Python et POO

Sam van Gool (vangool@irif.fr)

Préparation à l'agrégation d'informatique

October 8, 2024

Aujourd'hui : quelques détails en suspens

- Portée et espaces de nommage

Aujourd'hui : quelques détails en suspens

- Portée et espaces de nommage
- Iterateurs et générateurs

Aujourd'hui : quelques détails en suspens

- Portée et espaces de nommage
- Iterateurs et générateurs
- Typage et POO, Abstract base classes

Plan

Espaces de nommage

- Un *espace de nommage* est une table de correspondance entre des noms et des objets.

Espaces de nommage

- Un *espace de nommage* est une table de correspondance entre des noms et des objets.
- La *portée* d'un nom est la zone textuelle d'un programme Python où un espace de nommage est directement accessible (i.e. `var` et non pas `modulename.var`).

Espaces de nommage

- Un *espace de nommage* est une table de correspondance entre des noms et des objets.
- La *portée* d'un nom est la zone textuelle d'un programme Python où un espace de nommage est directement accessible (i.e. `var` et non pas `modulename.var`).
- Les portées sont déterminées de manière statique, mais utilisées de manière dynamique.

Espaces de nommage imbriqués

- L'ordre dans lequel l'interprète recherche les espaces de nommage :

Espaces de nommage imbriqués

- L'ordre dans lequel l'interprète recherche les espaces de nommage :
  - Les noms locaux à une fonction ;

Espaces de nommage imbriqués

- L'ordre dans lequel l'interprète recherche les espaces de nommage :
  - Les noms locaux à une fonction ;
  - Les noms dans des fonctions englobantes ;

Espaces de nommage imbriqués

- L'ordre dans lequel l'interprète recherche les espaces de nommage :
  - Les noms locaux à une fonction ;
  - Les noms dans des fonctions englobantes ;
  - Les noms globaux du module courant ;

Espaces de nommage imbriqués

- L'ordre dans lequel l'interprète recherche les espaces de nommage :
  - Les noms locaux à une fonction ;
  - Les noms dans des fonctions englobantes ;
  - Les noms globaux du module courant ;
  - La portée englobante : espace de nommage contenant les primitives (built-ins).

Exemple

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
```

py/scopes.py

Portée en cas de conflit de nom

Expliquer la différence entre

```
b = 1
def f():
    print(b)
f()
```

py/scopeconflict.py

et

```
b = 1
def f():
    print(b)
    b = 2
f()
```

py/scopeconflict2.py

Plan

Itérateurs

- Que fait l'interprète avec le code `for x in [1,2,3]`?

Itérateurs

- Que fait l'interprète avec le code `for x in [1,2,3]`?
- L'instruction `for` appelle la fonction `iter()` sur l'objet *conteneur*.

Itérateurs

- Que fait l'interprète avec le code `for x in [1,2,3]`?
- L'instruction `for` appelle la fonction `iter()` sur l'objet *conteneur*.
- La fonction `iter()` renvoie un objet *itérateur* qui définit la méthode `__next__()`.

Itérateurs

- Que fait l'interprète avec le code `for x in [1,2,3]`?
- L'instruction `for` appelle la fonction `iter()` sur l'objet *conteneur*.
- La fonction `iter()` renvoie un objet *itérateur* qui définit la méthode `__next__()`.
- La méthode `__next__()` retourne un élément du conteneur à chaque appel.

Itérateurs

- Que fait l'interprète avec le code `for x in [1,2,3]`?
- L'instruction `for` appelle la fonction `iter()` sur l'objet *conteneur*.
- La fonction `iter()` renvoie un objet *itérateur* qui définit la méthode `__next__()`.
- La méthode `__next__()` retourne un élément du conteneur à chaque appel.
- Lorsqu'il n'y a plus d'élément, `__next__()` lève une exception `StopIteration` qui indique à la boucle de l'instruction for de se terminer.

Itérateurs

- Que fait l'interprète avec le code `for x in [1,2,3]`?
- L'instruction `for` appelle la fonction `iter()` sur l'objet *conteneur*.
- La fonction `iter()` renvoie un objet *itérateur* qui définit la méthode `__next__()`.
- La méthode `__next__()` retourne un élément du conteneur à chaque appel.
- Lorsqu'il n'y a plus d'élément, `__next__()` lève une exception `StopIteration` qui indique à la boucle de l'instruction for de se terminer.
- On peut appeler `next()` directement sur un objet.

Exemple: iter et next

