مفهوم شیءگرایی

هدف از این فصل آشنایی با مفاهیم شیءگرایی است. پایتون زبانی شیءگرا[1]است که از تمام ویژگی‌ها و قابلیت‌های زبان‌های شیءگرا پشتیبانی می‌کند. در پایتون هر زمان که متغیری ایجاد می‌شود، متغیر نمونه یا شیء از یک کلاس است. به طور مثال با ایجاد متغیر نوع رشته، این متغیر نمونه‌ای از کلاس str خواهد بود. همین‌طور اگر متغیری از نوع لیست یا تاپل یا دیکشنری ایجاد کنید، این متغیرها نمونه‌ای از کلاس‌های list یا tuple یا dictionary خواهند بود. به همین دلیل ساخت و استفاده از کلاس‌ها و اشیاء در پایتون آسان است.

در تفکر شیءگرایی، سه اصطلاح مهم وجود دارد که عبارتند از:

- تحلیل شیءگرا (OOA) : فرآیند بررسی مساله ← (خروجی) مجموعه‌ای از نیازمندی‌ها
- طراحی شیءگرا (OOD) : به فرآیند تبدیل نیازمندی‌های شناسایی شده در مرحله تحلیل به تعدادی کلاس، طراحی شیءگرا گفته می‌شود.
- برنامه‌نویسی شیءگرا (OOP) : به فرآیند تبدیل طراحی انجام شده به برنامه‌ای که نیازهای مشتریان را برآورده می‌کند، برنامه‌نویسی شیءگرا گفته می‌شود.

برنامه‌نویسی شیءگرا

برنامه‌نویسی شیءگرا[2]یک الگوی تفکر در برنامه‌نویسی است که برگرفته از دنیای واقعی بوده و از دهه ۱۹۶۰ میلادی مطرح گشته است. به زبانی که از این الگو پشتیبانی کند، «زبان شیءگرا» گفته می‌شود. Simula 67 و Smalltalk نخستین زبان‌های برنامه‌نویسی شیءگرا هستند. ایده‌ی شیءگرایی در پاسخ به برخی از نیازها که الگوهای موجود پاسخ‌گو آن‌ها نبودند به وجود آمد. نیازهایی مانند توانایی حل تمامی مسائل پیچیده[3]، پنهان‌سازی داده[4]، قابلیت استفاده مجدد[5]، وابستگی کم‌تر به توابع، انعطاف بالا و غیره.

رویکرد برنامه‌نویسی شیءگرا «از پایین به بالا»[6] است؛ یعنی ابتدا واحدهایی کوچک از برنامه ایجاد می‌شوند و سپس با پیوند این واحدها، واحدهایی بزرگ‌تر و در نهایت شکلی کامل از برنامه به وجود می‌آید. در برنامه‌نویسی شیءگرا، هر برنامه در قالب موجودیت‌های کوچکی که در واقع همان اشیاء هستند و با یکدیگر تعامل دارند در نظر گرفته می‌شود.

---

1 . Object Oriented
2 . Object-Oriented Programming=OOP
3 . Complex
4 . Data Hiding

مزایای شیءگرایی

- بهینه شدن ساختار برنامه
- استفاده‌ی مجدد از کدها.
- کپسوله‌سازی
- وراثت

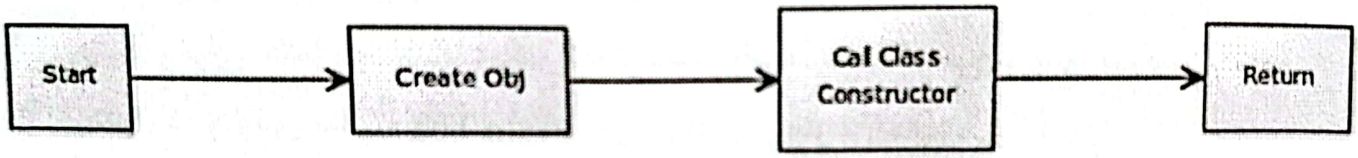
مفاهیم اساسی شیءگرایی در پایتون

- کلاس
- صفت
- متد
- شیء
- نمونه‌سازی[5]
- کپسوله‌سازی[7]
- وراثت[4]
- چندریختی[6]
- انتزاع یا تجرید[8] داده

✦ بهترین روش نوشتن تابع به صورت متد است.

