

| STAT243 Lecture 5.5 Object-Oriented Programming

| 1 OOP Principles

| 1.1 OOP 是什么？

Object-oriented programming (OOP) 在组织代码时需要考虑:

- 储存信息的 objects
- 以特定方式操作这些 objects 的 methods

其中 objects 属于 class, class 的组成部分包括:

- 存储信息的 fields (data)
- 操作 fields 的 methods (functions)

| 1.2 OOP principles

OOP 中的一些标准概念包括 encapsulation (封装), inheritance (继承), polymorphism (多态), 和 abstraction (抽象).

1. 封装:

- 防止用户从 object 外部直接访问 object 内的内部数据
- 不过 class 的设计允许用户通过程序员建立的接口访问数据 ('getter' 和 'setter' methods)
- 然而, Python 实际上并不真正强制执行 internal/private information 的概念

2. 继承:

- 允许一个类基于另一个类, 并添加 more specialized features

3. 多态:

- 允许对象或函数根据 context 有不同的行为
- 多态函数根据 input types 表现不同
- 一个例子是有一个名为 "algorithm" 的基类或超类, 以及各种特定的机器学习算法继承自该类, 所有这些类都可能有一个 "predict" method

4. 抽象:

- 涉及隐藏某些操作的细节, 为用户提供输入和获取输出的接口

⚠ Remark ▾

Class 通常用于有初始化类对象的 **constructors** 和移除对象的 **destructors**

| 1.3 Python 中的 Classes

🔗 Logic ▾

Python 提供了相当标准的方法来编写面向对象的代码

我们的示例是创建一个用于处理随机时间序列的 class:

- Class 的每个 object 都有控制时间序列随机行为的特定参数值
- 使用特定的 object, 我们可以模拟一个或多个时间序列



Python

```
1 import numpy as np
2
```

```

3  class tsSimClass:
4      """
5      时间序列模拟器的 class definition
6      """
7
8  def __init__(self, times, mean=0, cor_param=1, seed=1):
9      """constructor, 在调用 `tsSimClass(...)` 创建类实例时调用"""
10     self.n = len(times)
11     self.mean = mean
12     self.cor_param = cor_param
13
14     # 私有属性 (封装)
15     self._times = times
16     self._current_U = False
17
18     # 一些设置步骤
19     self._calc_mats()
20     np.random.seed(seed)
21
22 def __str__(self):
23     """print method"""
24     return f"An object of class `tsSimClass` with {self.n} time points."
25
26 def __len__(self):
27     return self.n
28
29 # 公共方法: getter 和 setter (封装)
30 def set_times(self, new_times):
31     self._times = new_times
32     self._current_U = False
33     self._calc_mats()
34
35 def get_times(self):
36     return self._times
37
38 # 主要公共方法
39 def simulate(self):
40     if not self._current_U:
41         self._calc_mats()
42     return self.mean + np.dot(self.U.T, np.random.normal(size=self.n))
43
44 # 私有方法
45 def _calc_mats(self):
46     """计算相关矩阵和 Cholesky 因子 (缓存)"""
47     lag_mat = np.abs(self._times[:, np.newaxis] - self._times)
48     cor_mat = np.exp(-lag_mat**2 / self.cor_param**2)
49     self.U = np.linalg.cholesky(cor_mat)
50     print("Done updating correlation matrix and Cholesky factor.")
51     self._current_U = True

```

使用类的示例:



Python

```

1  myts = tsSimClass(np.arange(1, 101), 2, 1)
2  # Done updating correlation matrix and Cholesky factor.
3
4  print(myts)
5  # An object of class 'tsSimClass' with 100 time points.
6
7  # 模拟时间序列

```

```
8 y1 = myts.simulate()
```

1.4 Copies 和 References

在以下示例中,

- `myts_ref` 是 `myts` 的 (浅) 拷贝, 两个 names 指向相同的底层对象, 但是在对 `myts_ref` 进行 assignment 时没有拷贝任何数据
- `myts_full_copy` 是对不同 object 的 reference, `myts` 的所有数据都必须拷贝到 `myts_full_copy`. 这需要额外的内存和时间, 但也更安全



