مباحث پیشرفته در برنامه نویسی شی گرا ¶

نامگذاری کلاس ها در پایتون

قبل از ادامه دادن به کلاس ها، باید با برخی از قواعد نامگذاری مهم آشنا شوید که پایتون در زمینه کلاس ها استفاده می کند. پایتون یک زبان انعطاف پذیر است که آزادی را دوست دارد و دوست ندارد محدودیت های صریح داشته باشد. به همین دلیل، این زبان و جامعه برنامه نویسان آن، بر روی روش های مرسوم نام گذاری تکیه می کنند تا به جای محدودیت ها، از قراردادها استفاده کنند.

توجه: بیشتر برنامه نویسان پایتون از قانون snake_case استفاده می کنند، که شامل استفاده از خط تیره (_) برای جداسازی چندین کلمه است. با این حال، قانون نامگذاری توصیه شده برای کلاس های پایتون PascalCase است، به طوری که هر کلمه با حروف بزرگ شروع شود.

در دو بخش زیر، در مورد دو قانون نامگذاری مهم صفات کلاس آموزش خواهید دید.

Public vs Non-Public Members

اولین قانون نامگذاری که باید در مورد آن اطلاعاتی داشته باشید، مربوط به این است که پایتون مانند جاوا و زبان های دیگر، بین ویژگی های خصوصی، محافظت شده و عمومی تفاوت قائل نیست. در پایتون، تمام ویژگی ها به یک شکل یا دیگر قابل دسترسی هستند. با این حال، پایتون یک قانون نامگذاری خوب دارد که باید از آن استفاده کنید تا اعلام کنید که یک ویژگی یا روش برای استفاده از خارج از کلاس یا شیء حاوی آن مناسب نیست.

قانون نامگذاری شامل اضافه کردن خط تیره به عنوان پیشوند به نام عضو است. بنابراین، در یک کلاس پایتون، شما قانون زیر را خواهید داشت:

عضویت	نام گذاری	مثال
Public	از الگوی معمولی نام گذاری استفاده نمایید.	`radius` و `calculate_area()`
Non-public	از _ بیش از نام ها استفاده نمایید.	`_radius` و `_calculate_area()`

عضو های عمومی بخش رسمی رابط یا API کلاس های شما هستند، در حالی که اعضای غیر عمومی قرار نبوده بخشی از آن API بخش باشد. این بدان معناست که شما نباید از اعضای غیر عمومی خارج از کلاس تعریف کنید.

لطفا توجه داشته باشید که قانون نامگذاری دوم فقط نشان می دهد که ویژگی به صورت مستقیم از خارج از کلاس حاوی استفاده نمی شود. با این حال، دسترسی مستقیم را جلو نمی گیرد. به عنوان مثال، می توانید obj._name را اجرا کنید و به محتوای name _ . دسترسی پیدا کنید. با این حال، این شیوه بد است و باید از آن خودداری کنید.

عضو های غیر عمومی فقط برای پشتیبانی از پیاده سازی داخلی یک کلاس خاص وجود دارند و ممکن است در هر زمان حذف شوند، بنابراین نباید به آنها اعتماد کنید. وجود این اعضا به نحوی است که به پیاده سازی کلاس بستگی دارد. بنابراین، شما نباید در کد مشتری به صورت مستقیم از آنها استفاده کنید. در صورت استفاده، کد شما ممکن است در هر لحظه خراب شود.

In [1]:

```
class Account:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self._balance = balance

    def __str__(self):
        return f'Account owner:   {self.owner}\nAccount balance: ${self._balance}'

    def get_balance(self):
        return self._balance

    def deposit(self, dep_amt):
        self._balance += dep_amt
        print('Deposit Accepted')

    def withdraw(self, wd_amt):
        if self._balance >= wd_amt:
            self._balance -= wd_amt
            print('Withdrawal Accepted')
        else:
            print('Funds Unavailable!')
```

In [2]:

```
acct1 = Account('Jose',100)
```

In [3]:

```
acct1.get_balance()
```

Out[3]:

```
100
```

In [4]:

```
acct1._balance
```

Out[4]:

```
100
```

Name Mangling

نام Mangling یکی دیگر از قواعد نامگذاری که می توانید در کلاس های پایتون ببینید و استفاده کنید، اضافه کردن دو زیرخط به نام و متد ها است. این قانون نامگذاری باعث فعال شدن چیزی به نام name mangling می شود.

Name mangling یک تبدیل خودکار نام است که نام کلاس را به نام عضو اضافه می کند، مانند _ClassName__attribute یا _ClassName__method. این باعث مخفی شدن نام ها می شود. به عبارت دیگر، نام های mangling برای دسترسی مستقیم در دسترس نیستند. آنها بخشی از API عمومی یک کلاس نیستند.

برای مثال، به کلاس نمونه زیر توجه کنید:

In [5]:

```
class Sample:
    def __init__(self):
        self.__x = 1
        self.y = 2

    def __foo(self):
        print("This is a private method.")

    def bar(self):
        print("This is a public method.")
        self.__foo()

obj1 = Sample()
```

In [6]:

```
obj1.y
```

Out[6]:

```
2
```

In [7]:

```
obj1.__x
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call las
t)
Cell In[7], line 1
----> 1 obj1.__x

AttributeError: 'Sample' object has no attribute '__x'
```

در این کلاس، x__. و foo()__ . دو زیرخط دارند، بنابراین نام های آنها به Sample__x._ و Sample__foo()._ تغییر می کند، همانطور که در خطوط برجسته مشاهده می کنید. پایتون به صورت خودکار پیشوند Sample_ را به هر دو نام اضافه کرده است. به دلیل این تغییر نام داخلی، نمی توانید با استفاده از نام های اصلی خود به ویژگی ها از خارج از کلاس دسترسی پیدا کنید. اگر سعی کنید این کار را انجام دهید، خطای AttributeError دریافت می کنید.

توجه: در مثال بالا، شما از تابع vars() داخلی استفاده می کنید، که یک دیکشنری از تمام عضو های مرتبط با شیء داده شده را برمی گرداند. این دیکشنری نقش مهمی در کلاس های پایتون بازی می کند. در بخش ویژگی dict. بیشتر درباره آن یاد خواهید گرفت.

این رفتار داخلی نام ها را پنهان می کند و حس خصوصی بودن ویژگی یا روش را ایجاد می کند. با این حال، آنها به صورت سختگیرانه خصوصی نیستند. شما می توانید به آنها از طریق نام های mangling دسترسی پیدا کنید:

In [8]:

```
1 vars(obj1)
```

Out[8]:

```
{'_Sample__x': 1, 'y': 2}
```

In [9]:

```
1 obj1._Sample__x
```

Out[9]:

```
1
```

هنوز هم می توانید به ویژگی ها یا روش های named-mangled با استفاده از نام های mangling دسترسی پیدا کنید، اگرچه این یک شیوه بد است و باید از آن در کد خود دوری کنید. اگر نامی را که از این شیوه در کد شخص دیگری مشاهده می کنید، ببینید، آنگاه سعی نکنید کد را به استفاده از نام خارج از کلاس حاوی آن وادار کنید.

Name mangling به خصوص زمانی مفید است که می خواهید اطمینان حاصل کنید یک ویژگی یا روش داده شده به طور تصادفی بازنویسی نمی شود. این یک راه برای جلوگیری از تداخل نام ها بین کلاس ها یا زیرکلاس ها است. همچنین برای جلوگیری از زیرکلاس های بازنویسی روش های بهینه سازی شده برای عملکرد بهتر مفید است.

واو! تا اینجای کار، شما مبانی کلاس های پایتون و چند تا بیشتر را فرا گرفتید. در لحظات آینده، به عمق کارکرد کلاس های پایتون خواهید پرداخت. اما قبل از آن، زمان آن رسیده است تا به چند دلیل بپردازید که باید درباره کلاس ها چیزهای بیشتر یاد بگیرید و در پروژه های پایتون خود از آنها استفاده کنید.

مزایای استفاده از کلاس ها در پایتون

آیا استفاده از کلاس ها در پایتون ارزش دارد؟ قطعاً! کلاس ها بلوک های ساختمانی برنامه نویسی شی گرا در پایتون هستند. آنها به شما اجازه می دهند تا از قدرت پایتون در حین نوشتن و سازماندهی کد خود بهره ببرید. با یادگیری درباره کلاس ها، شما قادر خواهید بود از تمام مزایایی که آنها فراهم می کنند، استفاده کنید. با کلاس ها، شما می توانید:

- مدل و حل مسائل پیچیده واقعی: در بسیاری از موارد، شی های موجود در کد شما به شی های واقعی جهان نقش می بینند. این می تواند به شما در فکر کردن درباره مسائل پیچیده کمک کند، که نتیجه آن بهبود روش حل مسائل برنامه نویسی شما خواهد بود.
- باز استفاده از کد و جلوگیری از تکرار: شما می توانید سلسله مراتبی از کلاس های مرتبط را تعریف کنید. کلاس های پایین سلسله مراتب، عملکردهای مشترک را فراهم می کنند که بعدها در زیرکلاس های سلسله مراتب استفاده مجدد خواهید کرد. این به شما اجازه می دهد تکرار کد را کم کنید و باز استفاده از کد را ترویج دهید.
- پوشش دادن داده های و رفتار های مرتبط در یک جانب: شما می توانید از کلاس های پایتون برای گروه بندی وابستگی های و روش های مرتبط در یک جانب، چگال شده استفاده کنید. این به شما در سازماندهی بهتر کدهای خود با استفاده از جانب های قابل استفاده و خودکار استفاده از آنچه حتی در پروژه های چندگانه قابل استفاده است، به شما کمک خواهد کرد.
- پوشش جزئیات پیاده سازی مفصل و اشیاء: شما می توانید از کلاس ها برای پوشش جزئیات پیاده سازی مفصل و اشیاء اصلی استفاده کنید. این به شما در فراهم کردن رابط های (API ها) قابل فهم برای کاربران خود برای پردازش داده های و رفتار های پیچیده کمک خواهد کرد.
- با رابط های مشترک چند ریختی را باز کنید: شما می توانید یک رابط خاص را در چند کلاس کمی متفاوت پیاده سازی کنید و آنها را به صورت قابل تعویض در کد خود استفاده کنید. این باعث می شود کد شما انعطاف پذیرتر و قابل تطبیق تر شود.

به طور خلاصه، کلاس های پایتون می توانند به شما در نوشتن کدهای سازماندهی شده، ساختار یافته، قابل نگهداری، قابل استفاده مجدد، انعطاف پذیر و دوستانه به کاربران شما کمک کنند. آنها یک ابزار عالی برای دسترسی به آنچه در دستان

Class attributes vs Instance attributes

همانطور که یاد گرفتید، کلاس ها زمانی عالی هستند که باید داده ها و رفتار را در یک موجودیت واحد بسته بندی کنید. داده ها به صورت ویژگی ها و رفتار به عنوان متدها ارائه خواهد شد. شما در حال حاضر ایده ای از ویژگی دارید. حالا وقت آن است که عمیق تر به آن بپردازید که چگونه می توانید ویژگی ها را در کلاس های سفارشی خود اضافه، دسترسی و تغییر دهید.

ابتدا باید بدانید که کلاس های شما می توانند دو نوع ویژگی در پایتون داشته باشند:

- ویژگی های کلاس: یک ویژگی کلاس، متغیری است که به صورت مستقیم در بدنه کلاس تعریف می شود. ویژگی های کلاس متعلق به کلاس حاوی آن است. داده های آن مشترک بین کلاس و تمام نمونه های آن است.
- ویژگی های نمونه: یک نمونه، متغیری است که درون یک متد تعریف می شود. ویژگي‌هاي نمونه به یک نمونه خاص از یک کلاس خاص تعلق دارد. داده های آن فقط برای آن نمونه در دسترس است و حالت آن را تعریف مي‌كند.

