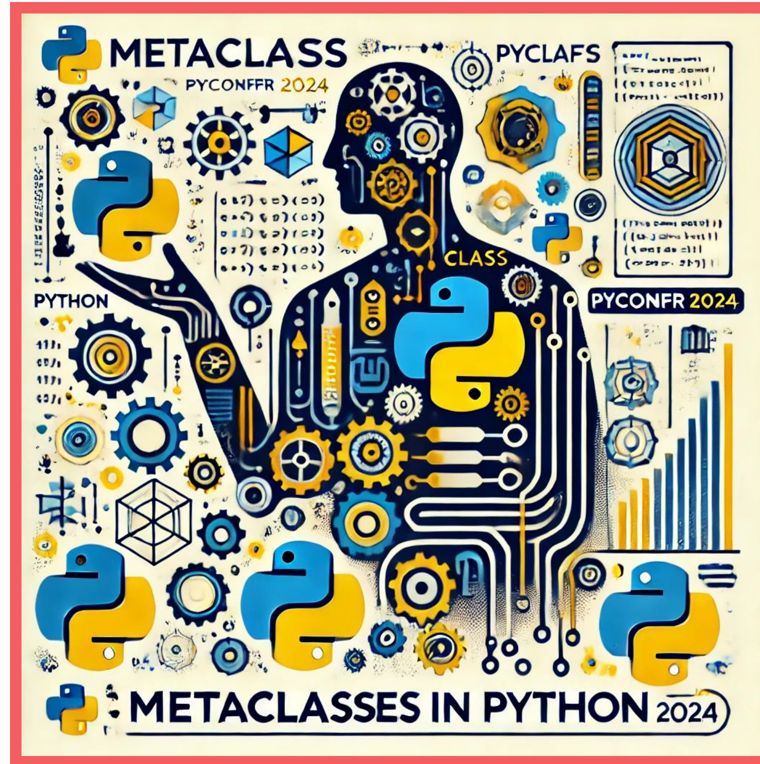
Les Métaclasses personnalisées…



…ou comment s'en passer !

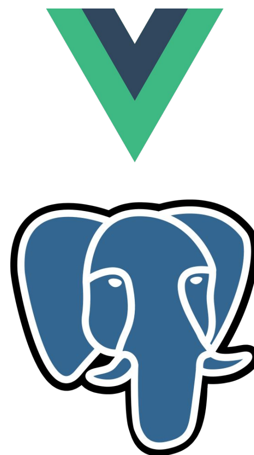
2024-09-20

Qui à déjà utiliser les Métaclasses en python ?

Il y a un piège ?!

Qui à déjà utilisé les métaclasses en python ?

Ensemble, nous concevons vos outils métiers, adaptés et intégrés à votre écosystème.



