## **الگوی طراحی یک راه حل کلی قابل تکرار برای یک مشکل معمول در طراحی نرم افزار است. الگوی طراحی یک طرح تمام شده نیست که بتواند مستقیماً به کد تبدیل شود بلکه توضیحات یا الگویی برای نحوه حل مسئله است که می تواند در شرایط مختلف مورد استفاده قرار گیرد. الگوهای طراحی می توانند با ارائه پارادایم های آزمایش شده و اثبات شده توسعه ، روند توسعه را تسریع کنند. طراحی موثر نرم افزار مستلزم در نظر گرفتن مواردی است که ممکن است در مراحل ابتدایی قابل مشاهده نباشد. استفاده مجدد از الگوهای طراحی کمک می کند تا از ایجاد شدن مسائل ظریفی که می توانند مشکلات اساسی ایجاد کنند جلوگیری شود. غالباً ، مردم فقط می دانند که چگونه برخی از تکنیک های طراحی نرم افزار را برای مشکلات خاص به کار بگیرند. به کارگیری این تکنیک ها برای طیف وسیعی از مشکلات دشوار است. الگوهای متداول طراحی می توانند با گذشت زمان بهبود یابند و آنها را نسبت به طرح های موقت مقاوم تر سازد**

**به طور خلاصه الگوهای طراحی پاسخی به مشکلات متداول در تولید نرم‌افزار است**

الگوهای طراحی به ۳ بخش تقسیم می‌شوند که عبارت اند از :creational و structural و behavioral

نکته ی بسیار مهم این است که الگوهای طراحی قطعه کد نیستند که ما آن‌ها را حفظ کنیم . الگوهای طراحی مفهوم هستند که باید آن‌ها را درک کنیم

creational : الگوهای طراحی هستند که روی نحوه ی ایجاد شدن یک object از یک class تمرکز میکنند

structual : الگوهای طراحی هستند که روی نحوه ی تعامل بین کلاس‌ها تمرکز میکنند

behavioral :

========================================================================

در پایتون متد call به ما اجازه میده که نمونه هایی که از کلاس ها ایجاد میکنیم رو بتونیم مثل یک فانکشن صدا بزنیم به زبان دیگر هنگامیکه نمونه‌ای که از کلاس ساخته شده است به صورت فانکشن صدا زده شود این متد فراخوانی می‌شود ((قطعه کد call.py))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

در پایتون متد new  مسئول ساخت یک کلاس جدید هست. متد  new تنها متدیست که قبل از متد init  اجرا شده و میتواند مانع فعال شدن متد init شود به زبان دیگر اگر بخواهید نحوه ی ایجاد شدن کلاس را تغییر دهید از \_\_new\_\_ استفاده میکنید

خیلی جاها گفته شده که \_\_init\_\_ سازنده ی کلاس هست اما این جمله اشتباه میباشد چرا که سازنده ی کلاس \_\_new\_\_ میباشد .

\_\_init\_\_ فقط initializer (آغازگر) میباشد و کارهای اولیه را انجام میدهد

توجه شود که \_\_new\_\_ اولین مقداری که دریافت میکند cls میباشد یعنی خود کلاس بر خلاف \_\_init\_\_ که اولین مقداری که میگیرد self میباشد

دقت کنید که بعد از اینکه تغییراتی در \_\_init\_\_ به وجود آورید حتماً باید فانکشن new کلاس پدر را return کنید هماننده کد زیر :

return super().\_\_new\_\_(cls, \*args, \*\*kwargs)

به طور کلی نحوه ی کارکردن \_\_new\_\_ به این شکل میباشد:زمانی که نمونه‌ای از کلاس در حال ایجاد شدن میباشد \_\_new\_\_ سریع فعال می‌شود و کلاس خودش را به عنوان object به cls ارسال میکند ( به خودش ارسال میکند ) و در ادامه میتوانیم استفاده هایی را که مد نظر داشتیم را بکنیم

