

Python Essential

Інтроекція та рефлексія

Python Essential

Після уроку обов'язково



Повторіть цей урок у форматі відео на [ITVDN.com](http://itvdn.com)

Доступ можна отримати через керівництво вашого навчального центру



Перевірте, як Ви засвоїли цей матеріал на [TestProvider.com](http://testprovider.com)

Інтроекція та рефлексія

Інтроспекція та рефлексія

Поняття інтроспекції

Інтроспекція (англ. type introspection) — можливість отримати тип та структуру об'єкта під час виконання програми. Особливе значення має у мові Objective C, проте є майже у всіх мовах, які дають змогу маніпулювати типами об'єктів як об'єктами першого класу; серед мов, що підтримують інтроспекцію: C++ (з RTTI), Go, Java, Kotlin, JavaScript, Perl, Ruby, Smalltalk; у PHP і Python інтроспекція інтегрована в саму мову. Інтроспекція може бути використана для реалізації ad-hoc-поліморфізму.

У Python інтроспекція може бути функціонально реалізована за допомогою вбудованих методів `type()` і `dir()` або вбудованого модуля `inspect`, або йти безпосередньо від імені об'єкта за допомогою вбудованих атрибутів `__class__` та `__dict__`. Користуватися інтроспекцією в Python особливо зручно завдяки парадигмі «все є об'єктом». Будь-яка сутність, будучи об'єктом, має метадані (дані про об'єкт), які називаються атрибутами, і пов'язані з цією сутністю функціональності, які називаються методами. У Python новий клас за замовчуванням є сам собою об'єктом метакласу `type`.



Інтроспекція та рефлексія

type(тип)

Важливо розуміти, що тип, як і інші сутності Python, теж є об'єктом.

Починаючи з версії Python 3.6: підкласи, які не перевизначають, `type.__new__` більше не можуть використовувати форму з одним аргументом для отримання типу об'єкта.

Найбільш важливі атрибути та функції мови для роботи з інтроспекцією наступні:

`__class__`

`__base__`

`__bases__`

`__dict__`

`dir()`

`type()`

`type(», (), {})`

`isinstance()`

`issubclass()`



Інтроекція та рефлексія

Інтроекція

У Python найпоширенішою формою інтроекції є використання функції `dir()` для виведення списку атрибутів об'єкту:

```
class MyClass(object):  
    def __init__(self, surname, name, age):  
        self.surname = surname  
        self.name = name  
        self.age = age  
  
    def __str__(self):  
        return {self.surname}, {self.name}, {self.age}
```

```
print(dir(MyClass("Petrenko", "Dmytro", 20)))
```



Інтроспекція та рефлексія

Реалізація інтроспекції

Будь-які інші ключові аргументи, зазначені у визначенні класу, передаються у всі операції metaclass, описані нижче.

Коли виконується визначення класу, відбуваються такі кроки:

- вирішення записів MRO;
- визначається відповідний метаклас;
- готується простір імен класу;
- виконується тіло класу;
- створюється об'єкт класу.



Інтроекція та рефлексія

Роздільна здатність записів MRO

Якщо база, що з'являється у визначенні класу, не є екземпляром type, то по ній шукається метод `__mro_entries__`. Якщо його знайдено, він викликається з вихідним базовим кортежем. Цей метод має повертати кортеж класів, який використовуватиметься замість цієї бази. Кортеж може бути порожнім, у разі вихідна база ігнорується.



Інтроспекція та рефлексія

Визначення підходящого метакласу

Відповідний метаклас визначення класу визначається так:

- якщо не вказано ні бази, ні явний метаклас (як аргумент), то використовується `type()`;
- якщо зазначений явний метаклас і не є екземпляром `type()`, він використовується безпосередньо як метаклас;
- якщо екземпляр `type()` заданий як явний метаклас або визначені бази, використовується найбільш похідний метаклас.

