برنامه نویسی شیئ گرا

(Object Oriented Programming-OOP)

به قسمت چهارم آموزش پایتون از پایه خوش آمدید.در این قسمت من فرض میکنم که مطالب قبلی مربوط به متغییرها ، انواع داده ای و توابع را فرا گرفته اید.اگر چنین نیست ، سه قسمت قبلی همین سری از آموزش ها را مطالعه کنید.

امروز میخواهیم به کاوش در مورد مفهوم برنامه نویسی شیئ گرا بپردازیم.

OOP راهی قدرتمند برای مدیریت کدهایتان است و شما در صورت رسیدن به درک عمیقی از مفاهیم آن قادر خواهید بود کدنویسی (برنامه نویسی) خود را بهبود بخشید.

برنامه نویسی شیئ گرا چیست ؟

پایتون از ابتدا به صورت یک زبان برنامه نویسی شیئ گرا طراحی شد.اما در حقیقت " شیئ گرایی " به چه معناست؟ تعابیر گوناگونی برای این عبارت وجود دارد و شما میتوانید برای چندین ساعت در مورد آن به صحبت بنشینید و سعی کنید آن را در قالب عبارات پیچیده و با بیان تفاوت ها و ریزه کاری ها در پیاده سازی شرح دهید.هرچند که من در اینجا سعی میکنم یک مرور سریع را روی این مفاهیم برای شما داشته باشم.

به طور کلی ، شیئ گرایی یک مفهوم در دنیای برنامه نویسی است که بر اساس آن ، اشیائی که ما در حال کار بر روی آنها هستیم نسبت به منطق مورد نیاز برای دستکاری این اشیاء از اهمیت بیشتری برخوردار می باشند.

به طور سنتی یک برنامه به صورت یک دستورالعمل و یا مجموعه ای از دستورالعمل ها مشاهده می شود که می توان با دنبال کردن این دستورالعمل ها در یک ترتیب مشخص مراحل تکمیل و اجرای یک " وظیفه " را از ابتدا تا انتها پیمود.

این رویکرد امروزه نیز میتواند درست باشد و حتی در بسیاری از برنامه های ساده نیازی حیاتی به شمار می آید.این رویکرد با

نام "برنامه نویسی رویه ای" شناخته میشود.

به عبارتی دیگر ، هنگامی که برنامه بزرگتر و پیچیده تر میشود ، منطق لازم برای رویه ای نوشتن آن بسیار درهم تنیده و غیرقابل درک میشود.گاهی رویکرد برنامه نویسی شیئ گرا میتواند در این چنین مواردی به کمک شما بیاید.
در رویکرد شیئ گرا به جای اینکه از اشیاء تنها به عنوان ظرف هایی برای اطلاعات و داده ها و همچنین قسمتی از دستورالعمل های رویه ای استفاده شود ، آنها را در کانون فرایند برنامه نویسی قرار میدهیم.

در قدم اول ما به تعریف شیئی که میخواهیم روی آن کار کنیم میپردازیم و نیز چگونگی ارتباط آن با سایر اجزا را مشخص میکنیم.سپس از طریق منطق به آن جسم و جان میبخشیم ، تا برنامه به شکل درستی اجرا شود.

وقتی که من در مورد اشیاء صحبت میکنم به معنی این است که باید بتوانم در مورد تمام اشیاء قابل شمارش این صحبت ها را بسط دهم.یک شیئ میتواند به یک "شخص" (که به وسیله ی ویژگی هایی نظیر نام ، سن ، آدرس و غیره مشخص میشود) و یا یک "شرکت" (که به وسیله ویژگی هایش ، نظیر تعداد کارمندان و چیزهای مشابه آن تعریف میشود) اشاره داشته باشد. حتی میتواند به مفاهیمی بسیار انتزاعی مانند یک دکمه در واسط کاربری یک رایانه اشاره کند.

در اینجا من قصد ندارم که تمامی مفاهیم بحث شده را به طور مفصل پوشش دهم چون در این صورت باید کل شب را برای آن وقت بگذارم.اما امیدوارم که شما بتوانید در پایان این آموزش ها درک عمیقی از مفاهیم ساده ی مورد نیاز برای شروع برنامه نویسی شیئ گرا در پایتون پیدا کنید. جنبه ی مثبت این قضیه این است که مفاهیم فوق در بسیاری از محیط های برنامه نویسی نسبتا مشابه هستند.بر این اساس مهاجرت از یک زبان به زبان دیگر راحت خواهد بود.

