Contents

[Introuction to Programming with Matlab 2](#_Toc96851349)

[Logical Indexing 2](#_Toc96851350)

[Data type change 5](#_Toc96851351)

[Structs 6](#_Toc96851352)

[Cells 9](#_Toc96851353)

[Mastering Programming with Matlab 13](#_Toc96851354)

[Recursion 13](#_Toc96851355)

[Variable number of Arguments 15](#_Toc96851356)

[Function Handles and Nested Functions 16](#_Toc96851357)

[Linear Algebra 18](#_Toc96851358)

[Live Scripts 23](#_Toc96851359)

[Algorithmic Complexity 23](#_Toc96851360)

[Efficiency & Speed up 25](#_Toc96851361)

[Object Orinted Programming (FaraDars) 25](#_Toc96851362)

# 

# Introduction to Programming with Matlab

## Logical Indexing

این ویژگی مختص نرم افزار MATLAB بوده و در زبان های دیگر به این سادگی قابل انجام نیست. متلب هر چیزی که غیر صفر باشه رو به عنوان True و صفر رو به عنوان False در نظر میگیره (به صورت logical). نیز True رو به صورت یک و False رو به صورت صفر نشون میده.

>> [4 -1 7 5 3] > [5 -9 6 5 -3]

ans =

0 1 1 0 1

این ماتریس (array) ans یک ماتریس هست که Data type اون logical هست چون از عملگر مقایسه استفاده شده. مثلا از عملگر های منطقی مثل && و || و ~ هم استفاده کنیم جوابی که میاد از نوع logical هستش و این فانکشن هم خود متلب داره به این صورت که

>> Hossein = logical ([ 1 -2 0 9.25 -78 0])

Hossein =

1 1 0 1 1 0

یعنی یک آرایه رو گرفت به صورت یک آرایه لاجیکال پس داد.

حالا یک مثال سخت تر

>> c = [2>1, 2<1, ~ (3>2 && 4>5)]

c =

1 0 1

>> a = 1:3

a =

1 2 3

>> a(c)

ans =

1 3

این دستور برای متلب یعنی فقط باید اون المان هایی از a رو انتخاب کنه که c مقدار منطقی true داره. یعنی ما به a عدد اندیس ها رو ندادیم که بگیم عضو های اول و سوم رو نشون بده، به جاش یک وکتور logical دادیم. به این کار میگن logical indexing.

>> v = [1 5 9 -9 8 -96 5 0 36 -54 -45]

v =

1 5 9 -9 8 -96 5 0 36 -54 -45

>> v >= 0

ans =

1 1 1 0 1 0 1 1 1 0 0

این ans هم logical هست؛ بدین صورت که جاهایی که منفی هست تو وکتور، اونا رو صفر گذاشته. خوب حالا مثلا بخواهیم اعداد غیر منفی وکتور v رو در یک وکتور دیگه ذخیره کنیم بدین صورت هست:

>> w = v (v >= 0)

w =

1 5 9 8 5 0 36

این کار با یک حلقه for هم قابل انجامه. اینجا هم کد نویسی رو کم کردیم و هم اینکه حدود ده تا بیست برابر سریعتر انجام میشه. مشخصه که لازم نیست که حتما در عملگر مقایسه ای مون از v استفاده کنیم. میتونیم از هر وکتور دیگه ای هم استفاده کنیم که بالاخره یک وکتور logical بده، میتونیم اصلا دو تا وکتور رو با هم مقایسه کنیم.

کاری که میکردیم این نکته رو داره که ما اون وکتور اصلی رو تغییر اش نمیدیم بلکه یک وکتور جدید میسازیم که یک سری تفاوت هایی داره به اون وکتور اصلیه. حالا اگه بخواهیم وکتور اصلیه رو هم تغییر بدیم بدین صورته.

خوب قاعدتا چیزی که بخواهیم تغییراش بدیم بایدسمت چپ تساوی باشه. الان این کد زیر بدین صورته که المان های منفی رو به جاش صفر میذاره.

>> v = [56 34 -2 7 20 -95 0 -45]

v =

56 34 -2 7 20 -95 0 -45

>> v(v<0) = 0

v =

56 34 0 7 20 0 0 0

میتونیم اعداد مختلفی هم به جای اعداد منفی بذاریم که خوب سمت راست رو یک وکتور میذاریم (تخصیص به ترتیب ستونی هست).

در دو بعدی هم ما از اینا استفاده میکنیم

A =

1 2 3

4 5 6

>> B = A(A>2)

B =

4

5

3

6

یک نکته مهم دیگه اینه که متلب براش ستون مهم تر از ردیف هست(column majored order) . یعنی اول ستون اول رو چک میکنه بعد میره ستون دوم و همینطور. در واقع بدین صورت هست که اندیس گذاری منطقی رو وقتی برای آرایه با بیش از یک بعد استفاده میکنیم اول اونو به یک ستون تبدیل میکنه و بعد کارش رو انجام میده. البته وقتی اندیس گذاری منطقی در سمت چپ باشه گفتیم که فقط مقدار المان ها رو تغییر میده و المانی رو حذف نمیکنه پس همچنان بعد هاش حفظ میشه.

>> A(A>3) = 0

A =

1 2 3

0 0 0

یک نکته خیلی مهم اینه که این قانونه. یعنی اینجوری نیست که اینجا چون ابعاد به هم نمیخوره ماتریس دو بعدی رو حفظ کرده باشه. ممکنه اندیس سمت راست تساوی طوری باشه که هیچ المانی حذف نشه ولی باز هم به یک ستون تبدیل میکنه اونو.

A =

1 2 3

4 5 6

>> B = A(A>0)

B =

1

4

2

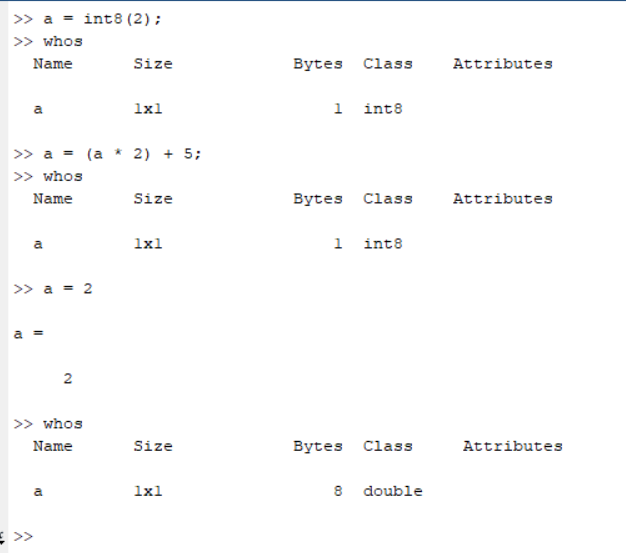
5

3

6

## Data type change

یک نکته راجع به تغییر دیتا تایپ ها وجود داره بدین صورت که یک متغییر نوع اش با توجه به مقدار اش تغییر میکنه به طور اتوماتیک. ما میتونیم یک رشته رو به متغییری که یک عدد رو داره تخصیص بدیم، نوع اون متغییر از نوع عددی به char تغییر میکنه. همینطور در مورد انواع مختلف متغییر های عددی هم به همین صورته.



متغییر a به عنوان یک متغییر از نوع int8 تعریف شده، وقتی با اعداد دیگه جمع و تفریق میشه همین دیتا تایپ رو داره ولی وقتی عدد 2 رو به متغییر اختصاص میدیم 🡸 الان دیگه سمت راست تساوی فقط یک عدد هست:2 . قبلا یک متغییر از نوع int8 بود که با عدد های دیگه که تو اون int8 جا میشدن جمع و ضرب شده بود و حاصل هم همون int8 بود ولی الان 2 خالی **به طور پیشفرض** دیتا تایپ اش double هست پس دیتا تایپ متغییر هم تغییر میکنه.

## Structs

المان های یک آرایه گفتیم که باید همه شون از یک نوع باشن. حالا اگر دیتا هایی که نوع هاشون مختلف هست رو بخواییم با هم گروه کنیم از struct استفاده میکنیم.

