# Standards of Programing

## easy way

ویرایش 1

نویسنده : احمد عبدالله زاده

# Table of Contents

# معرفی

با سلام در این کتاب قصد داریم استاندارد های مختلف برنامه نویسی را توضیح دهیم. انواع نامگذاری, کامنت گذاری و تمام اصولی که برای نوشتن یک کد ساده و قابل توسعه لازم است را با مثال و به زبانی ساده و قابل فهم برای تمام برنامه نویسان تازه کار یا با تجربه توضیح داده خواهد شد.

---

این کتاب وابسته به زبان برنامه نویسی خاصی نیست و تلاش شده در هر قسمت از چندین زبان برنامه نویسی رایج مثالهایی آورده بشه تا بر فهم بیشتر موارد گفته شده بیفزاییم.

---

کتاب را میتوانید در gitbook از اینجا مشاهده کنید.

برای دانلود نسخه pdf اینجا کلیک کنید.



این کتاب رو احمد عبدالله زاده شروع کرده ولی امیدوارم در ادامه راه دوستان دیگری هم به پروژه اضافه بشن.

## آدرس پروژه در گیت هاب

پروژه در گیت هاب

### دسترسی به من

من در گیت هاب
من در لینکداین

# مقدمه: دلیل اهمیت یادگیری برنامه نویسی و استانداردها

## از دلایل اهمیت برنامه نویس:

**زبان برنامه نویسی به زبان مشترک آینده تبدیل خواهد شد.**

پیشرفت هوش مصنوعی، اینترنت اشیا و بسیاری دیگراز تکنولوژی هایی که با استفاده از برنامه نویسی بوجود آمده و پیشرفت کرده اند باعث اتوماتیک شدن کارها و بیکارشدن میلیونها نفر شده اند، اما موقعیت های شغلی بسیاری را نیز بوجود آورده اند.

در دنیای امروزه که رایانه به عضوی جدایی ناپذیر از زندگی روزمره ما تبدیل شده است دانستن برنامه نویسی بسیار با اهمیت است.

فرقی نمیکند در چه رشته ی دانشگاهی تحیصل میکنید یا مشغول کار در چه زمینه ای هستید, دانستن برنامه نویسی میتواند شمارا در خلق ایده های جدید برای پیشرفت در کار یا آماده کردن و تحلیل مقالات علمی کمک کند و یا باعث افزایش اشتغال زایی شود.

یادگیری برنامه نویسی فقط برای سازندگان نرم افزار ها نیست.

## همانطور که استیو جابز می‌گوید :

> تمام مردم باید برنامه‌نویسی یاد بگیرند، چون برنامه‌نویسی به شما یاد می‌دهد که چگونه فکر کنید.

---

فرقی نمی کند در چه زمینه ای مشغول به کار هستید دانستن برنامه نویسی باعث برتری شما نیست به دیگران میشود.

**این گفته ی لیلا سِکا** مدیر ارشد در *Salesforce* (شرکت سان‌فرانسیسکویی که در زمینه‌ی پرداش ابری فعالیت می‌کند) می باشد، او همچنین در مصاحبه‌ای با بیزینس انسایدرز اذعان کرد که ندانستن برنامه‌نویسی از بزرگ‌ترین اشتباهات زندگی اوست.

## لیلا سکا همچنین می‌گوید:

> کد و برنامه‌نویسی، زبان کامپیوتر است؛ من می‌خواهم با کامپیوتر صحبت کنم و زبانش را یاد بگیرم. یادگیری برنامه‌نویسی باعث موفقیت بیشتر در کار من خواهد شد.

امروزه پیشرفت تکنولوژی و نفوذ آن در دیگر زمینه های علمی شما را ملزم میکند که حتی اگر توسعه دهنه ی نرم افزار نیستید شروع به یادگیری برنامه نویسی و مفاهیم ابتدایی تکنولوژی هایی مانند **api**, **web**, **coud** و ... کنید.

## اما این کار چه سودی میتواند داشته باشد؟

**افزایش خلاقیت**
**یادگیری تکنیک حل مشکل**
**درک بهتری از تکنولوژی**
**توانایی در گفتار تخصصی**

دانستن برنامه نویسی میتواند به شما کمک کند که راهکاری برای ساده کردن بخشی از کاری که روزانه انجام میدهید طراحی کنید یا ایده ای را که مدتها در ذهن دارید را پیاده سازی کنید و یا تحلیل کنید که چقدریک ایده قابل پیاده سازی است و همچنین چه هزینه ای برای اجرا نیاز دارد.