آغاز به کار :

همانطور که در قسمت قبل اشاره کردم ، اولین کاری که در برنامه نویسی شیئ گرا باید انجام دهید ، تعریف اشیائی است که میخواهیم با آنها کار کنیم.اولین قدمی که ما برای انجام این کار برمیداریم ، تعریف کردن ویژگی های شیئ مربوطه در

قالب یک "کلاس" است.شما یک کلاس را همانند یک الگو یا قالب در نظر بگیرید.یعنی یک راهنما و نقشه ی راه برای شیئی که قرار است از روی آن ساخته شود. هر شیئ متعلق به یک کلاس است و ویژگی های آن را به ارث میبرد.اما چند شیئ ساخته شده از روی یک کلاس هر یک برای خود رفتار مستقلی خواهند داشت.

به عنوان یک مثال ساده ، شما ممکن است کلاسی به اسم person و با ویژگی هایی مانند سن و نام داشته باشید. همچنین ممکن است نمونه ای از این کلاس (یک شیئ از کلاس) ساخته باشید که در این صورت مبین یک شخص یکتا در جامعه خواهد بود.نام این شخص میتواند Andy و سن او ۲۳ سال باشد.اما شما میتوانید شخص دیگری نیز از همین کلاس person شبیه سازی کنید که نام او مثلا Lucy باشد و ۱۸ سال سن داشته باشد.

حقیقتا فهم این مسائل بدون دیدن چند تمرین در این زمینه سخت خواهد بود. پس اجازه دهید به صورت جدی شروع به نوشتن چند خط کد کنیم.

تعریف کلاس :

ما برای تعریف یک کلاس بر اساس ساده ترین مدل برنامه نویسی پایتون از لغت class استفاده میکنیم که به دنبال آن نام کلاس جدیدمان را می آوریم.

من در اینجا میخواهم یک کلاس جدید به نام pet بسازم.

ما بلافاصله یک علامت کالن (:) را در انتهای نام کلاس قرار میدهیم و پس از آن هر چیز که درون محدوده ی تعریف کلاس قرار گیرد باید با یک جای خالی فاصله بیاید(به صورت دندانه دار).توجه کنید که در هر صورت به دنبال کلمه ی class هیچ پرانتزی نوشته نمیشود :

```
class pet:
```

خب ما الان یک کلاس داریم اما تا زمانی که چیزی درون آن قرار ندهیم موجود بی استفاده ای به شمار میرود.برای شروع بیایید یک جفت ویژگی به آن نسبت دهیم.برای این کار میتوانید تعدادی متغییر را به آسانی درون کلاس تعریف کنید.(من برای این کار یک متغییر عددی را به عنوان شمارشگر تعداد پاهای حیوان خانگی در نظر میگیرم). مانند همیشه اسم متغییر

به گونه ای باید انتخاب شود که به راحتی بتوانید اطلاعاتی در مورد متغییر به دست آورید. اجازه دهید اسم آن را number_Of_edge بگذاریم. الان باید یک مقدار به آن اختصاص دهیم وگرنه یک خطا دریافت میکنیم.

من اینجا از مقدار ۰ استفاده میکنم(در این مورد توجه به اینکه تمام حیوانات خانگی تعداد پاهای یکسانی ندارند زیاد اهمیت ندارد — مثلا تعداد پاهای ماهی و سگ برابر نیستند — در هر صورت ما مجبوریم که برای هر نمونه از این کلاس مقدار متغییر را تغییر دهیم).

```
class pet:
      number_of_legs = 0
```

نمونه ها و عناصر متغییر :

معمولا نمیتوان با یک کلاس به صورت مستقیم کار کرد. یعنی در ابتدا مجبوریم یک نمونه از آن را ساخته و سپس به دستکاری ان بپردازیم.

می توانیم نمونه ی ساخته شده را درون یک متغییر قرار دهیم.

بیایید در خارج کلاس (یعنی بیرون از ساختار دندانه دار کلاس) نمونه را درون متغییری به نام doug قرار دهیم. به راحتی برای ساخت یک نمونه ی جدید از کلاس ، نام آن را تایپ کنید و به دنبال آن یک جفت پرانتز قرار دهید.