تفاوت struct , array: 1. استراکت فیلد داره نه المنت 2. به فیلد های استراکت با name دسترسی پیدا میکنیم نه اندیس 3. فیلد های یک استراکت میتونن نوع های مختلفی داشته باشن. فیلد یک استراکت خودش میتونه یک استراکت باشه (استراکت یک استراکت داخلش داشته باشه).

>> r.snn = 12345678

r =

snn: 12345678

متغییر r رو ساخت از نوع استراکت. یک فیلد بهش داد به اسم snn و سپس 12345678 رو داخل فیلد snn متغیر استراکت گذاشت. پس همون نقطه در واقع مهمه.

>> r.name = 'Hossein'

r =

snn: 12345678

name: 'Hossein'

>> r.address.city = 'Mashhad'

r =

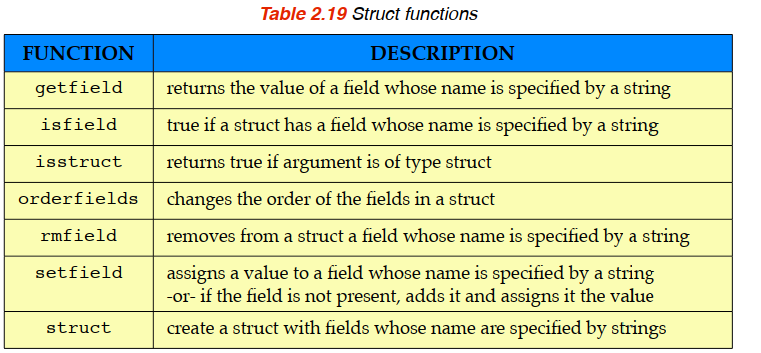
snn: 12345678

name: 'Hossein'

address: [1x1 struct]

یک فیلد دیگه تحت عنوان name برای متغیر r ساخت و یک رشته توش گذاشت. یک فیلد دیگه هم تحت عنوان address براش ساخت و یک استراکت (دومین نقطه) توش گذاشت. رشته رو هم در فیلد city فیلد address متغیر r گذاشت.

تازه فیلد استراکت میتونه یک آرایه هم باشه و هم چین یک المان یک آرایه میتونه یک استراکت باشه البته قانون هم چنان هست که همه المان های آرایه باید از یک تایپ باشن.



>> account.number = 1234567;

>> account.owner.name = 'Joe Smith';

>> account.owner.email = 'Joe@gmail.com';

>> account.balance = 5000;

>> account

account =

number: 1234567

owner: [1x1 struct]

balance: 5000

\* ما الان اینجا یک استراکت ساختیم که سه تا فیلد داره، یکی از فیلد ها خودش یک استراکت هست که بحث شده قبلا. حالا اگر بخواهیم Account رو تبدیل به یک array of structs بکنیم فقط کافیه بهش یک المان دیگه بدیم :

>> account(2).number = 7654321

account =

1x2 struct array with fields:

number

owner

balance

\* حالا که ما یک struct array داریم متلب فقط اسم فیلد ها رو نشون میده، همچنین استراکت جدید (المان دوم آرایه) همه همون فیلد های المان اول رو داره. این به این دلیله که باید تایپ هر دو المان یک باشه اونا به طور پیشفرض خالی اند.

>> account(1)

ans =

number: 1234567

owner: [1x1 struct]

balance: 5000

>> account(2)

ans =

number: 7654321

owner: []

balance: []

استراکت ها بایستی فیلد های یکسانی داشته باشن ولی value های داخل اون فیلد ها میتونه تایپ ویا سایز مختلفی داشته باشه در واقع هر المان آرایه Account باید فیلد های number, owner, balance رو داشته باشه، قانون homogenity تا همین درجه اعمال میشه و به value اون فیلد ها کار نداره.

\* بخواهیم یک استراکت بسازیم هم خیلی ساده اینجوره:

>> course = struct('Area', 'CS', 'number', 103, 'title', 'Matlab for students')

course =

Area: 'CS'

number: 103

title: 'Matlab for students'

## Cells

هر متغییری یک جایی رو در حافظه اشغال میکنه و هر مکان مموری آدرس یونیک خودش رو داره. مموری کامپیوتر رو میشه به عنوان یک وکتور طولانی از memory cell ها در نظر گرفت و آدرس هر سل ایندکس هست که با 1 شروع میشه و عدد بعدی سل های همسایه رو مشخص میکنه. حال **Pointer** متغییری هست که آدرس سل رو ذخیره میکنه، حالا متلب پوینتر رو هم سل در نظر میگیره! در خصوص سل ها قوانین به شدت سخت گیرانه ای وجود داره.

سل یک دیتا تایپ قدرتمند هست و مثل استراکت ها امکان ذخیره دیتای ناهمگن رو میده ولی بر خلاف استراکت اینجا اندیس داریم نه فیلد.

مثلا اگر بخواهیم یک صفحه نوشته رو ذخیره کنیم طول خط ها با هم متفاوته و نمیشه تو ماتریس ذخیره شون کنیم، اینا رو با سل هندل میکنیم. یک سل که چندین المان داره، هر المان مثلا یک خط اون صفحه هستش که سایز های مختلفی داره هر المان این دیتا. مثلا این اسکریپت این نوشته رو داخل متغییر page قرار میده، در واقع هر خط رو داخل یک المان این متغییر میذاره.

%% The Ultimate Legend of Big John

page{1} = 'You could find';

page{2} = 'Tall, dark hair';

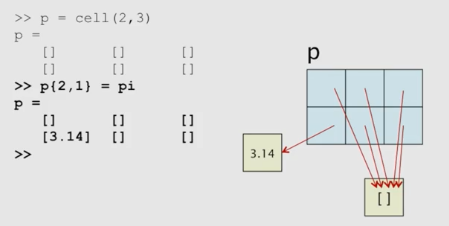
page{3}= 'They say he pulled';

page{4} = 'And when he''d stick in';

page{5}= 'On big John';

میتونیم تک تک به این المان ها دسترسی داشته باشیم و مثلا پرینت شون کنیم. البته که این خیلی ناهمگن نبود.

یک مثال دیگه: اول فانکشن cell رو فراخوانی میکنیم تا یک cell array خالی بسازه؛ منظور از خالی اینه که هر المان آدرس array خالی رو در خودش داره، هر المان به اون آرایه خالی point میکنه حالا به یکی از این المان ها یک عددی چیزی اختصاص بدیم در واقع آدرس اون عدد در اون المان ذخیره میشه و نه خود اون عدد.



3.14 رو فانکشن pi ساخته و جایی در حافظه ذخیره کرده حالا المان 2,1 اون سل مون به اون آدرس جایی در حافظه که اون اسکالر رو داره اشاره میکنه یعنی اون آدرس رو در خودش ذخیره میکنه. نکته مهم اینه که گرچه آدرس در اون المان ذخیره شده ما نه میتونیم به اون آدرس نگاه کنیم و نه تغییرش بدیم، فقط میتونیم اون چیزی که تو اون آدرس هست رو retrive کنیم.

>> p = cell(2, 3)

p =

[] [] []

[] [] []

>> p {2, 3} = [2 4; 6 8; 10 12];

>> p{1, 2} = sum(p{2, 3});

>> p{2, 3}

ans =

2 4

6 8

10 12

>> p{1, 2}

ans =

18 24

این مثال برای این بود که هم استفاده از {} در سمت راست تساوی مشخص بشه و هم اینکه ببینیم آدرس ها رو جمع نمیکنه. حالا تو این p میتونیم char هم بذاریم

>> p{1, 1} = 'awsome'

p =

'awsome' [1x2 double] []

[] [] [3x2 double]

وقتی به یک رشته یا آرایه اشاره میکنیم در سل ها در واقع آدرس اولین المان اون آرایه در اون مکان ذخیره میشه یعنی آدرس a اونجا ذخیره شده. حالا یک چیز جالب:

>> class(p)

ans =

cell

>> class(p{1, 1})

ans =

char

>> class(p(1, 1))

ans =

cell

>> p(1, 1)

ans =

'awsome'

>> p{1, 1}

ans =

awesome

در واقع تو سل ها بین پرانتز و کروشه فرق هست وقتی از پرانتز استفاده کنیم برای retrive کردن دیدیم که رشته رو داخل کوتیشن گذاشت، اگر عدد بود داخل براکت میذاشت که نشون بده این در واقع به اون آبجکت point میکنه ولی وقتی از کروشه استفاده کردیم خود رشته رو نشون داد چون تایپ اون char بود دیگه. حالا این متغییر p در واقع داده های یکسانی رو داره، همه شون از نوع cell هستن 😊

No two cell pointers can point at the same object.