---

## چرا کد نویسی استاندارد توصیه میشود؟

کدنویسی بصورت استاندارد فقط توصیه نمیشود بلکه ضروری و لازم است.

نرم افزار های بزرگ و پرکاربرد توسط یک شخص و در زمان اندکی ساخته نشده اند, یک نرم افزار ازاولین روزی که شروع به رشد میکند تا وقتی که به بلوغ میرسد و مورد استفاده همگان قرار میگیرد هر روزبزرگ و بزرگتر میشود , بارها تغییر میکند, برنامه نویسان مختلفی روی آن کار میکنند و حتی ممکن است بارها بازنویسی شود.

اگر کدها درهم باشند, نام متغیرها, توابع و دیگر اجزای کد بی معنی باشد هرگز یک نرم افزار رشد نمیکند زیرا درک و فهم کد دیگران سخت و وقت گیر میشود و در این شرایط حتی فهم وتغییر کدی که مدتی قبل خودمان نوشته ایم نیز بسیار دشوار میشود.

اگر **لینوس توروالدز** در هنگام نوشتن کرنل لینوکس کدی نامرتب و غیر استاندارد مینوشت هرگز کرنل لینوکس به نقطه ای که در حال حاضر در آن قرار دارد میرسید؟ **هرگز**

## در این شرایط استفاده از یک استاندارد جامع واجب است.

> رعایت استانداردها باعث ایجاد کدی مرتب, قابل فهم و حرفه ای میشود.

# فصل اول: خوانا بنویسیم

## چگونه بدون دانستن کامل یک استاندارد برنامه نویسی خواناتر کد بزنیم؟

این فصل با توجه به اشکالات رایج برنامه نویسان تازه کار نوشته شده و با توجه به استاندارد یا کتاب خاصی نمیباشد.

هدف این فصل توجه به استاندارد خاصی نیست و به سرعت میتوانید با رعایت نکات توضیح داده شده کدی تمیز و مرتب بنویسید.

---

فصل اول شامل قوانین کلی بدون توجه به یک استاندارد خاص می باشد, اصولی که برای نوشتن یک کد **تمیز** لازم است .

---

# قواعد:

## ۱. استفاده از tab:

به منظور متمایز کردن کدهای درون **بلاک** از دیگر قسمتهای کد و خوانایی بیشتر برنامه استفاده میشود, هر قسمت با یه tab درون بلاک خودش قرار میگیره.

> **بلاک**:
>
> بلاک به قسمتهایی از کد کد گفته میشه که خود آن دارای بدنه ای از کد میشود مانند حلقه ها, شرطها , توابع, کلاس ها و ...

*C*:

```
if(condition){
    return true;
}
else{
    return false;
}
```

**Block in Block:**

```
if(condition1){
    if(condition2){
        return true;
    }
    else{
        return false;
    }
}
```

در بعضی از زبان ها استفاده از tab جزو قواعد اصلی زبان است و استفاده نکردن از آن موجب بروز error میشود مثل **پایتون**:

*python*:

```
if condition:
    return True
else:
    return False
```

## ۲. استفاده از comment:

کد خوب کدیه که انقدر تمیز و مرتب نوشته شده باشه که نیازی به کامنت گذاری نداشته باشه ولی با این حال بهتره توی برنامه‌مون از کامنت استفاده کنیم تا هم کار رو برای برنامه نویس های احتمالی که ممکنه در آینده بخوان روی کد ما کار کنن آسون کنیم و هم اگه در آینده خودمون خواستیم روی کد کار کنیم خیلی کارمون سریع تر باشه.

نوشتن توضیحاتی در مورد هدف هر فایل یا کتابخانه, نوع استفاده و هدف ایجاد هر متغییر, تابع و دیگر اجزای کد, نوشتن نام و توضیحاتی در مورد نویسنده کد از جمله کاربرد های کامنت میباشد.

> **کامنت**:
>
> کامنت توضیحاتی هستند که در برنامه قرار میگیرند و در مورد خود برنامه یا یک قسمت خاص از برنامه توضیحاتی را می دهند.
>
> خطوطی که کامنت میشوند توسط سیستم اجرا نشده و تاثیری در خروجی برنامه و یا زمان اجرای برنامه ندارند.
>
> برای مطالعه بیشتر روی کامنت ها به این لینک مراجعه کنید.

**در زبان های برنامه نویسی از علایم مختلفی برای کامنت گذاری استفاده میشه:**

در python از #
از // برای کامنت کردن یک خط و از /**/ برای کامنت کردن بیش از یک خط در زبانهای java, c#, c++, c, css, ...
استفاده میشود
در متلب از %
...