Pierre Verkest

gh: petrus-v
in: pierre-verkest
pierre@verkest.fr

Agenda

- Concepts
- Cas pratique

Explorons le builtin: type

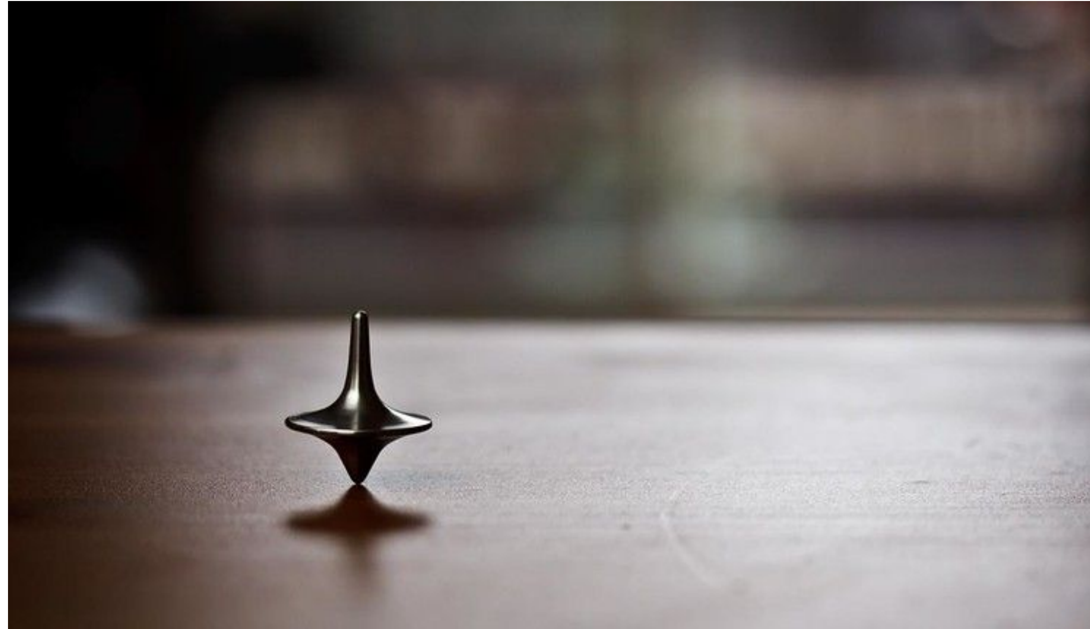
```
>>> class Car:
...     pass

>>> type(Car())
<class '__main__.Car'>

>>> type(Car)
<class 'type'>
>>> Car.__class__
<class 'type'>
```

Différent de l’héritage :

```
>>> Car.__bases__
(<class 'object'>,)
```

Inception

```
>>> type(type(Car))
<class 'type'>

>>> Car.__class__.__class__
<class 'type'>

>>> type(type)
<class 'type'>
```



Définition: Métaclasse

Métaclasse

En programmation, une **métaclasse** est une classe dont les instances sont des classes. Autrement dit, une métaclasse est la classe d'une classe.

instance of
type
instance of
object
inherits from
instance of
Car
instance of
my_car

Construire une classe dynamiquement : `type`

```
>>> class Car(object):
...     wheels = 4
...     color = "green"
...

>>> type(Car())
<class '__main__.Car'>

>>> Car.__class__
<class 'type'>
```