هر دو نوع ویژگي، مورد استفاده خود را دارند. ویژگي‌هاي نمونه، به طور قطع، پرکاربردترین نوع ویژگي است که در برنامه نویسي روزانه خود استفاده خواهید کرد، اما ویژگي‌هاي کلاس نیز مفید خواهد بود.

Class attributes

ویژگی های کلاس، متغیرهایی هستند که به صورت مستقیم در بدنه کلاس تعریف می شوند اما خارج از هر متدی هستند. این ویژگی ها به کلاس خود و نه به شیء خاصی از آن کلاس مربوط هستند.

تمام شیء هایی که از یک کلاس خاص ایجاد می کنید، ویژگی های کلاس را با مقادیر اصلی یکسان به اشتراک می گذارند. به دلیل این، اگر یک ویژگی کلاس را تغییر دهید، آن تغییر بر تمام شیء های مشتق شده تأثیر خواهد گذاشت.

به عنوان مثال، فرض کنید می خواهید یک کلاس ایجاد کنید که شمارش داخلی نمونه های ایجاد شده را نگه دارد. در آن صورت، می توانید از ویژگي‌هاي کلاس استفاده کنید:

In [10]:

```
class ObjectCounter:
    num_instances = 0
    def __init__(self):
        #ObjectCounter.num_instances += 1
        type(self).num_instances += 1
```

In [11]:

```
x1 = ObjectCounter()
x2 = ObjectCounter()
x3 = ObjectCounter()
x4 = ObjectCounter()
```

In [12]:

```
ObjectCounter.num_instances
```

Out[12]:

```
4
```

In [13]:

```
counter = ObjectCounter()
counter.num_instances
```

Out[13]:

```
5
```

In [14]:

```
ObjectCounter.num_instances
```

Out[14]:

```
5
```

ObjectCounter یک ویژگی کلاس .num_instances را نگه می دارد که به عنوان یک شمارنده از نمونه ها عمل می کند. هنگامی که پایتون این کلاس را تجزیه و تحلیل می کند، شمارنده را به صفر مقداردهی اولیه می کند و آن را تنها می گذارد. ایجاد نمونه های این کلاس به معنای خودکار فراخوانی روش ()__ init __ . و افزایش .num_instances به 1 است.

مهم است که بدانید می توانید از ویژگی های کلاس با استفاده از کلاس یا یکی از نمونه های آن دسترسی پیدا کنید. به همین دلیل می توانید از شیء شمارنده برای بازیابی مقدار .num_instances استفاده کنید. با این حال، اگر نیاز به تغییر ویژگی کلاس دارید، باید از خود کلاس به جای یکی از نمونه های آن استفاده کنید.

به عنوان مثال، اگر از self برای تغییر .num_instances استفاده کنید، با ساخت یک ویژگی نمونه جدید، ویژگی کلاس اصلی را بازنویسی خواهید کرد:

In [15]:

```
class ObjectCounter:
    num_instances = 0
    def __init__(self):
        self.num_instances += 1
```

In [17]:

```
x1 = ObjectCounter()
x2 = ObjectCounter()
x3 = ObjectCounter()
x4 = ObjectCounter()

ObjectCounter.num_instances
```

Out[17]:

```
0
```

In [18]:

```
x1.num_instances
```

Out[18]:

```
1
```

In [19]:

```
x2.num_instances
```

Out[19]:

```
1
```

شما نمی توانید از طریق نمونه های کلاس، ویژگی های کلاس را تغییر دهید. این کار باعث ایجاد ویژگی های نمونه جدید با همان نام ویژگی های کلاس اصلی می شود. به همین دلیل ObjectCounter.num_instances در این مثال 0 برمی گرداند. شما ویژگی کلاس را در خط برجسته شده بازنویسی کرده اید.

بطور کلی، باید از ویژگی های کلاس برای به اشتراک گذاشتن داده ها بین نمونه های یک کلاس استفاده کنید. هر تغییر در یک ویژگی کلاس به تمام نمونه های آن کلاس قابل مشاهده است.

Instance Attributes

ویژگی های نمونه، متغیرهایی هستند که به یک شیء خاص از یک کلاس داده شده اند. مقدار ویژگی نمونه به خود شیء متصل است. بنابراین، مقدار ویژگی به نمونه خود اختصاص دارد.

پایتون به شما اجازه می دهد تا ویژگی های جدید را به شیء های موجود که قبلاً ایجاد کرده اید، اضافه کنید. با این حال، شما بیشتر ویژگی های نمونه را درون متد های نمونه تعریف می کنید، که متد هایی هستند که self را به عنوان آرگومان اول خود دریافت می کنند.

برای مثال کلاس car زیر را در نظر بگیرید که تعدادی ویژگی نمونه دارد

In [20]:

```
class Car:
    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
        self.started = False
        self.speed = 0
        self.max_speed = 200
```

در این کلاس، شما هفت ویژگی نمونه را درون __init__() تعریف می کنید. ویژگی های .year، .model، make و .color مقادیر را از آرگومان های __init__() دریافت می کنند، که آرگومان هایی هستند که باید برای سازنده کلاس ، Car() ، به منظور ایجاد شیء های محسوس، گذاشته شوند.

سپس، شما ویژگی های speed.، started. و max_speed. را با مقادیر منطقی که از کاربر نمی آیند، به صورت صریح مقداردهی اولیه می کنید.

In [21]:

```
toyota_camry = Car("Toyota", "Camry", 2022, "Red")
ford_mustang = Car("Ford", "Mustang", 2022, "Black")
```

In [22]:

```
print(toyota_camry.make, toyota_camry.model, toyota_camry.year, toyota_camry.max_spe
```

```
Toyota Camry 2022 200
```

In [23]:

```
print(ford_mustang.make, ford_mustang.model, ford_mustang.year, ford_mustang.max_spe
```

```
Ford Mustang 2022 200
```

In [24]:

```
Car.make
```

```
---------------------------------------------------------------------------
-
AttributeError                            Traceback (most recent call las
t)
Cell In[24], line 1
----> 1 Car.make

AttributeError: type object 'Car' has no attribute 'make'
```

The dict. Attribute

در پایتون، هر دو کلاس ها و نمونه ها دارای یک ویژگی خاص به نام `__dict__` هستند. این ویژگی یک دیکشنری را که حاوی اعضای قابل نوشتن کلاس یا نمونه زیرین است، نگه می دارد. به یاد داشته باشید، این اعضا می توانند ویژگی ها یا متد ها باشند. هر کلید در `__dict__` یک نام ویژگی را نشان می دهد. مقدار مرتبط با یک کلید خاص، مقدار ویژگی مربوطه را نشان می دهد.

در یک کلاس، **dict**. شامل ویژگی های کلاس و متد ها خواهد بود. در یک نمونه، `__dict__` ویژگی های نمونه را نگه می دارد.

زمانی که شما به عضو کلاس از طریق شیء کلاس دسترسی پیدا می کنید، پایتون به طور خودکار برای نام عضو در `__dict__` کلاس جستجو می کند. اگر نام در آنجا وجود نداشته باشد، آنگاه شما یک AttributeError دریافت خواهید کرد.

بطور مشابه، زمانی که شما به عضو نمونه از طریق یک شئ محسوس از یک کلاس دسترسی پیدا می کنید، پایتون برای نام عضو در `__dict__` نمونه جستجو می کند. اگر نام در آنجا ظاهر نشود، پایتون در `__dict__` کلاس جستجو می کند. اگر نام پیدا نشود، آنگاه شما یک NameError در خودتان خواهید دید.

اینجا چندین کلاس برای آشنایی بیشتر در نظر گرفته ایم تا نحوه عملکرد این مکانیزم را بهتر درک نمایید.:

In [25]:

```
class SampleClass:
    class_attr = 100

    def __init__(self, instance_attr):
        self.instance_attr = instance_attr

    def method(self):
        print(f"Class attribute: {self.class_attr}")
        print(f"Instance attribute: {self.instance_attr}")
```

در این کلاس، شما یک ویژگی کلاس با مقدار 100 تعریف می کنید. در متد __init__() ، شما یک ویژگی نمونه را تعریف می کنید که مقدار آن از ورودی کاربر است. در نهایت، شما یک روش برای چاپ هر دو ویژگی تعریف می کنید.

حالا وقت آن رسیده است تا محتوای **dict** را در شئ کلاس بررسی کنید. به جلو بروید و کد زیر را اجرا کنید:

In [26]:

```
SampleClass.class_attr
```

Out[26]:

```
100
```

In [27]:

```
SampleClass.__dict__
```

Out[27]:

```
mappingproxy({'__module__': '__main__',
              'class_attr': 100,
              '__init__': <function __main__.SampleClass.__init__(self, in
stance_attr)>,
              'method': <function __main__.SampleClass.method(self)>,
              '__dict__': <attribute '__dict__' of 'SampleClass' objects>,
              '__weakref__': <attribute '__weakref__' of 'SampleClass' obj
ects>,
              '__doc__': None})
```

In [28]:

```
SampleClass.__dict__["class_attr"]
```

Out[28]:

```
100
```

خطوط مشخص شده نشان می دهند که هر دو ویژگی کلاس و متد در دیکشنری **dict**. کلاس هستند. توجه کنید که چگونه می توانید از **dict**. برای دسترسی به مقدار ویژگی های کلاس با مشخص کردن نام ویژگی در پرانتز مربعی استفاده کنید، به عنوان معمول در یک دیکشنری به کلید ها دسترسی پیدا می کنید.

در نمونه ها، دیکشنری **dict**. فقط شامل ویژگی های نمونه خواهد بود:

In [29]:

```
instance = SampleClass("Hello!")
```

In [30]:

```
instance.instance_attr
```

Out[30]:

```
'Hello!'
```

In [31]:

```
instance.method()
```

```
Class attribute: 100
Instance attribute: Hello!
```

In [32]:

```
instance.__dict__
```

Out[32]:

```
{'instance_attr': 'Hello!'}
```

In [33]:

```
vars(instance)
```

Out[33]:

```
{'instance_attr': 'Hello!'}
```

In [34]:

```
instance.__dict__["instance_attr"]
```

Out[34]:

```
'Hello!'
```

In [36]:

```
instance.__dict__["instance_attr"] = "Goodbye"
```

In [37]:

```
instance.instance_attr
```

Out[37]:

```
'Goodbye'
```

در این مثال، دیکشنری **dict**. نمونه شامل instance_attr. و مقدار خاص آن برای شیء در دست است. باز هم، شما می توانید با استفاده از **dict**. و نام ویژگی در پرانتز مربعی به هر ویژگی نمونه موجود دسترسی پیدا کنید.

شما می توانید بصورت پویا دیکشنری **dict**. نمونه را تغییر دهید. این بدان معناست که شما می توانید ارزش ویژگی های نمونه موجود را از طریق **dict**. تغییر دهید، همانطور که در مثال نهایی بالا انجام دادید. شما حتی می توانید با استفاده از دیکشنری **dict**. خود نمونه، ویژگی های جدید را به یک نمونه اضافه کنید.

استفاده از **dict**. برای تغییر مقادیر ویژگی های نمونه، به شما اجازه می دهد که در هنگام سیم بندي ویژگي هاي در پایتون، خطای RecursionError را که به علت بازگشت تابع بیش از حد به وجود می آید، جلوگیری کنید.

ویژگی های کلاسی و نمونه ای پویا

در پایتون، شما می توانید ویژگی های جدید را به کلاس ها و نمونه های خود به صورت دینامیک اضافه کنید. این امکان به شما اجازه می دهد تا در پاسخ به نیازها یا زمینه های تغییر کننده، داده ها و رفتار جدید را به کلاس ها و اشیاء خود اضافه کنید. همچنین به شما اجازه می دهد تا کلاس های موجود را برای نیازهای خاص و پویا سفارشی کنید.