Найбільш похідний метаклас вибирається з явно зазначеного метакласу (якщо є) та метакласів (тобто `type(cls)`) всіх зазначених базових класів. Найпохідніший метаклас – це той, який є підтипом усіх цих метакласів-кандидатів. Якщо жоден із метакласів-кандидатів не відповідає цьому критерію, визначення класу завершиться помилкою `TypeError`.



Інтроекція та рефлексія

Підготовка простору класів

Після того, як відповідний метаклас ідентифіковано, готується простір імен класу. Якщо метаклас є атрибут `__prepare__`, він іменується як `namespace=metaclass.__prepare__(name, base, ** kwds)`, де додаткові ключові аргументи, якщо такі є, беруться з визначення класу.

Метод `__prepare__` має бути реалізований як метод класу `classmethod()`. Простір імен, що повертається `__prepare__`, передається в `__new__`, але коли створюється останній об'єкт класу, простір імен копіюється до нового словника `dict`.

Якщо метаклас не має атрибуту `__prepare__`, тоді простір імен класу ініціалізується як порожнє впорядковане відображення (словник).



Інтроспекція та рефлексія

Виконання тіла класу

Тіло класу виконується (приблизно) як `exec(body, globals(), namespace)`. Ключова відмінність від звичайного виклику функції `exec()` полягає в тому, що лексична область видимості дозволяє тілу класу (включаючи будь-які методи) посилатися на імена з поточної та зовнішньої областей, коли визначення класу відбувається всередині функції.

Однак, навіть коли визначення класу відбувається всередині функції, методи, визначені всередині класу, не можуть бачити імена, визначені в області класу. Доступ до змінних класу повинен здійснюватись через перший параметр методів екземпляра чи класу або через неявне посилання `__class__` з лексичною областю видимості, описану у "Створенні об'єкта класу".



Інтроспекція та рефлексія

Створення класу

Як тільки простір імен класу заповнений шляхом виконання тіла класу, об'єкт класу створюється шляхом виклику `metaclass(name, bases, namespace, **kwargs)`. Додаткові ключові слова `**kwargs`, передані тут, такі ж, як і ті, що передані в `__prepare__`.

Цей об'єкт класу є тим, який посилатиметься форма з нульовим аргументом `super().__class__` – це неявна посилання на замикання, створена компілятором, якщо якісь методи у тілі класу посилаються або на `__class__`, або на `super()`. Це дозволяє нульовій формі аргументу функції `super()` правильно ідентифікувати визначений клас на основі лексичної області видимості, тоді як клас або екземпляр, який був використаний для виконання поточного виклику, ідентифікується на основі першого аргументу, переданого методу.



Інтроспекція та рефлексія

Створення класу

Подробиці реалізації CPython: у CPython 3.6 і пізніших версіях осередок `__class__` передається метакласу як запис `__classcell__` у просторі імен класів. Якщо він присутній, він повинен бути поширений до `type.__new__`, щоб клас був правильно ініціалізований. Невиконання цієї вимоги призведе до помилки `RuntimeError` у Python 3.8.

При використанні метакласу `type` за замовчуванням або будь-якого метакласу, який зрештою викликає `type.__new__` після створення об'єкта класу, викликаються такі додаткові кроки налаштування:

- `type.__new__` збирає всі дескриптори у просторі імен класу, які визначають метод `__set_name__()`;
- всі ці методи `__set_name__` викликаються з визначеним класом і наданим ім'ям цього конкретного дескриптора;
- нарешті, викликається хук `__init_subclass__()` для безпосереднього батька нового класу як дозвіл його методів.

Після того, як об'єкт класу створено, він передається декораторам класу, включеним у визначення класу (якщо є) та отриманий об'єкт прив'язується до локального простору імен як певний клас.

Коли новий клас створюється за `type.__new__`, то об'єкт, вказаний як параметр простору імен, копіюється у нове впорядковане зіставлення (словник), а вихідний об'єкт відкидається. Нова копія упаковується в проксі лише для читання, що стає атрибутом `__dict__` об'єкта класу.