متد سازنده

هر بار که یک شیء ایجاد می‌شود یک متد فراخوانی می‌شود. آن متد سازنده نام دارد. متد سازنده با تابع init ایجاد می‌شود. به عنوان پارامتر کلمه کلیدی self را می‌نویسیم که به خودش (شیء) اشاره دارد. فرآیند به صورت بصری زیر است:



شکل(۱-۹). فرآیند متد سازنده

بنابراین، تابع __init__() تابعی است که برای پارامترها مقدار اولیه تعریف می‌کند و در زمان ایجاد شیء به صورت خودکار فراخوانی می‌شود و بر اساس پارامترهایی که برای آن مشخص شده شیء ایجاد شده، مقداردهی می‌شود.

به مثال زیر توجه کنید. در داخل سازنده دو متغیر name و age مقداردهی اولیه شده است. سپس یک آبجکت از نوع کلاس تعریف شده و به محض فراخوانی، متغیرهای آن مقداردهی اولیه می‌شوند.

```
class Human:
    def __init__(self):
        self.name = 'ali'
        self.age = 30
obj = Human()
print('name=',obj.name,'\tage=',obj.age)
```

به مثال بعدی توجه کنید:

```
class Student:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def avg(self):
        return (self.a + self.b) / 2
a=int(input('Enter a= '))
```

فرم کلی آن به صورت زیر:

```
class base:
        pass
class derived(base):
        pass
```

نکات وراثت در کلاس

- اگر کلاس پایه در ماژول دیگری ذخیره شده باشد کلاس فرزند را به شکل زیر می‌نویسیم:

```
Class derived(module name.class name):
```

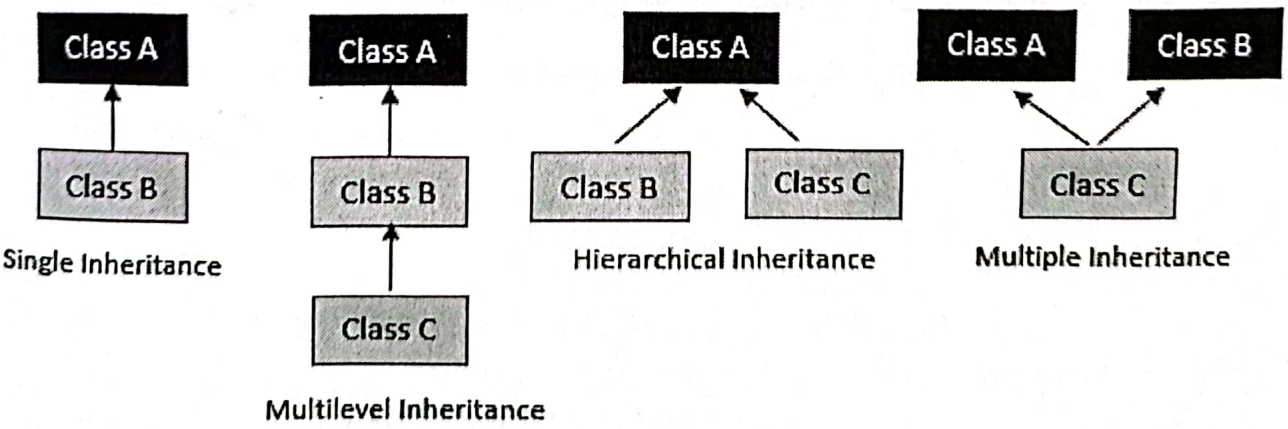
- اگر هر کدام از ماژول‌ها در یک درایو ذخیره شده باشد باید قبل از نام ماژول باید آدرس‌دهی کنیم.
- تغییر متد کلاس پایه توسط کلاس فرزند این زمانی انجام می‌شود که ما نخواهیم خود کلاس پایه تغییری کند.