(( قطعه کد new.py ))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

متا کلاس ها در پایتون برای تغییر در نحوه رفتار دیگر کلاس ها به کار میرن

در بالا اشاره شد که کلاسی که هنوز ساخته نشده است خودش را به عنوان object به خودش ارسال میکند این موارد چگونه ممکن است به کمک meta class ها

هنگامیکه خروجی قطعه کد زیر را اجرا کنیم خروجی به این مفهوم میباشد که این کلاس از نوع type میباشد . این type همان typeی میباشد که مشخص میکند هر چیزی از چه نوعی میباشد :

a1.\_\_class\_\_.\_\_class\_\_

<class ‘type’>

print(type(213))

ــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ

type به 2 شکل زیر استفاده می‌شود که با استفاده از دستور help(type) قابل مشاهده میباشد:

type(object) -> the object’s type

type(name, base, dict) -> a new type

حالت دوم به این معنی میباشد که با استفاده از type میتوانیم کلاس جدید ساخته می‌شود به این صورت که name اسم کلاس جدیدمان میباشد , base یک تاپل میباشد که قرار است این کلاس جدید از آن‌ها ارث بری میکند , dict فانکشن هایی است که قرار است این کلاس جدیدمان داشته باشد

متد type بالاترین و قویترین meta class ی است که در پایتون وجود دارد

تمامی کلاس‌هایی رو که ایجاد میکنیم به صورت پیش‌فرض type را به عنوان meta class خودشون دارند و برای ساخته شدن از این meta class استفاده میکنند و برای ساخته شدنشون از type استفاده میکنند برای همین است که در \_\_new\_\_ میتوانیم خوده کلاس را به clsی که ما فکر میکنیم هنوز ساخته نشده است ارسال کنیم که در‌واقع در پس زمینه type کلاس را میسازد

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ما میتوانیم meta classهای خود را ایجاد کنیم کافیست meta class ما از type ارث بری کند و برای استفاده به کلاس مورد نظر اینگونه استفاده می‌شود :

class MyMetaClass(type): ......

class one (metaclass = MyMetaClass): .....

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

در قطعه کد metaclass.py کلاسی به عنوان metaclass میسازیم که باعث می‌شود که از کلاس db فقط و فقط یک نمونه ساخته شود

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

دیزاین پترن singleton به ما این امکان رو میده که بتونیم کلاس ها رو جوری بسازیم که از اون کلاس فقط بشه یک نمونه ساخت.و اگر نمونه ی دیگری ساخته شود همان نمونه ی اول را برمیگرداند

خوده پایتون هم از singleton استفاده میکند . زمانی که دوباره یک import خاصی انجام می‌شود

(( قطعه singleton.py ))

========================================================================

Abstract Class ,Abstract Method

کلاس های انتزاعی در پایتون به کلاس هایی گفته میشوند که پایه کلاس های دیگر هستند. کلاس های انتزاعی به خودی خود کار خاصی انجام نمیدهند فقط رفتار کلی کلاس های فرزند را مشخص میکنند

abstract class ها کلاسهایی هستند که کلاس‌های فرزندشون رو مجبور میکنند که متدد های خودشون (متدهای abstract class) رو یکبار دیگر owerwrite کنند

قطعه کد (abstract.py)

ما در قطعه که در مثال اول دوتا کلاس داریم به اسم A و ‌‌B که کلاس B داره از کلاس A ارث بری میکنه.

در کلاس A یک متد به نام show وجود داره. در کلاس B میتونه متد show رو نادیده بگیره و بازنویسیش نکنه.  
اما اگه بخواید حالتی رو بوجود بیارید که کلاس B مجبور به بازنویسی متد show شود میتونید از AbstractClass استفاده کنید.