به عنوان مثال، شما می توانید از این ویژگی پایتون استفاده کنید زمانی که در زمان تعریف خود کلاس، ویژگی های مورد نیاز یک کلاس مشخص نمی باشد.

کلاس زیر را در نظر بگیرید، که قصد دارد یک ردیف داده از جدول پایگاه داده یا یک فایل CSV را ذخیره کند:

In [38]:

```
class Record:
    """Hold a record of data."""
```

در این کلاس، هیچ ویژگی یا متد تعریف نکرده اید زیرا نمی دانید کلاس چه داده هایی را ذخیره می کند. خوشبختانه، شما می توانید ویژگی ها و حتی متد ها را به این کلاس به صورت دینامیک اضافه کنید.

به عنوان مثال، فرض کنید یک ردیف داده را از یک فایل employees.csv با استفاده از csv.DictReader خوانده اید. این کلاس داده ها را می خواند و آنها را در یک شئ شبیه به دیکشنری برمی گرداند. حال فرض کنید که شما دارای دیکشنری داده های زیر هستید:

In [40]:

```
john = {
    "name": "John Doe",
    "position": "Python Developer",
    "department": "Engineering",
    "salary": 80000,
    "hire_date": "2020-01-01",
    "is_manager": False,
}
```

سپس، شما می خواهید این داده ها را به یک نمونه از کلاس Record خود اضافه کنید و باید هر فیلد داده را به عنوان ویژگی نمونه نشان دهید. به این صورت می توانید این کار را انجام دهید:

In [41]:

```
john_record = Record()

for field, value in john.items():
    setattr(john_record, field, value)
```

In [42]:

```
1 john_record.name
```

Out[42]:

```
'John Doe'
```

In [43]:

```
1 john_record.department
```

Out[43]:

```
'Engineering'
```

In [44]:

```
1 john_record.__dict__
```

Out[44]:

```
{'name': 'John Doe',
 'position': 'Python Developer',
 'department': 'Engineering',
 'salary': 80000,
 'hire_date': '2020-01-01',
 'is_manager': False}
```

همچمنین می توانیم با استفاده از نوتشین نقطه ای متدها و ویژگی های جدیدی را اضافه نماییم. در واقع می توانیم یک شی را با هر ویژگی یا متدی از صفر بسازیم

In [45]:

```
1 class User:
2     pass
3 
4 jane = User()
5 
6 jane.name = "Jane Doe"
7 jane.job = "Data Engineer"
8 jane.__dict__
```

Out[45]:

```
{'name': 'Jane Doe', 'job': 'Data Engineer'}
```

In [46]:

```
# Add methods dynamically
def __init__(self, name, job):
    self.name = name
    self.job = job

User.__init__ = __init__

User.__dict__
```

Out[46]:

```
mappingproxy({'__module__': '__main__',
              '__dict__': <attribute '__dict__' of 'User' objects>,
              '__weakref__': <attribute '__weakref__' of 'User' objects>,
              '__doc__': None,
              '__init__': <function __main__.__init__(self, name, job)>})
```

In [47]:

```
linda = User("Linda Smith", "Team Lead")
linda.__dict__
```

Out[47]:

```
{'name': 'Linda Smith', 'job': 'Team Lead'}
```

@Class Methods With classmethod

شما همچنین می توانید متدهای کلاس را به کلاس های پایتون سفارشی خود اضافه کنید. یک متد کلاس یک متد است که شیء کلاس را به عنوان اولین آرگومان خود به جای self دریافت می کند. در این صورت، آرگومان باید با cls نامیده شود، که همچنین یک قرارداد قوی در پایتون است. بنابراین، شما باید به آن چسبیده باشید.

شما می توانید با استفاده از دکوراتور @classmethod متدهای کلاس را ایجاد کنید. فراهم کردن سازنده های چندگانه برای کلاس های شما، یکی از معمول ترین موارد استفاده از متدهای کلاس در پایتون است.

به عنوان مثال، فرض کنید می خواهید یک سازنده جایگزین به ThreeDPoint خود اضافه کنید تا بتوانید به سرعت نقاط را از tuples یا لیست های مختصات ایجاد کنید:

In [48]:

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

    def __iter__(self):
        yield from (self.x, self.y, self.z)

    @classmethod
    def from_sequence(cls, sequence):
        return cls(*sequence)

    def __repr__(self):
        return f"{type(self).__name__}({self.x}, {self.y}, {self.z})"
```

در متد کلاس .from_sequence()، یک دنباله از مختصات را به عنوان آرگومان دریافت می کنید، یک شی ThreeDPoint از آن ایجاد می کنید و شی را به فراخواننده باز می گردانید. برای ایجاد شی جدید، از آرگومان cls استفاده می کنید که حاوی یک مرجع ضمنی به کلاس فعلی است که پایتون به صورت خودکار در متد شما درج می کند.

اینجاست که چگونه این متد کلاس کار می کند:

In [51]:

```
yy = ThreeDPoint(9, -6, 3)
```

In [52]:

```
yy
```

Out[52]:

```
ThreeDPoint(9, -6, 3)
```

In [53]:

```
x = ThreeDPoint.from_sequence((4, 8, 16))
```

In [54]:

```
x
```

Out[54]:

```
ThreeDPoint(4, 8, 16)
```

In [55]:

```
ThreeDPoint.from_sequence((4, 8, 16))
```

Out[55]:

```
ThreeDPoint(4, 8, 16)
```

In [56]:

```
point = ThreeDPoint(7, 14, 21)
point.from_sequence((3, 6, 9))
```

Out[56]:

```
ThreeDPoint(3, 6, 9)
```

In [57]:

```
point
```

Out[57]:

```
ThreeDPoint(7, 14, 21)
```

در این مثال، کلاس ThreeDPoint را به طور مستقیم برای دسترسی به متد کلاس ()from_sequence. استفاده می کنید. توجه داشته باشید که همچنین می توانید با استفاده از یک نمونه محکم، مانند point در مثال، به متد دسترسی پیدا کنید. در هر یک از تماس های ()from_sequence.، شما یک نمونه کاملاً جدید از ThreeDPoint دریافت خواهید کرد. با این حال، برای بهترین وضوح و جلوگیری از ابهام، باید به متدهای کلاس از طریق نام کلاس مربوطه دسترسی پیدا کنید.

static Methods With @staticmethod

کلاس های پایتون شما همچنین می توانند دارای متدهای استاتیک باشند. این متدها نمونه یا کلاس را به عنوان آرگومان دریافت نمی کنند. بنابراین، آنها توابع معمولی هستند که در یک کلاس تعریف شده اند. شما همچنین می توانید آنها را به عنوان تابع مستقل خارج از کلاس تعریف کرده باشید.

به طور معمول، به جای یک تابع معمولی خارج از کلاس، یک متد استاتیک را تعریف خواهید کرد زمانی که آن تابع به صورت نزدیک با کلاس شما مرتبط است و می خواهید آن را برای راحتی یا سازگاری با API کد خود بسته بندی کنید. به خاطر داشته باشید که فراخوانی یک تابع از فراخوانی یک متد کمی متفاوت است. برای فراخوانی یک متد، شما باید یک کلاس یا شیء را مشخص کنید که این متد را فراهم می کند.

اگر می خواهید یک متد استاتیک در یکی از کلاس های سفارشی خود بنویسید، آنگاه باید از دکوراتور @staticmethod استفاده کنید. به روش ()show_intro_message. در زیر نگاه کنید:

In [58]:

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

    def __iter__(self):
        yield from (self.x, self.y, self.z)

    @classmethod
    def from_sequence(cls, sequence):
        return cls(*sequence)

    @staticmethod
    def show_intro_message(name):
        print(f"Hey {name}! This is your 3D Point!")

    def __repr__(self):
        return f"{type(self).__name__}({self.x}, {self.y}, {self.z})"
```

متد استاتیک show_intro_message(). یک نام به عنوان آرگومان دریافت می کند و پیامی را روی صفحه نمایش چاپ می کند. توجه داشته باشید که این فقط یک مثال بازیگوشانه از نحوه نوشتن متدهای استاتیک در کلاس های شما است.

متدهای استاتیک مانند show_intro_message(). بر روی نمونه فعلی، self، یا کلاس فعلی، cls عمل نمی کنند. آنها به عنوان توابع مستقل در یک کلاس قرار دارند. به طور معمول، آنها را درون یک کلاس قرار می دهید زمانی که به صورت نزدیک با آن کلاس مرتبط هستند اما لزوماً بر کلاس یا نمونه های آن تأثیر ندارند.

اینجاست که چگونگی عملکرد این متد است:

In [59]:

```
ThreeDPoint.show_intro_message("Pythonista")
```

```
Hey Pythonista! This is your 3D Point!
```

In [60]:

```
point = ThreeDPoint(2, 4, 6)
```

In [61]:

```
point.show_intro_message("Python developer")
```

```
Hey Python developer! This is your 3D Point!
```

Getter and Setter Methods vs Properties

زبان‌های برنامه‌نویسی مانند جاوا و سی پلاس‌پلاس ویژگی‌های خود را به عنوان بخشی از رابط برنامه‌نویسی عمومی کلاس‌های خود نشان نمی‌دهند. به جای آن، این زبان‌های برنامه‌نویسی از روش‌های getter و setter برای دسترسی به ویژگی‌ها استفاده فراوان می‌کنند.

استفاده از روش‌های دسترسی به و به‌روزرسانی ویژگی، encapsulation را ترویج می‌دهد. encapsulation یک اصل OOP بنیادین است که توصیه می‌کند حالت یا داده یک شیء را از دنیای خارج، جلوگیری از دسترسی مستقیم حفاظت کند. حالت شئ فقط از طریق یک رابط عمومی که شامل متدهای getter و setter است، قابل دسترس است.

به عنوان مثال، فرض کنید جاوجوب Person یا ویژگی .name دارید. شما می توانید .name را به عنوان یک ویژگی

In [68]:

```
class Person:
    def __init__(self, name):
        self.set_name(name)

    def get_name(self):
        return self._name

    def set_name(self, value):
        self._name = value
```

در این مثال، ()get_name. روش getter است و به شما اجازه می دهد به ویژگی پایه دسترسی پیدا کنید. به طور مشابه، ()set_name. روش setter است و به شما اجازه می دهد تا مقدار فعلی _name. را تغییر دهید. ویژگی _name. غیرعمومی است و جایی است که داده های واقعی ذخیره می شوند.

اینجاست که چگونه می توانید از کلاس Person خود استفاده کنید:

In [69]:

```
jane = Person("Jane")
jane.get_name()
```

Out[69]:

```
'Jane'
```

In [70]:

```
jane.set_name("Jane Doe")
jane.get_name()
```

Out[70]:

```
'Jane Doe'
```

In [71]:

```
vars(jane)
```

Out[71]:

```
{'_name': 'Jane Doe'}
```

In [72]:

```
jane._name
```

Out[72]:

```
'Jane Doe'
```

در اینجا، شما یک نمونه از Person با استفاده از سازنده کلاس و "Jane" به عنوان نام مورد نیاز ایجاد می کنید. این بدان معناست که می توانید از روش ()get_name. برای دسترسی به نام Jane و روش ()set_name. برای به روزرسانی آن استفاده کنید.

الگوی getter و setter در زبان هایی مانند جاوا و C++ رایج است. علاوه بر ترویج تعلقات و API های متمرکز بر فراخوانی روش، این الگو به شما امکان می دهد تا به سرعت رفتار شبیه به تابع را به ویژگی های خود اضافه کنید بدون اینکه تغییرات شکست آور در API های خود را به دنبال داشته باشید.

با این حال، این الگو در جامعه پایتون کمتر محبوب است. در پایتون، کاملاً عادی است که ویژگی ها را به عنوان بخشی از API عمومی یک شئ فراهم کنید. اگر همیشه نیاز به اضافه کردن رفتار شبیه به تابع به بالای یک ویژگی عمومی دارید، آنگاه می توانید آن را به جای آنکه با یک متد API جایگزین شود، به یک خصوصیت تبدیل کنید.