مزایای ارث‌بری

- جلوگیری از دوباره نوشتن کد
- کوچک شدن برنامه
- بالا رفتن سرعت کدنویسی

انواع وراثت در پایتون

انواع وراثت بستگی به تعداد کلاس‌های فرزند و والدین دارد. چهار نوع وراثت در پایتون وجود دارد که در شکل(۳-۹) نشان داده شده است:



شکل(۳-۹). انواع وراثت در پایتون

وراثت یگانه (Single Inheritance)

وراثت منفرد یک کلاس مشتق شده را قادر می‌سازد تا ویژگی‌های یک کلاس والد را به ارث ببرد، بنابراین قابلیت استفاده مجدد کد و افزودن ویژگی‌های جدید به کد موجود را ممکن می‌سازد. در شکل(۴-۹)، Class B از Class A ارث‌بری می‌کند.

چندریختی

زمانی ما از چند ریختی‌ها استفاده می‌کنیم که بخواهیم یک تابع هم‌نام در کلاس‌های مختلف کار متفاوتی انجام دهد. برای مثال اگر یک کلاس به نام شکل داشته باشیم و کلاس‌های فرزند آن کلاس مستطیل، دایره و مثلث باشند، و درون هریک از کلاس‌های فرزندمان یک تابع با نام رسم شکل وجود داشته باشد. نکته قابل توجه این است که نام تابع در همه کلاس‌ها یکسان است اما در هر یک از کلاس‌ها متفاوت عمل می‌کند.

هنگام تعریف تابع‌های هم‌نام به این نکته توجه داشته باشید که این توابع باید یا تعداد متغیرهایشان متفاوت باشد یا نوع متغیرهایشان. توابعی که هم‌ریخت هستند در هنگام فراخوانی مشخص می‌شود که کدام یک فراخوانی شده‌اند.

دو نوع چندریختی وجود دارد: چندریختی در برنامه و چندریختی در شیءگرایی

output:

It is a shape

It is a rectangle

It is a triangle

کپسوله‌سازی

قبل از این که به کپسوله‌سازی بپردازیم، بهتر است که سطوح دسترسی به اعضای یک کلاس را بررسی کنیم.

سطوح دسترسی به اعضای یک کلاس

سطوح دسترسی به اعضای یک کلاس عبارتند از:

- نوع Public
- نوع Protected
- نوع Private

اعضای کلاس از نوع Public

در این نوع از دسترسی، متغیر یا تابع در کلاس‌های دیگر و نمونه‌های ایجاد شده از کلاس قابل دسترسی است. همه‌ی متغیرها و متدهای کلاس به صورت پیش‌فرض، public هستند. به مثال زیر توجه کنید:

```
class person:
    def __init__(self):
        self.name=''
        self.code=0
obj=person()
obj.name='Mina'
obj.code=123
print(obj.name,obj.code)
```

به کد زیر توجه کنید:

```
class person:
    id = None
    name = None

    def __init__(self, id, name):
        self.id = id        #public attribute
        self.name = name    #public attribute

    #public class function
    def show(self):
        print('id: {},Name: {}'.format(self.id, self.name))
obj = person(10,'Ali')
obj.show()

output: id: 10,Name: Ali

obj.id = 12;obj.name ='Reza'
obj.name = 14;obj.name ='Zahra'
obj.show()

output: id: 12, Name: Zahra
```

همان‌طور که در دو مثال فوق ملاحظه می‌کنید چون متغیرهای name و id از نوع Public تعریف شده‌اند به راحتی می‌توان عملیات چاپ، حذف، به‌روزرسانی را انجام داد.

اعضای کلاس از نوع Protected

در این نوع از دسترسی، متغیر یا تابع، در کلاسی که در آن تعریف شده و کلاس‌هایی که از این کلاس ارث برده‌اند قابل دسترس است. در زبان‌هایی مانند Java و C++ برای این‌که اعضای یک کلاس فقط در کلاس‌های فرزند قابل دسترسی باشند. فقط کافی بود از کلمه‌ی Protected در ابتدای آن‌ها استفاده کنیم. اما این مکانیسم در پایتون با قواعد نام‌گذاری[1] است بدین صورت که کافی است یک آندرلاین (_) در ابتدای نام متغیرها و متدهای کلاس با دسترسی Protected قرار دهید. به مثال زیر توجه کنید:

```
class person:
    _id=None
    _name=None
    def __init__(self,id,name):
        self._id=id                 #protected attribute
        self._name=name             #protected attribute
    #protected class function
    def _show(self):
```