>> c1 = {[1 2]; 'apple'}

c1 =

[1x2 double]

'apple'

>> c2 = c1;

>> c1{2, 1} = 'orange'

c1 =

[1x2 double]

'orange'

>> c2

c2 =

[1x2 double]

'apple'

اینجا وقتی گفتیم c2 = c1 بدین معنی نیست که المان های c2 به همون آبجکت هایی اشاره میکنن که c1 اشاره میکنه بلکه یک کپی ازش ساخته میشه و دیگه به بعد c2 مستقل از c1 عمل میکنه.

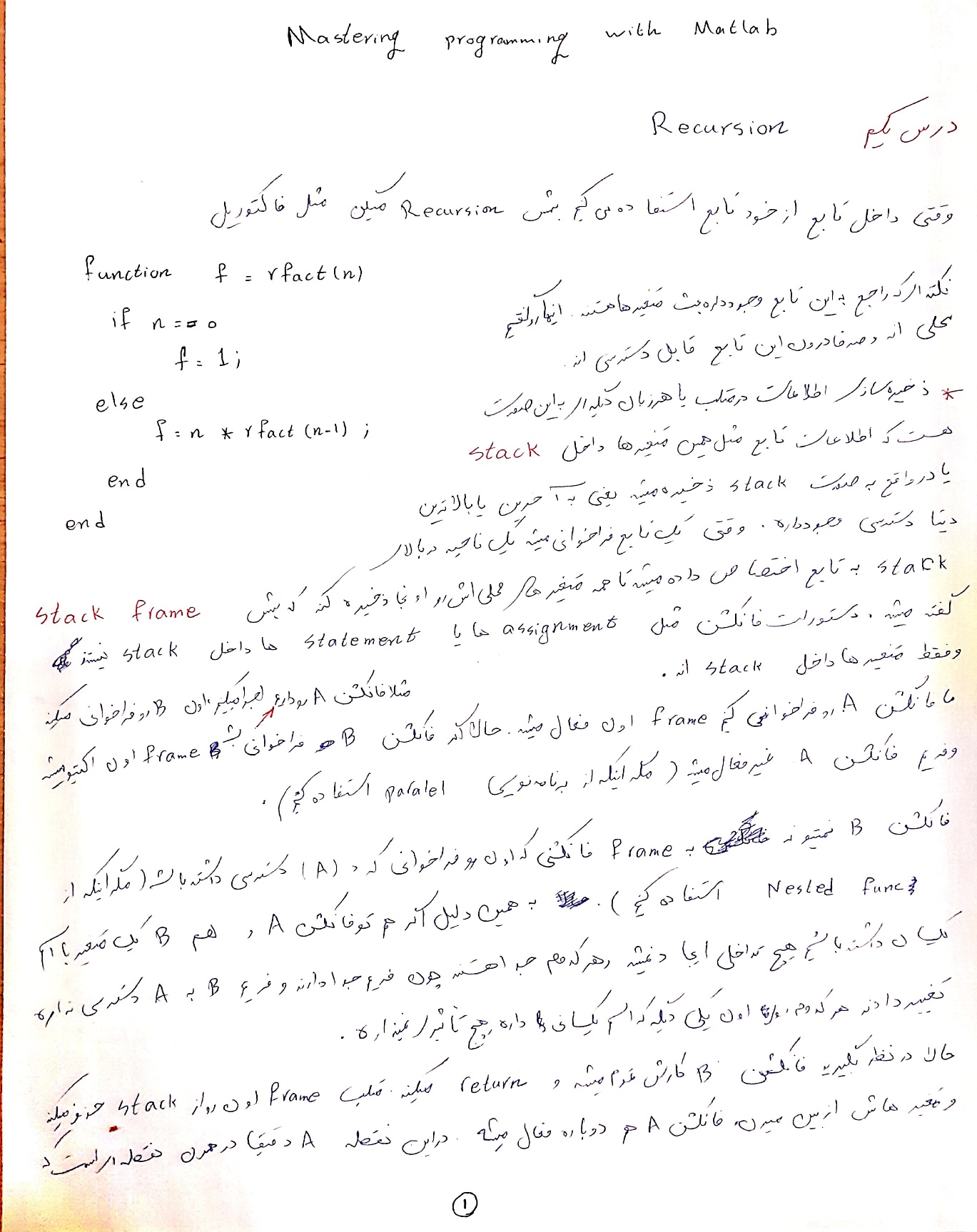
فرادرس:

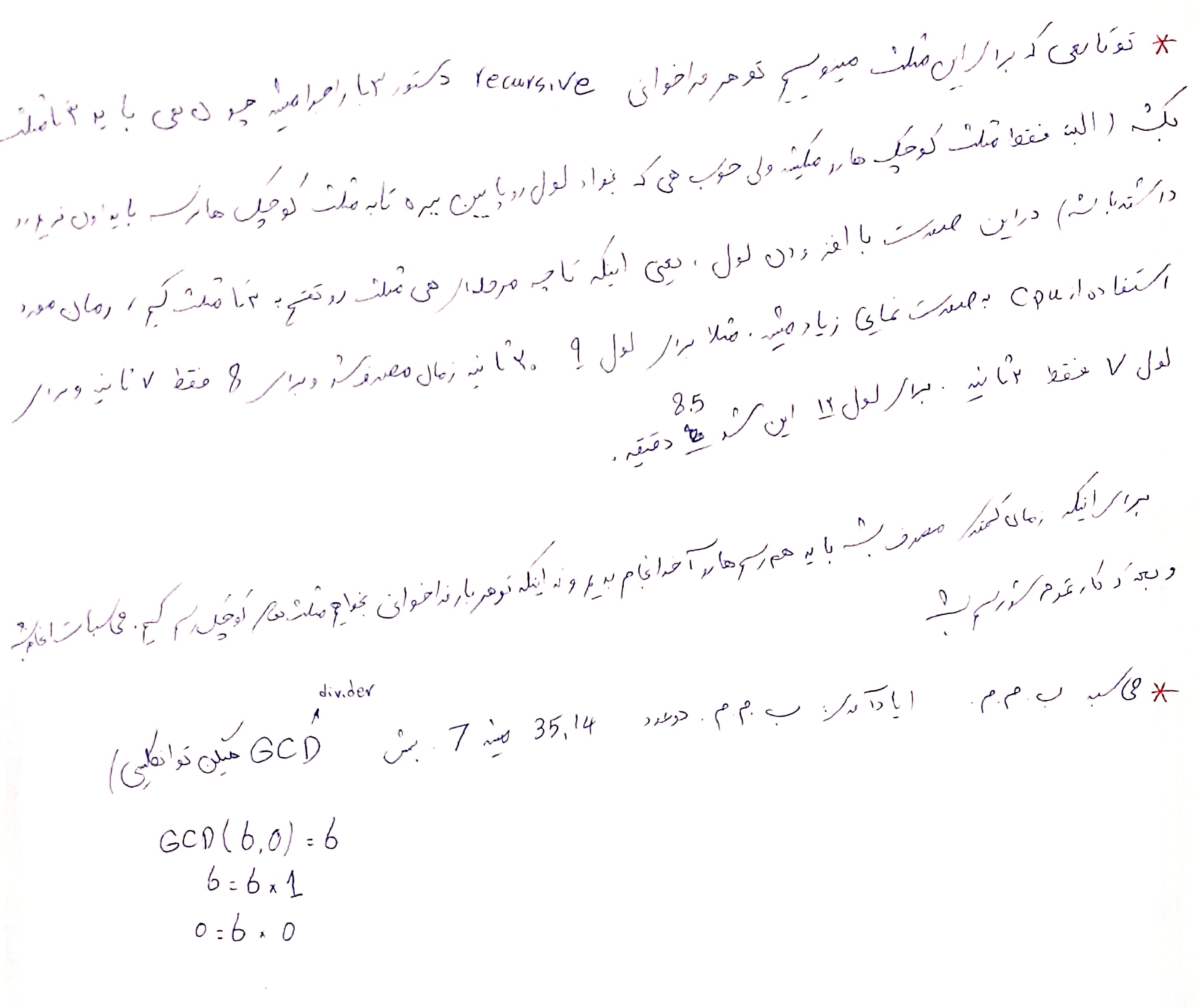
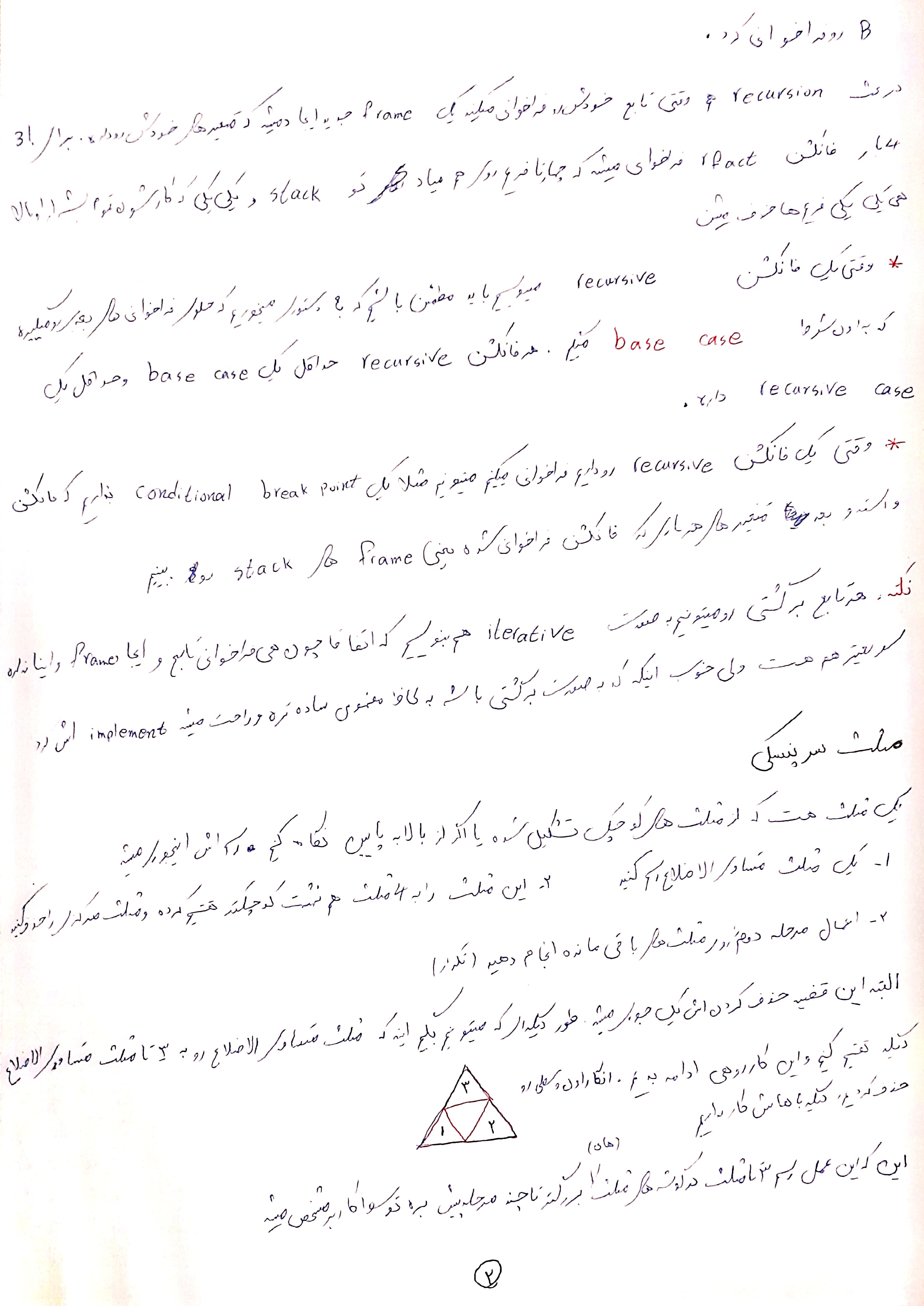
آرایه سلولی در واقع همون ماتریس هست ولی المان هاش هر چیزی میتونه باشه؛ عدد، کاراکتر، رشته، یک آرایه سلولی دیگه. این نوع داده بدون نام هست و به یک ساختار مستطیلی محدود هستش، ولی مثلا مشخصات دانشجو ها رو بخواهیم ذخیره کنیم میشه ولی یه کم وقت گیر میشه. دیگه نهایتا میشه مثل یک فایل اکسل.

از نوع داده structure در این نوع موارد استفاده میکنیم، یک داده ساختار یافته است که بسیار حیاتی و کارآ هست.

# Mastering Programming with Matlab

## Recursion





## Variable number of Arguments

مثلا برای تابع fprintf ما می‌تونیم به تعداد بینهایت ورودی بهش بدیم. اینجا میخواهیم خودمون چنین فانکشنی بنویسیم. nargin به ما تعداد ورودی های فراهم شده برای فانکشن رو میده، اینکه وقتی فانکشن فراخوانی شد چند تا ورودی براش فراهم شد. nargout هم تعداد خروجی هایی که خواسته شده هنگامی که تابع فراخوانی شد رو میده. اگر فانکشنی داشته باشیم که بخواهیم محدودیتی برای تعداد المان های ورودی اش نداشته باشه از varargin استفاده می‌کنیم:

function print\_all(varargin)

for ii = 1:nargin

fprintf('Here is input argument number %d: %d\n', ii, varargin{ii})

end

end

nargin همچنان تعداد ورودی هایی که هنگام فراخوانی فانکشن بهش داده شده رو میده همچنین varargin یک cell vector هست که تمام ورودی ها رو ذخیره میکنه و باید از کروشه استفاده کنیم تا به عناصر اش دستیابی داشته باشیم.

وقتی از varargin استفاده میکنیم همچنین میتوانیم از ورودی های دیگه هم تو تعریف فانکشن استفاده کنیم ولی در این صورت varargin حتما باید آخر باشه.

همچنین ممکنه گاهی نیاز پیدا کنیم که تعداد خروجی هامون محدود نباشه. این هم میتونیم توسط varargout انجام بدیم بدین صورت:

function varargout = distribute (v)

for ii = 1:length(v)

varargout{ii} = v(ii);

end

end

این فانکشن یک بردار رو به عنوان ورودی میگیره و اعضای اون رو اختصاص میده به تک تک خروجی هایی که فانکشن کالر درخواست داده.

## Function Handles and Nested Functions

معمولا برای صدا زدن یک تابع اسم اش رو مینویسیم اما میتونیم از روش دیگه‌ای هم استفاده کنیم. Function handle به متغییری میگیم که باهاش تابع رو فراخوانی میکنیم. این کار بعضی جا ها کاربرد داره از جمله: وقتی از یک فانکشن به عنوان ورودی یک فانکشن دیگه استفاده می‌کنیم، Callback fun مثلا تو یوزر اینترفیس وقتی یک کلید رو میزنیم چه فانکشنی اجرا بشه، ایجاد یک فانکشن بدون نام، فانکشن داخل یک m-file یک سابفانکشن داخل یک m-file رو دیگه فراخوانی کنه.

برای ایجاد یک فانکشن هندل صرفا از یک @ قبل اسم فانکشن استفاده میکنیم.

فانکشن fplot: تقریبا مثل فانکشن plot هست اما به عنوان ورودی اول یک فانکشن هندل میگیره و اون فانکشن رو در محدوده ای که به عنوان ورودی دوم بهش میدیم رسم میکنه:

fplot(@tan, [0, pi])

نسبت به plot یکسری تفاوت هایی داره از جمله اینکه نقاط ناپیوستگی رو خیلی بهتر باهاشون برخورد میکنه.

فانشکن بی نام: نمیتونه توش ساختار های کنترلی باشه

>> poly = @(x) 2\*x.^3 - x.^2 + 2\*x - 12

poly =

function\_handle with value:

@(x)2\*x.^3-x.^2+2\*x-12

ما میخواستیم یک تابع چند جمله ای داشته باشیم. تو یک خط این تابع رو تعریف کردیم و هند لاش مشخص شد. بعد @ آرگیومنت های ورودی رو تو پرانتز میذاریم و بعدش هم که دیگه statement رو مینویسیم.