در مثال دوم کلاس C رو به یک کلاس انتزاعی تبدیل کردیم. کلاس ABC به شما اجازه میده کلاس هاتون رو انتزاعی کنید تا بقیه کلاس ها فرزند مجبور به بازنویسی متدهاش بشن. در این مثال کلاس D متد show رو بازنویسی نکرده و انتظار داریم که پایتون بهمون ارور بده. اما اگه کد بالا رو اجرا کنید میبینید که اتفاقی نمیفته. چرا؟ چون پایتون اعتقادی به زور کردن یک موضوع نداره. اون کلاس ABC که کلاس A داره ازش ارث بری میکنه بیشتر یک حالت نمایشی داره تا برنامه نویس بفهمه که داره از یک کلاس انتزاعی ارثبری میکنه. اما اگه احیانا دلتون بخواد واقعا برنامه نویس رو مجبور به بازنویسی کنید میتونید متدهاتون رو هم به شکل انتزاعی دربیارید که این کار را در مثال سوم مشاهده میکنید

این‌طور که به نظر میرسه ما نباید در کلاس‌ها و متدهای abstract کار خاصی را انجام دهیم و فقط لازم است که حالت کلی را در این کلاس‌ها پیاده‌سازی کنیم اما میتوانیم فانکشنالیتی خاصی به این کلاس‌ها اضافه کنیم و با استفاده از متد super به این فانکشنالیتی دسترسی پیدا کنیم که در مثال آخر این فایل میتوانید ببینید

========================================================================

دیزاین پترن factory این امکان رو به ما میده که بتونیم superclass بسازیم که مسئول ساخت object باشه و به subclass اجازه بدیم که بتونن نوع آبجکتی که ساخته میشه رو تغییر بدن

در قطعه کد factory.py مثال اولی که زده شده است ۲ مشکل اساسی دارد : اول اینکه تمام پروسه edit رو سپردیم دست if و else ها اگر بتوانیم فرمت های بیشتری را پشتیبانی کنیم باید به تعداد این فرمت ها if و else اضافه کنیم دوم اینکه پروسه edit را داخل بلاک if قرار داده‌ایم باید برای هر edit format فانکشن مخصوص به خود را داشته باشد سوم اینکه متد edit داره کار زیادی رو انجام میده هر formatی که پشتیبانی می‌شود edit میکند (نکته مهمی که در پایتون باید ریایت کرد این است که هر فانکشنی باید یک کار خاصی رو انجام بده )

مثال‌های بعدی هر کدام یک مشکل را بر طرف میکنند :   
مثال دوم این است که edit هر format داخل فانکشن مخصوص به خود قرار دارد مثال سوم این است که تابع get فقط یک کار را انجام دهد (هر تابع یک کار انجام دهد)

هر factory method از ۳ component باید تشکیل شود: ۱.client: آن بخشی از کد که کاربران فقط با آن در ارتباط هستند و با بقیه ی کد کاری ندارد در مثال سوم تابع edit این کار رو انجام میدهد ۲.creator: آن بخشی از کد که تصمیم میگیره که چه نوع objectی ساخته شود در مثال سوم تابع get\_edit این کارو انجام میدهد ۳.product: آن بخشی از کد که در مرحله آخر برای ما یک چیز خاصی رو تولید میکند که در مثال سوم دو تابع edit\_json و edit\_xml این کارو برای ما انجام میدهد بخش دیگری هم وجود دارد که جز این ۳ component نمیباشند به نام identifier که creator بر اساس آن تصمیم میگیرد که چه کاری انجام دهد که format برای این منظور میباشد

========================================================================

در پایتون دیزاین پترن abstract factory به ما این امکان رو میده که بتونیم آبجکتی بسازیم که بتونه به چندین شکل مختلف ایجاد بشه ما باید این دسته بندی هارو به شکل مناسبی کنترل کنیم که در صورت تغییر یک دسته بندی نیاز نباشد زیاد در کد تغییر دهیم

abstract factory به این شکل است که چندین factory method در کنار هم قرار میدهیم قطعه کد abstractfactory.py