در اینجاست که بسیاری از توسعه دهندگان پایتون کلاس Person را خود را مینویسند:

In [65]:

```
class Person:
    def __init__(self, name):
        self.name = name
```

این کلاس روش getter و setter برای ویژگی name. ندارد. به جای آن، ویژگی را به عنوان بخشی از API خود فراهم می کند. بنابراین، می توانید آن را به صورت مستقیم استفاده کنید:

In [66]:

```
jane = Person("Jane")
jane.name
```

Out[66]:

```
'Jane'
```

In [67]:

```
jane.name = "Jane Doe"
jane.name
```

Out[67]:

```
'Jane Doe'
```

در این مثال، به جای استفاده از یک متد setter برای تغییر مقدار name.، ویژگی را به صورت مستقیم در یک عبارت اختصاص استفاده می کنید. این عمل در کد پایتون رایج است. اگر کلاس Person شما به یک نقطه ای برسد که نیاز به اضافه کردن رفتار شبیه به تابع به بالای name. دارید، آنگاه می توانید ویژگی را به یک خصوصیت تبدیل کنید.

برای مثال، فرض کنید نیاز دارید ویژگی را با حروف بزرگ ذخیره کنید. سپس می توانید شبیه به زیر عمل کنید:

In [73]:

```
class Person:
    def __init__(self, name):
        self.name = name
        #self.set_name(name)

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value.upper()
```

این کلاس name. را به عنوان یک خصوصیت با روش getter و setter مناسب تعریف می کند. هنگامی که شما به مقدار ویژگی دسترسی یا آن را به روز می کنید، پایتون به ترتیب این متد ها را به صورت خودکار فراخوانی می کند. متد setter از بالا بردن مقدار ورودی قبل از اختصاص دادن آن به name_. مراقبت می کند:

In [74]:

```
jane = Person("Jane")
jane.name
```

Out[74]:

```
'JANE'
```

In [75]:

```
vars(jane)
```

Out[75]:

```
{'_name': 'JANE'}
```

In [76]:

```
jane.name = "Jane Doe"
jane.name
```

Out[76]:

```
'JANE DOE'
```

خصوصیات پایتون به شما اجازه می دهند تا رفتار شبیه به تابع را به ویژگی های خود اضافه کنید در حالی که همچنان از آن ها به عنوان ویژگی های عادی به جای روش ها استفاده می کنید. توجه کنید که هنوز می توانید با استفاده از یک عبارت اختصاص به name. مقادیر جدید را اختصاص دهید، به جای فراخوانی متد. اجرای این عبارت، روش setter را فعال می کند که مقدار ورودی را بالا می برد.

بررسی کلاس های ویژه از کتابخانه استاندارد پایتون

در کتابخانه استاندارد پایتون، شما ابزارهای زیادی را که چالش های مختلف را حل می کنند، پیدا خواهید کرد. در میان تمام این ابزارها، شما چندین ابزار خواهید دید که شما را در زمان نوشتن کلاس های سفارشی به صورت مؤثرتری کمک می کنند.

به عنوان مثال، اگر شما به دنبال یک ابزار هستید که شما را از نوشتن بسیاری از کدهای boilerplate مربوط به کلاس نجات دهد، آنگاه شما می توانید از data classes و ماژول dataclasses استفاده کنید.

به طور مشابه، اگر به دنبال یک ابزار هستید که به شما اجازه می دهد به سرعت تعریف شمارش های بر پایه کلاس از ثابت ها را داشته باشید، آنگاه می توانید چشم خود را به ماژول enum و نوع های مختلف کلاس شمارش خود بپردازید.

در بخش های زیر، شما با اصول استفاده از data classes و enumerations برای نوشتن بهینه، قابل اطمینان و کلاس های ویژه در پایتون آشنا خواهید شد.

Data Classes

پایتون در کلاس‌های داده‌ای خود، به ذخیره داده‌ها می‌پردازد. با این حال، آن‌ها همچنین تولیدکننده کد هستند که برای شما در پشت صحنه، کد boilerplate بسیاری از کلاس‌های مربوط به شما را تولید می‌کنند.

به عنوان مثال، اگر از زیرساخت کلاس داده استفاده کنید تا یک کلاس سفارشی بنویسید، آنگاه نیازی به پیاده‌سازی متدهای ویژه مانند `__init__()`، `__repr__()` و `__eq__()` و `__hash__()` نخواهید داشت. کلاس داده، آن‌ها را برای شما خواهد نوشت. علاوه بر این، کلاس داده این متدها را با رعایت بهترین شیوه‌های کاربردی و جلوگیری از خطاهای احتمالی نوشته است.

همانطور که می‌دانید، متدهای ویژه، قابلیت‌های مهمی را در کلاس‌های پایتون پشتیبانی می‌کنند. در مورد کلاس‌های داده، شما یک نمایش رشته دقیق، قابلیت‌های مقایسه، قابلیت هش کردن و غیره خواهید داشت.

اگرچه نام کلاس داده ممکن است این نوع کلاس را به محدود بودن در بردارنده داده‌ها محدود کند، اما همچنین متدهای دیگری را ارائه می‌دهد. بنابراین، کلاس‌های داده شبیه به کلاس‌های عادی هستند اما با قابلیت‌های فوق العاده.

برای ایجاد یک کلاس داده، به سادگی @decorator dataclass را از ماژول dataclasses وارد کنید. شما از این decorator در تعریف کلاس خود استفاده خواهید کرد. در این حالت، شما نیاز به نوشتن یک متد `__init__()` نخواهید داشت. شما فقط با استفاده از type hints فیلدهای داده را به عنوان ویژگی‌های کلاس تعریف خواهید کرد.

بطور مثال، به این صورت می‌توانید کلاس ThreeDPoint را به عنوان یک کلاس داده بنویسید:

In [77]:

```python
from dataclasses import dataclass

@dataclass
class ThreeDPoint:
    x: int | float
    y: int | float
    z: int | float

    @classmethod
    def from_sequence(cls, sequence):
        return cls(*sequence)

    @staticmethod
    def show_intro_message(name):
        print(f"Hey {name}! This is your 3D Point!")
```

این پیاده‌سازی جدید ThreeDPoint از @decorator dataclass پایتون استفاده می‌کند تا کلاس عادی را به کلاس داده تبدیل کند. به جای تعریف یک متد `__init__()`، شما ویژگی‌های نمونه را با نوع مربوطه آن‌ها لیست می‌کنید. کلاس داده به شما کمک خواهد کرد تا یک مقداردهی مناسب برای شما بنویسد. توجه داشته باشید که شما `__iter__()` یا `__repr__()` را نیز تعریف نمی‌کنید.

بعد از تعریف فیلدهای داده یا ویژگی‌ها، می‌توانید شروع به اضافه کردن روش‌های مورد نیاز خود کنید. در این مثال، شما روش کلاس `.from_sequence()` و روش استاتیک `.show_intro_message()` را حفظ می‌کنید.

برای بررسی قابلیت‌های اضافی که @dataclass به این نسخه از ThreeDPoint اضافه کرده است، به جلو بروید و کد زیر را اجرا کنید:

In [78]:

```
point_1 = ThreeDPoint(1.0, 2.0, 3.0)

point_2 = ThreeDPoint(2, 3, 4)

point_1 == point_2
```

Out[78]:

```
False
```

In [79]:

```
point_3 = ThreeDPoint(1, 2, 3)

point_1 == point_3
```

Out[79]:

```
True
```

In [80]:

```
point_1
```

Out[80]:

```
ThreeDPoint(x=1.0, y=2.0, z=3.0)
```

In [81]:

```
point_3
```

Out[81]:

```
ThreeDPoint(x=1, y=2, z=3)
```

In [82]:

```
from dataclasses import astuple

astuple(point_1)
```

Out[82]:

```
(1.0, 2.0, 3.0)
```

Enumerations

شما در چندین زبان برنامه‌نویسی، نوع داده‌ای enum یا enumeration را خواهید یافت. Enums به شما اجازه می‌دهند تا مجموعه‌ای از ثابت‌های نامگذاری شده را ایجاد کنید که به عنوان اعضا شناخته می‌شوند و می‌توانند از طریق خود enum دسترسی پیدا کنند.

پایتون نوع داده‌ای enum داخلی ندارد. خوشبختانه، پایتون 3.4 ماژول enum را معرفی کرده است تا کلاس Enum را برای پشتیبانی از enum های عمومی فراهم کند.

روزهای هفته، ماه‌ها و فصول سال، کدهای وضعیت HTTP، رنگ‌های چراغ ترافیک و برنامه‌های قیمت‌گذاری یک سرویس وب، همگی نمونه‌های خوبی از ثابت‌هایی هستند که می‌توانید در enum گروه بندی کنید. به طور خلاصه، شما می‌توانید از enums برای نشان دادن متغیرهایی استفاده کنید که مقدار گیری آن‌ها تحت یک مجموعه محدود از مقادیر ممکن است.

کلاس Enum، در میان کلاس‌های مشابه دیگر در ماژول enum، به شما اجازه می‌دهد تا به سرعت و با کارآمدی enum های سفارشی یا گروه‌های ثابت مشابه با ویژگی های خوب و جذاب از خود استفاده کنید. علاوه بر ثابت عضو، enums همچنین ممکن است دارای روش‌های عمل با آن ثابت باشد.

برای تعریف یک enum سفارشی، می‌توانید از کلاس Enum به عنوان زیرکلاس استفاده کنید. در ادامه مثالی از یک enum را مشاهده می‌کنید که روزهای هفته را گروه بندی می‌کند:

In [83]:

```
from enum import Enum

class WeekDay(Enum):
    MONDAY = 1
    TUESDAY = 2
    WEDNESDAY = 3
    THURSDAY = 4
    FRIDAY = 5
    SATURDAY = 6
    SUNDAY = 7
```

اگر سعی کنید مقدار یک عضو enum را تغییر دهید، خطای AttributeError دریافت خواهید کرد. بنابراین، اعضای enum به صورت دقیق ثابت هستند. شما می‌توانید بر روی اعضای enum به صورت مستقیم حلقه بزنید زیرا enum ها به طور پیش فرض حلقه را پشتیبانی می‌کنند.

In [84]:

```
1 WeekDay.MONDAY = 0
```

```
---------------------------------------------------------------------------
-
AttributeError                            Traceback (most recent call las
t)
Cell In[84], line 1
----> 1 WeekDay.MONDAY = 0

File C:\ProgramData\Anaconda3\lib\enum.py:480, in EnumMeta.__setattr__(cl
s, name, value)
    478 member_map = cls.__dict__.get('_member_map_', {})
    479 if name in member_map:
--> 480     raise AttributeError('Cannot reassign members.')
    481 super().__setattr__(name, value)

AttributeError: Cannot reassign members.
```

In [85]:

```
1 list(WeekDay)
```

Out[85]:

```
[<WeekDay.MONDAY: 1>,
 <WeekDay.TUESDAY: 2>,
 <WeekDay.WEDNESDAY: 3>,
 <WeekDay.THURSDAY: 4>,
 <WeekDay.FRIDAY: 5>,
 <WeekDay.SATURDAY: 6>,
 <WeekDay.SUNDAY: 7>]
```

شما می‌توانید با استفاده از نحو مختلف، به صورت مستقیم به اعضای آن‌ها دسترسی پیدا کنید:

In [86]:

```
1 WeekDay.MONDAY
```

Out[86]:

```
<WeekDay.MONDAY: 1>
```

In [89]:

```
1 WeekDay(1)
```

Out[89]:

```
<WeekDay.MONDAY: 1>
```

In [87]:

```
1 WeekDay(2)
```

Out[87]:

```
<WeekDay.TUESDAY: 2>
```

In [90]:

```
1 WeekDay["WEDNESDAY"]
```

Out[90]:

```
<WeekDay.WEDNESDAY: 3>
```

در مثال اول، شما با استفاده از نمایش نقطه‌ای به یک عضو enum دسترسی پیدا می‌کنید که بسیار شفاف و خوانا است. در مثال دوم، شما با فراخوانی enum با مقدار آن عضو به عنوان یک آرگومان، به یک عضو دسترسی پیدا می‌کنید. در نهایت، شما از نحو شبیه به دیکشنری برای دسترسی به یک عضو دیگر با نام استفاده می‌کنید.