Інтроспекція та рефлексія

Створення класу

Подробиці реалізації CPython: у CPython 3.6 і пізніших версіях осередок `__class__` передається метакласу як запис `__classcell__` у просторі імен класів. Якщо він присутній, він повинен бути поширений до `type.__new__`, щоб клас був правильно ініціалізований. Невиконання цієї вимоги призведе до помилки `RuntimeError` у Python 3.8.

При використанні метакласу `type` за замовчуванням або будь-якого метакласу, який зрештою викликає `type.__new__` після створення об'єкта класу, викликаються такі додаткові кроки налаштування:

- `type.__new__` збирає всі дескриптори у просторі імен класу, які визначають метод `__set_name__()`;
- всі ці методи `__set_name__` викликаються з визначеним класом і наданим ім'ям цього конкретного дескриптора;
- нарешті, викликається хук `__init_subclass__()` для безпосереднього батька нового класу як дозвіл його методів.

Після того, як об'єкт класу створено, він передається декораторам класу, включеним у визначення класу (якщо є) та отриманий об'єкт прив'язується до локального простору імен як певний клас.

Коли новий клас створюється за `type.__new__`, то об'єкт, вказаний як параметр простору імен, копіюється у нове впорядковане зіставлення (словник), а вихідний об'єкт відкидається. Нова копія упаковується в проксі лише для читання, що стає атрибутом `__dict__` об'єкта класу.



Інтроспекція та рефлексія

Метакласи

Метаклас найчастіше використовується як фабрика класів, а взагалі можливості використання метакласів безмежні.

Напрямки використання:

- перерахування,
- ведення журналу,
- перевірку інтерфейсу,
- автоматичне делегування,
- автоматичне створення властивостей,
- проксі,
- фреймворки,
- автоматичне блокування/синхронізацію ресурсів.

Якщо не потрібні складні зміни класу, метакласи використовувати не варто. Просто змінити клас можна двома способами:

- Руками
- Декораторами класу
- У 99% випадків краще використовувати ці методи, а 98% зміни класу взагалі не потрібні.

Причина складності коду, який використовує метакласи, полягає не в самих метакласах. Код складний тому, що зазвичай метакласи використовуються для складних завдань, заснованих на успадкуванні, інтроспекції та маніпуляції такими змінними як `__dict__`.



Інтроспекція та рефлексія

Метакласи

Якщо не потрібні складні зміни класу, метакласи використовувати не варто. Просто змінити клас можна двома способами:

- вручну;
- за допомогою декораторів класу

Практично завжди краще використовувати ці методи, а у більшості випадків – зміни класу взагалі не потрібні.

Причина складності коду, який використовує метакласи, полягає не в самих метакласах. Код складний тому, що зазвичай метакласи використовуються для складних завдань, заснованих на успадкуванні, інтроспекції та маніпуляції такими змінними як `__dict__`.



Інтроспекція та рефлексія

Рефлексія

Рефлексія – це здатність комп'ютерної програми вивчати і модифікувати свою структуру та поведінку (значення, мета-дані, властивості та функції) під час виконання.

Найбільш важливі атрибути та функції мови для роботи з рефлексією наступні:

`hasattr()`

`delattr()`

`setattr()`

`getattr()`

Інтроспекція та рефлексія

Реалізація у Python

```
class MyClass(object):
    def __init__(self, surname, name, age):
        self.surname = surname
        self.name = name
        self.age = age

    def __str__(self):
        return {self.surname}, {self.name}, {self.age}

print(dir(MyClass("Petrenko", "Dmytro", 20)))
```

```
# Без рефлексії
m_c = MyClass(1, 2, 3)
m_c.__str__()

# З рефлексією
class_name = "MyClass"
method = "__str__"
m_c = globals()[class_name]()
getattr(m_c, method)()

# C eval
eval("MyClass().__str__()")
```

Інтроекція та рефлексія

Функція eval()

`eval(expression[, globals[, locals]])`

Функція приймає перший аргумент, який називається виразом, який містить вираз, який потрібно обчислити. `eval()` також приймає два необов'язкові(опціональні) аргументи:

- глобальні
- локальні

`eval()` використовує аргументи для оцінки виразів Python на льоту.