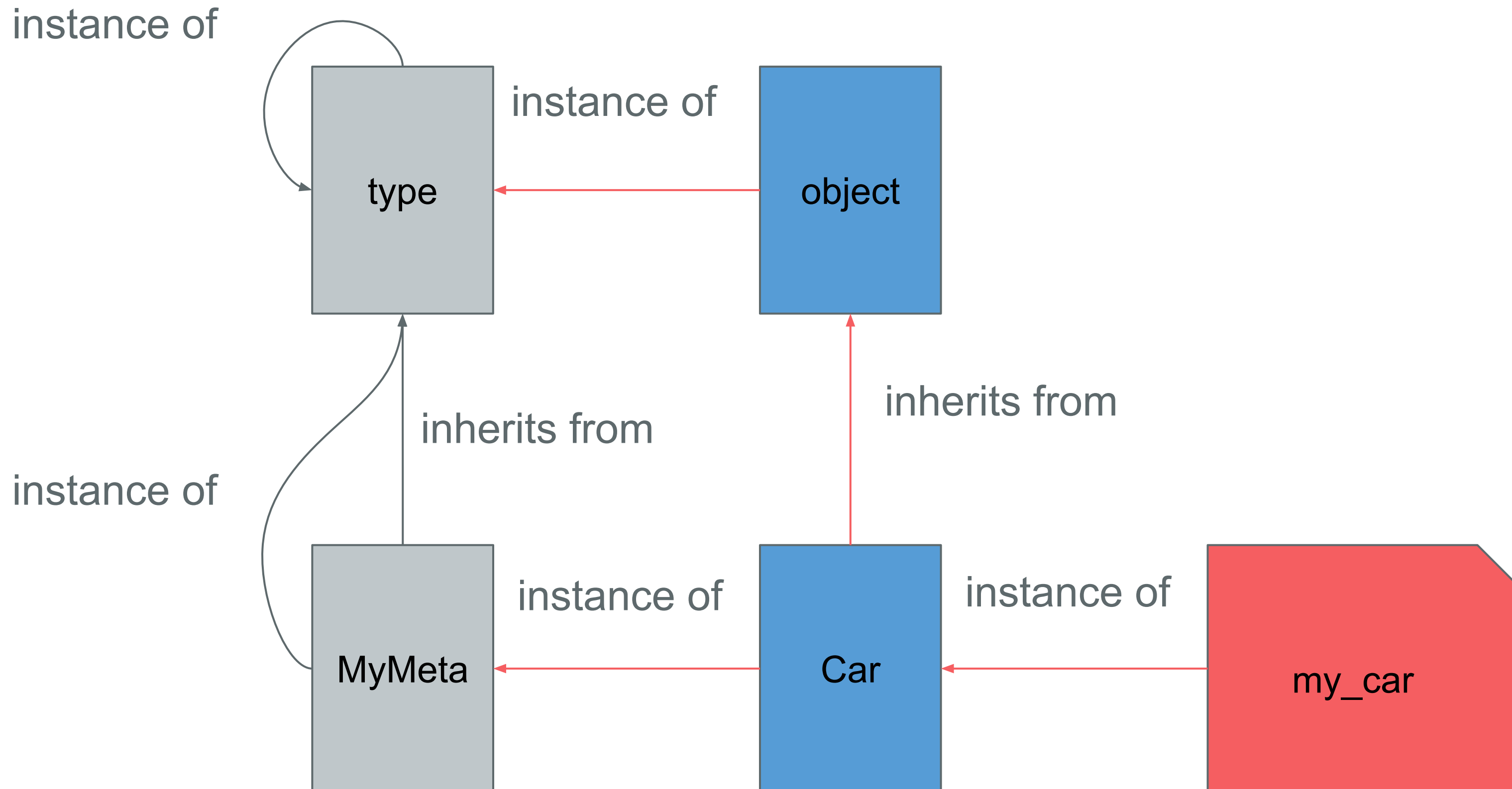
```
>>> Car = type(
...     "Car",
...     (object, ),
...     {
...      "wheels": 4,
...      "color": "green",
...     },
... )

>>> type(Car())
<class '__main__.Car'>

>>> Car.__class__
<class 'type'>
```

Métaclasse personnalisée ou comment construire une classe avec une sous classe de `type`

```python
>>> class MyMeta(type):
...     pass
...

>>> class Car(metaclass=MyMeta):
...     wheels = 4
...     color = "green"
...

>>> Car.__class__
<class '__main__.MyMeta'>
```

```python
>>> Car = MyMeta(
...     "Car",
...     (object, ),
...     {"color": "green"})

>>> Car.__class__
<class '__main__.MyMeta'>
```

Instanception



Les sous classes héritent des méta classes

```
>>> class MyMeta(type):
...     pass

>>> class Vehicle(metaclass=MyMeta):
...     pass

>>> class Car(Vehicle):
...     pass

>>> Car.__class__
<class '__main__.MyMeta'>
```

```
>>> class MyMeta(type):
...     pass

>>> class MyMeta2(type):
...     pass

>>> class Vehicle(metaclass=MyMeta):
...     pass

>>> class Mixin(metaclass=MyMeta2):
...     pass

>>> class Car(Vehicle, Mixin):
...     pass

TypeError: metaclass conflict: the metaclass
of a derived class must be a (non-strict)
subclass of the metaclasses of all its bases
```