اگر می‌خواهید به اجزای یک عضو دسترسی دقیق داشته باشید، می‌توانید از ویژگی های name. و value. استفاده کنید که در زمینه حلقه بسیار مفید هستند:

In [91]:

```
1 WeekDay.THURSDAY.name
```

Out[91]:

```
'THURSDAY'
```

In [92]:

```
1 WeekDay.THURSDAY.value
```

Out[92]:

```
4
```

In [93]:

```
1 for day in WeekDay:
2     print(day.name, "->", day.value)
```

```
MONDAY -> 1
TUESDAY -> 2
WEDNESDAY -> 3
THURSDAY -> 4
FRIDAY -> 5
SATURDAY -> 6
SUNDAY -> 7
```

در نهایت ما می توانیم متدهای دلخواه خودمان را نیز اضافه نماییم:

In [94]:

```python
from enum import Enum

class WeekDay(Enum):
    MONDAY = 1
    TUESDAY = 2
    WEDNESDAY = 3
    THURSDAY = 4
    FRIDAY = 5
    SATURDAY = 6
    SUNDAY = 7

    @classmethod
    def favorite_day(cls):
        return cls.FRIDAY

    def __str__(self):
        return f"Current day: {self.name}"
```

In [95]:

```python
WeekDay.favorite_day()
```

Out[95]:

```
<WeekDay.FRIDAY: 5>
```

In [96]:

```python
print(WeekDay.FRIDAY)
```

```
Current day: FRIDAY
```

سلسه مراتب کلاس ها

استفاده از inheritance، شما می‌توانید سلسله‌مراتب کلاس‌ها را طراحی و ساختاردهی کنید. سلسله‌مراتب کلاس یک مجموعه از کلاس‌های نزدیک به هم هستند که از طریق inheritance به یکدیگر متصل شده‌اند و در یک ساختار شبیه به درخت قرار دارند.

کلاس یا کلاس‌های بالای سلسله‌مراتب، کلاس‌های پایه هستند، در حالی که کلاس‌های پایین‌تر، کلاس‌های مشتق یا زیرکلاس هستند. سلسله‌مراتب مبتنی بر inheritance، رابطه is-a-type-of بین زیرکلاس‌ها و کلاس پایه خود را بیان می‌کنند.

در هر سطح از سلسله‌مراتب، ویژگی‌ها و رفتار از سطوح بالاتر به ارث می‌برند. بنابراین، کلاس‌های بالای سلسله‌مراتب، کلاس‌های جنرال با عملکرد مشترک هستند، در حالی که کلاس‌های پایین تر در سلسله مراتب، تخصص بیشتری دارند. آن‌ها ویژگی ها و رفتار از superclass خود را به ارث می برند و همچنین ویژگی ها و رفتار خود را نیز اضافه می کنند.

طبقه بندی تاکسونومی حیوانات یک مثال رایج است برای توضیح سلسله مراتب کلاس. در این سلسله مراتب، شما یک کلاس Animal جنرال دارید. در زیر این کلاس، شما ممکن است زیرکلاس های Fish، Bird، Mammal و غیره داشته باشید. این زیرکلاس ها کلاس های خاص تر از Animal هستند و ویژگی ها و روش های آن را به ارث می برند. آن ها نیز ممکن است ویژگي هاي خود را داشته باشند.

برای ادامه با سلسله مراتب، شما ممکن است زیرکلاس Mammal، Bird و Fish را داشته باشید و زیرکلاس های مشتق شده با خصوصیات حتي خصوصیات خصوصي تر را ایجاد کنید. در اینجا یک نمونه بازیچه کوتاه است:

In [97]:

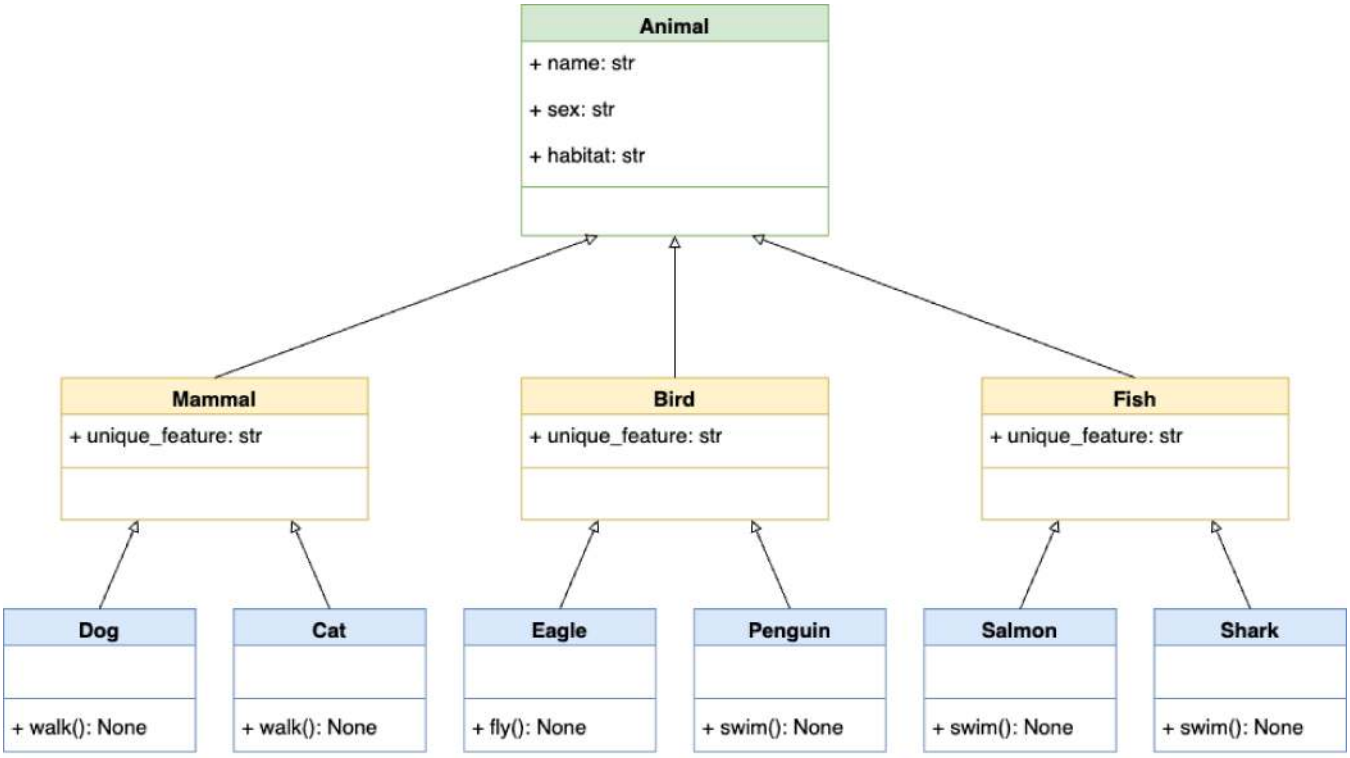
```python
class Animal:
    def __init__(self, name, sex, habitat):
        self.name = name
        self.sex = sex
        self.habitat = habitat

class Mammal(Animal):
    unique_feature = "Mammary glands"

class Bird(Animal):
    unique_feature = "Feathers"

class Fish(Animal):
    unique_feature = "Gills"

class Dog(Mammal):
    def walk(self):
        print("The dog is walking")

class Cat(Mammal):
    def walk(self):
        print("The cat is walking")

class Eagle(Bird):
    def fly(self):
        print("The eagle is flying")

class Penguin(Bird):
    def swim(self):
        print("The penguin is swimming")

class Salmon(Fish):
    def swim(self):
        print("The salmon is swimming")

class Shark(Fish):
    def swim(self):
        print("The shark is swimming")
```

در بالای سلسله مراتب، شما کلاس Animal را دارید. این کلاس پایه سلسله مراتب شما است. این ویژگی های name.و sex. و habitat. را دارد که به ترتیب شیء های رشته ای هستند. این ویژگی ها برای تمام حیوانات مشترک هستند.

سپس شما با به ارث بردن از Animal، کلاس های Mammal، Bird و Fish را تعریف می کنید. این کلاس ها یک ویژگی کلاس unique_feature. دارند که ویژگی تمایز دهنده هر گروه حیوانات را نگه می دارد.

سپس شما موجودات خاص مانند Dog و Cat را ایجاد می کنید. این کلاس ها دارای روش های خاصی هستند که به تمام سگ ها و گربه ها مشترک است. به طور مشابه، شما دو کلاس را تعریف می کنید که به ترتیب از Bird و دو کلاس دیگر به ارث برده شده است.

اینجا یک نمودار کلاس شبیه به درخت است که به شما در فهم رابطه سلسله مراتب بین کلاس ها کمک خواهد کرد:



هر سطح در سلسله مراتب می تواند و به طور معمول، ویژگی ها و قابلیت های جدیدی را بر روی آنچه که پدرانش ارائه می دهند، اضافه کند. اگر از بالا به پایین از نمودار عبور کنید، آنگاه از کلاس های عمومی به کلاس های تخصصی حرکت خواهید کرد.

Extended vs Overridden Methods

زمانی که از ارث بری استفاده می کنید، ممکن است با یک مسئله جالب و چالش برانگیز روبرو شوید. در برخی از موارد، یک کلاس پدر تنها در سطح پایه، یک قابلیت خاص را فراهم می کند و شما ممکن است بخواهید آن قابلیت را در زیر کلاس های خود گسترش دهید. در دیگر موارد، ویژگی در کلاس پدر برای زیر کلاس مناسب نیست.

در این موارد، می توانید از یکی از راهبردهای زیر استفاده کنید، به تفکیک مورد خاص شما:

extend یک متد به ارث برده شده در یک زیر کلاس، به این معنی است که شما قابلیت فراهم شده توسط کلاس پدر را دوباره استفاده کرده و قابلیت جدید را بر روی آن اضافه خواهید کرد override یک متد به ارث برده شده در یک زیر کلاس، به این معنی است که شما به طور کامل قابلیت از کلاس پدر را دور خواهید انداخت و قابلیت جدید را در زیر کلاس فراخواهید داد اینجا یک مثال از سلسله مراتب کلاس کوچک است که روش اول را برای فراهم کردن قابلیت های گسترده بر اساس قابلیت های به ارث برده شده، به کار می برد:

In [98]:

```python
class Aircraft:
    def __init__(self, thrust, lift, max_speed):
        self.thrust = thrust
        self.lift = lift
        self.max_speed = max_speed

    def show_technical_specs(self):
        print(f"Thrust: {self.thrust} kW")
        print(f"Lift: {self.lift} kg")
        print(f"Max speed: {self.max_speed} km/h")
```

In [99]:

```python
class Helicopter(Aircraft):
    def __init__(self, thrust, lift, max_speed, num_rotors):
        super().__init__(thrust, lift, max_speed)
        self.num_rotors = num_rotors

    def show_technical_specs(self):
        super().show_technical_specs()
        print(f"Number of rotors: {self.num_rotors}")
```

In [100]:

```python
h1 = Helicopter(25, 300, 150, 2)
```

In [101]:

```python
h1
```

Out[101]:

```
<__main__.Helicopter at 0x1a1bdd16e90>
```

در این مثال، شما کلاس هواپیما را به عنوان کلاس پایه تعریف می کنید. در `__init__()` ، چند ویژگی نمونه ایجاد می کنید. سپس متد `.show_technical_specs()` را تعریف می کنید که اطلاعاتی در مورد مشخصات فنی هواپیما چاپ می کند.