Щоб оцінити вираз на основі рядка, `eval()` Python виконує наступні кроки:

- розбирає вираз
- оцінює його як вираз Python
- інтерпретує його в байт-код
- повертає результат оцінювання

Примітка. Ви також можете використовувати `exec()` для динамічного виконання коду Python. Основна відмінність між `eval()` і `exec()` полягає в тому, що `eval()` може лише виконувати або оцінювати вирази, тоді як `exec()` може виконувати будь-який фрагмент коду Python.

Інтроспекція та рефлексія

Реалізація у Python

```
class MyClass(object):  
    my_attr = 0
```

```
m_c = MyClass()  
print(getattr(m_c, "my_attr"))  
# AttributeError: 'MyClass' object has no attribute 'my_attr2'  
# print(getattr(m_c, "my_attr2"))
```

```
class MyClass:  
    my_attr1 = 1  
    my_attr2 = 2  
    my_attr3 = 3
```

```
m_c = MyClass()  
print(hasattr(m_c, "my_attr1"))  
print(hasattr(m_c, "my_attr2"))  
print(hasattr(m_c, "my_attr3"))  
print(hasattr(m_c, "my_attr4"))  
print(isinstance(m_c.my_attr1, int))  
print(isinstance(m_c.my_attr2, int))  
print(isinstance(m_c.my_attr3, int))  
print(isinstance(m_c.my_attr1, str))  
print(isinstance(m_c.my_attr2, str))  
print(isinstance(m_c.my_attr3, str))
```

Інтроспекція та рефлексія

Реалізація у Python

```
class MyClass1:  
    pass
```

```
class MyClass2(MyClass1):  
    pass
```

```
print(isinstance(MyClass1(), MyClass1))  
print(type(MyClass1()) == MyClass1)  
print(isinstance(MyClass2(), MyClass1))  
print(type(MyClass2()) == MyClass1)  
print()  
print(issubclass(MyClass1, MyClass2))  
print(issubclass(MyClass2, MyClass1))
```

Інтроепекція та рефлексія

Реалізація у Python

```
class A(object):  
    pass
```

```
class B(A):  
    pass
```

```
class C(B, A):  
    pass
```

```
print(C.__base__)  
print(C.__bases__)
```

Інтроспекція та рефлексія

Модуль inspect

Основне призначення модуля inspect — давати додатку інформацію про модулі, класи, функції, трасувальні об'єкти, фрейми виконання та кодові об'єкти. Саме модуль inspect дозволяє заглянути "на кухню" інтерпретатора Python.

Модуль має функції для перевірки належності об'єктів різних типів, з якими він працює:

Функція	Перевірений тип
inspect.isbuiltin	Вбудована функція
inspect.isclass	Клас
inspect.iscode	Код
inspect.isdatadescriptor	Описувач даних
inspect.isframe	Фрейм
inspect.isfunction	Функція
inspect.ismethod	Метод
inspect.ismethoddescriptor	Описувач методу
inspect.ismodule	Модуль
inspect.isroutine	Функція або метод
inspect.istraceback	Трасувальний об'єкт

Дивіться наші уроки у відеоформаті

ITVDN.com



ITVDN

IT VIDEO DEVELOPERS NETWORK

Перегляньте цей урок у відеоформаті на освітньому порталі [ITVDN.com](http://itvdn.com) для закріплення пройденого матеріалу.

Усі курси записані сертифікованими тренерами, які працюють у навчальному центрі CyberBionic Systematics.

Перевірка знань

TestProvider.com



TestProvider

TestProvider – це online-сервіс перевірки знань з інформаційних технологій. Завдяки йому Ви можете оцінити Ваш рівень і виявити слабкі місця. Він буде корисним як у процесі вивчення технології, так і загальної оцінки знань IT-фахівця.

Після кожного уроку проходите тестування для перевірки знань на TestProvider.com.

Успішне проходження фінального тестування дасть Вам змогу отримати відповідний Сертифікат.

Python Essential

Q&A

Інформаційний відеосервіс для розробників програмного забезпечення