*python:*

```python
ix = 0        # index to scan array
count = 1    # counter of positive values
```

*C:*

```c
int ix;        // index to scan array
int count;    // counter of positive values
int positiveCounter(){
    // it counts all positive numbers

}
```

## ۳. استفاده از ثوابت:

متغیر های را که در کد مقدار ثابت و تاثیر گذاری دارند را بصورت **ثابت** در ابتدای کد تعریف کنید تا در صورتی که نیاز باشد مقدار آن متغیر را تغییر دهید فقط یک بار این کار را انجام دهید و لازم نباشد در طول کد بارها این کار را انجام دهید.

ثوابت را معمولا با حروف بزرگ تعریف میکنند تا برنامه نویسان در هر کجای کد که با این متغییر ها برخورد میکنند بدانند که با یک متغییر از نوع ثابت برخورد کرده اند که دارای دسترسی **فقط خواندنی** (read only) میباشد و نمیتوانند مقدار آنرا تغییر دهند.

> برای مطالعه بیشتر در مورد ثوابت به این لینک مراجعه کنید.

*C:*

```c
#define SPEED 9600
const SPEED = 9600;
```

*python:*

```python
SPEED = 9600

def setSpeed(sp):
    while sp < SPEED:
        ...
```

## ۴. فضای سفید:

رها کردن بعضی از خط ها باعث افزایش خوانایی کد میشود.

برای مثال میان تعریف دو بدنه تابع یا بین فراخوانی کتابخانه های مورد نیاز برنامه و شروع کدهای برنامه میتوان با رها کردن یک خط به خوانایی کد کمک کنیم.

*C*:

```
#include<stdio.h>

void main(){

    int start = 0;
    int end = 100;

    int i;

    for(i = start; i < end; i++){
        printf("standard code :)");
    }
}
```

*python*:

```
import request

def getLink(api):

    codes = requests.get(api)
    jsonResult = json.loads(codes.content)

    link = jsonResult["images"][0]["url"]

    except:

        print ("net is off")    # it runs if net is off .
    else:

        downLink = baseUrl + link   # make download link .
        print (downLink)

api = "https://www.bing.com/HPImageArchive.aspx?format=js&idx=0&n=1&mkt=en-US"
getLink(api)
```

## ۵. گذاشتن فاصله بعد از عملگرها:

استفاده از فاصله قبل و بعد از عملگرها (* + - = /).

In Correct:

```
sum=sum+i;
```

Correct:

```
sum = sum + i;
```

فقط بعد از **جداکنندها**(: , ;) از فاصله استفاده کنید.

In Correct:

```
l = [2,3 , 4]
```

:Correct

```
l = [2, 3, 4]
```

### ۶. قبل و بعد از پرانتز نیازی به استفاده از فاصله نیست.

*:python*

```
def testFunc():
    print("Hello, World!!!")
```

# فصل دوم: نامگذاری متغیرها

## روش های مختلفی برای نامگذاری یک متغیر وجود دارد که میتوانید بسته به نیاز یا سلیقه یکی از آنها را انتخاب کنید.

انتخاب نام مناسب برای متغیرها بسیار مهم است زیرا نام هر متغیر **توضیح** مختصری از دلیل تعریف آن متغیرمی باشد.

این استانداردها در تمامی زبانهای برنامه نویسی یکسانند.

---

### متغیر چیست؟

متغیر یک نام است که به اطلاعات ذخیره شده در یک قسمت از حافظه اشاره میکند.

متغیر ها شامل انواع مختلفی هستند که نوع داده ای اطلاعات ذخیره شده در حافظه را مشخص میکنند, برای مثال stringو integer و float و ...

> برای مطالعه بیشتر در مورد متغیرها به این لینک مراجعه کنید.

---

# انواع روش های نامگذاری متغیرها:

## ۱. نامگذاری متغیر های `private` و `protected` :

بهتر است قبل از نام متغیر هایی که بصورت `private` یا `protected` تعریف میشود از **underscore** استفاده کنیم تا در طول برنامه به برنامه نویس یادآوری شود که متغیر مورد نظر دارای دسترسی `private` یا `protected` میباشد.