بعد، شما هلیکوپتر را تعریف می کنید و از هواپیما به عنوان پدر بهره می برید. متد `__init__()` هلیکوپتر، رومتدش مربوطه هواپیما را با فراخوانی `super()` برای مقداردهی اولیه ویژگی های thrust.و lift. و max_speed. گسترش می دهد. شما قبلاً چنین چیزی را در بخش قبل دیده بودید.

هلیکوپتر همچنین قابلیت متد `.show_technical_specs()` را گسترش می دهد. در این حالت، شما ابتدا با استفاده از `super().show_technical_specs()` را از هواپیما فراخوانی می کنید. سپس یک فراخوانی جدید به `print()` اضافه می کنید که اطلاعات جدید را به توصیف فنی هلیکوپتر در دست اضافه می کند.

اینجا نحوه عملکرد نمونه های هلیکوپتر است:

In [102]:

```python
sikorsky_UH60 = Helicopter(1490, 9979, 278, 2)
```

In [103]:

```
sikorsky_UH60.show_technical_specs()
```

```
Thrust: 1490 kW
Lift: 9979 kg
Max speed: 278 km/h
Number of rotors: 2
```

زمانی که روی یک نمونه هلیکوپتر `show_technical_specs()` را فراخوانی می کنید، اطلاعات ارائه شده توسط کلاس پایه، هواپیما، و همچنین اطلاعات خاص اضافه شده توسط خود هلیکوپتر را دریافت می کنید. شما قابلیت های هواپیما را در زیر کلاس خود، هلیکوپتر، گسترش داده اید.

حال زمان آن رسیده است که نگاهی به این بیندازید که چگونه می توانید یک متد را در یک زیر کلاس بازنویسی کنید. به عنوان مثال، بگویید که یک کلاس پایه به نام Worker دارید که چندین ویژگی و متد مانند مثال زیر را تعریف می کند:

In [104]:

```
class Worker:
    def __init__(self, name, address, hourly_salary):
        self.name = name
        self.address = address
        self.hourly_salary = hourly_salary

    def show_profile(self):
        print("== Worker profile ==")
        print(f"Name: {self.name}")
        print(f"Address: {self.address}")
        print(f"Hourly salary: {self.hourly_salary}")

    def calculate_payroll(self, hours=40):
        return self.hourly_salary * hours
```

در این کلاس، شما چند ویژگی نمونه را برای ذخیره داده های مهم در مورد کارگر فعلی تعریف می کنید. همچنین متد `show_profile()` را برای نمایش اطلاعات مربوط به کارگر ارائه می دهید. در نهایت، شما یک متد کلی `calculate_payroll()` را برای محاسبه حقوق کارگران از حقوق ساعتی و تعداد ساعات کارکرد شان ارائه می دهید.

در دوره توسعه بعدی، برخی از الزامات تغییر می کنند. حالا متوجه می شوید که مدیران حقوق خود را به روش دیگری محاسبه می کنند. آنها یک پاداش ساعتی دارند که شما باید قبل از محاسبه مبلغ نهایی به حقوق ساعتی عادی آن اضافه کنید.

پس از فکر کردن به مسئله، تصمیم می گیرید که Manager باید `calculate_payroll()` را به طور کامل بازنویسی کند. در اینجا پیاده سازی است که شما با آن آمده اید:

In [ ]:

```
class Manager(Worker):
    def __init__(self, name, address, hourly_salary, hourly_bonus):
        super().__init__(name, address, hourly_salary)
        self.hourly_bonus = hourly_bonus

    def calculate_payroll(self, hours=40):
        return (self.hourly_salary + self.hourly_bonus) * hours
```

در مقدمه مدیر، شما پاداش ساعتی را به عنوان یک آرگومان در نظر می گیرید. سپس به طور معمول روش `__init__()` والدین را فراخوانی کرده و ویژگی نمونه `.hourly_bonus` را تعریف می کنید. در نهایت، شما `.calculate_payroll()` را با یک پیاده سازی کاملاً متفاوت که از قابلیت های به ارث برده شده استفاده نمی کند، بازنویسی می کنید.

وراثت چندگانه

در پایتون، شما می توانید از چندین وراثت استفاده کنید. این نوع وراثت به شما اجازه می دهد کلاسی را ایجاد کنید که از چندین والدین به ارث برده شده است. زیر کلاس دسترسی به ویژگی ها و روش های همه والدین خود دارد.

وراثت چندگانه به شما اجازه می دهد کد را از چندین کلاس موجود استفاده مجدد کنید. با این حال، شما باید با مراقبتی از پیچیدگی وراثت چندگانه برخورد کنید. در غیر این صورت، مشکلاتی مانند مشکل الماس را ممکن است مواجه شوید. در بخش ترتیب روش (MRO) بیشتر درباره این موضوع بیاموزید.

اینجا یک مثال کوچک از وراثت چندگانه در پایتون است:

In [105]:

```
class Vehicle:
    def __init__(self, make, model, color):
        self.make = make
        self.model = model
        self.color = color

    def start(self):
        print("Starting the engine...")

    def stop(self):
        print("Stopping the engine...")

    def show_technical_specs(self):
        print(f"Make: {self.make}")
        print(f"Model: {self.model}")
        print(f"Color: {self.color}")

class Car(Vehicle):
    def drive(self):
        print("Driving on the road...")

class Aircraft(Vehicle):
    def fly(self):
        print("Flying in the sky...")

class FlyingCar(Car, Aircraft):
    pass
```

در این مثال، شما یک کلاس Vehicle با ویژگی های make.و model. و color. می نویسید. این کلاس همچنین متد های `.start()` و `.stop()` و `.show_technical_specs()` را دارد. سپس شما یک کلاس Car ایجاد می کنید که از Vehicle به ارث برده شده است و با یک متد جدید به نام `.drive()` توسعه می یابد. همچنین شما یک کلاس Aircraft ایجاد می کنید که از Vehicle به ارث برده شده است و یک متد `.fly()` را اضافه می کند.

در نهایت، شما یک FlyingCar class تعریف می کنید تا یک ماشین را نشان دهد که می توانید آن را در جاده رانده یا در آسمان پرواز کنید. آیا این جالب نیست؟ توجه داشته باشید که این کلاس هر دو Car و Aircraft را در لیست کلاس های والد خود شامل می شود. بنابراین، عملکرد خود را از هر دو superclass به ارث خواهد برد.

اینجا نحوه استفاده از FlyingCar class را مشاهده می کنید:

In [106]:

```
space_flyer = FlyingCar("Space", "Flyer", "Black")
```

In [107]:

```
space_flyer.show_technical_specs()
```

```
Make: Space
Model: Flyer
Color: Black
```

In [108]:

```
space_flyer.start()
```

```
Starting the engine...
```

In [109]:

```
space_flyer.drive()
```

```
Driving on the road...
```

In [110]:

```
space_flyer.fly()
```

```
Flying in the sky...
```

در این قطعه کد، شما ابتدا یک نمونه از FlyingCar ایجاد می کنید. سپس تمام متد های آن را فراخوانی می کنید، از جمله متد های به ارث برده شده. همانطور که می بینید، وراثت چندگانه باعث افزایش استفاده مجدد کد می شود و به شما اجازه می دهد تا به طور همزمان از عملکرد چندین کلاس پایه استفاده کنید. به هر حال، اگر این FlyingCar را واقعاً پرواز کنید، مطمئن شوید که در حال پرواز موتور را نمی خاموشید!

نحوه تمایز متدها در فراخوانی Method Resolution Order (MRO)

زمانی که از وراثت چندگانه استفاده می کنید، ممکن است با مواردی روبرو شوید که یک کلاس از دو یا چند کلاس به ارث برده شده است که پایه یکسانی دارند. این به عنوان مشکل الماس شناخته می شود. مسئله واقعی زمانی پدیدار می شود که والدین چندین نسخه از همان متد را فراهم می کنند. در این صورت، سخت است تعیین کردن اینکه کدام نسخه از آن روش زیرکلاس در نهایت استفاده خواهد کرد.

پایتون با استفاده از ترتیب روش خاص (MRO) با این مشکل برخورد می کند. پس MRO در پایتون چیست؟ این الگوریتمی است که به پایتون می گوید چگونه در یک زمینه وراثت چندگانه به دنبال روش های به ارث برده شده بگردید. MRO پایتون تعیین می کند کدام پیاده سازی روش یا ویژگی را باید استفاده کنید زمانی که چندین نسخه از آن در سلسله مراتب کلاس وجود دارد.

MRO پایتون بر اساس ترتیب کلاس های والد در تعریف زیرکلاس است. به عنوان مثال، Car قبل از Aircraft در FlyingCar class در بخش قبل قرار دارد. MRO همچنین رابطه وراثت بین کلاس ها را در نظر می گیرد. به طور کلی، پایتون به دنبال متد ها و ویژگی ها به ترتیب زیر می گردد:

- کلاس فعلی
- سوپر کلاس های سمت چپ
- سوپر کلاس بعدی که از چپ به راست، تا آخرین سوپر کلاس لیست شده است
- سوپر کلاس های کلاس های به ارث برده شده
- کلاس شیء

مهم است به یاد داشته باشید که زیرکلاس ها در جستجو اول قرار می گیرند. علاوه بر این، اگر شما چندین والدین داشته باشید که هر کدام نسخه متفاوتی از یک متد یا ویژگی خاص را پیاده سازی کنند، پایتون آنها را با ترتیبی که در تعریف کلاس لیست شده اند جستجو می کند.

برای توضیح MRO، سلسله مراتب کلاس نمونه زیر را در نظر بگیرید:

In [112]:

```
class A:
    def method(self):
        print("A.method")

class B(A):
    def method(self):
        print("B.method")

class C(A):
    def method(self):
        print("C.method")

class D(B, C):
    pass
```

در این مثال، D از B و C به ارث برده شده است که از A به ارث برده شده است. تمام سوپر کلاس ها در سلسله مراتب نسخه متفاوتی از `.method()` را تعریف می کنند. کدام یک از این نسخه ها در نهایت توسط D فراخوانی خواهد شد؟ برای پاسخ به این سوال، به آرامی `.method()` را روی یک نمونه D فراخوانی کنید:

In [113]:

```
D().method()
```

```
B.method
```

وقتی شما `.method()` را روی یک نمونه D فراخوانی می کنید، B.method را در صفحه خود مشاهده می کنید. این بدان معناست که پایتون ابتدا `.method()` را در کلاس B پیدا کرده است. این نسخه از `.method()` است که شما فراخوانی می کنید. شما نسخه های C و A را نادیده می گیرید.

شما می توانید MRO فعلی یک کلاس داده شده را با استفاده از ویژگی خاص `.__mro__` بررسی کنید:

In [114]:

```
D.__mro__
```

Out[114]:

```
(__main__.D, __main__.B, __main__.C, __main__.A, object)
```

در خروجی، می توانید ببینید که پایتون با عبور از D خود، سپس B، سپس C، سپس A و در نهایت object، کلاس پایه تمام کلاس های پایتون، به دنبال متد ها و ویژگی ها در D جستجو می کند.

ویژگی `__mro__` می تواند به شما در تغییر کلاس های خود و تعریف MRO خاص کلاس شما کمک کند. روش تغییر این

Mixin Classes

یک کلاس mixin متدهایی را فراهم می کند که می توانید در بسیاری از کلاس های دیگر استفاده کنید. کلاس های mixin نوع جدیدی را تعریف نمی کنند، بنابراین قرار نیست به صورت نمونه سازی شوند. شما از قابلیت آنها برای اضافه کردن ویژگی های اضافی به سرعت به کلاس های دیگر استفاده می کنید.