یک نکته دیگه که هست اینه که متغیر های داخل این فانکشن در بیرون قابل دسترسی نیستند ولی متغیر های بیرونی داخل این فانکشن قابل دسترسی اند. این قضیه دسترسی فانکشن به متغییر بیرونی مثل اینه که متغییر گلوبال داشته باشیم تو فانکشن های معمولی مون. حالا یک نکته اینه که ما متغییر رو بیرون تعریف میکنیم، تو فانکشن ازش استفاده میشه بعدش اگر متغییر رو تغییر بدیم برای فانکشن مهم نیست و اگر دوباره فانکشن رو فراخوانی کنیم از مقدار اولیه متغییر استفاده میکنه.

برای ورودی چنین فانکشنی که خوب مساله ای نبود. حالا اگر بخواهیم این چنین فانکشنی چندتا خروجی بده یک راه اینه که داخل تعریف از یک تابع built in استفاده کنیم که خودش چندتا خروجی میده. یک راه دیگه اینه که از deal استفاده کنیم که به ازای هر اینپوت آرگیومنت اش یک output argument میده. بدین صورت

>> xyz = @(x, y) deal (x\*y, x+y)

xyz =

function\_handle with value:

@(x,y)deal(x\*y,x+y)

>> [p, s] = xyz(10, 20)

p =

200

s =

30

تا حالا سه نوع فانشکن باهاش کار کردیم: main function, subfunction, anynomous function یک نوع دیگه فانکشن nested function هست. مثل سابفانکشن هست و داخل یک m-file با یک main function وجود داره ولی در واقع is defined inside the body of another function که بهش parent function میگیم. معمولا parent function همون main function هست و یکی یا چند تا nested func داخلش هست.

nested function به متغیر های parent func دسترسی داره و اون هم به متغیر های داخل nested func اش دسترسی داره (البته از ورژن 2019b به بعد به شرطی که متغییر اول در داخل main func تعریف شده باشه یا هم اینکه اون متغییر جز متغییر های ورودی فانکشن parent تعریف شده باشه). پس به صورت efficient تری با هم کار میکنن.

function [y1, y2] = Nested\_Function (x)

c = 10;

sub(c, x);

y1 = inner(x);

function out = inner(in)

out = c\*in;

end

c = 11;

sub(c, x)

y2 = inner(x);

end

function sub(in1, in2)

fprintf('Multiplying %d times %d\n', in1, in2)

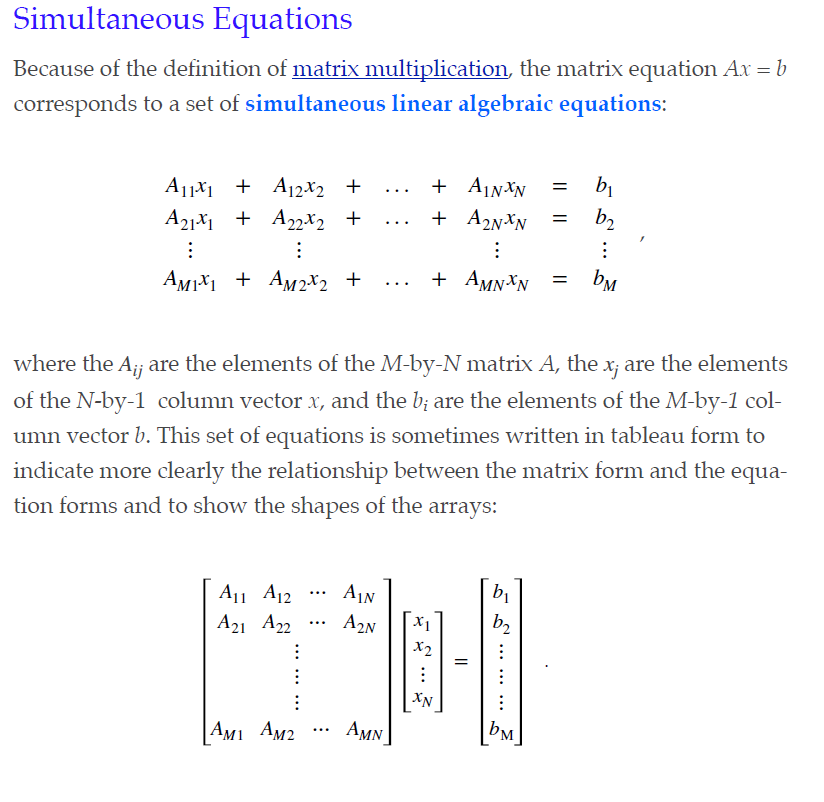
end

اینجا ما یک main function داریم که parent func برای nested func مون هم هست. یک nested func داریم به اسم inner که دو بار فراخوانی شده. یک سابفانکشن هم داره که جهت پرینت پیام ها ازش استفاده شده. متغییر c اینجا متغییر مهمی هست. در دوتا فانکشن بهش دسترسی پیدا میشه تو خود متلب این رنگ آبی کمرنگ داره. این رنگ رو برای متغییر های گلوبال هم استفاده میکنه متلب. البته اونجا از keyword مخصوص خودش global استفاده میکردیم و البته اونها خیلی هم مستعد ایجاد باگ هستند.

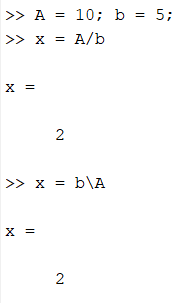
یک nested func میتونه خودش یک nested func داشته باشه. مثلا فانکشن اصلی A دو تا nested func داره به نام B, C که خود B یک nested func داره به نام D. حالا A میتونه B, C رو صدا بزنه ولی D رو نمیتونه و فقط B میتونه D رو صدا کنه. از طرفی B, C میتونن همدیگه را فراخوانی کنن. از طرفی D میتونه A و هم B رو فراخوانی کنه(recursive) و به متغییر های هر دو دسترسی داره. ولی B, C که میتونن همیدیگه رو فراخوانی کنن ب متغییر های local همدیگه دسترسی ندارن. Acsess to functions goes up and down and side to side but acsess to variables goes only up and down.

## Linear Algebra

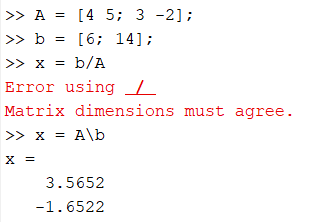
در مسائل علمی و مهندسی بسیار با این نوع معادلات مواجهه میشویم. معادلات به صورت Ax = b که A ماتریس دو بعدی و x و b ماتریس های برداری ستونی هستند این معادلات را تشکیل میدهند.



نکته: تو بحث تقسیم های معمولی بین \ و / تو متلب تفاوت هست:



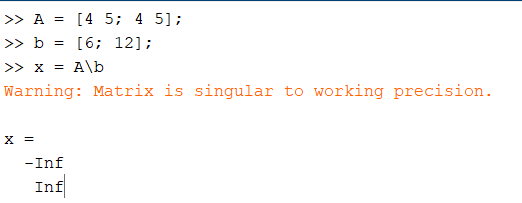
در واقع b تو مخرج هست بدین صورت که در واقع زیر اون اسلش قرار گرفته دیگه، در هر دو حالت.



این هم برای حالت ماتریسی. یا در واقع N معادله N مجهول. اینجا فوروارد اسلش کار نمیکنه. برای فوروارد اسلش عملگر ها باید تعداد ستون هاشون یکی باشه و برای بک اسلش باید تعداد ردیف هاشون یکی باشه.

اپراتور بک اسلش یکی از ویژگی ها و قدرت های متلب جهت حل چنین مسائلی است. بقیه زبان ها چنین قابلیتی را به این راحتی در دسترس ندارند.