> `private` و `protected` :
>
> در برنامه نویسی شی گرا میتوانیم برای هر method یا property دسترسی public, private, protected, ... تعریف کنیم.
>
> **برای یادگیری بیشتر برنامه نویسی شی گرا روی لینک کلیک کنید.**

*:#C*

```
class A{
    protected int _someVariable;
    private int _counter;

    public string name;
}
```

*:PHP*

```
private $_someVariable;

class MyClass {

   protected function __behindTheScenesMethod() {}
}
```

## ۲. نامگذاری ثوابت(constant):

متغیر های را که در کد مقدار ثابت و تاثیر گذاری دارند را بصورت **ثابت** در ابتدای کد تعریف کنید تا در صورتی که نیاز باشد مقدار آن متغیر را تغییر دهید فقط یک بار این کار را انجام دهید و لازم نباشد در طول کد بارها این کار را انجام دهید.

ثوابت را معمولا با حروف بزرگ تعریف میکنند تا برنامه نویسان در هر کجای کد که با این متغییر ها برخورد میکنند بدانند که با یک متغییر از نوع ثابت برخورد کرده اند که دارای دسترسی **فقط خواندنی** (read only) میباشد و نمیتوانند مقدار آنرا تغییر دهند.

> برای مطالعه بیشتر در مورد ثوابت به این لینک مراجعه کنید.

*:javaScript*

```
var TAXRATE = .0825;
```

*:PHP*

```
define('TAXRATE', .0825);
```

*:C*

```
const SPEED = 9600;
// or
#define SPEED 9600
```

## ۳. روش single letter Prefixies:

در این روش با استفاده از یک حرف در ابتدای نام متغیر نوع آن متغیر را مشخص میکنیم.

از `i` برای نوع integer و از `s` برای string , به همین ترتیب کاراکتر اول هر نوع را در ابتدای نام آن متغیر میگذاریم.

*:javaScript*

```
var sName = "Lieutenant Commander Geordi La Forge";
var iAge = 22
```

## ۴. روش camelCase:

این روش به این ترتیبه که اگر نام تابع ما از چند کلمه تشکیل شده باشد حرف نخست کلمه اول را با حرف کوچک و حرف نخست بقیه کلمات رو با حرف بزرگ مینویسیم.

*:javaScript*

```
var preacherOfSockChanging = 'Lieutenant Dan';
```

*:python*

```
 preacherOfSockChanging = 'Lieutenant Dan'
```

*:C*

```
int  preacherOfSockChanging = 'Lieutenant Dan';
```

## ۴. روش underScore:

در این روش اگر نام متغیر ما از چند کلمه تشکیل شده باشد بین هر کلمه یک **underscore** میگذاریم.

*:javaScript*

```
var preache_of_sock_changing = 'Lieutenant Dan';
```

*python*:

```python
preache_of_sock_changing = 'Lieutenant Dan'
```

*C*:

```c
int preache_of_sock_changing = 'Lieutenant Dan';
```

## ۵. روش Pascal Casing:

در این روش اگر نام متعیر از چند کلمه تشکیل شده باشه ابتدای هر کلمه رو با حرف بزرگ شروع میکنیم.

*javaScript*:

```javascript
var FirstName = "mehdi"
```

*python*:

```python
FirstName = "mehdi"
```

*C*:

```c
int NameOfVariable = 123;
```

## ۶. روش مجارستانی (Hungerian):

در این روش برای هر نوع شی موجود یک پیشوند درنظر گرفته می‌شود تا از روی نام شی بتوان به نوع آن پی برد. در ادامه و پس از این پیشوندها سایر کلمات بر اساس روش **Pascal Casing** نوشته می‌شوند.

*C*:

```c
char* strFirstName = "ahmad";
int intAge = 22;
```

*python*:

```python
strFirstName = "ahmad"
intAge = 22
```

# فصل سوم: قواعد نامگذاری توابع و کلاسها

این قواعد بسیار مشابه نامگذاری متغیرها هستند اما از ابتدا آنهارا مرور میکنیم.

---

## تابع:

تابع بلاکی از کد است که وظیفه ای دارد و کاری را انجام میدهد.

تابع در واقع ساختاری از کد را در یک خط خلاصه میکند و معمولا نتیجه ای را برمیگرداند.

> برای مطالعه بیشتر در مورد تابع اینجا کلیک کنید.

---

### در اینجا قواعد نامگذاری تابع و کلاس را بیان میکنیم:

یک نکته بسیار مهم در نامگذاری توابع استفاده از یک **فعل** در نام تابع است که بیانگر عملی است که تابع انجام میدهد.

## به مثال های زیر توجه کنید:

فرض کنید تابعی دارید که عمل **گرفتن (get)** انجام میدهد, برای مثال گرفتن نام کاربر: `getUserName`
یا تابعی که نام گرفته شده را در یک متغیر محلی **قرار میدهد(set)**: `setUserName`
تابعی که عمل **reload** اطلاعات از دیتابیس را انجام میدهد: `reloadData`

در هر سه مثال بالا نام توابع دارای افعال(**set**, **get**, **reload**) هستند که وظیفه توابع را توصیف میکنند.