========================================================================

دیزاین پترن prototype به ما کمک میکنه که بتونیم یک کپی از آبجکتی که قبلا ساخته شده بگیریم تا نیاز نباشه که آبجکت جدیدی اینجاد بشه که در مصرف منابع صرفه جویی بشه نکته مهمی که وجود دارد این است که ما میتوانیم دقیقاً کپی که از object میگیریم همانند خود object باشد و هم میتوانیم در بعضی از خصوصیات تغییر دهیم

قطعه کد prototype.py

برای اطلاعات بیشتر درباره copy میتوانید فایل Shallow and Deep Copy در   
Important-points-of-Python-and-Django-and-etc را مطالعه شود

========================================================================

دیزاین پترن builder زمانی استفاده میشه که ما یک محصولی قراره بسازیم که اون محصول از اجزای کوچکتری ساخته شده که اون اجزای کوچکتر هم باید ایجاد بشن تا ما بتونیم از اونها استفاده کنیم

در design pattern builder یک کلاس مهمی وجود دارد به نام director که تمام اجزایی که ساخته شده است را کنار هم قرار میدهد

همان‌طور که در کد مشاهده میکنید مراحل ساخت یک object پیچیده رو به قسمت‌های کوچکتر تقسیم کردیم . ۳ کلاس ‌Body و Engine و Wheel قسمت‌های کوچکتر میباشند که اول آن‌ها ساخته می‌شوند .بعد بخش‌های بزرگ‌تر که کلاس‌های ‌Builder و Benz و Bmw میباشند ساخته می‌شود . بعد حالت کلی آن ساخته می‌شود که کلاس Car این حالت کلی میباشد. بعد با استفاده از Director متد ها را به ترتیب صدا زدیم یعنی اول body رو صدا زدیم و آن را داخلcar ریختیم و در آخر خوده car را return کردیم (director آن متد هایی که لازم است را به ترتیب صدا میزند)  
دقت کنید که builderهایمان(Benz و Bmw) از متدهای یکسان استفاده میکند

قطعه کد builder.py

========================================================================

دیزاین پترن adapter به ما این امکان رو میده که بتونیم با ایجاد کردن یک کلاس واسط، کلاس هایی که نمیتونن با هم کار کنند رو به هم ربط بدیم



دیزاین پترن adapter هم دقیقاً مثل شکل بالا میباشد ما وقتی دو کلاس داشته باشیم که به هیچ صورتی نشود بین آن‌ها ارتباط برقرار کرد در این حالت مینوانیم از adapter استفاده کنیم

این دیزاین پترن از ۳ بخش تشکیل می‌شود :  
client : در‌واقع کاربری هست که میخواهد محصولی تغییر کند الان ما دوست داریم که این ۳شاخه تغییر کند پس ما client هستیم   
adaptee : اون محصولی هست که میخواهیم تغییر کند الان این ۳شاخه adaptee میباشد   
adapter : اون چیزی هست که اون تغییر درخواستی ما را انجام میدهد

قطعه کد adapter.py

========================================================================

در پایتون از args و kwargs برای هندل کردن آرگومان های پیش بینی نشده استفاده میشود. این آرگومانها میتوانند به شکل ساده یا دیکشنری ارسال شوند

به مثال زیر دقت کنید

def show(name):

print( f'Hello {name}' )

show('amir', 'ali')

در مثال بالا یه دونه فانکشن داریم به اسم show که انتظار داره یک مقدار رو بهش بفرستیم و اون یک مقدار رو بگیره و چاپ کنه. اما ما دوتا مقدار رو فرستادیم و این کد به ما ارور میده.

اینجا میتونید از \*args استفاده کنید. args به شما اجازه میده که پارامترهای اضافی رو بگیرید.

def show(name, \*args):

print( f'Hello {name}' )

show('amir', 'ali')

اگه کد بالا رو اجرا کنید دیگه به شما ارور نمیده چون args اومده و پارامترهای اضافی رو داخل خودش نگهداشته. این مقادیر اضافی را به صورت تاپل در خودش ذخیره میکند

حالا اگه من به فانکشن show به صورت key-value مقدار بدم چه اتفاقی میوفته؟

show('amir', 'ali', age=20)

کد بالا رو اگه اجرا کنید باز هم به شما ارور میده. اینجا دیگه args نمیتونه برای شما کاری انجام بده.

