_nombre)
```

Cela permet de faire

```
>>> demande_nombre()
Entrer un nombre : 11
Maximum 10 dépassé.
```

7 Métaclasses

Les métaclasses sont les classes qui instancient d'autres classes. Par défaut, une seule métaclasse est définie : la métaclasse `type`. On s'en rend compte en demandant le type des classes que l'on crée.

```
class MaClasse:
    pass

print(type(MaClasse)) # <class 'type'>
```

Deuxième partie
Modules

1 Librairie standard

1.1 re

Le module `re` permet d'utiliser les expressions régulières en Python.

Documentation Documentation Python 3

1.1.1 Ecrire une expression régulière

Les expressions régulières sont un excellent moyen de retrouver des motifs complexes dans une chaine de caractères. On écrit les motifs à rechercher grâce à plusieurs caractères spéciaux :

Spécification du caractère

- « `.` » désigne n'importe quel caractère.
- « `[]` » permet de dire quels caractères on veut trouver (`[a-e]` : `a`, `b`, `c`, `d` ou `e` ; `[a-eA-E]` idem avec les majuscule comprises ; `[+-*]` : soit `*` soit `+` soit `-`).
- « `\w` » équivaut à `[a-zA-Z0-9_]`.
- « `\W` » désigne tout caractère non alpha-numérique.
- « `\d` » équivaut à `[0-9]`.
- « `\D` » désigne tout caractère non numérique.
- « `\s` » désigne un espace.

Place du motif dans la chaine

- « `^` » (se place au début) signifie que le début de la chaine doit correspondre au motif.
- « `$` » (se place à la fin) signifie que la fin de la chaine doit correspondre au motif.

Nombre d'apparition(s) consécutive(s)

- « `{n}` » indique que le caractère précédent doit apparaitre `n` fois.
- « `{n,m}` » indique que le caractère précédent doit apparaitre entre `n` et `m` fois.
- « `*` » indique que le caractère précédent n'apparait pas ou apparait sans maximum d'occurences (`ab*` correspond à `a`, `ab`, ou bien `abbbbbb`, etc.).
- « `+` » indique que le caractère précédent apparait au moins une fois (`ab+` correspond à `ab`, `abb`, ou bien `abbbbbb`, etc.).
- « `?` » indique que le caractère précédent apparait au plus une fois (équivalent à `{0,1}`).

Les quatre derniers qualificateurs sont dits gourmands : ils valident autant de caractères que possible. Par exemple pour `"aaaaa"`, `a{3,5}` validera la chaine en entier. Pour une version non gourmande, on suit le qualificateur d'un `?` : `*?`, `+?`, `??` et `{n,m}?`. Un qualificateur non gourmand valide le moins de caractères possibles.

Pour contrôler le nombre d'apparitions d'un groupe de caractères, on met ceux-ci entre parenthèses (`(abc)+` : `abc`, `abcabc`, etc.). Cela crée un groupe de caractères, on peut le nommer en suivant la parenthèses ouvrante de `?P<nom>`. Cela est utile par exemple quand on veut remplacer des caractères. On peut séparer des expression régulières par un `|` afin d'indiquer que plusieurs possibilités sont possibles.

1.1.2 Méthodes

re.compile

On compile une expression régulière en utilisant la fonction `compile`. Cette fonction retourne un objet expression régulière (regex) sur lequel on peut évaluer diverses méthodes. Si l'on cherche une phrase, la syntaxe sera :

```
import re

regex = re.compile(r"[A-Z]\w*\s?(\w+\s?)*.")
```

Remarque On utilise le préfixe `r` devant la chaine de caractère pour éviter d'avoir à écrire `\\` au lieu d'un unique `\`.

regex.finditer

On peut rechercher toutes les occurences du motif grâce à la méthode `re.finditer(motif, chaine)`. Cela retourne un objet iterable. On accède aux objets en appelant `next(iterable)`, qui retourne un objet expression rationnelle. Celui-ci contient plusieurs chaines de caractères (une pour chaque groupe du motif), on y accède en appelant les différents groupes : `objet.group(numéro ou nom)`.

Exemple On veut extraire les phrases d'une chaîne de caractères.

Script

```
chaine = r"Je suis une phrase. Moi aussi"
regex = re.compile(r"[A-Z]\w*\s?(?:\w+\s?)*.")
resultats = regex.finditer(chaine)
while True:
    try:
        print(next(resultats).group(0))
    except:
        break
```

Sortie

```
Je suis une phrase.
Moi aussi.
```

regex.sub

On peut remplacer les motifs par d'autres motifs en utilisant la méthode `re.sub`. Elle prend en paramètres :

1. le motif (chaîne de caractères ou objet expression rationnelle.)
2. le remplacement (peut être une fonction)
3. la chaine à traiter
4. `count`=le nombre d'occurences à remplacer

et renvoie la chaine de caractères modifiée. Lorsque l'on veut appeler un groupe de caractères nommé avec `(?P<nom>)`, on y fait référence dans la chaine de remplacement par `\g<nom>`.

Exemple