### در سه مثال بالا از روش **camelCase** برای نامگذاری استفاده شده است, اما میتوانید از دیگر استانداردها نیز استفاده کنید.

---

## قواعد نامگذاری توابع:

### ۱. روش **camelCase**:

این روش به این ترتیبه که اگر نام تابع ما از چند کلمه تشکیل شده باشد حرف نخست کلمه اول را با حرف کوچک و حرف نخست بقیه کلمات رو با حرف بزرگ مینویسیم.

*:python*

```python
class Test:

    def setUserName(self, name):
        self.name = name

    def getUserName(self):
        return self.name
```

*:C*

```c
void setUserName(char* name){
```

```
    // codes
}

char* getUserName(){

    // codes
}
```

## ۲. روش underScore:

در این روش اگر نام تابع ما از چند کلمه تشکیل شده باشد بین هر کلمه یک **underscore** میگذاریم.

*python*:

```python
class Test:

    def set_user_name(self, name):
        self.name = name

    def get_user_name(self):
        return self.name
```

*C*:

```
void set_user_name(char* name){

    // codes
}

char* get_user_name(){

    // codes
}
```

## ۳. روش pascalCase:

در این روش اگر نام تابع از چند کلمه تشکیل شده باشد ابتدای هر کلمه رو با حرف بزرگ شروع میکنیم.

*python*:

```python
class Test:

    def SetUserName(self, name):
        self.name = name

    def GetUserName(self):
        return self.name
```

*C*:

```
void SetUserName(char* name){
    // codes
}

char* GetUserName(){

    return name;
}
```

### ۴. magic functions:

در برخی زبانهای برنامه نویسی توابعی به اسم **magic function** وجود دارند که با دو **underScore** در ابتدا و دو **underScore** در انتهای نام تابع تعریف میشوند.

*python*:

```
from os.path import join

class FileObject:

    def __init__(self, filepath='~', filename='sample.txt'):
        # open a file filename in filepath in read and write mode
        self.file = open(join(filepath, filename), 'r+')

    def __del__(self):
        self.file.close()
        del self.file
```

### ۵. نامگذاری توابع `private` و `protected` :

بهتر است قبل از نام توابعی که بصورت `private` یا `protected` تعریف میشود از **underscore** استفاده کنیم تا در طول برنامه به برنامه نویس یادآوری شود که تابع مورد نظر بصورت `private` یا `protected` تعریف شده است.

*PHP*:

```
// This variable is not available outside of the class
private $_someVariable;

class MyClass {
   // This method is only available from within this class, or
   // any others that inherit from it.

   protected function __behindTheScenesMethod() {}
   private function __getUserInfo(){}
}
```

---

# قواعد نامگذاری کلاسها:

## کلاس:

در برخی از زبانها قاعده ی جدیدی بنام **شی گرای** بوجود آمد, به این صورت که بخشی از کدها درون بلاکی بنام `class` قرار میگیرند.

در برنامه نویسی شی گرا ساختار اصلی برنامه شی ها هستند در واقع داده ها و توابعی که قرار است روی آن داده ها کار کنند در قالبی بنام شی قرار گرفته و یک واحد را تشکیل داده و کپسوله میشوند, پس توابع خارجی دیگر نمیتوانند به آن داده ها دسترسی داشته و آنها را تغییر دهند.

**برنامه نویسی شی گرا به نسبت از برنامه نویسی تحت تابع کارامدتر و پیچیده تر است.**

### دلایل برتری برنامه نویسی شی گرا:

قابلیت سازماندهی بهینه تر کدها
قابلیت شکستن برنامه به اجزای ساده تر و کوچکتر
امنیت بالاتر

> برای مطالعه بیشتر روی کلاس ها اینجا کلیک کنید.

در نامگذاری کلاسها نیز مانند نامگذاری توابع و متغیرها از روشهای **underScore**, **pascalCase**, **camelCase**, ... استفاده میشود.

تنها نکته ای که در نامگذاری `class` باید به آن توجه کنیم شروع کلمه اول نام کلاس با حرف بزرگ میباشد.

*python*:

```python
class Father:
    ## codes

class Child(Father):
    ## codes
```

*PHP*:

```php
// This variable is not available outside of the class
private $_someVariable;

class MyClass {
   // This method is only available from within this class, or
   // any others that inherit from it.
   protected function __behindTheScenesMethod() {}
}
```

# فصل چهارم: اصول comment گذاری

**Comment گذاری در برنامه نویسی اصولی دارد که در اینجا به آن میپردازیم.**

---

## Comment:

کامنت توضیحاتی هستند که در جاهای مختلف کد قرار میگیره و در مورد خود برنامه یا یک متغیر یا تابع و یا فایل خاصی از برنامه توضیحاتی را می دهند.

