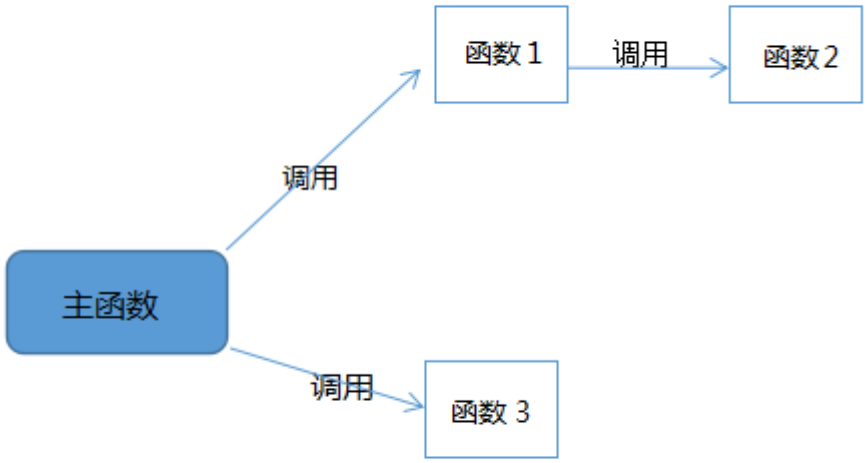
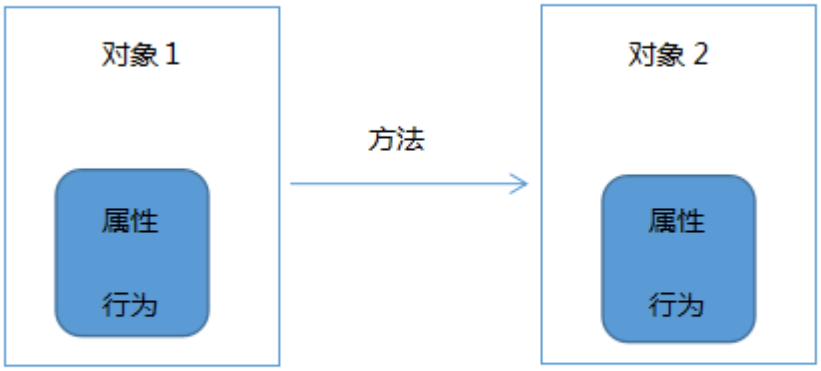
面向对象

面向对象目前仍然是最主流的编程范式，它强调将程序中的各种功能分离，用不同模块的交互、通信来模拟现实逻辑。我们的思考方向，也就从一件事怎么做，转换为了谁来做。



面向过程



面向对象

引入对象的动机

我们曾在lesson 1中说过，数据 + 函数就可以构成程序的全部。引入对象正是为了更好地管理数据与函数，且对于很多程序而言，**对象中存储的数据仅仅是作为一组函数的配置参数，管理函数才是引入对象的最终目的**。

常见的教程在讲面向对象时，经常以狗、圆形、汽车等实体引入，告诉你如何用类模拟它们，比如在类中设置成员变量来表示狗的品种、圆的半径、汽车的品牌等，然后在类中设置成员函数来模拟狗叫、求面积、汽车启动这样的事件...这样的说法有利于你快速理解类和对象本身的含义，但你很快会发现你几乎不会在你的代码中用到这样"典型"的对象，你更多会看到的是，连接器、优化器、求解器等相对抽象的东西，它们看起来与其说是一个对象，不如说只是一组**有复杂内部状态的函数**。这也正是我们引入对象的动机：**将一组共享状态的函数连同它的状态封装起来，以整体的形式与外部交互**。

再谈闭包: 丐版的类

在引入对象前，我们并非没有工具进行一组函数的状态管理

我们可以使用闭包的结构来管理有相同依赖参数一组函数，并且定义一个生成闭包的函数来调整闭包中的参数配置，这样也就形成了一个丐版的类。反过来说，正是因为闭包实现的繁琐，我们使用类来完成我们的目标