شما می توانید به قابلیت یک کلاس mixin به متد های مختلف دسترسی پیدا کنید. یکی از این روش ها، inheritance است. با این حال، به ارث بردن از کلاس های mixin به معنای یک رابطه is-a نیست زیرا این کلاس ها نوع های قابل تعریف را بیان نمی کنند. آنها فقط وظایف خاص خود را بسته بندی می کنند که قرار است در کلاس های دیگر استفاده مجدد شود.

برای توضیح نحوه استفاده از کلاس های mixin، فرض کنید در حال ساخت یک سلسله مراتب کلاس با یک کلاس Person در بالاترین سطح هستید. از این کلاس، شما به کلاس های مشتق شده مانند Professor، Student، Employee و چندین دستگاه دیگر مشتق خواهید شد. سپس متوجه می شوید که تمام زیرکلاس های Person نیاز به متدهای serialize داده خود در فرمت های مختلف، از جمله JSON و pickle دارند.

با توجه به این، شما به نوشتن یک SerializerMixin class فکر می کنید که این وظایف را بر عهده دارد. در زیر آنچه را پیدا می کنید:

In [115]:

```
import json
import pickle

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class SerializerMixin:
    def to_json(self):
        return json.dumps(self.__dict__)

    def to_pickle(self):
        return pickle.dumps(self.__dict__)

class Employee(SerializerMixin, Person):
    def __init__(self, name, age, salary):
        super().__init__(name, age)
        self.salary = salary
```

در این مثال، Person کلاس والدین است و SerializerMixin یک کلاس mixin است که قابلیت سریال سازی را فراهم می کند. کلاس Employee از هر دو SerializerMixin و Person به ارث می برد. بنابراین، این متدهای `.to_json()` و `.to_pickle()` را به ارث خواهد برد، که می توانید از آنها برای سریال سازی نمونه های Employee در کد خود استفاده کنید.

در این مثال، Employee یک Person است. با این حال، یک SerializerMixin نیست زیرا این کلاس نوع شیء را تعریف نمی کند. فقط یک کلاس mixin است که قابلیت های سریال سازی را بسته بندی می کند.

نحوه عملکرد Employee بصورت زیر خواهد بود:

In [116]:

```
1 john = Employee("John Doe", 30, 50000)
2 john.to_json()
```

Out[116]:

```
'{"name": "John Doe", "age": 30, "salary": 50000}'
```

In [117]:

```
1 john.to_pickle()
```

Out[117]:

```
b'\x80\x04\x95+\x00\x00\x00\x00\x00\x00\x00}\x94(\x8c\x04name\x94\x8c\x08J
ohn Doe\x94\x8c\x03age\x94K\x1e\x8c\x06salary\x94MP\xc3u.'
```

اکنون کلاس Employee شما قادر به سریال سازی داده های خود با استفاده از فرمت های JSON و pickle است. عالی است! آیا می توانید به هر کلاس mixin مفید دیگری فکر کنید؟

تا این لحظه، شما بسیاری در مورد inheritance ساده و چندگانه در Python یاد گرفته اید. در بخش بعدی، شما به برخی از مزایای استفاده از inheritance در هنگام نوشتن و سازماندهی کد خود می پردازید.

مزایای استفاده از وراثت

- قابلیت استفاده مجدد(Reusability): شما می توانید به سرعت کد کاری را از یک یا چند کلاس والدین به عنوان زیرکلاس های خود به تعداد دلخواه استفاده کنید.
- ماژولاریتی(Modularity): شما می توانید از inheritance برای سازماندهی کد خود در سلسله مراتب کلاس های مرتبط استفاده کنید.
- قابلیت نگهداری(Maintainability): شما می توانید به سرعت مشکلات را در یک کلاس والدین رفع کنید یا ویژگی های جدید را اضافه کنید. این تغییرات به صورت خودکار در تمام زیرکلاس های آن در دسترس خواهند بود. Inheritance همچنین تکرار کد را کاهش می دهد.
- Polymorphism: شما می توانید زیرکلاس هایی را ایجاد کنید که می توانند جایگزین کلاس والدین خود شوند و قابلیت های مشابه یا معادل را فراهم کنند.
- قابلیت گسترش(Extensibility): شما می توانید به سرعت یک کلاس قبلی را با اضافه کردن داده های جدید و رفتار به زیرکلاس های خود گسترش دهید.

راهکارهای جایگزین وراثت در پایتون

Inheritance، به خصوص inheritance چندگانه، می تواند یک موضوع پیچیده و سخت برای درک باشد. خوشبختانه، inheritance تنها تکنیکی نیست که به شما اجازه می دهد تا قابلیت استفاده مجدد را در برنامه نویسی شیء گرا داشته باشید. شما همچنین دارای composition هستید که یک رابطه has-a بین کلاس ها را نشان می دهد.

Composition به شما اجازه می دهد یک شیء را از اجزای آن بسازید. شیء composite دسترسی مستقیم به رابط هر اجزای خود ندارد. با این حال، می تواند از پیاده سازی هر اجزای استفاده کند.

Delegation یک تکنیک دیگر است که می توانید برای ارتقاء قابلیت استفاده مجدد کد در برنامه های OOP خود استفاده کنید. با delegation، می توانید رابطه های can-do را نشان دهید، جایی که یک شیء بر روی یک شیء دیگر برای انجام یک وظیفه خاص تکیه می کند.

در بخش های بعد، شما بیشتر در مورد این تکنولوژی ها و نحوه استفاده از آن ها در کدهای شئ گرا خود برای ساخت کدهای قابل اطمینان و قابل انعطاف خواهید آموخت.

Composition

همانطور که قبلاً گفته شد، می توانید از composition برای مدل کردن یک رابطه has-a بین اشیاء استفاده کنید. به عبارت دیگر، از طریق composition، می توانید با ترکیب اشیاء که به عنوان اجزا عمل می کنند، اشیاء پیچیده ای را ایجاد کنید. توجه داشته باشید که این اجزا به عنوان کلاس های مستقل معنایی نخواهند داشت.

ترجیح composition بر inheritance به طراحی کلاس های قابل انعطاف تر منجر می شود. بر خلاف inheritance، composition در زمان اجرا تعریف می شود، به این معنی که می توانید به صورت پویا یک جزء فعلی را با یک جزء دیگر از همان نوع جایگزین کنید. این ویژگی باعث می شود تا رفتار composite در زمان اجرا قابل تغییر باشد.

در مثال زیر، شما از composition برای ساخت یک کلاس IndustrialRobot از Body و Arm components استفاده می کنید:

In [119]:

```python
class IndustrialRobot:
    def __init__(self):
        self.body = Body()
        self.arm = Arm()

    def rotate_body_left(self, degrees=10):
        self.body.rotate_left(degrees)

    def rotate_body_right(self, degrees=10):
        self.body.rotate_right(degrees)

    def move_arm_up(self, distance=1):
        self.arm.move_up(distance)

    def move_arm_down(self, distance=1):
        self.arm.move_down(distance)

    def weld(self):
        self.arm.weld()

class Body:
    def __init__(self):
        self.rotation = 0

    def rotate_left(self, degrees=10):
        self.rotation -= degrees
        print(f"Rotating body {degrees} degrees to the left...")

    def rotate_right(self, degrees=10):
        self.rotation += degrees
        print(f"Rotating body {degrees} degrees to the right...")

class Arm:
    def __init__(self):
        self.position = 0

    def move_up(self, distance=1):
        self.position += distance
        print(f"Moving arm {distance} cm up...")

    def move_down(self, distance=1):
        self.position -= distance
        print(f"Moving arm {distance} cm down...")

    def weld(self):
        print("Welding...")
```

در این مثال، شما یک کلاس IndustrialRobot را از اجزای آن، Body و Arm، ساخته اید. کلاس Body حرکات افقی را فراهم می کند، در حالی که کلاس Arm نماینده بازوی ربات است و حرکت عمودی و قابلیت جوشکاری را فراهم می کند.

اینجاست که چگونه می توانید از IndustrialRobot در کد خود استفاده کنید:

In [120]:

```
1 robot = IndustrialRobot()
```

In [122]:

```
1 robot
```

Out[122]:

```
<__main__.IndustrialRobot at 0x1a1bddb7e20>
```

In [123]:

```
1 robot.rotate_body_left()
```

```
Rotating body 10 degrees to the left...
```

In [124]:

```
1 robot.move_arm_up(15)
```

```
Moving arm 15 cm up...
```

In [125]:

```
1 robot.move_arm_up(15)
```

```
Moving arm 15 cm up...
```

In [126]:

```
1 robot.rotate_body_right(20)
```

```
Rotating body 20 degrees to the right...
```

In [127]:

```
1 robot.move_arm_down(5)
```

```
Moving arm 5 cm down...
```

In [128]:

```
1 robot.weld()
```

```
Welding...
```

In [129]:

```
1 vars(robot)
```

Out[129]:

```
{'body': <__main__.Body at 0x1a1bddb5540>,
 'arm': <__main__.Arm at 0x1a1bddb58d0>}
```

In [130]:

```
1 robot.body
```

Out[130]:

```
<__main__.Body at 0x1a1bddb5540>
```

In [131]:

```
1 robot.body.rotation
```

Out[131]:

```
10
```

In [132]:

```
1 robot.arm.position
```

Out[132]:

```
25
```

عالی! ربات شما به عنوان انتظار کار می کند. به شما اجازه می دهد بدن و بازوی خود را بر اساس نیازهای حرکتی خود حرکت دهید. همچنین به شما اجازه می دهد تا قطعات مکانیکی مختلف را با هم جوش بدهید.

یک ایده برای ساخت ربات حتی پیشرفته تر، پیاده سازی چندین نوع بازو با فناوری های جوشکاری مختلف است. سپس می توانید با اجرای `robot.arm = NewArm()` بازو را تغییر دهید. حتی می توانید یک متد `.change_arm()` به کلاس ربات خود اضافه کنید. چطور به عنوان یک تمرین یادگیری به نظر می رسد؟

بر خلاف composition، inheritance کل رابط اجزای را نمایش نمی دهد، بنابراین encapsulation را حفظ می کند. در عوض، اشیاء composite فقط به عملکردهای لازم خود از اجزای خود دسترسی و استفاده می کنند. این ویژگی باعث می شود طراحی کلاس شما قابل اطمینان و قابل اعتماد تر شود زیرا عضو های نامورده را نشان نخواهد داد.

با توجه به مثال ربات، فرض کنید چندین ربات مختلف در یک کارخانه دارید. هر ربات می تواند قابلیت های مختلف مانند جوشکاری، برش، شکل دادن، پولیش و غیره داشته باشد. همچنین چندین بازو مستقل دارید. بعضی از آنها می توانند همه آن عملکردها را انجام دهند. بعضي ديگر فقط يك زيرمجموعه از عملکردها را انجام مي دهند.

حال بگویید که يك ربات خاص فقط قادر به جوشكاري است. با اين حال، اين ربات مي تواند از بازوهای مختلف با فن آوري های جوشكاري مختلف استفاده كند. در صورت استفاده از inheritance، پس از آن كه ربات به عمليات ديگري مثل برش و شكل دادن دسترسي پيدا كرده است، مي تواند حادثه يا خراب شدگي ايجاد كند.

در صورت استفاده از composition، ربات جوشكار فقط به قابليت جوشكاري بازو دسترسي خواهد داشت. با اين حال، composition مي تواند به شما در محافظت از كلاس هاي خود در برابر استفاده ناخواسته كمك كند.

Delegation

Delegation یک تکنیک دیگری است که می توانید به عنوان جایگزین inheritance استفاده کنید. با delegation، می توانید رابطه های can-do را مدل کنید، جایی که یک شیء یک کار را به شیء دیگری منتقل می کند، که از اجرای کار مراقبت می کند. توجه داشته باشید که شیء منتقل شده مستقل از delegator وجود دارد.