البته در اینجا لازم نیست نگران استفاده از پرانتزها باشید چون بعدا خواهید دید که همانند توابع ، این راهی ست که در لحظه ی ساخت نمونه از کلاس میتوان مقادیری را به متغییرهای آن فرستاد.

```
class pet:
      number_of_legs = 0

doug = pet()
```

ما الان یک نمونه از کلاس pet را داریم.آیا الان توانایی دستکاری در ویژگی های آن را پیدا کرده ایم؟

برای اشاره به خصوصیات یک شیئ در ابتدا باید به پایتون بگوییم منظورمان کدام شیئ است(یا کدام نمونه از کلاس).بنابراین ما در این مثال ، اکنون خود را با تایپ کردن doug که نام نمونه است شروع میکنیم.بعد از آن یک نقطه قرارمیدهیم تا مشخص کنیم قصد دسترسی به یکی از محتویات نمونه ی doug را داریم.پس از نقطه ، نام خصوصیت موردنظر

خود را می آوریم.اگر تمام مراحل فوق را درست انجام داده باشید باید الان همچین چیزی جلویتان باشد :

```
doug.number_of_legs
```

ما میتوانیم این کار را به طور مشابه برای سایر متغییرهایمان نیز تکرار کنیم.من الان میخواهم doug را به عنوان یک سگ مقدار دهی کنم و به متغییر آن مقدار ۴ بدهم(چون سگ دارای ۴ پا می باشد)

اگر موافق باشید در پایان کار هم یک خط کد برای چاپ تعداد پاهای حیوان خانگی مان بنویسیم تا از صحت کارمان مطمئن شویم :

```
class pet:
      number_of_legs = 0

doug = pet()
doug.number_of_legs = 4
print "Doug has %s legs." % doug.number_of_legs
```

اگر کد فوق را اجرا کنید میبینید که خروجی مورد نظر برای شما نمایش داده خواهد شد.این کد کلاس pet را تعریف میکند ، یک نمونه از آن میسازد و آن را درون متغییر doug ذخیره میکند و سپس به درون نمونه ی ایجاد شده میرود و خاصیت به ارث برده ی آن را با ۴ مقداردهی میکند.(خاصیت number_of_legs از کلاس اصلی به ارث برده شده است)

شما از طریق این مثال ساده ی کاربردی توانستید یک ساختمان داده ی پیمانه ای بسازید که استفاده از آن نیز بسیار ساده است.

معرفی منطق :

بسیار خوب !!! تاکنون شما با اساسی ترین مفاهیم کلاس ها و اشیاء آشنا شده اید اما در اینجا میخواهیم به صورت جدی تر از کلاس ها به عنوان ساختمان داده ها و یا ظرف هایی برای متغییرها استفاده کنیم. اینها همگی خوب و عالی هستند اما اگر بخواهیم به پیاده سازی و اجرای وظایف پیچیده تری بپردازیم ، به راهی احتیاج داریم که بتوانیم منطقی کلی را بر روی اشیاء و کلاس ها حاکم کنیم.ما برای این کار از متدها استفاده میکنیم.

اساسا متدها همان توابع هستند با این تفاوت که درون کلاس ها قرار میگیرند.شیوه ی تعریف این متدها دقیقا مشابه تعریف یک تابع است که البته فرق آن این جاست که باید درون یک کلاس تعریف شوند و در واقع متعلق به یک کلاس باشند. هرگاه شما بخواهید متدی را فراخوانی کنید مجبور هستید که ابتدا به یک شیئ از کلاس موردنظر رجوع کنید. دقیقا مشابه حالتی که قبلا برای دسترسی به متغییرها بیان کردیم.

من برای شرح این قضیه یک مثال کوتاه در مورد همان کلاس pet میزنم.بیایید یک کلاس به نام sleep بسازیم که در اولین فراخوانی اش یک پیغام را در خروجی چاپ میکند. همانند شیوه ی تعریف توابع ، لغت def را در ابتدا مینویسم. سپس به دنبال آن نام متدی را می آوریم که قصد ساخت آن را داریم.در ادامه یک جفت پرانتز باز و بسته قرار میدهیم و در آخر هم علامت کالن را می نویسیم و با رفتن به سطر بعدی ، خط جدیدی را شروع میکنیم.