1.类与实例对象

考虑这样一个场景，我们希望有一种方法来为不同代码段的执行计时，方便我们进行性能分析。已知python的time库允许我们获取当前时间，一个简单的想法是，在目标代码段前后分别获取当前时间，然后相减。这样做的缺点是必须手动管理获取到的时间，比如在循环外新建一个列表来存储。

我们想到，是否能像一个真正的计时器一样，把获取时间与存储时间、甚至分析时间等功能集成起来。我们可以把计时器的工作模式抽象出来：按下开始键开始计时，按下暂停键获得时长，支持多次按下暂停键进行多次计时，也支持重新开始计时。接下来，我们实现一个Timer类来模拟这些功能。

```
import time



'''

计时器

'''
```

```
class Timer(object):

        # 初始函数，创建类时即调用

        def __init__(self):

                self.times = []

                self.start()

        # 开始计时

        def start(self):

                self.tik = time.time()

        # 结束计时并返回结果

        def stop(self):

                self.times.append(time.time() - self.tik)

                return self.times[-1]

        # 计算平均时长

        def avg(self):

                return sum(self.times) / len(self.times)

        # 计算总时间

        def sum(self):

                return sum(self.times)

        # 计算前n段的时长

        def cumsum(self):

                return np.array(self.times).cumsum().tolist()
```

上述代码定义了一个类，类名为Timer，意味着你为你的程序定义了一种新的复合数据类型，你应该将Timer视为int、str、list这样的类型名

1.1 构造函数与析构函数

```
timer1 = Timer() # 创建一个Timer类型的实例对象
```

__init__函数会在你创建实例时被调用，self参数指调用者自己，用于访问调用者自己。你可能会觉得有点矛盾，调用者此时还没有被初始化，如何被访问呢？这个问题比较复杂，我们暂时不讨论，只需要记住必须传入self作为参数即可。

在Timer的实现中我们的**init**只要求传入self作为参数，被调用时不需要再传入任何参数了。我们也可以在**init**中加入其他参数，如下

```
class Timer(object):
        def __init__(self, max_times):
                self.max_times = max_times
        ...


timer1 = Timer(15)
```

事实上，构造函数的实现内容并没有硬性规定，但基本上用于完成以下任务

- 想要要用到哪些成员变量，给成员变量做初始化
- 若有外部参数传入，做必要的参数检查
- 若有必要，更新相关的静态变量

1.1.1 静态变量（类属性）

一种依赖于类而不依赖于实例的变量，用于记录例如实例个数这样的类级别的参数

如果想计算程序里同时存在的该类的实例数量，可以在**init**函数前创建静态变量num，在**init**和**del**时更新num，从而达到监控实例数量的效果

静态变量在类外部既可以用类名调用，也可以用实例名调用，在内部只能用类名调用

这里的**del**函数我们不常定义，它会在实例被释放时调用，做一些清理工作

```
class Timer(object):
        num = 0
        def __init__(self):
                Timer.num += 1
                if Timer.num > 10:
                        raise ValueError('同时存在太多实例对象')
                self.times = []
                self.start()
        def __del__(self):
                Timer.num -= 1
...


timer1 = Timer()
print(Timer.num)
print(timer1.num)
```

1.1.2 成员变量（实例属性）

上述代码中的times与tik都是成员变量，它们是与实例绑定的，**每个实例都有自己的一份成员变量**

在类里，我们访问成员变量的方式是*self.xx*；在类外，我们访问成员变量的方式是*实例名.xx*

在其他支持面向对象的语言中，权限是一个极为重要的概念，大家认为应该将成员变量尽可能设为私有的，即只能由类内部的代码访问，外部需要修改或访问时使用getter或setter函数来处理，来保证我们能追踪类状态的变化、保证成员变量的合法性等。

python的成员变量是默认公开的，也就是可以被外部访问，虽然python也支持对成员变量的权限进行设置，但我仍然推荐你直接使用公开的成员变量即可，这是google的python代码规范支持的做法，所以不必担心。如果你确实有保证成员变量合法性等需求，又不愿回到getter/setter的模式，可以自行了解property装饰器的使用