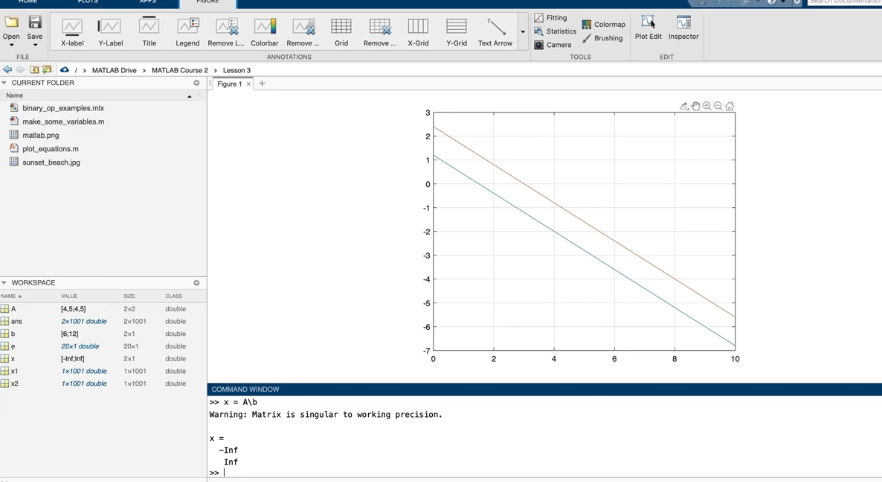
البته یک نکته ای که وجود داره اینه که اگر دستگاه معآدلات جواب نداشته باشه متلب میاد و یک اخطار میده بدین صورت:



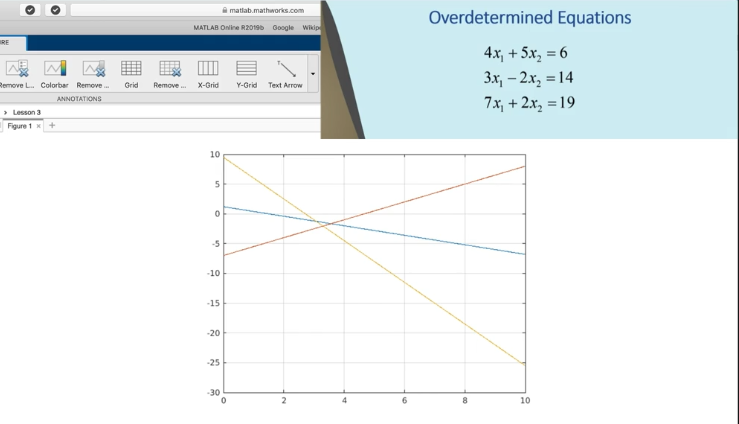
و جواب رو هم به صورت بینهایت میده.

البته نکته ای که وجود دارده اینه که این یک اخطار هست و نه یک error پس تو یک تابع باشه بقیه محاسبات انجام میشه.

خوب اینجا متلب داره میگه که عدد های حاصل برای جواب از بزرگترین عددی که متلب میشناسه بزرگتر شده پس به صورت بینهایت نشون میده. حالا این دو تا متغییر مون یکی شون مثبت بینهایت شده و یکی منفی بینهایت. چرا که این دو تا معادله رو اگه به صورت خط بکشیم شیب هر دو تا خط منفی هست.

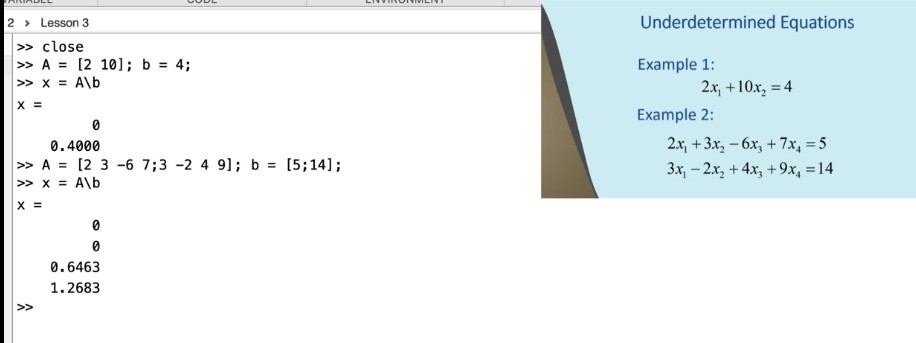


پس این دستگاه معادلات جواب نداره چرا که خطوط متناظر با این معادلات همدیگه رو قطع نمیکنند. حالا همون دستگاه دو معادله دو مجهول قبلی که جواب داشت. اگر یک معادله دیگه به اون دستگاه اضافه کنیم اون موقع هم ممکنه دیگه جواب های حاصله مون برای معادله سوم صدق نکنه. پس در این حالت هم دستگاه جواب نداره.



معمولا وقتی تعداد معادلات از تعداد مجهولات بیشتر میشه همچین اتفاقی پیش میاد و دیگه دستگاه جواب نداره. در چنین حالتی هم متلب باز جواب میده. این جواب در واقع مقادیری هست که با کمترین خطا بخواد هر سه تا معادله رو ستیسفای کنه. در واقع جوابی که کمترین میزان sum of squared error رو داشته باشه.

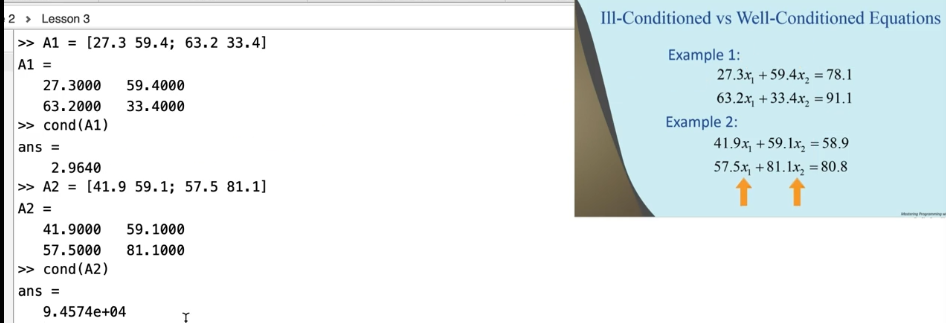
خوب حالا حالت بعدی وقتی هست که تعداد معادلات از تعداد مجهولات کمتر باشه. در چنین حالتی یا اصلا جواب وجود نداره یا بینهایت جواب وجود داره. برای حل چنین مسائلی کاری که متلب میکنه اینه که تا جایی که بتونه مجهولات بیشتری رو برابر صفر قرار میده و بعد بقیه رو حساب میکنه. به ازای هر مجهولی که از معدلات بیشتر داریم اون مجهول رو صفر میکنه.



خوب حالا بحث بعدی که باید در نظر گرفته بشه بحث پایداری جواب هاست. در مسائلی که در معادله A\*x = b رو داریم و مقادیر ماتریس b مقادیر اندازه گیری شده هستند. باید تاثیر تغییرات در مقادیر b روی جواب های x مورد بررسی قرار بگیره. اگر اختلافات اندک در مقادیر b باعث بشه مقادیر x تغییرات زیادی داشته باشند you have got a problem در چنین حالتی میگیم A مورد داره، A is ill conditioned، . تو متلب با استفاده از تابع cond که مخفف condition هست میتونیم ببینیم شرایط سیستم مون چجوریاست. ما میتونیم شرایط A رو توسط تابع cond(A) بدست بیاریم. به b و x هم کاری نداریم اصلا. هر ماتریسی یک cond ی داره. هرچه عدد حاصل از این تابع بیشتر شد یعنی شرایط خراب تر هست. حالا اون عددی که میده در واقع

The condition number for each set of equations is the maximum factor by which the percent error can increase from the inputs to the outputs.

Output همون مقادیر X هست و input هم مقادیر سمت راست تساوی ها یا همون b هستند. مقادیر ماتریس A معمولا مقادری اند که محاسبه شده اند و مقادیر ماتریس b مقادیری اند که اندازه گیری میشوند.



برای مثال اول اگر اندازه گیری ها با یک درصد خطا همراه باشه مقادیر محاسبه شده ماتریس b "حداکثر" 2.964 درصد خطا دارند. ولی در مثال دوم این مقدار خیلی بیشتره و شرایط به شدت وخیم هست. برای بهبود شرایط مثال دوم کاری که باید بکنیم اینه که یک معادله دیگه به سیستم اضافه کنیم. یعنی سه معادله، دو مجهول. در چنین شرایطی دیدیم که متلب میتونه جواب خوبی بدست بده. پس اگر عدد cond ماتریس بهتر شده باشه خیلی خوبه دیگه.

