Notes d'apprentissage du tutoriel Python

À travers les apprentissages précédents, nous avons déjà acquis une certaine compréhension de Python. Maintenant, en nous basant sur la documentation officielle, nous allons continuer à approfondir nos connaissances sur Python.

Contrôle du flux de code

type

```
print(type(1))
```

```
<class 'int'>
```

```
print(type('a'))
```

```
<class 'str'>
```

La fonction `type` est très utile pour afficher le type d'un objet.

range

La fonction `range` en Python est utilisée pour générer une séquence de nombres. Elle est souvent utilisée dans les boucles `for` pour itérer sur une séquence de nombres. Voici quelques exemples d'utilisation de `range` :

```
# Génère une séquence de 0 à 4
for i in range(5):
    print(i)

# Génère une séquence de 2 à 5
for i in range(2, 6):
    print(i)

# Génère une séquence de 1 à 10 avec un pas de 2
for i in range(1, 11, 2):
    print(i)
```

La fonction `range` peut prendre jusqu'à trois arguments :

1. **start** : La valeur de départ de la séquence (inclus). Par défaut, c'est 0.
2. **stop** : La valeur de fin de la séquence (exclus). Ce paramètre est obligatoire.
3. **step** : Le pas entre chaque nombre de la séquence. Par défaut, c'est 1.

Il est important de noter que `range` ne génère pas une liste, mais un objet de type `range`, qui est un itérable. Cela signifie qu'il génère les nombres à la demande, ce qui est plus efficace en termes de mémoire, surtout pour des séquences de grande taille.

Pour convertir un objet `range` en liste, vous pouvez utiliser la fonction `list` :

```
# Convertit un range en liste
numbers = list(range(5))
print(numbers)  # Affiche [0, 1, 2, 3, 4]
```

En résumé, `range` est un outil puissant pour générer des séquences de nombres de manière efficace, surtout dans des contextes où vous avez besoin d'itérer sur une plage de valeurs.

La fonction `range` est extrêmement utile.

```
for i in range(5):
  print(i, end = ' ')
```

```
0 1 2 3 4
```

```
for i in range(2, 6, 2):
  print(i, end = ' ')
```

```
2 4
```

Regardez la définition de la fonction `range`.

```
class range(Sequence[int]):
    start: int
    stop: int
    step: int
```

Visible est une classe.

```
print(range(5))
```

```
range(0, 5)
```

Au lieu de :

```
[0,1,2,3,4]
```

Continuez.

```
print(list(range(5)))
```

```
[0, 1, 2, 3, 4]
```

Pourquoi. Regardez la définition de `list`.

```
class list(MutableSequence[_T], Generic[_T]):
```

La traduction en français de cette ligne de code n'est pas nécessaire, car il s'agit d'une définition de classe en Python qui utilise des concepts génériques et des annotations de type. Les noms de classes et de modules comme `list`, `MutableSequence`, et `Generic` sont des éléments spécifiques au langage Python et ne sont pas traduits.

La définition de `list` est `list(MutableSequence[_T], Generic[_T]):`. Et la définition de `range` est `class range(Sequence[int])`. `list` hérite de `MutableSequence`, tandis que `range` hérite de `Sequence`.

En continuant à chercher plus bas, voici ce que l'on trouve.

```
Sequence = _alias(collections.abc.Sequence, 1)
MutableSequence = _alias(collections.abc.MutableSequence, 1)
```

Ici, nous ne comprenons pas la relation entre les deux. Mais nous avons probablement compris pourquoi nous pouvons écrire `list(range(5))`.

Paramètres de fonction

Voici des connaissances supplémentaires sur les fonctions.

```
def fn(a = 3):
  print(a)

fn()

```shell
3
```

Cela permet de donner une valeur par défaut à un paramètre.