1. convention

```
        print('id: {}' . format(self._id))
#subclasses
class employee(person):
    def __init__(self,id,name):
        person.__init__(self,id,name)
    def displayname(self):
        print(self._name)
        self._show()
obj=person(10,'Ali')
obj._show()

output: id: 10

obj1=employee(12,'Reza')
obj1.displayname()
output:
Reza
id: 12
```

اعضای کلاس از نوع Private

در این نوع از دسترسی، متغیر یا تابع فقط در کلاسی که در آن تعریف شده است قابل دسترسی است. اعلان Private برای اعضای یک کلاس به این معناست که کسی نباید خارج از کلاس از آن‌ها استفاده نماید. پایتون از یک تکنیک به نام Name mangling استفاده می‌کند. این تکنیک باعث می‌شود تا هر عضوی که با دو آندرلاین (_) شروع یا پایان یافته است، به عنوان _<className><memberName> شناخته شود. به مثال زیر توجه کنید:

```
class person:
    def __init__(self,id,name):
        self.__id=id            #private attribute
        self. name=name         #public attribute
    #private class function
    def __show(self):
        print('id: {},Name: {}' . format(self.__id,self.__name))

obj=person(10,'Ali')
obj.__show()
```

output: AttributeError: 'person' object has no attribute '__show'

همان‌طور که ملاحظه می‌کنید؛ برای دسترسی به متد ___show () پیام خطا صادر شده است. چون این متد از نوع private تعریف شده، به همین دلیل امکان دسترسی در حالت عادی وجود ندارد.

```
print(obj.__id)
output: AttributeError: 'person' object has no attribute '__id'
```

برای دسترسی به ویژگی id__ نیز پیام خطا صادر شده است. زیرا این ویژگی نیز از نوع private است.

برای دسترسی به اعضای یک کلاس که از نوع private هستند، می‌توانید از قاعده زیر استفاده کنید:

instance_classname_attribut/method

```
obj._person__show()
print(obj._person__name)
```

output:
id: 10,Name: Ali
Ali

کپسوله‌سازی چیست؟

کپسوله‌سازی به معنی جلوگیری از تغییر مستقیم داده‌ها است. با استفاده از کپسوله‌سازی دسترسی مستقیم برنامه‌نویسان به متدها و سایر المان‌های یک شیء محدود می‌شود. این از تغییر مستقیم داده‌ها که به آن کپسوله کردن می‌گویند جلوگیری می‌کند. در واقع کپسوله‌سازی در شیءگرایی امکانی است برای پنهان‌سازی داده‌ها. در این شرایط اشیاء بدون این‌که از درون یکدیگر و چگونگی کارکرد هم کوچک‌ترین آگاهی داشته باشند به تعامل با یکدیگر می‌پردازند. برای اعمال تغییرات می‌توان از روش دریافت‌کننده (getter) و تنظیم‌کننده (setter) و حذف‌کننده (deleter) کمک گرفت. به مثال زیر توجه کنید:

```
class car:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print(f'Selling Price={self.__maxprice}')

pride=car()
pride.sell()

#change the price
pride.__maxprice=1000
pride.sell()
```

خروجی به صورت زیر:

Selling Price=900
Selling Price=900

در این مثال؛

۱. از روش ()__init__ برای ذخیره حداکثر قیمت فروش استفاده کردیم.
۲. سعی کرده‌ایم مقدار maxprice__ را خارج از کلاس تغییر دهیم. ولی از آنجایی که maxprice__ یک متغیر خصوصی است، این تغییر در خروجی دیده نمی‌شود.
۳. برای تغییر مقدار، باید از یک تابع تنظیم‌کننده استفاده کنیم مانند ()setMaxPrice که قیمت را به عنوان یک پارامتر می‌گیرد. پس مثال را به صورت زیر تکمیل می‌کنیم:

انتزاع یا تجرید

تجرید در برنامه‌نویسی شیءگرا به همراه مفهوم چندریختی می‌آید و توسط دو مفهوم «کلاس‌های مجرد یا انتزاعی»[1] و «متدهای مجرد یا انتزاعی»[2] ارایه می‌گردد.