Python

```
1 myts_ref = myts # 'myts_ref' 和 'myts' 是同一底层对象的名称
2
3 import copy
4 myts_full_copy = copy.deepcopy(myts) # 完整的深拷贝
5
6 ## Now let's change the values of a field.
7 myts.set_times(np.arange(1,1001,10))
8 # Done updating correlation matrix and Cholesky factor.
9
10 myts.get_times()[0:4]
11 # array([ 1, 11, 21, 31])
12
13 myts_ref.get_times()[0:4] # the same as `myts`
14 # array([ 1, 11, 21, 31])
15
16 myts_full_copy.get_times()[0:4] # different from `myts`
17 # array([1, 2, 3, 4])
```

1.5 Encapsulation 封装

- Private fields 的使用保护它们不被用户修改
- Python 并不真正提供此功能, 但按照约定, 名称以下划线开头的属性被视为私有
- 在模块中, 以下划线开头的对象是私有属性的弱形式. 用户可以访问它们, 但 `from foo import *` 不会导入它们.

1.6 Inheritance 继承

继承可以是减少代码重复并以逻辑方式组织代码的强大方法



Python

```
1 class Bear:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5
6     def __str__(self):
7         return f"A bear named '{self.name}' of age {self.age}."
8
9     def color(self):
10        return "unknown"
11
12 class GrizzlyBear(Bear):
13     def __init__(self, name, age, num_people_killed=0):
14         super().__init__(name, age)
15         self.num_people_killed = num_people_killed
16
17     def color(self):
```

```

18         return "brown"
19
20 # 使用示例
21 yog = Bear("Yogi the Bear", 23)
22 print(yog) # A bear named 'Yogi the Bear' of age 23.
23 yog.color() # unknown
24
25 num399 = GrizzlyBear("Jackson Hole Grizzly 399", 35)
26 print(num399) # A bear named 'Jackson Hole Grizzly 399' of age 35.
27 num399.color() # brown
28 num399.num_people_killed # 0

```

`GrizzlyBear` 类具有超出基类继承的额外字段/方法. Python 先使用特定于 `GrizzlyBear` 类的方法 (如果存在), 然后再回退到 `Bear` 类的方法

⚠ Remark ▾

以上是多态性的一个例子, `GrizzlyBear` 类的实例是多态的, 因为它们可以同时具有 `GrizzlyBear` 和 `Bear` 类的行为: `color` 方法是多态的, 因为它可以用于两个类, 但根据不同的类定义了不同的行为

| 2 Attributes

| 2.1 Attributes 是什么?

fields 和 methods 都是属性.

| 2.2 Class attributes 与 Instance attributes

Class attributes 允许我们操作与 class 的所有 instances 相关的信息

在下例中, `count` 是类属性, 而 `name` 和 `age` 是实例属性

```

Python
1 class Bear:
2     count = 0 # 类属性
3
4     def __init__(self, name, age):
5         self.name = name # 实例属性
6         self.age = age # 实例属性
7         Bear.count += 1
8
9 yog = Bear("Yogi the Bear", 23)
10 yog.count # 1
11
12 smokey = Bear("Smokey the Bear", 77)
13 smokey.count # 2

```

| 2.3 添加属性

可以在某些情况下 add instance attributes on the fly

```

Python
1 yog.bizarre = 7
2 yog.bizarre # 7
3
4 def foo(x):
5     print(x)

```

```
6
7 foo.bizarre = 3
8 foo.bizarre # 3
```

3 Generic function OOP

Logic ▾

Generic function (泛型函数) 的使用很方便, 它允许我们使用熟悉的函数处理各种对象, 可以理解为函数层面的多态

3.1 Generic function 的例子

考虑 Python 中的 `len` 函数. 它似乎神奇地适用于各种对象.