```
def fn(end: int, start = 1):
  i = start
  s = 0
  while i < end:
    s += i
    i += 1
  return s
```

Le code reste en anglais, car il s'agit d'un langage de programmation. La fonction `fn` prend un paramètre obligatoire `end` (de type `int`) et un paramètre optionnel `start` (par défaut à 1). Elle initialise une variable `i` à la valeur de `start` et une variable `s` à 0. Ensuite, elle entre dans une boucle `while` qui continue tant que `i` est inférieur à `end`. À chaque itération, elle ajoute la valeur de `i` à `s` et incrémente `i` de 1. Enfin, la fonction retourne la valeur de `s`.

```
print(fn(10))
```

```
45
```

`end` est un paramètre obligatoire. Notez qu'il est important de placer les paramètres obligatoires en premier.

```
def fn(start = 1, end: int):

    def fn(start = 1, end: int):
                                ^
SyntaxError: un argument non par défaut suit un argument par défaut
```
```

Notez que `end` est un `non-default argument`. `start` est un `default argument`. Cela signifie qu'un argument non par défaut suit un argument par défaut. En d'autres termes, les arguments non par défaut doivent être placés avant tous les arguments par défaut. `start` étant un argument par défaut, cela signifie que si aucune valeur n'est fournie, il prendra automatiquement une valeur par défaut.

```
def fn(a, /, b):
  print(a + b)
```

fn(1, 3)

```
Ici, le `/` est utilisé pour séparer les types de paramètres. Il existe deux formes de passage de parame

```python
def fn(a, /, b):
  print(a + b)

fn(a=1, 3)
```

En Python, la fonction `fn` est appelée avec deux arguments : `a=1` et `3`. Cependant, cette syntaxe est incorrecte car les arguments positionnels (comme `3`) doivent être placés avant les arguments nommés (comme `a=1`). Voici la correction :

```
fn(3, a=1)
```

Cela appelle la fonction `fn` avec `3` comme premier argument positionnel et `a=1` comme argument nommé.

```
    fn(a=1, 3)
             ^
SyntaxError: argument positionnel suit un argument mot-clé
```

Cela ne fonctionnera pas ainsi. `a=1` signifie que le paramètre est passé par mot-clé. Cela le traite comme un argument nommé. Alors que `b` est un argument positionnel.

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      ----------    ----------     ----------
        |             |                  |
```
```

```
        |        Positionnel ou mot-clé    |
        |                                  - Mot-clé uniquement
         -- Positionnel uniquement
```

Notez qu'en définissant la fonction ici, l'utilisation de `/` et `*` implique déjà le type de passage des paramètres. Par conséquent, il faut respecter les règles de passage.

```
def fn(a, /, b):
  print(a + b)
```

En Python, la fonction `fn` est définie avec deux paramètres : `a` et `b`. Le symbole `/` dans la signature de la fonction indique que le paramètre `a` est un paramètre positionnel uniquement, ce qui signifie qu'il ne peut pas être passé en utilisant un argument nommé. Le paramètre `b`, quant à lui, peut être passé soit par position, soit par nom. La fonction affiche ensuite la somme de `a` et `b`.

```
fn(1, b=3)
```

Aucune erreur n'a été signalée avec ce qui précède.

```
def fn(a, /, b, *, c):
  print(a + b + c)
```

fn(1, 3, 4)

```shell
    fn(1, 3, 4)
TypeError: fn() prend 2 arguments positionnels mais 3 ont été donnés
```

`fn` ne peut accepter que 2 arguments positionnels, mais 3 ont été fournis.

```
def fn(a, /, b, *, c):
  print(a + b + c)
```

La fonction `fn` est définie avec trois paramètres : `a`, `b`, et `c`. La syntaxe utilisée ici introduit des spécificités dans la manière dont ces paramètres peuvent être passés :

- `a` est un paramètre positionnel uniquement, indiqué par le `/`. Cela signifie que `a` ne peut être passé qu'en tant qu'argument positionnel, et non en tant qu'argument nommé.

- `b` est un paramètre qui peut être passé soit positionnellement, soit en tant qu'argument nommé.
- `c` est un paramètre nommé uniquement, indiqué par le `*`. Cela signifie que `c` doit être passé en tant qu'argument nommé.

La fonction additionne les trois paramètres et affiche le résultat.