Création de la classe ou l'instance de méta-classe

```
>>> class MyMeta(type):
...   def __new__(cls, name, bases, attributes, **kwargs):
...     new_class = super().__new__(cls, name, bases, attributes)
...     print("MyMeta.new", cls, name, bases, attributes, kwargs, "create", new_class)
...     return new_class
...

>>> class Car(metaclass=MyMeta):
...   pass
...

MyMeta.new <class '__main__.MyMeta'> Car () {'__module__': '__main__', '__qualname__':
'Car', } {} create <class '__main__.Car'>
```

Un peu de sémantique: Méta-classe => Méta-programmation

Méta programmation:

La **métaprogrammation**, nommée par analogie avec les métadonnées et les métaclasses[réf. souhaitée], désigne l'écriture de programmes qui manipulent des données décrivant elles-mêmes des programmes. Dans le cas particulier où le programme manipule ses propres instructions pendant son exécution, on parle de programme auto-modifiant.

Paramètre de classe

```
>>> def MyMeta(name, bases, attributes, extra=""):
...   match extra.lower():
...     case "function":
...       return lambda name: print(
...                          "your car is an", name)
...     case "string":
...       return "Hello the world"
...   return type(name, bases, attributes)
```

```
>>> class Car(metaclass=MyMeta, extra=""):
...   def __init__(self, name):
...     print("my car is an", name)

>>> Car("AX")
my car is an AX
>>> Car
<__main__.Car object at 0x7fe2ea0c8050>
```

```
>>> class Car(metaclass=MyMeta, extra="function"):
...   pass

>>> Car("AX")
your car is an AX
>>> Car
<function MyMeta.<locals>.<lambda> at
0x7fe2ea0956c0>
```

```
>>> class Car(metaclass=MyMeta, extra="string"):
...   pass

>>> Car
'Hello the world'
```

Exécution de la définition d’une classe

META → prepare → new → mro → init

```
... class MyMeta(type):
...
...   def __prepare__(cls, name, bases):
...       print("Prepare dict object")
...       return super().__prepare__(name, bases)
...
...   def mro(cls):
...       print("Compute mro for", cls)
...       return super().mro()
...
...   def __new__(cls, name, bases, attributes):
...       print("Creating new class")
...       c = super().__new__(
...           cls, name, bases, attributes
...       )
...       print("Class", c, "created !")
...       return c
...
...   def __init__(cls, *args, **kwargs):
...       print("Init new class", cls)
...       super().__init__(*args, **kwargs)
```

```
... class Car(metaclass=MyMeta):
...     pass
```

```
Prepare dict object
Creating new class
Compute mro for <class '__main__.Car'>
Class <class '__main__.Car'> created !
Init new class <class '__main__.Car'>
```

Réalisation d'un ORM GoDjan

Objectifs:

- Tenir un registre des modèles Python devant créer des tables SQL
- Générer les instructions CREATE TABLE
- Fournir une méthod `save` qui génère des instructions INSERT

Cas pratique

Attribute descriptor

```
>>> class Field:
...
...   def __init__(self, sql_type, default=None):
...     self.sql_type = sql_type
...     self.default = default
...
...   def __set_name__(self, owner, name):
...     self.public_name = name
...     self.private_name = '_' + name
...
...   def __get__(self, obj, objtype=None):
...     value = getattr(
...       obj,
...       self.private_name,
...       self.default
...     )
...         return value
...
...   def __set__(self, obj, value):
...     setattr(obj, self.private_name, value)
...
```

```
>>> class Car:
...   color = Field("VARCHAR")
...   wheels = Field("INTEGER", default=4)
...
...   def __init__(self, color):
...     self.color = color
...
...   def __str__(self):
...          return f"My car is {self.color} with {self.wheels}
wheel(s)"
...


>>> car = Car(“blue”)
>>> car.__dict__
{'_color': 'blue'}
>>> str(car)
'My car is blue with 4 wheel(s)'
```