```python
i = iter([1,2,3,4,5,6])
print(f"first element: {next(i)}")
for x in i:
    print(f"element consumed by for: {x}")
    try:
        y = next(i)
        print(f"consuming one more element: {y}")
    except StopIteration:
        print("iterator exhausted")
```

py/iternext.py

Créer son propre itérateur

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

py/reverseit.py

Générateurs

- Une syntaxe plus simple pour déclarer les itérateurs.

Générateurs

- Une syntaxe plus simple pour déclarer les itérateurs.
- Un générateur est une fonction dont l'exécution est interrompue par `yield` (et non par `return`).

Générateurs

- Une syntaxe plus simple pour déclarer les itérateurs.
- Un générateur est une fonction dont l'exécution est interrompue par `yield` (et non par `return`).
- L'appel d'une fonction de générateur crée un *generator iterator* (itérateur de générateur).

Exemple de générateur

```
def reverse_except_last(word):
    for index in range(len(word)-2,-1,-1):
        yield word[index]
for char in reverse_except_last("hello"):
    print(char)
```

py/genexa.py

Enumération

```python
from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

py/enumexa.py

- Ensemble de noms symboliques (*membres*) liés à des *valeurs*,

Enumération

```python
from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

py/enumexa.py

- Ensemble de noms symboliques (*membres*) liés à des *valeurs*,
- Itérable dans l'ordre de définition,

Enumération

```python
from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

py/enumexa.py

- Ensemble de noms symboliques (*membres*) liés à des *valeurs*,
- Itérable dans l'ordre de définition,
- Appelable pour renvoyer les valeurs de ses membres,

Enumération

```python
from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

py/enumexa.py

- Ensemble de noms symboliques (*membres*) liés à des *valeurs*,
- Itérable dans l'ordre de définition,
- Appelable pour renvoyer les valeurs de ses membres,
- Indexable pour renvoyer les noms de ses membres.

Exemple

```
from enum import Enum, auto
class Color(Enum):
    RED = auto()
    GREEN = auto()
    BLUE = auto()
for c in Color:
    print(c.name)
print(Color['RED'].value)
```

py/enumexa2.py

Plan

Typage et POO

- L'un des avantages de la programmation orientée objet est qu'elle permet au programmeur de renforcer le système de types du langage avec des types personnalisés.

Typage et POO

- L'un des avantages de la programmation orientée objet est qu'elle permet au programmeur de renforcer le système de types du langage avec des types personnalisés.
- Dans Python, cette idée n'a été introduite que progressivement. Le passage d'un système de types dynamique à un système de types statique est lent.

Typage et POO

- L'un des avantages de la programmation orientée objet est qu'elle permet au programmeur de renforcer le système de types du langage avec des types personnalisés.
- Dans Python, cette idée n'a été introduite que progressivement. Le passage d'un système de types dynamique à un système de types statique est lent.
- Aujourd'hui, Python combine les deux avec un système appelé gradual typing : si une variable est de type inconnu, on y attribue le type `Any`.

Protocoles dynamiques, statiques et ABC

- Comme nous l'avons vu la semaine dernière, la documentation Python définit des protocoles *informels* ou *dynamiques*, tels que `Sequence`.

Protocoles dynamiques, statiques et ABC

- Comme nous l'avons vu la semaine dernière, la documentation Python définit des protocoles *informels* ou *dynamiques*, tels que `Sequence`.
- Depuis Python 3.8, il y a une classe `typing.Protocol`. Un protocol *statique* est une sous-classe de cette classe.

Protocoles dynamiques, statiques et ABC

- Comme nous l'avons vu la semaine dernière, la documentation Python définit des protocoles *informels* ou *dynamiques*, tels que `Sequence`.
- Depuis Python 3.8, il y a une classe `typing.Protocol`. Un protocol *statique* est une sous-classe de cette classe.
- Nous n'aborderons pas les protocols statiques ici. Référence : Protocols.

Protocoles dynamiques, statiques et ABC

- Comme nous l'avons vu la semaine dernière, la documentation Python définit des protocoles *informels* ou *dynamiques*, tels que `Sequence`.
- Depuis Python 3.8, il y a une classe `typing.Protocol`. Un protocol *statique* est une sous-classe de cette classe.
- Nous n'aborderons pas les protocols statiques ici. Référence : Protocols.
- Dans la plupart des cas, les *abstract base classes* sont suffisantes.

Duck typing?