«کلاس انتزاعی» در پایتون به کلاسی گفته می‌شوند که پایه کلاس‌های دیگر هستند. کلاس‌های انتزاعی به خودی خود کار خاصی انجام نمی‌دهند فقط رفتار کلی کلاس‌های فرزند را مشخص می‌کنند. کلاس انتزاعی شامل یک یا چند «متد انتزاعی» است. «متد انتزاعی» متدی است که اعلان[3] شده ولی بدنه آن تعریف نشده است یعنی پیاده‌سازی ندارد. کلاس‌های انتزاعی قابلیت نمونه‌سازی ندارند و نمی‌توان از آن‌ها شیء ایجاد نمود؛ چرا که هدف از توسعه آن‌ها قرار گرفتن در بالاترین سطح (یا چند سطح بالایی) درخت وراثت، به عنوان کلاس پایه برای ارث‌بری کلاس‌های پایین‌تر می‌باشد. ایده طراحی کلاس انتزاعی در تعیین یک نقشه توسعه برای کلاس‌های فرزند آن است؛ تعیین صفات و متدهای لازم ولی واگذاردن تعریف متدها بر عهده کلاس‌های فرزند است.

به عنوان نمونه سه کلاس «طوطی»، «گربه» و «کبوتر» را در نظر بگیرید. این کلاس‌ها جدا از رفتارهای خاص خود (مانند: «آواز خواندن» در طوطی ، «شنا کردن» در ماهی، «پریدن» در گربه)، در یک سری رفتار به مانند «نفس کشیدن»، «غذا خوردن» و ...مشترک هستند. راه درست توسعه این کلاس‌ها تعیین یک «کلاس پایه» برای رفتارهای مشترک و ارث‌بری هر سه آن‌ها می‌باشد. ولی از آنجا که هر یک، این رفتارهای مشترک را به گونه‌ای دیگر انجام می‌دهد؛ راه درست‌تر آن است که یک «کلاس مجرد» به عنوان «کلاس پایه» آن‌ها در نظر بگیریم؛ در این حالت هر کدام از کلاس‌ها ضمن دانستن رفتارهای لازم می‌تواند آن‌ها را متناسب با خواست خود تعریف نماید(پیاده‌سازی این مثال به عهده دانشجو می‌باشد).

چرا از کلاس‌های پایه انتزاعی استفاده کنیم؟

با تعریف یک کلاس پایه انتزاعی، می‌توانید یک رابط برنامه کاربردی مشترک (API) برای مجموعه‌ای از کلاس‌های فرعی تعریف کنید. این قابلیت به ویژه در شرایطی مفید است که شخص ثالثی قرار است پیاده‌سازی‌هایی را ارائه کند، مانند افزونه‌ها.

نحوه‌ی کار کلاس‌های پایه انتزاعی

به طور پیش‌فرض، پایتون کلاس‌های انتزاعی را ارائه نمی‌دهد. پایتون دارای ماژولی است که پایه‌ای را برای تعریف کلاس‌های پایه انتزاعی (ABC) فراهم می‌کند و نام ماژول abc است. به مثال زیر توجه کنید: فرض کنید سه کلاس به نام‌های Triangle ، Pentagon و Hexagon داریم که از کلاس Polygon ارث‌بری می‌کنند.

```
class Polygon:
    def num_of_side(self):
        pass
class Triangle(Polygon):
    pass
class Pentagon(Polygon):
    pass
class Hexagon(Polygon):
    pass

t = Triangle()
t.num_of_side()

p = Pentagon()
p.num_of_side()

h = Hexagon()
h.num_of_side()
```

با اجرای کد فوق در خروجی چیزی نمایش داده نمی‌شود. زیرا متد کلاس پایه بدنه آن پیاده‌سازی نشده است.

اکنون اگر بدنه متد کلاس پایه را به صورت زیر پیاده‌سازی کنیم:

```
class Polygon:
    def num_of_side(self):
```