Cas pratique

Attribute descriptor

```
>>> PYTHON_SQL_MAP = {str: "VARCHAR", int: "INTEGER"}
...
>>> class Field:
...
...   def __init__(self, sql_type=None, default=None):
...     self.default = default
...     self.sql_type = sql_type
...
...   def __set_name__(self, owner, name):
...     self.public_name = name
...     self.private_name = '_' + name
...     if self.sql_type is None:
...       self.sql_type = PYTHON_SQL_MAP.get(
...         owner.__annotations__.get(name, str),
...         "VARCHAR"
...       )
...
...   def __get__(self, obj, objtype=None):
...     return getattr(obj, self.private_name, self.default)
...
...   def __set__(self, obj, value):
...     setattr(obj, self.private_name, value)
...
...   def sql_definition(self):
...     SQL = f"{self.public_name} {self.sql_type}"
...     if self.default:
...       SQL += f" DEFAULT {self.default}"
...     return SQL
...
```

```
>>> class Car:
...   color: str = Field()
...   wheels: int = Field(default=4)
...
...   def __init__(self, color):
...     self.color = color
...
...   def __str__(self):
...          return f"My car is {self.color} with {self.wheels}
wheel(s)"
...


>>> str(Car("blue"))
'My car is blue with 4 wheel(s)'

>>> A.__dict__["wheels"].sql_type
'INTEGER'
```

Cas pratique

GoDjan ORM

```
>>> def get_fields(model_cls):
...   fields = {}
...   for base_cls in set(model_cls.__bases__) - {object}:
...     fields |= get_fields(base_cls)
...   fields |= {
...     v.public_name: v
...     for v in model_cls.__dict__.values()
...     if isinstance(v, Field)
...   }
...   return fields
...

>>> def _fields(cls):
...   return get_fields(cls).values()
...

>>> def _create_table_statement(cls):
...   fields_part = ", ".join(
...     field.sql_definition() for field in cls._fields()
...   )
...   return f"CREATE TABLE {cls.Meta.table_name} ({fields_part})"
...

>>> def _save_statement(self):
...   fields_name = []
...   values = []
...   for field in self._fields():
...     fields_name.append(field.public_name)
...     values.append(str(getattr(self, field.public_name)))
...   return (
...     f"INSERT INTO {self.Meta.table_name} "
...     f"({', '.join(fields_name)}) VALUES "
...     f"({', '.join(values)});"   )
...
```

SQL Injection

```
>>> class ORM:
...
...   models_registry = []
...
...   @classmethod
...   def register_model(cls, model_cls):
...     if model_cls.Meta.table_name is not None:
...       print("Register model", model_cls.__name__)
...       cls.models_registry.append(model_cls)
...
...   @classmethod
...   def create_tables(cls):
...     for model_cls in cls.models_registry:
...       print(
...         "Create table for model",
...         model_cls.__name__,
...         "with SQL:",
...         model_cls._create_table_statement()
...       )
...
```

Cas pratique

Méta-classe

```
>>> class MetaModel(type):
...
...   @classmethod
...   def __prepare__(cls, name, bases=None, kwargs=None):
...    ns = super().__prepare__(
...      cls, name, bases=bases, kwargs=kwargs
...    )
...    ns.update({
...      "_fields": classmethod(_fields),
...      "_create_table_statement": classmethod(
...         _create_table_statement
...      ),
...      "_save_statement": _save_statement,
...    })
...    return ns
...
...   def __new__(cls, name, bases, attrs, **kwargs):
...     model_cls = super().__new__(
...       cls, name, bases, attrs, **kwargs
...     )
...     ORM.register_model(model_cls)
...     return model_cls
...

>>> class GoDjanModel(metaclass=MetaModel):
...
...   class Meta:
...     table_name = None
...
```