مطابق معمول هر چیزی که قرار است درون متد قرار گیرد حتما باید به صورت دندانه دار نسبت به سطح بالایی بیاید.

در اینجا تفاوت دیگر متد و تابع آشکار می شود : متد همیشه ی ، همیشه ی ، همیشه باید یک آرگومان به نام self مابین دو پرانتز داشته باشد.هنگامی که پایتون متد را فراخوانی میکند ، اتفاقی که می افتد این است که شیئ فعلی را به عنوان اولین آرگومان به متد sleep ارسال میکند.(پاس میدهد)

دلیل این مطلب را بعدا خواهید فهمید اما آنچه که لازم است الان بدانید توجه به این نکته است که به یک متد همیشه آرگومانی با نام self به عنوان اولین آرگومان الصاق می شود.(اگر بخواهید که آرگومان های بیشتری اضافه کنید میتوانید آنها را یکی پس از دیگری بیاورید و این کار دقیقا مشابه توابع دارای آرگومان های چندگانه است). اگر شما این آرگومان را الصاق نکنید ، در زمان اجرای برنامه یک خطا دریافت خواهید کرد. دلیل این است که پایتون شیئ کنونی را به متد پاس میدهد(آرگومان self) و متد پاسخ میدهد: " *آهای آقا!!! من قرار نیست هیچ آرگومانی را از شما تحویل بگیرم.اصلا میدونی که داری چیکار میکنی؟!* ". این هم دقیقا مشابه حالتی است که بخواهید آرگومانی را به تابعی پاس دهید که هیچ آرگومانی نمی پذیرد.

در اینجا هر آنچه را که تاکنون انجام داده ایم میتوانید ببنید:

```
class pet:
```

```
	number_of_legs = 0

	def sleep(self):

doug = pet()
```

عبارت print را اینگونه داخل بدنه ی متد مینویسیم :

```
class pet:
	number_of_legs = 0

	def sleep(self):
		print "zzz"

doug = pet()
```

حالا اگر بخواهیم از این متد استفاده کنیم ، می توانیم به آسانی از طریق یک نمونه ی کلاس pet به آن دسترسی داشته باشیم. مشابه کاری که با متغییر number_of_edge انجام دادیم ، ابتدا نام نمونه را می نویسیم (در اینجا doug) ، سپس نقطه (.) و در آخر نام متد به همراه پرانتزهای آن.توجه داشته باشید که درست است ما متد sleep را بدون هیچ آرگومانی فراخوانی میکنیم اما پایتون به صورت خودکار داخل پرانتز را با آرگومان self پر میکند.

```
class pet:
	number_of_legs = 0

	def sleep(self):
		print "zzz"


doug = pet()
doug.sleep()
```

اگر کد را اجرا کنید میتوانید پیغام zzz را در صفحه ی خروجی مشاهده کنید.

داده ها :

نظرتان چیست متد جدیدی را بنویسیم که تعداد پاهای حیوان خانگی را نشان دهد؟این متد به ما می آموزد که چطور میتوانیم از طریق متدها به دستکاری داده های درون class بپردازیم.همچنین نشان میدهد که چرا ما ملزم به استفاده از آن آرگومان self کذایی هستیم.بیایید متد جدیدمان را با نام count_legs بسازیم.

اینجاست که آرگومان self وارد عمل می شود. به یاد دارید وقتی که میخواستیم از متغییر number_of_legs در خارج از

کلاس استفاده کنیم ، مجبور بودیم به جای نوشتن number_of_lengs ، بنویسیم doug.number_of_legs ؟جواب کاربردی این است که ما مجبوریم برای استفاده از محتویات یک متغیر از کلاس ، از طریق نمونه ی ساخته شده ی آن کلاس اقدام کنیم.

بهرحال ما در هنگام نوشتن کلاس نمیدانیم که این اجزا توسط کدام یک از نمونه ها استفاده می شود (چون ممکن است نمونه های زیادی از یک کلاس در برنامه موجود باشد). بنابراین برای حل این مشکل از متغییر self استفاده میکنیم. Self تنها یک ارجاع به شیئی ست که در حال حاضر مشغول کار بر روی آن هستیم.پس برای دسترسی به متغییر کلاس کنونی شما میتوانید به آسانی از لغت self و یک نقطه بعد از آن استفاده کنید.

