Strollby

PyCascades
VANCOUVER, BC 2023 | SEATTLE • PORTLAND • VANCOUVER

Metaprogramming In Python

Using metaclasses

Adarsh Divakaran
Product Engineer at Strollby - UST

Outline

Metaprogramming refers to the potential for a program to have knowledge of itself or to manipulate itself

In Python, metaprogramming can be achieved through several techniques like **decorators**, **descriptors, metaclasses**, and **introspection** using the built-in functions

Python provides a lot of flexibility for metaprogramming, and many popular libraries and frameworks use metaprogramming techniques to provide **powerful and flexible abstractions**.

Python supports a form of metaprogramming for classes called **metaclasses**.

```
>>> number = 1


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

```
>>> class Plugin:
...     pass
...
...
>>> audio_plugin = Plugin()

>>> type(audio_plugin)
<class '__main__.Plugin'>
```

Decorators • Descriptors • Metaclasses

Decorators

- A decorator in python is a design pattern that allows us to modify the functionality of a function by wrapping it in another function.

- Decorators lets us *DRY* up our callables

```python
def my_decorator(func):
    def inner(*args, **kwargs):
        print("Decorator called!")
        return func(*args, **kwargs)

    return inner

@my_decorator
def example():
    return "Example called!"

result = example()

# Decorator called! Example called!
```

Decorators

```python
def exception_handler(func):

    def inner_function(*args, **kwargs):
        try:
            func(*args, **kwargs)
        except TypeError:
            # Report error
            logging.warning(f"{func.__name__} raised TypeError")

    return inner_function

@exception_handler
def add(a, b):
    print(a + b)

>>> add("", 2)
WARNING:root:add raised TypeError
```

Decorators - Class Decorators

```python
import functools
def singleton(cls):
    """Make a class a Singleton class"""
    cls._instance = None  # Adds a new attribute to class, to store the instance
    @functools.wraps(cls)
    def wrapper_singleton(*args, **kwargs):
         if not cls._instance:
             cls._instance = cls(*args, **kwargs)
         return cls._instance

    return wrapper_singleton
```

Decorators - Class Decorators

```
>>> @singleton
... class Plugin:
...     pass
...

>>> plugin_1 = Plugin()
>>> id(plugin_1)
4387817120

>>> plugin_2 = Plugin()
>>> id(plugin_2)
4387817120
```

Descriptors

- A descriptor is an attribute value that has one of the methods in the descriptor protocol. Those methods are __get__(), __set__(), and __delete__().

- They are the mechanism behind properties, methods, static methods, class methods, and super(). They are used throughout Python itself.

Descriptors - Example

```
# ReadOnlyAttribute is a class that implements the descriptor protocol
>>> class ReadOnlyAttribute:
...     def __get__(self, obj, type=None) -> object:
...         print(f"Accessing the attribute  of {obj.__class__} to get the value")
...         return 0
...     def __set__(self, obj, value) -> None:
...         raise AttributeError("Cannot change the value")
>>> class Plugin:
...     attribute1 = ReadOnlyAttribute()
>>> new_plugin = Plugin()
>>> # Once ReadOnlyAttribute is instantiated as an attribute of Plugin, it can be considered a
descriptor.

>>> x = new_plugin.attribute1
Accessing the attribute  of <class '__main__.Plugin'> to get the value
>>> print(x)
0
```

Dunder Methods

- Dunder methods are methods that allow instances of a class to interact with the built-in functions and operators of the language.
- Typically, dunder methods are not invoked directly by the programmer, making it look like they are called by magic. *That is why dunder methods are also referred to as "magic methods" sometimes.*
- Dunder methods are not called magically, though. They are just called implicitly by the language, at specific times that are well-defined, and that depend on the dunder method in question.

```python
#Initialization and Construction
__new__(cls, other) #To get called in an object's instantiation.
__init__(self, other)   #To get called by the __new__ method.
__del__(self)   #Destructor method.

#Unary operators and functions
__pos__(self)   #To get called for unary positive e.g. +someobject.
__neg__(self)   #To get called for unary negative e.g. -someobject.
__abs__(self)   #To get called by built-in abs() function.
__round__(self,n)   #To get called by built-in round() function.
__floor__(self) #To get called by built-in math.floor() function.
__ceil__(self)  #To get called by built-in math.ceil() function.