- Python 通过调用 argument 所属的 class 的 `__len__` 方法来实现 `len` 函数.
- `__len__` 是一个 **Double-UNDERscore 双下划线** 方法.

类似的情况也发生在运算符上:

```
Python
1 x = 3
2 x + 5 # 8
3 x.__add__(5) # 8
4
5 x = 'abc'
6 x + 'xyz' # 'abcxyz'
7 x.__add__('xyz') # 'abcxyz'
```

3.2 为什么使用泛型函数

Python 开发人员本可以将 `len` 编写为具有一堆 if 语句的常规函数, 以便处理不同类型的输入对象. 但这有一些缺点:

1. 需要编写进行 checking 的代码
2. 所有不同情况的代码都存在于一个可能很长的函数中
3. 最重要的是, `len` 仅适用于现有类

4 dunder methods

4.1 常见的 dunder methods

dunder (双下划线) 方法是一种特殊方法, 在以下情况中, Python 将调用这些方法

- 在类的实例上调用某些函数
- 调用其他标准操作

一些重要的双下划线方法:

- `__init__`: 构造函数
- `__len__`: 被 `len()` 调用
- `__str__`: 被 `print()` 调用
- `__repr__`: 在调用对象名称时调用
- `__call__`: 如果实例作为函数调用调用
- `__add__`: 被 `+` 运算符调用
- `__getitem__`: 被 `[]` 切片运算符调用

4.2 dunder methods 示例

```

Python
1 class Bear:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5
6     def __str__(self):
7         return f"A bear named {self.name} of age {self.age}."
8
9     def __repr__(self):
10        return f"Bear(name={self.name}, age={self.age})"
11
12    def __add__(self, value):
13        self.age += value
14        return None
15
16 yogi = Bear("Yogi the Bear", 23)
17 print(yogi)    # 调用 __str__: A bear named Yogi the Bear of age 23.
18 yogi          # 调用 __repr__: Bear(name=Yogi the Bear, age=23)
19 yogi + 12      # 调用 __add__
20 print(yogi)    # A bear named Yogi the Bear of age 35.

```

4.3 Python object protocols: 示例 1 - iterators

支持 iteration 的 container class 应提供 `__iter__` 和 `__next__` 方法来实现迭代器协议。

```

Python
1 mytuple = ("apple", "banana", "cherry")
2 for item in mytuple:
3     print(item)
4
5 # 手动创建迭代器
6 myit = iter(mytuple) # 等价于 mytuple.__iter__()
7 type(myit) # <class 'tuple_iterator'>
8
9 print(next(myit)) # apple
10 print(next(myit)) # banana
11 myit.__next__()  # cherry, 等价于 next(myit)

```

⚠ Remark ▾

Tuple 是 iterable container, 但它们本身不是 iterator

4.4 Python 对象协议: 示例 2 - 使用 with 的上下文管理器

在 Python 中读写文件的标准方式使用 `with`:

```

Python
1 with open('myfile.txt', 'r') as file:
2     lines = file.readlines()

```

这创建了一个 "context manager", 等效于:

```

Python
1 file = open('myfile.txt', 'r')
2 try:

```

```
3     lines = file.readlines()
4 finally:
5     file.close()
```

通过对 class 提供 `__enter__` 和 `__exit__` 方法, 可以使用 `with` 实现 context manager protocol



Python

```
1 import time
2
3 class MyTimer(object):
4     def __enter__(self):
5         print(f"Starting at {time.ctime()}.")
6
7     def __exit__(self, exception_type, exception_value, traceback):
8         print(f"Ending at {time.ctime()}.")
9
10 with MyTimer():
11     x = np.random.normal(size=50000000)
12     del x
13
14 # Starting at Mon Sep 29 08:34:10 2025.
15 # Ending at Mon Sep 29 08:34:12 2025.
```