Python __slots__

Summary: in this tutorial, you will learn about the Python `__slots__` and how how to use it to make your class more efficient.

Introduction to the Python __slots__

The following defines a `Point2D` class that has two attributes including `x` and `y` coordinates:

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Point2D({self.x},{self.y})'
```

Each instance of the `Point2D` class has its own `__dict__` attribute that stores the instance attributes (https://www.pythontutorial.net/python-oop/python-instance-variables/) . For example:

```
point = Point2D(0, 0)
print(point.__dict__)
```

By default, Python uses the dictionaries (https://www.pythontutorial.net/python-basics/python-dictionary/) to manage the instance attributes. The dictionary allows you to add more attributes to the instance dynamically at runtime. However, it also has a certain memory overhead. If the `Point2D` class has many objects, there will be a lot of memory overhead.

To avoid the memory overhead, Python introduced the slots. If a class only contains fixed (or predetermined) instance attributes, you can use the slots to instruct Python to use a more compact data structure instead of dictionaries.

For example, if the `Point2D` class has only two instance attributes, you can specify the attributes in the slots like this:

```
class Point2D:
    __slots__ = ('x', 'y')

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Point2D({self.x},{self.y})'
```

In this example, you assign an iterable (https://www.pythontutorial.net/python-basics/python-iterables/) (a tuple (https://www.pythontutorial.net/python-basics/python-tuples/)) that contains the attribute names that you'll use in the class.

By doing this, Python will not use the `__dict__` for the instances of the class. The following will cause an `AttributeError` error:

```
point = Point2D(0, 0)
print(point.__dict__)
```

Error:

```
AttributeError: 'Point2D' object has no attribute __dict__
```

Instead, you'll see the `__slots__` in the instance of the class. For example:

```
point = Point2D(0, 0)
print(point.__slots__)
```

Output:

```
('x', 'y')
```

Also, you cannot add more attributes to the instance dynamically at runtime. The following will result in an error:

```
point.z = 0
```

Error:

```
AttributeError: 'Point2D' object has no attribute 'z'
```

However, you can add the class attributes (https://www.pythontutorial.net/python-oop/python-class-attributes/) to the class:

```
Point2D.color = 'black'
pprint(Point2D.__dict__)
```

Output:

```
mappingproxy({'__doc__': None,
              '__init__': <function Point2D.__init__ at 0x000001BBBA841310>,
              '__module__': '__main__',
              '__repr__': <function Point2D.__repr__ at 0x000001BBBA8413A0>,
              '__slots__': ('x', 'y'),
              'color': 'black',
              'x': <member 'x' of 'Point2D' objects>,
              'y': <member 'y' of 'Point2D' objects>})
```

This code works because Python applies the slots to the instances of the class, not the class.

Python __slots__ and single inheritance

Let's examine the slots in the context of inheritance.

The base class uses the slots but the subclass doesn't

The following defines the `Point2D` as the base class and `Point3D` as a subclass that inherits from the `Point2D` class:

```
class Point2D:
    __slots__ = ('x', 'y')

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Point2D({self.x},{self.y})'


class Point3D(Point2D):
    def __init__(self, x, y, z):
```

```
        super().__init__(x, y)
        self.z = z


if __name__ == '__main__':
    point = Point3D(10, 20, 30)
    print(point.__dict__)
```

Output:

```
{'z': 30}
```

The `Point3D` class doesn't have slots so its instance has the `__dict__` attribute. In this case, the subclass `Point3D` uses slots from its base class (if available) and uses an instance dictionary.

If you want the `Point3D` class to use slots, you can define additional attributes like this:

```
class Point3D(Point2D):
    __slots__ = ('z',)

    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z
```

Note that you don't specify the attributes that are already specified in the `__slots__` of the base class.

Now, the `Point3D` class will use slots for all attributes including x, y, and z.

The base class doesn't use __slots__ and the subclass doesn't

The following example defines a base class that doesn't use the `__slots__` and the subclass does:

```
class Shape:
    pass


class Point2D(Shape):
    __slots__ = ('x', 'y')

    def __init__(self, x, y):
        self.x = x
        self.y = y


if __name__ == '__main__':
    # use both slots and dict to store instance attributes
    point = Point2D(10, 10)
    print(point.__slots__)
    print(point.__dict__)

    # can add the attribute at runtime
    point.color = 'black'
    print(point.__dict__)
```

Output:

```
('x', 'y')
{'color': 'black'}
```

In this case, the instances of the `Point2D` class uses both `__slots__` and dictionary to store the instance attributes.

Summary

- Python uses dictionaries to store instance attributes of instances of a class. This allows you to dynamically add more attributes to instances at runtime but also create a memory overhead.
- Define `__slots__` in the class if it has predetermined instances attributes to instruct Python not to use dictionaries to store instance attributes. The `__slots__` optimizes the memory if the class has many objects.