#String Magic Methods
__str__(self)   #To get called by built-int str() method to return a string representation of a type.#
__repr__(self)  #To get called by built-int repr() method to return a machine readable representation of a type.
__dir__(self)   #To get called by built-int dir() method to return a list of attributes of a class.

#Attribute Magic Methods
__getattr__(self, name) #Is called when the accessing attribute of a class that does not exist.
__setattr__(self, name, value)  #Is called when assigning a value to the attribute of a class.
__delattr__(self, name) #Is called when deleting an attribute of a class.

#Operator Magic Methods Description
__add__(self, other)    #To get called on add operation using + operator
__sub__(self, other)    #To get called on subtraction operation using - operator.
__mul__(self, other)    #To get called on multiplication operation using * operator.
__floordiv__(self, other)   #To get called on floor division operation using // operator.
__truediv__(self, other)    #To get called on division operation using / operator.
__mod__(self, other)    #To get called on modulo operation using % operator.
__pow__(self, other[, modulo])  #To get called on calculating the power using ** operator.
__lt__(self, other) #To get called on comparison using < operator.
__le__(self, other) #To get called on comparison using <= operator.
__eq__(self, other) #To get called on comparison using == operator.
__ne__(self, other) #To get called on comparison using != operator.
__ge__(self, other) #To get called on comparison using >= operator.
```

Class creation in Python

Here's what happens whenever the keyword class is encountered:

- The body *(statements and functions)* of the class is isolated.
- The namespace dictionary of the class is created *(but not populated yet)*.
- The body of the class executes, then the namespace dictionary is populated with all of the attributes, methods defined, and some additional useful info about the class.
- The metaclass is identified
- The metaclass is then called with the name, bases, and attributes of the class to instantiate it. ‘type’ is the default metaclass in Python

Type

- type is the built-in metaclass Python uses.
- Parent of all classes

Usage:

- type(obj): If a single argument is passed, it returns the type of the given object.
- type (name, bases, attrs): returns a new data type or, if simple, a new class,

Type

Python uses the type class to create other classes.

```
type(name, bases, dict) -> a new type
```

The constructor has three parameters for creating a new class:

- name is the name of the class e.g., Plugin
- bases is a tuple that contains the base classes of the new class.

  For example, if the Plugin inherits from the BasePlugin class, so the bases contains one class (BasePlugin,)

- dict is the class namespace

Type

```python
class Plugin:
    plugin_id = 10


class SubPlugin(Plugin):
    sub_plugin_id = 20

    @classmethod
    def print_id(cls):
        print(cls.sub_plugin_id)
```

```python
class Plugin:
    plugin_id = 10

@classmethod
def print_id_implementation(cls):
    print(cls.sub_plugin_id)

SubPlugin = type("SubPlugin",  # class name
                 (Plugin,),    # bases
                 {
                     "print_id": print_id_implementation,
                     "sub_plugin_id": 20
                 }             # namespace dict
                 )
```

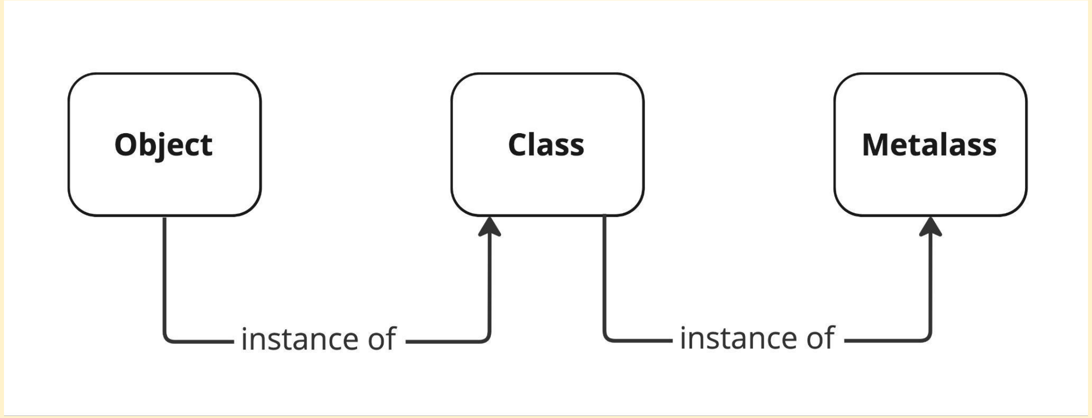
Type