## Live Scripts

اول چند تا از تفاوت‌های اسکریپت و فانکشن رو یادآوری می‌کنیم.

یکی اینکه اسکریپت‌ها ورک اسپیس خاص خودشون رو ندارن و در لول کامند لاین کار می‌کنن. یکی دیگه هم اینکه اسکریپت ها نه اینپوت آرگیومنت دارند و نه آرگیومنت خروجی.

به خاطر این دوتا تفاوت اسکریپت ها صرفا برای ساخت برنامه‌های کوچک به کار میرن.

## Algorithmic Complexity

تو چنین فانکشنی که برای محاسبه nامین عدد سری فیبوناتچی ساخته شده برای هر خروجی که بخواد بده یه مقداری بیشتر از حد نیاز داره recursive call میکنه.

function f = fibo(n)

if n <= 2

f = 1;

else

f = fibo(n-2) + fibo(n-1);

end

end

ما میتونیم یک متغییر persistent تعریف کنیم که در کال های متوالی فانکشن مقدار اولیه اش رو فراموش نکنه بدین صورت:

function [f, cnt] = fibocnt(n)

persistent count; %must specify persistent

if isempty(count) %first time it is set to [] br MATLAB

count = 1; %so we set it to 1

else

count = count + 1; %subsequent times we increament by 1

end

if n <= 2

f = 1;

else

f = fibocnt(n-2) + fibocnt(n-1);

end

cnt = count; %persistent variables cant be an output argument

end

این به ما میگه که برای محاسبه هر عنصر دنیاله فیبوناتچی چندبار فانکشن فراخوانی شده. اگر [f c] = fibocnt(30) فراخوانی بشه میبینم که برای محاسبه سی امین عدد این دنباله، فانکشن 16625651 فراخوانی شده که نشان خیلی بدی هست. راه حل اینه که مثلا هر سری که فراخوانی میشه عددها رو ذخیره کنه و فقط یک فراخوانی recursive داشته باشه.

زمانی که فانکشن برای ران شدن طی میکنه به عوامل زیادی وابسته است که مهم ترین اونها یکی ورودی ها هستند و یکی هم الگوریتم مورد استفاده. بعضی الگوریتم ها هستند که وقتی سایز ورودی دو برابر بشه زمان طی شده برای حل مساله 4 برابر میشه. بعضی الگوریتم ها خطی هستند و با دوبرابر شدن سایز ورودی زمان حل هم دوبرابر میشه. اینها رو با تحلیل کد و الگوریتم میشه متوجه شد. این قضیه رو کلا بهش میگن time complexity که با محاسبه کردن اش میتونیم تخمین میزنیم زمان مورد نیاز برای حل مساله های بزرگتر چقدر هست.

الگوریتمی که با n برابر شدن سایز ورودی زمان حل اش n2‌میشه رو میگیم از مرتبه N2‌هست و اینجوری مینویسیم:‌ O(N2) . بعضی الگوریتم ها O(v=logN) هستند؛ مثل الگوریتمی که میخواد یک عدد رو در یک لیست مرتب شده پیدا کنه و هی لیست رو نصف میکنه تا به عدد مدنظر برسه.

یک نکته مهم اینه که زمانی که طول میکشه تا یک برنامه اجرا بشه به میزان پیچیدگی اون هم وابسته است. یک برنامه برپایه الگوریتم با O(N) معمولا ساده تر از الگوریتم با O(logN) هست. برای مسائل کوچکتر (ورودی های کمتر) شاید حتی برنامه با time complexity بالاتر ولی ساده تر سریعتر به جواب برسه.

بعضی الگوریتم ها، مثل الگوریتم‌های مرتب کردن، از مرتبه c N log(N) هستند و بعضی دیگه هم از مرتبه 2N که دیگه بدترین حالت هست. پس کلا شش نوع الگوریتم بحث شد. برای بحث ضرب ماتریس ها O(N3/2) هست.

## Efficiency & Speed up

در عمل اول ما یک کد ساده مینویسیم که کار رو انجام بده. بعد اگه برای ورودی های بزرگ مشکل داشتیم و ارزش داشت که مشکل رو حل کنیم میریم سراغ تغییر کد. اول باید مشخص کنیم کجای کد مشکل داره و زمان زیادی رو داره مصرف میکنه. میتونیم از ابزار های پروفایل کردن کد موجود در خود متلب استفاده کنیم (متلب فانکشن profiler داره که میزان زمان مصرف شده توسط هرتابع، زیر تابع و هر خط کد رو میده.).

البته ممکنه مشکل از کد نباشه و از الگوریتم باشه. می‌تونیم الگوریتم رو تغییر بدیم یا هم که الگوریتم موجود رو طوری اصلاح کنیم که در عمل بهینه تر رفتار کنه. در این زمینه سوالی که باید بپرسیم از خودمون اینه که آیا کد داره کار غیرضروری‌ای انجام می‌ده؟ اگه آره باید اون رو حذف کنیم. یک نکته مهم دیگه هم بحث preallocation هست که حتما باید انجام بدیم.

یک بحث مهم دیگه در مورد متلب اینه که چون بر اساس ماتریس هست، built-in فانکشن های زیادی در این زمینه داره که خیلی سریع تر و بهینه تر کارها رو انجام میدن. کلا هرجا از لوپ های صریح استفاده کردیم باید از خودمون بپرسیم راهی نداشت که از لوپ های ضمنی یا همون بیلت این فانکشن ها برای این کار استفاده کنیم؟

نکته: متلب وقتی میخواد یک کد رو اجرا کنه به زبان ماشین تبدیل میکنه پس کدی که اجرا میشه کدی نیست که نوشته شده. حالا نکته مهم اینجاست که وقتی پروفایلر متلب رو روشن کنیم اون بهینه سازی های کامپایلر (تبدیل کد متلب به زبان ماشین) غیرفعال میشن. مثلا بهینه سازی هایی که با vectorize کردن کد‌ (استفاده از بیلت این فانکشن ها به جای حلقه های صریح) صورت میگیره تو پروفایلر نشون نمیده.

یک نکته دیگه هم اینکه وقتی کد به زبان ماشین ترجمه شد اون ترجمه تو یک فایل ذخیره میشه که با ریستارت کردن متلب اون هم حذف میشه. پس وقتی یک کد رو زمان میگیریم زمان اولین اجرا بیشتر از زمان اجراهای بعدی هست. از دقیقه 25 به بعد رو ندیدم.

## Object Orinted Programming (FaraDars)

بعضی زبان ها مثل BASIC, MatLab, php یک دستور میتونه به عنوان یک واحد مستقل عمل کنه و کامپیوتر کار انجام بده. مثل دستور پرینت. ولی برخی زبان ها مثل C, Pascal حتما یک تابع باید باشه. مثل C که void main() داره به عنوان تابع. یک پله بالاتر زبان هایی هستند که حتما یک کلاس لازم دارند تا اجرا بشن. تابع یک ساختاری هست که حاوی چندتا دستور است و یک ورودی میگیره دستور ها رو برای اون ورودی ها انجام میده و خروجی میده. کلاس هم مجموعه ای از توابع که به صورت فعال با هم ارتباط دارند. مثل C++, C#, VB.Net .

کلاس یک نوع داده جدید هست که برای انجام مجموعه‌ای از کارهای مرتبط، ذخیره سازی و پردازش داده‌ها به صورت متمرکز پیاده سازی می‌شود. مثلا شناسنامه یک کلاس هست. شناسنامه یک شخص خاص نمونه‌ای یک آبجکت اون کلاس هست. یا مثلا ماشین در حالت کلی کلاس هست. کارهایی هم میتونه انجام بده. حالا مثلا یک ماشین خاص که میبینیم یک آبجکت هست.

کلاس پراپرتی یا خاصیت داره، مثلا کلاس Student ممکنه پراپرتی های ID, Name, Age داشته باشه. میتونه یک کارهایی هم انجام بده مثل Select\_Course(), Study() یک متد هستند یا رویه‌ها که کار انجام میدن.

غیر از constructor هر متد دیگه‌ای که تعریف بشه ورودی اولش باید آبجکت باشه، آبجکتی هست که داریم باهاش کار میکنیم.