مانند مثال زیر :

```
class pet:
       number_of_legs = 0

       def sleep(self):
              print "zzz"

       def count_legs(self):
              print "I have %s legs" % self.number_of_legs

doug = pet()
doug.number_of_legs = 4
doug.count_legs()
```

در این مثال کاربرد self داخل متد این است که وقتی شما قدام به اجرای متد میکنید ، به صورت خودکار نام نمونه با self جایگزین می شود. بنابراین هنگامی که ()doug.count_legs را فراخوانی میکنیم ، self با doug جایگزین میشود.برای شرح اینکه این فرایند در حالت چند نمونه ای چگونه کار میکند ، نمونه ی دیگری از کلاس pet به نام nemo را به برنامه اضافه میکنیم.

```
class pet:
       number_of_legs = 0

       def sleep(self):
              print "zzz"

       def count_legs(self):
              print "I have %s legs" % self.number_of_legs


doug = pet()
doug.number_of_legs = 4
```

```
doug.count_legs()

nemo = pet()
nemo.number_of_legs = 0
nemo.count_legs()
```

این برنامه ابتدا پیغام خروجی ۴ و سپس ۰ را چاپ میکند.

در این روش ، متد کاملا به صورت دندانه دار نوشته شده است چون مرجع self به صورت داینامیکی تغییر میکند و کاملا به متن نوشته شده وابسته است و تنها به ما اجازه می دهد که متغییرهای درون شیئ فعلی را دستکاری کنیم.

فقط یک نکته : شما میتوانید برای متدهایتان هر اسمی بجز self انتخاب کنید.

استفاده از کلمه ی self یک توافق ساده بین برنامه نویسان پایتون است که باعث میشود کدهای نوشته شده توسط افراد مختلف ، هرچه بیشتر استاندارد و قابل فهم شود.نصیحت من این است که به توافق ها پایبند بمانید.

چند ویژگی پیشرفته :

ما اکنون مقدمات را پشت سر گذاشته ایم و آماده ایم که نگاهی به ویژگی های پیشرفته ی کلاس ها بیاندازیم و ببینیم که آنها چگونه می توانند در ساده سازی طراحی برنامه های ساخت یافته به ما کمک کنند.

مطلب بعدی که میخوام در مورد آن صحبت کنیم " ارث بری " است.ارث بری به فرایند ساخت یک کلاس بر پایه ی یک کلاس پایه گفته می شود که به کلاس جدید اجازه ی به ارث بردن ویژگی ها و مشخصات والد را می دهد.

کلاس جدید میتواند تمامی متدها و متغییرهای والد را داشته باشد.(گاهی به کلاس والد –پایه–هم گفته می شود).

بیایید مثال حیوانات خانگی خود را گسترش دهیم تا ببینیم که این تکنیک تا چه حد می تواند مفید باشد.اگر کلاس pet را به عنوان والد در نظر بگیریم ، میتوانیم کلاس فرزندی را بر پایه ی آن بسازیم که بتواند از pet ارث بری کند.

کلاس فرزند میتواند هرچیزی مثل سگ و یا ماهی باشد(گاهی کلاس فرزند می تواند همان pet باقی بماند اما دارای ویژگی های مخصوص خود باشد).

یک سگ حیوان خانگی است و میتواند تمام آنچه را که یک حیوان خانگی انجام میدهد را انجام دهد.(به فرض مثال سگ

میخورد ، می خوابد و تعدادی دست و پا دارد) اما ویژگی های منحصر به فردی نیز دارد که او را یک سگ کرده است.
سگ مو دارد ؛ در حالی که همه ی حیوانات خانگی این ویژگی را ندارند. سگ ممکن است تکه چوبی را بردارد اما همه ی حیوانات خانگی قادر به این کار نیستند.

خب به اصل مطلب برگردیم.گفتیم که میخواهیم کلاسی درون برنامه مان ایجاد کنیم که رفتار یک سگ را به نمایش بگذارد.
میتوان با استفاده از ارث بری متدها و متغییرهای pet را به کلاس جدید ارث داد.بنابراین کلاس doug می تواند متغییر number_of_legs و متد sleep را داشته باشد.