```
>>> print(SubPlugin.__class__)
<class 'type'>

>>> print(SubPlugin.__bases__)
(<class '__main__.Plugin'>,)

>>> print(SubPlugin.__dict__)
{'print_id': <classmethod(<function print_id_implementation at 0x10e6b3f60>)>, 'sub_plugin_id': 20, '__module__':
'__main__', '__doc__': None}
```

Metaclass

Since classes are `just objects` metaclasses are the way to customize their creation

The base metaclass is *type*
type's type is *type*



Metaclass

```
>>> class Plugin:
...     pass

>>> audio_plugin = Plugin()

>>> type(audio_plugin)
<class '__main__.Plugin'>

>>> type(Plugin)
<class 'type'>

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

Metaclass

A metaclass is, by definition, a class whose instance is another class.

- Metaclasses allow us to customize the process of creating a class and partially manage the process of creating an instance of a class.
- The metaclass is responsible for the generation of classes, so we can write our custom metaclasses to modify the way classes are generated by performing extra actions or injecting code.

Metaclass

Writing a custom metaclass involves two steps:

1. **Write a subclass of 'type'.**
2. Insert the new metaclass into the class creation process.

We subclass the type class and modify the dunder methods like __init__, __new__, __prepare__, and __call__ to modify the behavior of the classes while creating them. These methods have information like base class, name of the class, attributes, and their values.

Metaclass - Metamethod Invocation Order

The order in which the Python interpreter invokes the metamethods of the metaclass at the time of the creation of the class:

- The interpreter identifies and finds parent classes for the current class (if any).
- The interpreter defines the metaclass.
- The method MetaClass.__ prepare__ is called – it must return a dict-like object in which the attributes and methods of the class will be written. After that, the object will be passed to the method MetaClass.__ new__ through the argument attrs.
- The interpreter reads the body of the class and forms the parameters for transferring them to the MetaClass.

Metaclass - Metamethod Invocation Order

- The method MetaClass.__ new__ is is called - new__ is a constructor method, returns the created class object.
- The method MetaClass.__ init__ is invoked – an initializer method with which we can add additional attributes and methods to a class object.
- At this step, the class is considered to be created.

__new__ vs __init__ dunders

- __new__ is responsible for returning a new instance of our class.
- __init__, on the other hand, doesn't return anything. It's only responsible for initializing the instance after it's been created.
- A simple rule of thumb to remember: Use new when we need to control the creation of a new instance; use init when we need to control the initialization of a new instance.
- __new__ is used when we wants to define dict or bases tuples before the class is created. The return value of __new__is usually an instance of cls.

Metaclass - Example

```
class Meta(type):

    @classmethod
    def __prepare__(mcls, name, bases):
        print(f"Prepare called, {mcls=} {name=} {bases=}")
        return super().__prepare__(name, bases)

    def __new__(mcls, name, bases, namespace):
        print(f"New called, {name=} {bases=}  {namespace=}")
        return super().__new__(mcls, name, bases, namespace)

    def __init__(cls, name, bases, namespace):
        print(f"Init called for {cls}")
        cls.new_attrib = 2
        super().__init__(name, bases, namespace)
```

Metaclass - Example

```
>>> class A(metaclass=Meta):
...     attrib = 0
...
...     def method(self):
...         pass

Prepare called, mcls=<class '__main__.Meta'> name='A' bases=()

New called, name='A' bases=()  namespace={'__module__': '__main__', '__qualname__': 'A', 'attrib': 0, 'method':
<function A.method at 0x1058a88b0>}

Init called for <class '__main__.A'>

>>> print(F"Namespace dict of Class A {A.__dict__}")

Namespace dict of Class A {'__module__': '__main__', 'attrib': 0, 'method': <function A.method at 0x1058a88b0>,
'__dict__': <attribute '__dict__' of 'A' objects>, '__weakref__': <attribute '__weakref__' of 'A' objects>,
'__doc__': None, 'new_attrib': 2}
```

When to use metaclasses?