class Meta n'est pas une métaclasse

```
>>> class Vehicle(GoDjanModel):
...   color: str = Field()
...   wheels: int = Field()
...

>>> class CarModel(Vehicle):
...   wheels: int = Field(default=4)
...   seats: int = Field(default=5)
...
...   class Meta:
...     table_name = "MetaInjectionSchema.car_table"
...
...   def __str__(self):
...     return (
...       f"my car is {self.color} "
...       f"with {self.wheels} wheel(s) "
...       f"with {self.seats} seats."
...     )
...
Register model CarModel

>>> ORM.create_tables()
Create table for model CarModel with SQL: CREATE TABLE
MetaInjectionSchema.car_table (color VARCHAR, wheels INTEGER
DEFAULT 4, seats INTEGER DEFAULT 5)

>>> car = CarModel()
>>> car.color = "blue"
>>> car._save_statement()
'INSERT INTO MetaInjectionSchema.car_table (color, wheels, seats)
VALUES (blue, 4, 5);'
```

Décorateur

```
>>> class Model:
...
...     def __init__(self, table_name=None):
...         self.table_name = table_name
...
...     def __call__(self, model_cls):
...         model_cls.Meta = type(
...             "Meta",
...             (),
...             {"table_name": self.table_name}
...         )
...         model_cls._fields = classmethod(_fields)
...         model_cls._create_table_statement = classmethod(
...             _create_table_statement
...         )
...         model_cls._save_statement = _save_statement
...         ORM.register_model(model_cls)
...         return model_cls
...
```

```
>>> class Vehicle:
...   color = Field("VARCHAR")
...   wheels = Field("INTEGER")
...
...
... @Model(table_name="myschema.car")
... class Car(Vehicle):
...   wheels = Field("INTEGER", default=4)
...   seats = Field("INTEGER", default=5)
...
...   def __str__(self):
...     return (
...       f"my car is {self.color} “
...       f"with {self.wheels} wheel(s) with “
...       f"{self.seats} seats."
...
Register model Car


>>> ORM.create_tables()
Create table for model Car with SQL: CREATE TABLE myschema.car
(color VARCHAR, wheels INTEGER DEFAULT 4, seats INTEGER DEFAULT 5)

>>> car = Car()
>>> car.color = "red"
>>> car._save_statement()
'INSERT INTO myschema.car (color, wheels, seats) VALUES (red, 4,
5);'
```

Quand utiliser des métaclasses ou pas !

Tenir un registre de classe

initialiser des attributs

modifier la définition d'une classe

S'assurer de l'implémentation d'une sous classe (ou `__init_subclass__`)

maîtriser l’ordres des attributes (peut avoir de l’importance pour des binding C)

Rendre complètement opaque le comportement d'une classe et y mettre le bazard (jouer avec mro ^^)

Remerciement

"It's Pythons All The Way Down: Python Types & Metaclasses Made Simple" - Mark Smith (PyCon AU 2019)