خطوطی که کامنت میشوند توسط سیستم اجرا نشده و تاثیری در خروجی برنامه و یا زمان اجرای برنامه ندارند.

> برای مطالعه بیشتر روی کامنت ها به این لینک مراجعه کنید.

---

## اهمیت Comment گذاری:

برنامه نویسانی که زمان زیادی را روی یک پروژه صرف میکنند اهمیت کامنت گذاری را خیلی خوب میدانند.

برنامه با بزرگ و بزرگ شدن خود پیچیده تر میشود, گاهی اوقات نیاز است برنامه نویس به کد های گذشته خود باز گردند, یا برنامه نویسان جدید به پروژه اضافه میشوند, در اینجاست که اهمیت کامنت مشخص میشود.

ایجاد کامنت برای متغیرها, توابع, فایلها و هر جای پر اهمیت دیگر پروژه باعث نجات برنامه نویش از سردرگمی میشود.

---

**در زبان های برنامه نویسی از علایم مختلفی برای کامنت گذاری استفاده میشه:**

در python از #
از // برای کامنت کردن یک خط و از /**/ برای کامنت کردن بیش از یک خط در زبانهای java, c#, c++, c, css, ...
استفاده میشود
در متلب از %
...

## انواع روش های Comment گذاری:

### ۱. Inline Commenting:

**Inline Comment** توضیحاتی در خصوص یکی از خطهای برنامه میدهد, مثلا دلیل ایجاد یک متغیر یا دلیل وجود یک شرط در برنامه.

*python:*

```python
# counts number of apples
aplCount = 0

# contains sum of fruits
sums = 0
```

*javaScript:*

```javascript
// counts number of apples
```

```
aplCount = 0

// contains sum of fruits
sums = 0
```

:C

```
// counts number of apples
int aplCount = 0

// contains sum of fruits
int sums = 0
```

:javaScript

```
if(callAjax($params)) { // successfully run callAjax with user parameters
 ... code
}
```

:python

```
if id not in featureSet: # check existance of 'id' in 'featureSet'
    return False
```

## ۲. Descriptive Blocks:

وقتی در برنامه نیاز به گذاشتن کامنت بزرگتر از یک خط دارید از **Descriptive Blocks** استفاده میکنید.

:javaScript

```
/**
  * @desc opens a modal window to display a message
  * @param string $msg - the message to be displayed
  * @return bool - success or failure
*/
function modalPopup($msg) {
    ...
}
```

:python

```
## this function get datas of api
## api is contains json code
## 'address' contains api url
def apiData(address):

    codes = requests.get(address)
    jsonResult = json.loads(codes.content)

    return jsonResult
```

## ۳. Group/Class Comments:

در بالای فایل برنامه یا در بالای هر کتابخانه ای که برای برنامه خود مینویسید قرار میگیرد که شامل مستنداتی شامل برنامه یا آن فایل میشود,

**از قبیل**:

هدف طراحی برنامه یا کتابخانه مورد نظر

نام و آدرس ایمیل برنامه نویس
**requirements** ها که شامل دیگر فایل ها و کتابخانه هایی هستند که این برنامه برای اجرا شدن به آنها نیاز دارد.

*java:*

```
/**
  * @desc this class will hold functions for user interaction
  * examples include user_pass(), user_username(), user_age(), user_regdate()
  * @author Ahmad Abdollahzade ahmadabdollahzade74@gmail.com
  * @required settings.php
*/

abstract class myWebClass { }
```

*python:*

```
## @desc this class will hold functions for user interaction
## examples include user_pass(), user_username(), user_age(), user_regdate()
## @author Jake Rocheleau jakerocheleau@gmail.com
## @required settings.php

class getInfo:
    ...
```

## ۴. نحوه کامنت گذاری در کدهای html و css:

در **css** از // و /**/ برای کامنت گذاری استفاده میکنیم.

*css:*

```
//@keyframes foo {
  from, to { width: 500px; }
  50% { width: 400px; }
}
@keyframes bar {
  from, to { height: 500px; }
  50% { height: 400px; }
}
```

```
// Do some stuff.
.foo { animation: bar 1s infinite; }
/* Whoops, the .foo block is commented out! */
```

همچنین در html از <!--- در ابتدا و از--> در انتهای خط برای کامنت کردن یک خط یا مجموعه ای از خطوط استفاده میشود.