```
pass
```

1.2 datetime

`datetime`

Le module `datetime` permet de créer des objets représentant des dates et de faire des opérations. La classe `datetime.date` représente une date par son année, son mois et son jour : `jour = datetime.date(2017, 1, 1)` correspond à la date 1er janvier 2017. La classe `datetime.timedelta` permet de faire des opérations sur les dates. Ses objets sont représentés par un nombre de jours (on peut construire un `timedelta` avec des semaines/mois/années, le constructeur convertit en jours). Le module `datetime` peut ausi être utilisé pour utiliser des durées plus réduites, i.e. secondes, minutes, heures, etc.

Exemple

```
>>> import datetime
>>> j1 = datetime.date(2017, 1, 1)
>>> j2 = j1 + datetime.timedelta(30)
>>> j2
datetime.date(2017, 1, 31)
```

Documentation Documentation Python 3

1.3 turtle

Contient des classes pour dessiner des formes simples en faisant avancer des tortues. Elles peuvent avancer, reculer, tourner d'un certain angle. La classe `Turtle` permet de créer des objets tortues qui peuvent :

1. Avancer : `Turtle.forward(<nb de pixels>)`
2. Reculer : `Turtle.backward(<nb de pixels>)`
3. Tourner à droite ou à gauche (ex : `Turtle.right(<degrés>)`)
4. Changer de couleur (`Turtle.color(<couleur>)`) ou de forme (`Turtle.shape(<forme>)`).

Exemple

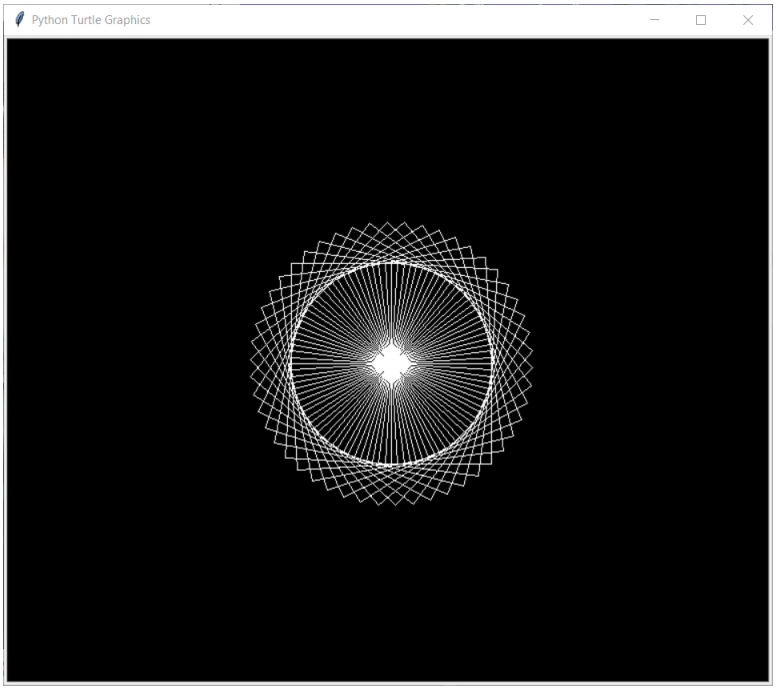
```
import turtle

Terrain = turtle.Screen()
Terrain.bgcolor("black")

Tortue = turtle.Turtle()
Tortue.speed(3)
Tortue.shape("turtle")
Tortue.color("white")

for i in range(50):
    for e in range(4):
        Tortue.forward(100)
        Tortue.right(90)
    Tortue.right(360/50)

Terrain.exitonclick()
```



Résultat

Documentation Documentation Python 3, Wikilivres

1.4 ctypes

Ce module sert à appeler des fonctions écrites en langage C dans des librairies DLL par exemple.

1.4.1 Boites de dialogue

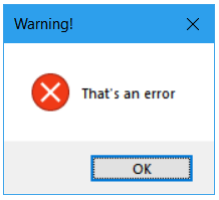
Le module ctypes peut servir à faire apparaitre des boites de dialogue. On peut modifier le comportement du script Python en fonction du bouton appuyé car la fonction faisant apparaitre ces boites renvoie un entier qui dépend du bouton appuyé. Diverses options sont disponibles :

```
# Button styles:
# 0 : OK
# 1 : OK | Annuler
# 2 : Abandonner | Recommencer | Ignorer
# 3 : Oui | Non | Annuler
# 4 : Oui | Non
# 5 : Recommencer | Annuler
# 6 : Annuler | Recommencer | Continuer

# To also change icon, add these values to previous number
# 16 Icone erreur
# 32 Icone question
# 48 Icone attention
# 64 Icone information
```

Exemple

```
ctypes.windll.user32.MessageBoxW(0, "That's an error", "Warning!", 16)
```



Résultat

1.5 keyboard

1.6 os

1.7 sys

2 Modules à télécharger

2.1 twilio

2.2 win10toast

2.3 splinter

2.4 pylint

2.5 autopy

Index