شاید شما تعجب کنید که چگونه ما بدون نوشتن این متدها و متغییرها در کلاس dog آنها را به صورت کامل از pet میگیریم؟

ارث بری ۲ ویژگی منحصربفرد را در اختیار ما قرار میدهد:
یک : اگر بخواهیم شیئی را از نوع pet بسازیم به صورتی که سگ نباشد ، میتوان این کار را انجام داد.
دو : شاید بخواهیم نوع دیگری از حیوانات خانگی را اضافه کنیم(مثلا یک ماهی)ما میتوانیم این کلاس دوم را نیز با ارث بری از pet ایجاد کنیم.هر دوی این کلاس های جدید می توانند متدها و متغییرهای pet را به اشتراگ بگذارند و همچنین می توانند در همان زمان ویژگی منحصر بفرد و اضافی خود را نیز داشته باشند.یعنی متدها و متغییرهایی که فقط برای اشیایی که از خودشان ساخته می شوند قابل دستیابی اند.

بیایید کمی عمیق تر وارد قضیه شویم.پس اجازه دهید چیزهایی بنویسیم که ماجرا روشن تر شود.در ابتدا کلاس جدیدی به اسم dog می نویسیم .با این تفاوت که این بار بین نام کلاس و علامت کالن یک جفت پرانتز باز و بسته قرار می دهیم و نام کلاسی که باید از آن ارث بری کنیم را بین آن دو می نویسیم.یعنی همانند توابع ، نام کلاس والد را به صورت یک آرگومان به آن پاس میدهیم.

در ادامه متد ساده ای را به این کلاس اضافه می کنیم تا طرز کار این فرایند را برایمان شرح دهد.من متد bark را می نویسم

تا عبارت woooof را چاپ کند.

```
class pet:
	number_of_legs = 0

	def sleep(self):
		print "zzz"

	def count_legs(self):
		print "I have %s legs" % self.number_of_legs

class dog(pet):
	def bark(self):
		print "Wooooof"
```

بیایید ببینیم الان با ساخت یک نمونه ی جدید از این کلاس چه اتفاقی می افتد.! دوباره با استفاده از نمونه ی doug که به کلاس dog اشاره میکند، متد جدید را اینگونه فراخوانی میکنیم:doug.bark()

```
class pet:
	number_of_legs = 0

	def sleep(self):
		print "zzz"

	def count_legs(self):
		print "I have %s legs" % self.number_of_legs

class dog(pet):
	def bark(self):
		print "Woof"

doug = dog()
doug.bark()
```

تا اینجای کار بسیار عالی پیش رفته ایم اما هنوز کار جدیدی انجام نداده ایم بجز اینکه یک کلاس به همراه یک متد درون آن ساخته ایم.

کاری که ارث بری برای ما انجام میدهد این است که تمام متدها و متغییرهای در دسترس کلاس pet را وارد شیئ dog میکند.پس بنابر این گفته ها می توانم اینگونه بنویسم :

```
class pet:
	number_of_legs = 0

	def sleep(self):
		print "zzz"

	def count_legs(self):
		print "I have %s legs" % self.number_of_legs
```

```
class dog(pet):
       def bark(self):
              print "Woof"

doug = dog()
doug.sleep()
```

پس از این مراحل متد sleep به درستی اجرا می شود . در حقیقت شیئ doug به هردو کلاس pet و dog تعلق دارد.برای اطمینان از اینکه متغییرها هم مانند متدها به ارث رسیده اند این کدها را امتحان میکنیم:

```
class pet:
       number_of_legs = 0

       def sleep(self):
              print "zzz"

        def count_legs(self):
              print "I have %s legs" % self.number_of_legs

class dog(pet):
       def bark(self):
              print "Woof"

doug = dog()
doug.number_of_legs = 4
doug.count_legs()
```

می توانید ببینید که doug مانند سابق به درستی کار میکند و این بدان معنی است که متغییرها به درستی به ارث برده شده اند.کلاس فرزند جدیدی که ساختیم یک نسخه ی ویژه ، منحصربفرد و تغییر یافته از والد خود است که چند عملکرد مخصوص به خود را به نمایش میگذارد ، در حالی که تمامی حالت ها و عملکردهای والد را نیز با خود دارد.

خسته نباشید!!!شما تا اینجا یک مرور سریع روی برنامه نویسی شیئ گرا انجام دادید.با ما در قسمت بعدی این سری از آموزش ها همراه باشید.جایی که قرار است به سراغ استفاده از پایتون در وب برویم!

مترجم : بهنام محمد کریمی

ایمیل : besa.1371@gmail.com