- Metaclasses propagate down hierarchies. Child classes use metaclass of their parent by default.
- It's mostly about wrapping/rewriting
    - Functions : Decorators
    - Classes : Class Decorators
    - Class hierarchies : Metaclasses

Metaclass Alternatives - Class Decorators

Class decorators

- Can be used to add custom behavior to classes
- Do not propagate class hierarchies

Metaclass Alternatives - Class Decorators

```
#Singleton pattern using Metaclasses
class SingletonMeta(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
            if cls not in cls._instances:
                    cls._instances[cls] = super(SingletonMeta,cls).__call__(*args, **kwargs)
            return cls._instances[cls]

class SingletonClass(metaclass=SingletonMeta):
    pass
```

Metaclasses Alternatives - PEP 487

PEP 487 – Simpler customisation of class creation

Abstract:

Currently, customising class creation requires the use of a custom metaclass. This custom metaclass then persists for the entire lifecycle of the class, creating the potential for spurious metaclass conflicts.

This PEP proposes to instead support a wide range of customisation scenarios through a new __init_subclass__ hook in the class body, and a hook to initialize attributes.

The new mechanism should be easier to understand and use than implementing a custom metaclass, and thus should provide a gentler introduction to the full power of Python's metaclass machinery.

Metaclasses Alternatives - PEP 487

PEP 487 sets out to take two common metaclass use-cases and make them more accessible without having to understand all the ins and outs of metaclasses.

While there are many possible ways to use a metaclass, the vast majority of use cases falls into just three categories: some initialization code running after class creation, the initialization of descriptors and keeping the order in which class attributes were defined.

The first two categories can easily be achieved by having simple hooks into the class creation:

1. An __init_subclass__ hook that initializes all subclasses of a given class. It is useful for both registering subclasses in some way, *and* for setting default attribute values on those subclasses.
2. upon class creation, a __set_name__ hook is called on all the attribute (descriptors) defined in the class

Metaclasses Alternatives - PEP 487

Class Registration using Metaclass

```
>>> # Library code
>>> registry = {}
>>> def register(cls):
...     if cls.__name__ != "Parent":
...         registry[cls.__name__] = cls

>>> class ParentMeta(type):
...     def __new__(cls, name, bases, attrs):
...         new_class = super(ParentMeta, cls).__new__(cls, name, bases, attrs)
...         register(new_class)  # register function
...         return new_class

>>> class Parent(metaclass=ParentMeta):
...     pass

>>> # User defined classes
>>> class Child1(Parent):
...     pass

>>> class Child2(Parent):
...     pass

>>> print(registry)
{'Child1': <class '__main__.Child1'>, 'Child2': <class '__main__.Child2'>}
```

Metaclasses Alternatives - PEP 487

Class Registration using PEP 487 __init_subclass__

```python
>>> # Library code
>>> registry = {}

>>> def register(cls):
...     registry[cls.__name__] = cls

>>> class Parent:
...     def __init_subclass__(cls, **kwargs):
...         super().__init_subclass__(**kwargs)
...         register(cls)

>>> # User defined classes
>>> class Child1(Parent):
...     pass

>>> class Child2(Parent):
...     pass

>>> print(registry)
{'Child1': <class '__main__.Child1'>, 'Child2': <class '__main__.Child2'>}
```

"Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why)."

- Tim Peters

Examples

Python ABC - ABCMeta

- To Make a class abstract, we inherit from class ABC defined in stdlib abc module.
- Metaclass of ABC is ABCMeta

ABCMeta

- Holds a registry of all abstract methods defined
- On instantiation, check if all abstract methods are implemented

Python ABC

```
>>> from abc import ABC, abstractmethod

>>> class BasePlugin(ABC):
...     @abstractmethod
...     def get_config(self):
...         pass

>>> class AudioPlugin(BasePlugin):
...     pass

>>> AudioPlugin()

Traceback (most recent call last):
  File "<input>", line 15, in <module>
TypeError: Can't instantiate abstract class AudioPlugin with abstract method get_config
```

```python
class ABCMeta(type):
    """Metaclass for defining Abstract Base Classes (ABCs).

    """
    ...

    def __new__(mcls, name, bases, namespace, /, **kwargs):
        cls = super().__new__(mcls, name, bases, namespace, **kwargs)
        # Compute set of abstract method names
        abstracts = {name
                     for name, value in namespace.items()
                     if getattr(value, "__isabstractmethod__", False)}
        for base in bases:
            for name in getattr(base, "__abstractmethods__", set()):
                value = getattr(cls, name, None)
                if getattr(value, "__isabstractmethod__", False):
                    abstracts.add(name)
        cls.__abstractmethods__ = frozenset(abstracts)

        ...

        return cls

    def register(cls, subclass): ...

    def _dump_registry(cls, file=None): ...
```