```
fn(a = 1, b=3, c=4)
```

```
    fn(a = 1, b=3, c=4)
TypeError: fn() a reçu des arguments positionnels passés comme arguments nommés : 'a'
```

`fn` a maintenant des paramètres qui ne peuvent être passés que par position, mais qui sont maintenant passés par mot-clé.

Paramètres sous forme de mapping

```
def fn(**kwds):
  print(kwds)
```

fn(**{'a': 1})

```shell
{'a': 1}
```

```
def fn(**kwds):
  print(kwds['a'])

d = {'a': 1}
fn(**d)
```

```
1
```

On peut voir que `**` sert à déballer les paramètres.

```
def fn(a, **kwds):
  print(kwds['a'])
```

```
d = {'a': 1}
fn(1, **d)
```

```
TypeError: fn() a reçu plusieurs valeurs pour l'argument 'a'
```

Lorsque vous appelez une fonction comme `fn(1, **d)`, cela se déploie en `fn(a=1, a=1)`. Cela entraînera donc une erreur.

```
def fn(**kwds):
  print(kwds['a'])

d = {'a': 1}
fn(d)
```

```
TypeError: fn() prend 0 arguments positionnels mais 1 a été donné
```

Si vous appelez une fonction comme `fn(d)`, elle sera traitée comme un argument positionnel, et non comme un argument mot-clé déployé.

```
def fn(a, / , **kwds):
  print(kwds['a'])
```

La fonction `fn` en Python prend un argument positionnel `a` et un ensemble d'arguments nommés `**kwds`. Le paramètre `a` est spécifié comme étant uniquement positionnel grâce à la barre oblique `/`, ce qui signifie qu'il ne peut pas être passé comme un argument nommé. Ensuite, la fonction imprime la valeur associée à la clé `'a'` dans le dictionnaire `kwds`.

```
d = {'a': 1}
fn(1, **d)
```

Cela fonctionne ainsi. Cela montre que les paramètres de position et les paramètres sous forme de mappage peuvent porter le même nom.

```
def fn(a, / , a):
  print(a)
```

En Python, la fonction ci-dessus est incorrecte car elle tente de définir deux paramètres avec le même nom `a`. Cela entraînera une erreur de syntaxe. Voici une version corrigée avec des noms de paramètres différents :

```
def fn(a, / , b):
    print(a, b)
```

Dans cette version, `a` est un paramètre positionnel uniquement (indiqué par `/`), et `b` est un paramètre nommé ou positionnel.

```
d = {'a': 1}
fn(1, **d)
```

```
SyntaxError: argument 'a' en double dans la définition de la fonction
```

Cela entraînera une erreur. Faites attention aux relations subtiles entre ces différentes situations.

```
def fn(a, / , **kwds):
  print(kwds['a'])
```

fn(1, **[1,2])

```shell
TypeError: __main__.fn() l'argument après ** doit être un mapping, pas une liste
```

** doit être suivi d'une carte.

Paramètres de type itérable

```
def fn(*kwds):
  print(kwds)
```

fn(*[1,2])

```shell
(1, 2)
```

```
def fn(*kwds):
  print(kwds)
```

fn(*1)

```shell
TypeError: __main__.fn() l'argument après * doit être un itérable, pas un int
```

`*` doit être suivi d'un `iterable`.

```
def fn(a, *kwds):
  print(type(kwds))
```

fn(1, *[1])

```shell
<class 'tuple'>
```

Imprimez le type. C'est aussi pourquoi la sortie ci-dessus est `(1,2)` et non `[1,2]`.

```
def fn(*kwds):
  print(kwds)
```

fn(1, *[1])

```shell
(1, 1)
```

Remarquez que lors de l'appel de `fn(1, *[1])`, les paramètres sont déployés, devenant `fn(1,1)`. Ensuite, lors de l'analyse de `fn(*kwds)`, `kwds` transforme `1,1` en un tuple `(1,1)`.