*html:*

```
<!-- You will not be able to see this text. -->

You can even comment out things in <!-- the middle of --> a sentence.

<!--
Or you can
comment out
a large number of lines.
-->
```

برای دیدن مثالهای بیشتر به این لینک مراجعه کنید.

# فصل پنجم: Simple Better Than Complex

**در این فصل به اصولی میپردازیم که بتوان کد پیچیده ای را به کد ساده ای تبدیل کنیم.**

---

## جلوگیری از خلاصه نویسی:

کد ساده همیشه بهتر از کد پیچیده و گیج کننده است, به سه مثال زیر دقت کنید:

:C

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
    return true;
}
else
{
    return false;
}
```

and

```
if (hours < 24 && minutes < 60 && seconds < 60)
    return true;
else
    return false;
```

and

```
return hours < 24 && minutes < 60 && seconds < 60;
```

هر سه کد یک کار را انجام میدهند,اما کدام یک خواناتر و قابل فهم تر است؟ **بدون شک کد اول**

---

در بسیاری از زبانها اگر در یک **Block** فقط یک دستور وجود داشته باشد میتوان از گذاشتن {} خوداری کرد و یا در زبان پایتون دستور را مقابل خود بلاک نوشت.

اما استفاده از این قابلیت ها اصلا توصیه نمیشود زیرا باعث عدم خوانایی کد میشود.

## برای مثال:

*:python*

```
if hours < 24 and minutes < 60 and seconds < 60: return True
else: return False
```

به کد بالا دقت کنید, درست است که این کد کار میکند اما کد زنی به این روش اصلا توصیه **نمیشود**.

## بهتر است کد بالا بصورت زیر نوشته شود:

```
if hours < 24 and minutes < 60 and seconds < 60:
    return True
```

```
else:
    return False
```

یا این کد را در نظر بگیرید:

:*C*

```
if (hours < 24)
    if (minutes < 60)
         if (seconds < 60)
            return true;
return false;
```

این کد ممکن است باعث گیج شدن برنامه نویس شود, و اصلا کد نویسی به این روش توصیه **نمیشود**.

### بهتر است کد بالا بصورت زیر نوشته شود:

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
    return true;
}
else
{
    return false;
}
```

---

## اصول Functional Programming:

> برای مطالعه در مورد Functional Programming اینجا کلیک کنید.

استفاده صحیح از توابع باعث مرتب شدن کدها و دسته بندی هر قسمت از کد و در نتیجه خوانایی بیشتر کد میشود.

یکی از مهمترین فواید استفاده از توابع سرعت بخشیدن به **Error** یابی برنامه میباشد.

> تابع خوب تابعی است که یک کار را انجام دهد.

برنامه زیر را در نظر بگیرید که فاقد تابع است, کدها درهم شده اند و وقتی برنامه رشد کند و به صدها خط برسد دیگر پیدا کردن یک قسمت از کد, تغییر برنامه, یا رفع یک مشکل کاری بسیار دشوار خواهد بود.

:*python*

```
def showColumns():
    # Gets arranged data and shows them in columns for each mac address

    makeColumns()
    column41S = addStar41()
    column70S = addStar70()
    column52S = addStar52()
    column53S = addStar53()

    c = 0
    print("             mac : 41")
    print("-----------------------------------")
    for i in column41S:
        print(str(c) + ") " + i[0] + "  " + i[1] + "  " + i[2])
        c += 1

    print("***********************************\n")

    c = 0
```

```
    print("             mac : 70")
    print("-----------------------------------")
    for i in column70S:
        print(str(c) + ") " + i[0] + "  " + i[1] + "  " + i[2])
        c += 1

    print("*************************************\n")

    c = 0
    print("             mac : 52")
    print("-----------------------------------")
    for i in column52S:
        print(str(c) + ") " + i[0] + "  " + i[1] + "  " + i[2])
        c += 1

    print("*************************************\n")

    c = 0
    print("             mac : 53")
    print("-----------------------------------")
    for i in column53S:
        print(str(c) + ") " + i[0] + "  " + i[1] + "  " + i[2])
        c += 1

    # Deletes extra data from RAM
    del column41S[:]
    del column70S[:]
    del column52S[:]
    del column53S[:]

    exitNum = input("Exit(y/n)? ")

    if(exitNum.upper() == 'Y'):
        exit()

    elif(exitNum.upper() == 'N'):

        columnMacForPlot = input("Enter column mac address: ")

        if columnMacForPlot == "41":

            X, Y = getRangeOfPlot(column41)

        elif columnMacForPlot == "70":

            X, Y = getRangeOfPlot(column70)

        elif columnMacForPlot == "52":

            X, Y = getRangeOfPlot(column52)

        elif columnMacForPlot == "53":

            X, Y = getRangeOfPlot(column53)

        else:
            plotData()

    else:
        plotData()

    plt.plot(X, Y, 'r')
    plt.xlabel("range")
    plt.ylabel("tempreture")
    plt.title('Tempreture plot')
    plt.show()
```