Enums

- Created similar to normal class, by inheriting from enum.Enum
- Enums have been added to Python 3.4 as described in PEP 435

Enums

```
>>> from enum import Enum

>>> class Title(Enum):
...     MR = 1
...     MRS = 2
...     MS = 3

>>>  type(Title)
<class 'enum.EnumMeta'>

>>>type(Enum)
<class 'enum.EnumMeta'>
```

Enums - Why Metaclass?

- Objects are not created for Enums.
- Validation before instantiation (eg: Duplicate keys)
- Usages like:
  - Title.MR - Attribute access
  - Title[MR] - Subscript access
  - Title(1) - __call__

```python
class EnumMeta(type):
    """
    Metaclass for Enum
    """
    @classmethod
    def __prepare__(metacls, cls, bases, **kwds): ...

    def __new__(metacls, cls, bases, classdict, **kwds): ...

    def __bool__(self): ...

    def __call__(cls, value, names=None, *, module=None, qualname=None, type=None, start=1): ...

    def __contains__(cls, member): ...

    def __delattr__(cls, attr): ...

    def __dir__(self): ...

    def __getattr__(cls, name): ...

    def __getitem__(cls, name): ...

    def __iter__(cls): ...

    def __len__(cls): ...
```

```python
class Enum(metaclass=EnumMeta):
    """
    Generic enumeration.

    Derive from this class to define new enumerations.
    """
    ...

    def _generate_next_value_(name, start, count, last_values): ...

    @classmethod
    def _missing_(cls, value): ...

    def __repr__(self): ...

    def __str__(self): ...

    def __dir__(self): ...

    def __format__(self, format_spec): ...

    def __hash__(self): ...
```

Django Models

```python
from django.db import models

class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()



"""
# Django Model Source
class Model(AltersData, metaclass=ModelBase):
    def __init__(self, *args, **kwargs): ...
"""
```

```python
class ModelBase(type):
    """Metaclass for all models."""

    def __new__(cls, name, bases, attrs, **kwargs):
        super_new = super().__new__

        # Ensure initialization is only performed for subclasses of Model
        # (excluding Model class itself).
        parents = [b for b in bases if isinstance(b, ModelBase)]
        if not parents:
            return super_new(cls, name, bases, attrs)

        # Modifying attrs - the namespace dict
        module = attrs.pop("__module__")
        new_attrs = {"__module__": module}
        ...
        attr_meta = attrs.pop("Meta", None)

        # Pass all attrs without a (Django-specific) contribute_to_class()
        # method to type.__new__()
        contributable_attrs = {}

        for obj_name, obj in attrs.items():
            # attributes manipulation
            if _has_contribute_to_class(obj):
                # checks for contribute_to_class attribute in the object
                contributable_attrs[obj_name] = obj
            else:
                new_attrs[obj_name] = obj
        new_class = super_new(cls, name, bases, new_attrs, **kwargs)
        ...
```

Metaprogramming in Python - More to Explore

Python has many popular useful features: dir(), help(), decorators, descriptors, metaclasses, & more

But also great support for more 'esoteric' uses...

- Handling code as abstract syntax trees - ast module
- Inspecting runtime objects - inspect module(eg: getsource), dir()
- Viewing the interpreter stack - inspect module, inspect.stack()
- Compilation to bytecode at runtime - dis module

References

- [Data model — Python 3 documentation]
- [Metaprogramming in Python - IBM Developer]
- [The Metaprogramming In Production On Python - Smartspate]
- [Bytepawn - Marton Trencseni – Building a toy Python Enum class]
- [Python 3 Metaprogramming - David Beazley]

Thank You

adarsh-d

adarshd905