```
def concat(*args, sep='/'):
  return sep.join(args)

print(concat('a','b','c', sep=','))
```

```
a,b,c
```

Expressions Lambda

`lambda` permet de stocker une fonction comme une variable. Vous souvenez-vous de ce qui a été dit dans l'article « Démystifier l'informatique » ?

```
def incrementor(n):
  return lambda x: x + n

f = incrementor(2)
print(f(3))
```

```
5
```

Prenons un autre exemple.

```
pairs = [(1, 4), (2, 1), (0, 3)]
```

pairs.sort(key = lambda pair: pair[1])

```
print(pairs)
```

```
[(2, 1), (0, 3), (1, 4)]
```

```
pairs = [(1, 4), (2, 1), (0, 3)]

pairs.sort(key = lambda pair: pair[0])

print(pairs)
```

```
[(0, 3), (1, 4), (2, 1)]
```

`pair[0]` trie selon le premier nombre. `pair[1]` trie selon le deuxième nombre.

Commentaires de documentation

```
def add():
  """ajouter quelque chose
  """
  pass

print(add.__doc__)
```

```
ajouter quelque chose
```

Signature de fonction

```
def add(a:int, b:int) -> int:
  print(add.__annotations__)
  return a+b
```

La fonction `add` prend deux arguments `a` et `b`, tous deux annotés comme étant de type `int`, et retourne leur somme. La ligne `print(add.__annotations__)` affiche les annotations de la fonction, qui dans ce cas seront `{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}`.

```
add(1, 2)
```

```
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Structures de données

Listes

```
a = [1,2,3,4]
```

a.append(5) print(a) # [1, 2, 3, 4, 5]

a[len(a):] = [6] print(a) # [1, 2, 3, 4, 5, 6]

a[3:] = [6] print(a) # [1, 2, 3, 6]

a.insert(0, -1) print(a) # [-1, 1, 2, 3, 6]

a.remove(1) print(a) # [-1, 2, 3, 6]

a.pop() print(a) # [-1, 2, 3]

```
a.clear()
print(a)  # []
```

a[:] = [1, 2] print(a.count(1)) # 1

a.reverse() print(a) # [2, 1]

```
b = a.copy()
a[0] = 10
print(b)   # [2, 1]
print(a)   # [10, 1]
```

b = a a[0] = 3 print(b) # [3, 1] print(a) # [3, 1]

Construction de listes

```python
print(3 ** 2)   # 9
print(3 ** 3)   # 27
```

Commençons par apprendre une opération, `**`. Cela signifie `puissance`.

```
sq = []
for x in range(10):
  sq.append(x ** 2)

print(sq)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Ensuite, essayons d'utiliser `map`.

```
a = map(lambda x:x, range(10))
print(a)
# <map object at 0x103bb0550>
print(list(a))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

sq = map(lambda x: x ** 2, range(10))
print(list(sq))
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

sq = [x ** 2 for x in range(10)]
print(sq)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On peut voir que `for` est très flexible.

```
a = [i for i in range(5)]
print(a)
# [0, 1, 2, 3, 4]
```

a = [i+j for i in range(3) for j in range(3)] print(a) # [0, 1, 2, 1, 2, 3, 2, 3, 4]

a = [i for i in range(5) if i % 2 == 0] print(a) # [0, 2, 4]

```
a = [(i,i) for i in range(3)]
print(a)
# [(0, 0), (1, 1), (2, 2)]
```

Construction de listes imbriquées

```
matrix = [[(i+j*4) for i in range(4)] for j in range(3)]
print(matrix)
# [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

t = [] for j in range(3): t.append([(i+j*4) for i in range(4)]) print(t) # [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]

Faites attention à la manière dont ces deux segments de code sont écrits. C'est-à-dire :

```python
[[(i+j*4) for i in range(4)] for j in range(3)]
```

Ce qui équivaut à :

```
for j in range(3):
   [(i+j*4) for i in range(4)]
```

Ce qui équivaut à :

```
for j in range(3):
  for i in range(4):
      (i+j*4)
```

Cela facilite donc la transposition de matrices.