1.2 成员函数（实例方法）

代码中的start、stop即为成员函数，你可以注意到它们的第一个参数均为self,也说明这些函数内可能需要访问当前实例的成员变量。当然，在调用这些函数时，你不需要真的传入一个对象，它会自动把调用者作为参数传入。

```
timer1.start() # timer1 已经被自动传给了start

time.sleep(1)

res = timer1.stop() # timer1 已经被自动传给了stop

print(res)
```

这里的实现都没有用到其他参数，这并不是十分典型的情况。除self外，成员函数的定义与和普通的函数没有任何区别，也可以定义接受外部参数。

1.3 静态方法（类方法）

当然了，我们想定义的函数并不需要访问成员变量或其他成员函数，但我们又想让它作为类的一部分存在来方便管理，则我们可以将其定义为静态方法

```
class Timer():

        ...

        ...
```

```
        @staticmethod
        def my_func(arg1, arg2):
                ...
```

同静态变量一样，静态方法也是类级别的，不依赖于任何实例，故可以用类名来调用，对是否有实例存在没有要求。

2. 继承(了解)

继承本是面向对象的核心特性，但对python来说它并不是一个必不可少的特性,因此我们只简单了解。

2.1 用继承提供统一接口

```
# 基类
class Animal:
        def speak(self):
                pass


# 子类，继承于父类Animal
class Dog(Animal):
        def speak(self):
                print("Woof!")


class Cat(Animal):
```

```
        def speak(self):
                print("Meow!")


def animal_sound(animal: Animal):
        animal.speak()


dog = Dog()
cat = Cat()


animal_sound(dog) # 输出: Woof!
animal_sound(cat) # 输出: Meow!
```

你会发现这里就算没有Animal类对Dog、Cat的接口(比如这里的speak方法)进行约束，我们的代码似乎也可以正常运行。这是因为python的函数并不对传入的参数进行类型检查，因此使用抽象基类来提供统一接口并不是必要的。

2.2 用组合替代继承实现代码复用

继承的另一个好处是便于代码复用，而这一优点事实上很大程度可以被组合替代，下面是用组合实现代码复用的一个实例。

```
class Engine:
        def start(self):
                print("引擎启动")
```

```
class Wheels:

        def roll(self):

                print("车轮滚动")



class Car:

        def __init__(self):

                self.engine = Engine()

                self.wheels = Wheels()



def drive(self):

        self.engine.start()

        self.wheels.roll()

        print("汽车行驶")



class Bicycle:

        def __init__(self):

                self.wheels = Wheels()



        def ride(self):

                self.wheels.roll()

                print("自行车骑行")
```

```
# 使用组合

my_car = Car()

my_car.drive()



my_bicycle = Bicycle()

my_bicycle.ride()
```

2.3 适合使用继承的场景

在我们有幸参与大型工程的开发前，我们或许很难感受到使用继承设计自己的代码的好处。我们使用继承，更多只是为了在他人写好的代码基础上做一些自定义的改动。

```
import torch

import torch.nn as nn



# 继承nn.Module,保留参数管理、自动微分、序列化等别人实现好的功能

class LinearModel(nn.Module):

        def __init__(self, input_size, output_size):

                # 用super调用父类的构造函数，保证从父类继承的功能能正常实现

                super(LinearModel, self).__init__()

                self.linear = nn.Linear(input_size, output_size)
```

```
        def forward(self, x):
                return self.linear(x)


# 创建模型实例
input_size = 10
output_size = 5
model = LinearModel(input_size, output_size)


# 打印模型结构
print(model)


# 模型的参数
for name, param in model.named_parameters():
        print(f"Parameter name: {name}, Shape: {param.shape}")
```

3. 其他可以探索的内容

以上仅仅是python面向对象中最基础与常用的特性，若你感兴趣，也可以继续探索以下内容

- 继承: 多重继承、抽象基类
- @property
- 类属性的绑定与**slots**
- 定制类

- 元类