برای گرفتن مقدارهای اضافی که به شکل key-value میان میتونید از \*\*kwargs استفاده کنید:

def show(name, \*args, \*\*kwargs):

print( f'Hello {name}' )

show('amir', 'ali', age=20)

بازم اگه کد بالا رو اجرا کنید دیگه ارور نخواهید داشت چون kwargs اونها رو داخل خودش ذخیره کرده.این مقادیر اضافی را به صورت دیکشنری در خودش ذخیره میکند

(( توجه شود که حتماً لازم نیست که از اسم های args و kwargs استفاده شود فقط \* و \*\* قبل از اسم‌ها استفاده شود ))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

تفاوت بین arguman و parameter :  
این دو کلمه معمولاً به جای هم استفاده می‌شوند اما یک تفاوت کوچکی باهم دارن . اون مقداری که ارسال میکنیم به یک function به آن arguman گفته می‌شود اما اون مقداری که از function میگیرید parameter گفته می‌شود

نکته ی بسیار مهم : هر چیزی که در پایتون وجود دارد یک کلاس است قسمت مهم این نکته این است که هر چیزی که در پایتون وجود دارد رفتار یکسانی دارند   
منظور از objects همان کلاس میباشد

هنگامیکه functionی (تابع) ساخته می‌شود میتوان با آن همانند object رفتار کرد یعنی میتوان آن را به یک متغییر داد اما باید دقت داشت که پرانتز ها را نگذاریم   
(( پرانتز ها به این معنی است که فانکشن مورد نظر را صدا بزن و آن را اجرا کن ))  
 نکته ی بعدی این است که object اون function با متغییری که به آن دادیم کاملاً متفاوت است یعنی اگر مثلاً show حذف شود اما x بازهم کار میکند اما اگر show صدا زده شود error خواهد داد

به اون functionهایی که پارامتر ورودیشان یک function دیگر است *higher-order گفته می‌شود   
خوده پایتون یک فانکشن heigher-order به نام map دارد*

*توجه شود که در فانکشن های تودرتو لازم است که در آخر فانکشن داخلی* return *شود   
نکته مهمی که درباره فانکشن های تودرتو وجود دارد این است که نمیتوان به طور مستقیم به فانکشن داخلی دسترسی داشت   
به فانکشن های داخلی inner و به فانکشن های خارجی outer گفته می‌شود*

*فانکشن های فرزند میتوانند به پارامترهای ورودی پدر دسترسی داشته باشند  
به این فانکشن های فرزند که به این صورت به پارامترهای پدر دسترسی دارند lexical closure گفته می‌شود*

*با استفاده از متد \_\_call\_\_ میتوانیم objectهایمان را callable کنیم*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*برای ساخت decoratorها باید از فانکشن های تودرتو استفاده کرد   
هنگامیکه بالای یک فانکشن یک decorator قرار میدهیم به طور خودکار function را به عنوان آرگومان به function مربوط به decorator ارسال میکند*

*نکته بسیار مهمی که باید به آن توجه کرد این است که decoratorها از پایین به بالا اجرا می‌شوند*

*اگر functionی که بر روی آن decorator قرار گرفته باشد خود پارامتر ورودی داشته باشد این پارامتر ورودی در فانکشن های inner فرستاده می‌شود (( خود فانکشن به فانکشن outer فرستاده می‌شود ))  
در‌واقع در مثال t همان text است*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*دیزاین پترن decorator به ما این امکان رو میده که بتونیم به کلاس هایی که قبلا ساختیم رفتارهایی رو اضافه کنیم بدون اینکه نیاز باشه اون کلاس رو تغییر بدیم*

========================================================================