```
matrix = [[(i+j*4) for i in range(4)] for j in range(3)]
print(matrix)
#  [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]

mt = [[row[j] for row in matrix] for j in range(4)]
print(mt)
# [[0, 4, 8], [1, 5, 9], [2, 6, 10], [3, 7, 11]]

print(list(zip(*matrix)))
[(0, 4, 8), (1, 5, 9), (2, 6, 10), (3, 7, 11)]
```

del

La commande `del` en Python est utilisée pour supprimer un objet. Cela peut être une variable, un élément d'une liste, une clé d'un dictionnaire, etc. Voici quelques exemples d'utilisation de `del` :

1. **Supprimer une variable :**

   ```
   x = 10
   del x
   # Tentative d'accès à x après suppression générera une erreur
   ```

2. **Supprimer un élément d'une liste :**

   ```
   ma_liste = [1, 2, 3, 4, 5]
   del ma_liste[2]  # Supprime l'élément à l'index 2 (valeur 3)
   print(ma_liste)  # Affiche [1, 2, 4, 5]
   ```

3. **Supprimer une clé d'un dictionnaire :**

   ```
   mon_dict = {'a': 1, 'b': 2, 'c': 3}
   del mon_dict['b']  # Supprime la clé 'b' et sa valeur associée
   print(mon_dict)  # Affiche {'a': 1, 'c': 3}
   ```

4. **Supprimer un slice d'une liste :**

```
ma_liste = [1, 2, 3, 4, 5]
del ma_liste[1:3]  # Supprime les éléments de l'index 1 à 2 (valeur 2 et 3)
print(ma_liste)  # Affiche [1, 4, 5]
```

La commande `del` est utile pour libérer de la mémoire en supprimant des objets qui ne sont plus nécessaires, mais il faut l'utiliser avec précaution pour éviter des erreurs dues à des tentatives d'accès à des objets supprimés.

```
a = [1, 2, 3, 4]

del a[1]
print(a)  # [1, 3, 4]

del a[0:2]
print(a) # [4]

del a
print(a) # NameError: name 'a' is not defined
```

Dictionnaire

```
ages = {'li': 19, 'wang': 28, 'he' : 7}
for name, age in ages.items():
    print(name, age)

# li 19
# wang 28
# he 7

for name in ages:
    print(name)

# li
# wang
# he

for name, age in ages:
     print(name)
```

ValueError : trop de valeurs à décompresser (2 attendues)

```
for i, name in enumerate(['li', 'wang', 'he']):
    print(i, name)
```

0 li

1 wang

2 he

```
print(reversed([1, 2, 3]))
# <list_reverseiterator object at 0x10701ffd0>

print(list(reversed([1, 2, 3])))
# [3, 2, 1]

### Modules

### Appel de modules via un script

```python
import sys

def f(n):
    if n < 2:
        return n
    else:
        return f(n-1) + f(n-2)

if __name__ == "__main__":
    r = f(int(sys.argv[1]))
    print(r)

% python fib.py 3
2
```
```

```
% python -m fib 5
5
```

dir

```
import fib

print(dir(fib))
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__
```

```
import builtins
print(dir(builtins))
```

Note : Le code reste en anglais, car il s'agit d'une commande Python standard. La traduction n'est pas nécessaire pour les noms de fonctions ou de modules en programmation.

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '**build_class**', '**debug**', '**doc**', '**import**', '**loader**', '**name**', '**package**', '**spec**', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

Package

Les paquets, c'est-à-dire les `packages`.

```shell
pk.py
fibp
   cal
      cal.py
   pt
       pt.py
```

cal.py :

```
def f(n):
    if n < 2:
        return n
    else:
        return f(n-1) + f(n-2)

def fl(n):
    return list(map(f, range(5)))
```

pt.py:

```
def p(l):
    print(l, end=' ')

def pln(l):
    print(l)
```

pk.py :

```
import fibp.cal.cal
import fibp.pt.pt
```

fibp.pt.pt.p(fibp.cal.cal.fl(10))

`pk.py` peut également être écrit de cette manière :

```python
from fibp.cal import cal
from fibp.pt import pt
```

pt.p(cal.fl(10)) “‘