```
class Artist:
    def draw(self): ...
class Lottery:
    def draw(self): ...
```

py/noduck.py

- Le duck typing impliquerait que `Artist` et `Lottery` soient le *même* type.

Duck typing?

```
class Artist:
    def draw(self): ...
class Lottery:
    def draw(self): ...
```

py/noduck.py

- Le duck typing impliquerait que `Artist` et `Lottery` soient le *même* type.
- Goose typing : déclarer des classes comme héritant d'un *type abstrait*.

ABC : exemple

```
from collections import abc
class Example(abc.Sized):
    def __len__(self):
        return 20
```

py/abcexa.py

- Une classe peut hériter de `abc.Sized` si elle implémente une méthode `len`.

ABC : exemple

```
from collections import abc
class Example(abc.Sized):
    def __len__(self):
        return 20
```

py/abcexa.py

- Une classe peut hériter de `abc.Sized` si elle implémente une méthode `len`.
- Cela permet d'effectuer une (certaine) vérification de type au moment de l'exécution.

ABC : exemple négatif

```
from collections import abc
class Example(abc.Sized):
    def __init__(self):
        self.x = 5
e = Example() # TypeError at runtime
```

py/abcexawrong.py

Quelques ABCs

ABC	Inherits from	Abstract Methods	Mixin Methods
Container [1]		`__contains__`	
Hashable [1]		`__hash__`	
Iterable [1] [2]		`__iter__`	
Iterator [1]	Iterable	`__next__`	`__iter__`
Reversible [1]	Iterable	`__reversed__`	
Generator [1]	Iterator	`send`, `throw`	`close`, `__iter__`, `__next__`
Sized [1]		`__len__`	
Callable [1]		`__call__`	
Collection [1]	Sized, Iterable, Container	`__contains__`, `__iter__`, `__len__`	
Sequence	Reversible, Collection	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
MutableSequence	Sequence	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited Sequence methods and `append`, `clear`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`

Observations

- L'héritage **multiple** est possible en Python :

```
class Collection(Sized, Iterable, Container):
```

Observations

- L'héritage **multiple** est possible en Python :

```
class Collection(Sized, Iterable, Container):
```

- Une classe précédemment définie peut être **enregistrée** comme héritant d'une ABC : `Container.register(MyClass)`

Observations

- L'héritage **multiple** est possible en Python :

```
class Collection(Sized, Iterable, Container):
```

- Une classe précédemment définie peut être **enregistrée** comme héritant d'une ABC : `Container.register(MyClass)`
- Les ABC sont principalement là pour être *utilisés*, et non pour être *définis* par le programmeur (contrairement à d'autres langages orientés objet).

Observations

- L'héritage **multiple** est possible en Python :

```
class Collection(Sized, Iterable, Container):
```

- Une classe précédemment définie peut être **enregistrée** comme héritant d'une ABC : `Container.register(MyClass)`
- Les ABC sont principalement là pour être *utilisés*, et non pour être *définis* par le programmeur (contrairement à d'autres langages orientés objet).
- Le module `abc` fournit une infrastructure pour définir des classes de base abstraites personnalisées, mais nous ne l'aborderons pas ici.

Register

```python
from collections.abc import Container, Sequence


class MyClass:
    def __contains__(self, x):
        return True


print(f"{issubclass(MyClass, Container)=}")
print(f"{issubclass(MyClass, Sequence)=}")
Sequence.register(MyClass)  # type:ignore
print(f"{issubclass(MyClass, Sequence)=}")
```

py/registerexa.py

- `register` est la responsabilité du programmeur, Python ne détecte pas qu'il manque des méthodes dans `MyClass` pour être une `Sequence`.