می توانید از delegation برای دستیابی به separation of concerns، code reuse و modularity استفاده کنید. به عنوان مثال، فرض کنید می خواهید یک ساختار داده stack ایجاد کنید. شما به فکر بهره برداری از لیست Python به عنوان یک راه

In [133]:

```
class Stack:
    def __init__(self, items=None):
        if items is None:
            self._items = []
        else:
            self._items = list(items)

    def push(self, item):
        self._items.append(item)

    def pop(self):
        return self._items.pop()

    def __repr__(self) -> str:
        return f"{type(self).__name__}({self._items})"
```

در `__init__()` ، شما یک شیء لیست به نام `._items` تعریف می کنید که می تواند داده های اولیه خود را از آرگومان های items بگیرد. شما از این لیست برای ذخیره داده ها در Stack حاوی استفاده خواهید کرد، بنابراین تمام عملیات مربوط به ذخیره، اضافه کردن و حذف داده ها را به این شیء لیست منتقل می کنید. سپس عملیات های معمول Stack و `.push()` و `.pop()` را پیاده سازی می کنید.

توجه کنید که این عملیات ها به راحتی مسئولیت های خود را در `._items.append()` و `._items.pop()` به ترتیب منتقل می کنند. کلاس Stack شما عملیات خود را به شیء لیست منتقل کرده است، که در حال حاضر می داند چگونه آن ها را انجام دهد.

مهم است به یاد داشته باشید که این کلاس بسیار قابل انعطاف است. شما می توانید شیء لیست در `._items` خود را با هر شیء دیگری جایگزین کنید، تا زمانی که `.pop()` و `.append()` روش های آن را پیاده سازی کند. به عنوان مثال، می توانید از یک شئ deque از ماژول collections استفاده کنید.

بدلیل استفاده از delegation برای نوشتن کلاس خود، پیاده سازي داخلي ليست در Stack قابل مشاهده یا به صورت مستقیم در دسترس نیست که encapsulation را حفظ مي كند:

In [134]:

```
stack = Stack([1, 2, 3])
```

In [135]:

```
stack
```

Out[135]:

```
Stack([1, 2, 3])
```

In [136]:

```
stack.push(4)
```

In [137]:

```
1 stack
```

Out[137]:

```
Stack([1, 2, 3, 4])
```

In [138]:

```
1 stack.pop()
```

Out[138]:

```
4
```

In [139]:

```
1 stack.pop()
```

Out[139]:

```
3
```

In [140]:

```
1 stack
```

Out[140]:

```
Stack([1, 2])
```

In [141]:

```
1 dir(stack)
```

Out[141]:

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_items',
 'pop',
 'push']
```

رابط عمومی کلاس Stack شما فقط شامل متد های مربوط به stack و `.push()` و `.pop()` است، همانطور که در خروجی تابع `dir()` مشاهده می کنید. این باعث می شود که کاربران کلاس شما از استفاده از متد های خاص لیست که با ساختار داده کلاسیک stack سازگار نیستند، جلوگیری شود.

در صورت استفاده از inheritance، کلاس فرزند شما، Stack، تمام قابلیت هاي پدر خود، لیست را به ارث خواهد برد:

In [142]:

```
class Stack(list):
    def push(self, item):
        self.append(item)
    def pop(self):
        return super().pop()
    def __repr__(self) -> str:
        return f"{type(self).__name__}({super().__repr__()})"

stack = Stack()
dir(stack)
```

Out[142]:

```
['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__module__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'push',
 'remove',
 'reverse',
 'sort']
```

در نهایت در پایتون روش سریعتر برای Delegate کردن وجود دارد. این متد در پایتون به نام `__getattr__()` وجود دارد. هر زمان که به یک ویژگی یا متد نمونه دسترسی پیدا کنید، پایتون به صورت خودکار این متد را فراخوانی می کند. شما می توانید از این متد برای هدایت درخواست به یک شی دیگر که ممکن است متد یا ویژگی مناسب را فراهم کند، استفاده کنید.

برای توضیح این تکنیک، به مثال mixin برگردید که از یک کلاس mixin برای فراهم کردن قابلیت های سریال سازی به

In [ ]:

```python
import json
import pickle

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class Serializer:
    def __init__(self, instance):
        self.instance = instance

    def to_json(self):
        return json.dumps(self.instance.__dict__)

    def to_pickle(self):
        return pickle.dumps(self.instance.__dict__)

class Employee(Person):
    def __init__(self, name, age, salary):
        super().__init__(name, age)
        self.salary = salary

    def __getattr__(self, attr):
        return getattr(Serializer(self), attr)
```

در این پیاده سازی جدید، کلاس سریال ساز نمونه ای که داده ها را فراهم می کند را به عنوان یک آرگومان دریافت می کند. Employee یک متد `__getattr__()` را تعریف می کند که از تابع `getattr()` داخلی برای دسترسی به متدهای کلاس Serializer استفاده می کند.

به عنوان مثال، اگر شما `.to_json()` را روی یک نمونه از Employee فراخوانی کنید، آن تماس به صورت خودکار به `.to_json()` در نمونه Serializer هدایت خواهد شد. این را امتحان کنید! این یک ویژگی خوب پایتون است.

شما در یک مثال سریع از delegation در پایتون سعی کرده اید تا چگونگی هدایت بخشی از مسئولیت های یک کلاس به یک کلاس دیگر را برای دسترسی مجدد به کد و جداسازی مسئولیت ها یاد بگیرید. باز هم باید توجه داشته باشید که این تکنیک به طور غیر مستقیم تمام ویژگی های و متدهای منتقل شده را نشان می دهد. بنابراین، با دقت از آن استفاده کنید.

Dependency Injection

"تزریق وابستگی الگوی طراحی است که می توانید از آن برای دستیابی به جداسازی کم از یک کلاس و اجزای آن استفاده کنید. با استفاده از این تکنیک، می توانید وابستگی های یک شیء را از خارج فراهم کنید، به جای ارث بری یا پیاده سازی آن ها در خود شیء. با این روش، می توانید کلاس های انعطاف پذیری را ایجاد کنید که قادر به تغییر رفتار خود به صورت پویا هستند، بسته به قابلیت های وارد شده.

در مثال ربات شما، می توانید از تزریق وابستگی برای جداسازی کلاس های Arm و Body از IndustrialRobot استفاده کنید، که باعث می شود کد شما انعطاف پذیرتر و چند منظوره تر شود.

مثال به روز شده در زیر آمده است:"

In [ ]:

```
class IndustrialRobot:
    def __init__(self, body, arm):
        self.body = body
        self.arm = arm
    # rest of the class methods ...
```

"در این نسخه جدید از IndustrialRobot، فقط دو تغییر کوچک در `__init__()` انجام دادید. اکنون این روش body و arm را به عنوان آرگومان ها می پذیرد و مقادیر آن ها را به ویژگی های نمونه مربوطه، `.body` و `.arm` ، اختصاص می دهد. این به شما اجازه می دهد تا شیء های body و arm مناسب را در کلاس تزریق کنید تا بتواند کار خود را انجام دهد.

اینجاست که چگونه می توانید از IndustrialRobot با این پیاده سازی جدید استفاده کنید:"

In [ ]:

```
robot = IndustrialRobot(Body(), Arm())
```

"در کل، عملکرد کلاس همانند نسخه اول شما باقی می ماند. تنها تفاوت این است که اکنون باید شیء های body و arm را به سازنده کلاس منتقل کنید. این مرحله یک روش رایج برای پیاده سازی تزریق وابستگی است."

ایجاد Abstract Base Classes (ABCs) و Interfaces

"گاهی اوقات، می خواهید سلسله مراتب کلاس را ایجاد کنید که تمام کلاس ها یک رابط یا API پیش تعریف شده را پیاده سازی کنند. به عبارت دیگر، می خواهید مجموعه خاصی از روش ها و ویژگی های عمومی را تعریف کنید که تمام کلاس های سلسله مراتب باید آن ها را پیاده سازی کنند. در پایتون، می توانید این کار را با استفاده از چیزی به نام یک کلاس پایه انتزاعی (ABC) انجام دهید.

ماژول abc در کتابخانه استاندارد چندین ABC و ابزار مرتبط دیگر را صادر می کند که می توانید از آن ها برای تعریف کلاس های پایه سفارشی استفاده کنید که تمام زیرکلاس های خود را برای پیاده سازی رابط های خاص الزام می دارند.

شما نمی توانید به طور مستقیم ABCs را نمونه سازی کنید. شما باید آن ها از طریق زیرکلاس تعریف و استفاده نمایید. به نوعی، ABCs به عنوان الگوهای برای دیگر کلاس ها برای به ارث بردن عمل می کنند.

برای نشان دادن چگونگی استفاده از ABCs در پایتون، فرض کنید می خواهید یک سلسله مراتب کلاس برای نمایش شکل های مختلف، مانند دایره، مربع و غیره ایجاد کنید. شما تصمیم می گیرید که تمام کلاس ها باید `.get_area()` و `.get_perimeter()` را دارا باشند. در این شرایط، شما می توانید با چنین کلاس پایه اولیه زیر شروع کنید:"

In [143]:

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def get_area(self):
        pass

    @abstractmethod
    def get_perimeter(self):
        pass
```

"کلاس Shape از abc.ABC به ارث می برد، به این معنی که یک کلاس پایه انتزاعی است. سپس شما با استفاده از تزئین کننده @abstractmethod، روش های get_area() و get_perimeter() را تعریف می کنید. با استفاده از تزئین کننده @abstractmethod، شما اعلام می کنید که این دو روش رابط عمومی هستند که تمام زیرکلاس های Shape باید پیاده سازی کنند.

حال می توانید کلاس Circle را ایجاد کنید. در اینجا رویکرد اول به این کلاس وجود دارد:"

In [144]:

```python
from abc import ABC, abstractmethod
from math import pi

# ...

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def get_area(self):
        return pi * self.radius ** 2
```

"در این قطعه کد، شما با به ارث بردن از Shape، کلاس Circle را تعریف می کنید. در این نقطه، فقط روش get_area() را اضافه کرده اید. حالا به جلو بروید و کد زیر را اجرا کنید:"

In [145]:

```python
circle = Circle(100)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[145], line 1
----> 1 circle = Circle(100)

TypeError: Can't instantiate abstract class Circle with abstract method ge
t_perimeter
```

In [146]:

```
from abc import ABC, abstractmethod
from math import pi

# ...

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def get_area(self):
        return pi * self.radius ** 2

    def get_perimeter(self):
        return 2 * pi * self.radius
```

این بار، کلاس Circle شما تمام متد های مورد نیاز را پیاده سازی می کند. این متد ها برای تمام کلاس های سلسله مراتب شکل شما مشترک هستند. پس از تعریف پیاده سازی های سفارشی مناسب برای تمام متد های انتزاعی، می توانید به ایجاد نمونه Circle بپردازید، همانطور که در مثال زیر استفاده شده است:

In [147]:

```
circle = Circle(100)
```

In [148]:

```
circle.radius
```

Out[148]:

```
100
```

In [149]:

```
circle.get_area()
```

Out[149]:

```
31415.926535897932
```

In [150]:

```
circle.get_perimeter()
```

Out[150]:

```
628.3185307179587
```

پس از پیاده سازی متد های سفارشی برای جایگزین کردن پیاده سازی های انتزاعی `.get_area()` و `.get_perimeter()` ، می توانید Circle را در کد خود نمونه سازی و استفاده کنید.

حال برای تمرین می توانید یک کلاس مربع با استفاده رابط انتزاعی shape تعریف کنید.

In [151]:

```python
class Square(Shape):
    def __init__(self, side):
        self.side = side

    def get_area(self):
        return self.side ** 2

    def get_perimeter(self):
        return 4 * self.side
```

خدا قوت!! عالی بود، تمام مباحث مربوط به شی گرایی در پایتون، ار صفر تا صد رو یادگرفتیم 😉😉😉