میتوان تابع بالا را به چندین تابع کوچکتر تبدیل کرد:

```
def addStar(column):
    # Puts * for columns that dont have data after one minute from last column

    dataTime = column[0][2].split(" ")[1]
    tmp = int(dataTime.split(":")[1])

    columnS = column[:]

    for i in columnS:

        dataTime = i[2].split(" ")[1]
        minutes = int(dataTime.split(":")[1])

        if abs(tmp - minutes) > 1:
            columnS.insert(columnS.index(i), ['*', '*', '*'])

        tmp = minutes

    return columnS


def showData(column, mac):
    # Shows data of each mac address in a column

    c = 0
    print("             mac : {0}".format(mac))
    print("-----------------------------------")
    for i in column:
        print(str(c) + ") " + i[0] + "  " + i[1] + "  " + i[2])
        c += 1

    print("***********************************")


def showColumns():
    # Gets arranged data and shows them in columns for each mac address

    makeColumns()
    column41S = addStar(column41)
    column70S = addStar(column70)
    column52S = addStar(column52)
    column53S = addStar(column53)

    showData(column41S, 41)
    showData(column70S, 70)
    showData(column52S, 52)
    showData(column53S, 53)

    # Deletes extra data from RAM
    del column41S[:]
    del column70S[:]
    del column52S[:]
    del column53S[:]

    plotData()


def getRangeOfPlot(column):
    # Gets range of X and Y from user and makes them for each mac addresses logs

    Y = []
    fromRow = int(input("From: "))
    toRow = int(input("To: "))

    if fromRow >= 0 and toRow < len(column41):

        X = range(fromRow, toRow)
```

```
        for i in range(fromRow, toRow):
            Y.append(column[i][0])

    else:
        plotData()

    return X, Y

def plotData():
    # Draws plot for each mac address

    exitNum = input("Exit(y/n)? ")

    if(exitNum.upper() == 'Y'):
        exit()

    elif(exitNum.upper() == 'N'):

        columnMacForPlot = input("Enter column mac address: ")

        if columnMacForPlot == "41":

            X, Y = getRangeOfPlot(column41)

        elif columnMacForPlot == "70":

            X, Y = getRangeOfPlot(column70)

        elif columnMacForPlot == "52":

            X, Y = getRangeOfPlot(column52)

        elif columnMacForPlot == "53":

            X, Y = getRangeOfPlot(column53)

        else:
            plotData()

    else:
        plotData()

    plt.plot(X, Y, 'r')
    plt.xlabel("range")
    plt.ylabel("tempreture")
    plt.title('Tempreture plot')
    plt.show()

plotData()
```

در برنامه اول کد ما شامل یک تابع است که در مثال بعد تبدیل به پنج تابع شده است که درون یکدیگر صدا زده شده اند این کار باعث کمتر شدن تعداد خطهای برنامه و درنتیجه مرتب شدن برنامه و میشود.

> برای مشاهده کد کامل برنامه بالا به این لینک مراجعه فرمایید.

# فصل ششم: استفاده از ابزار برای استاندارد سازی کدها

در این فصل ابزاری را برای بررسی و استاندارد سازی کدها معرفی میکنیم.

---

## ۱. Coala:

کار این ابزار اصلاح و استاندارد سازی کدها برای تمام زبانهاست بر اساس PEP8 میباشد.

> برای آشنایی با PEP8 به این لینک مراجعه کنید.

میتوانید coala را در اینجا مشاهده کنید.

### نصب :

```
pip3 install coala-bears
```

### اجرا:

```
cd project; coala --files="**/*.py" --bears=PEP8Bear --save
```

---

## ۲. autopep8:

یک ماژول پایتون است که بصورت اتوماتیک کدهای پایتون را به فرمت PEP8 میبرد.

### نصب:

```
pip install autopep8
```

میتوان این ماژول را روی Editor های مثل atom یا vscode جداگانه نصب کرد.

---

> اکثر Ide های مفروف ابزاری به منظور استاندارد سازی و حتی شخصی سازی آنرا دارند.