La doc python

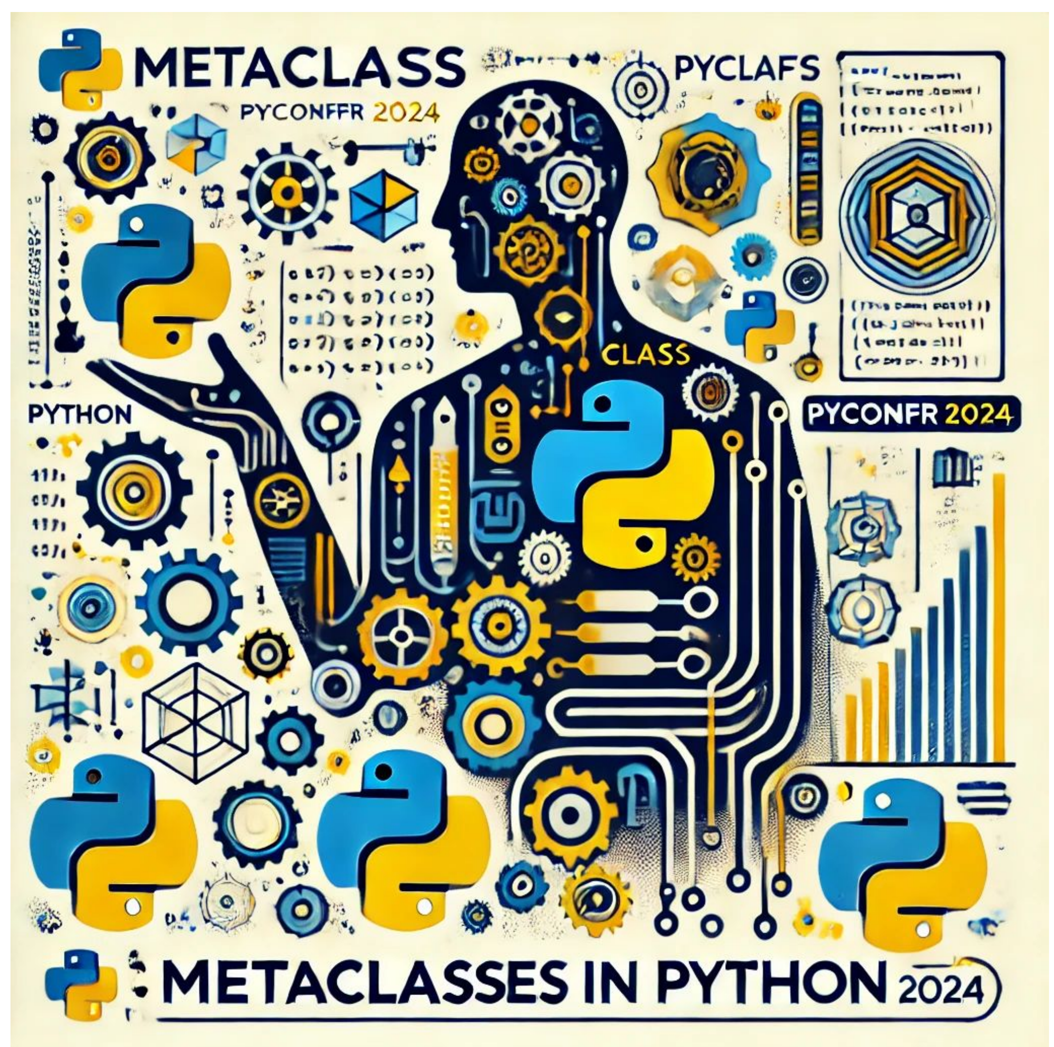
Jean-Sébastien Suzanne

Merci

Questions ?

Merci

"Crée une image illustrant une présentation pour une conférence sur les métaclasses en Python pour le PyConFr 2024. L'image doit refléter l'idée que les métaclasses sont responsables de la création des classes. Montre une métaclasse sous forme d'une figure abstraite ou d'un concept visuel complexe, manipulant ou générant des blocs de code ou des diagrammes de classes Python. La scène devrait inclure des éléments comme des lignes de code Python stylisées (en mettant en avant le mot-clé `class`), des engrenages ou des circuits symbolisant la construction interne des classes. Utilise des couleurs liées à Python (comme bleu, jaune et blanc) et intègre subtilement le logo de Python pour rappeler le cadre de la conférence. Ajoute aussi une bannière avec 'PyConFr 2024' et 'Métaclasses en Python' pour donner un contexte clair à l'image."



"Crée une image pour une présentation sur les métaclasses en Python, en utilisant une analogie spatiale. L'image doit représenter une métaclasse comme une galaxie, qui crée des planètes (symbolisant les classes), et ces planètes engendrent des satellites (représentant les instances). Montre des objets célestes interagissant entre eux dans un univers spatial, avec une galaxie générant plusieurs planètes, elles-mêmes entourées de satellites. Intègre subtilement des lignes de code Python, notamment `class Planet(metaclass=Univers):`, pour renforcer la thématique. Utilise des couleurs inspirées de l'espace et de Python, comme des nuances de bleu, jaune, et blanc, avec une ambiance futuriste et cosmique."