مثلا این یک کلاس هست. ضرایب یک چند جمله ای بهش داده میشه و میتونه ریشه ها رو حساب کنه.

classdef Polynomial

properties

Coeffs

end

methods

function obj = Polynomial(c) %Constructor method

if ~exist('c', 'var')

c = 0;

end

obj.Coeffs = c;

end

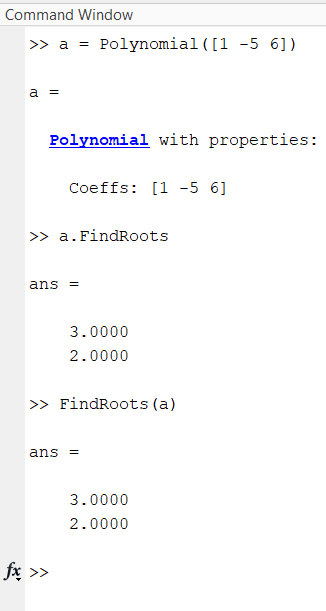
function r = FindRoots(obj) %Method for finding Polynomial roots

r = roots(obj.Coeffs);

end

end

end



الان اون ریشه‌هایی که میده رو میشه گفت پراپرتی های وابسته هستن. ضرایب که پراپرتی های مستقل بودند و ریشه ها به اون ضرایب وابسته ‌اند. حالا مثلا میخواهیم درجه این چند جمله‌ای رو به ما بده.هم به صورت متد تعریف میکنیم و هم به صورت خاصیت (نکته: چیزی که محاسبه کردنش مسلتزم انجام محاسبات طولانی هست رو نباید به صورت پراپرتی تعریف کنیم):

classdef Polynomial

properties

Coeffs

end

methods

function obj = Polynomial(c) %Constructor method

if ~exist('c', 'var')

c = 0;

end

obj.Coeffs = c;

end

function r = Roots(obj) %Method for finding Polynomial roots

r = roots(obj.Coeffs);

end

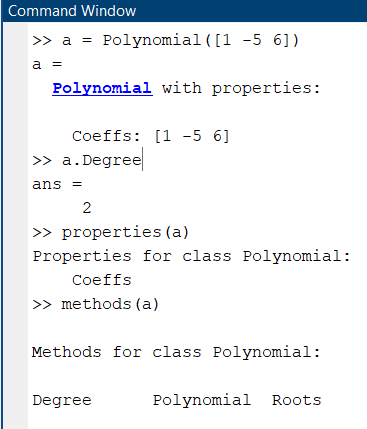
function n =Degree(obj) %Method for getting the degree of Polynomial

n = numel(obj.Coeffs)-1;

end

end

end



حالااگه بخواهیم به صورت خاصیت هم باشه اون درجه و ریشه‌ها:

classdef Polynomial

properties %Independent properties

Coeffs

end

properties(Dependent=true)

Degree

end

methods

function obj = Polynomial(c) %Constructor method

if ~exist('c', 'var')

c = 0;

end

obj.Coeffs = c;

end

function r = Roots(obj) %Method for finding Polynomial roots

r = roots(obj.Coeffs);

end

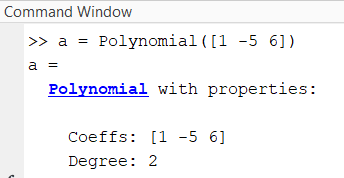
function n = get.Degree(obj)

n=numel(obj.Coeffs)-1;

end

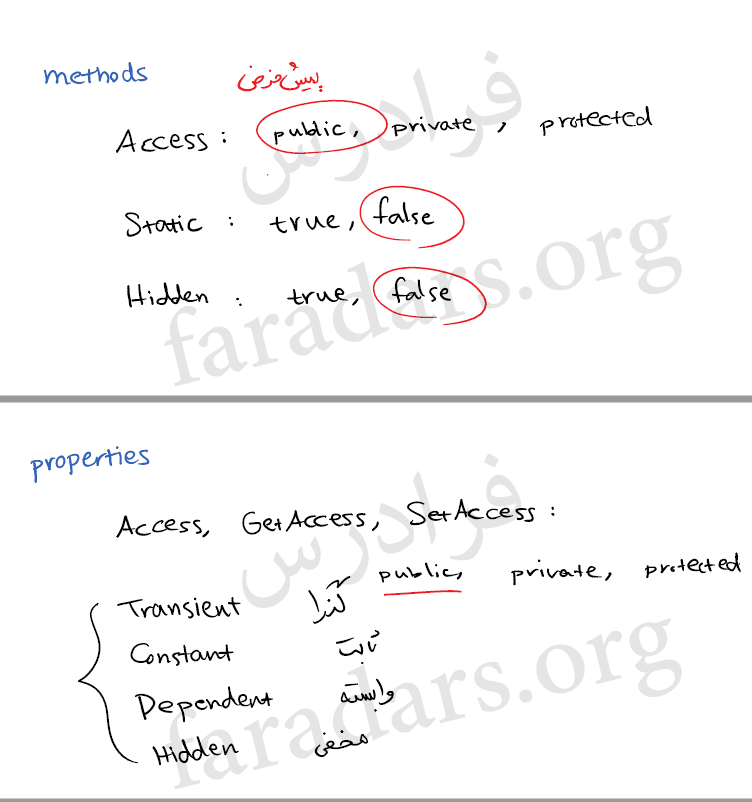
end

end



درجه رو در لحظه حساب میکنه و میده. نیاز به فراخوانی تابع نیست دیگه، خودش ران میشه. برای همین مثلا ریشه ها رو نمیاریم تو پراپرتی ها. مثلا یک چند جمله ای درجه بالا باشه محاسبه ریشه هاش طول میکشه.

خوب ما یک جا بحث Dependent بودن یا نبودن پراپرتی رو داشتیم. متدها و پراپرتی ها یکسری چیزهای دیگه هم دارند که میتونیم تعریف کنیم.

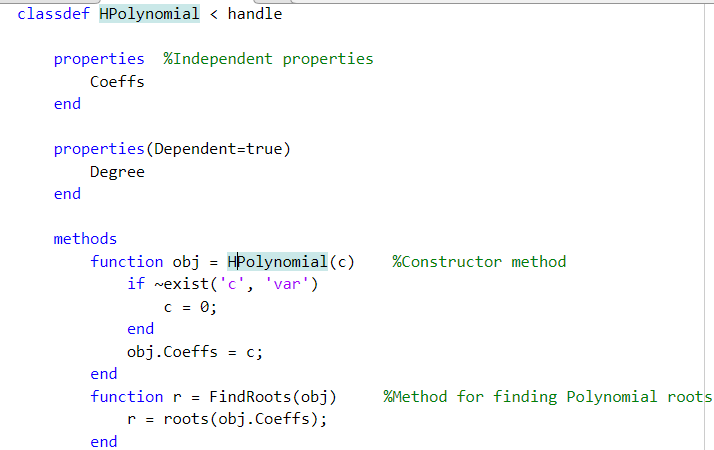


تو بحث متدها اگر متدی Protected باشه فقط تو اون کلاس و کلاس هایی که از اون کلاس ارث بری دارند قابل دسترسی است. هم‌چنین بحث استاتیک هم اینجوریه که اگر چیزی به کلاس وابسته باشه و به نمونه ها وابسته نباشه اون استاتیک هست. توابع استاتیک دیگه اون آبجکت به عنوان ورودی براشون لازم نیست.

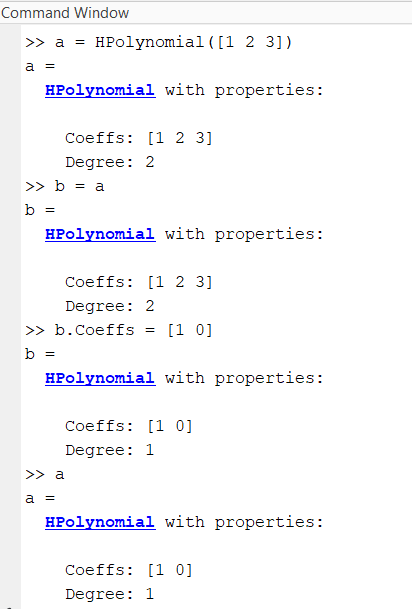
راجع به پراپرتی ها هم مثلا یک پراپرتی GetAccess اش میتونه پابلیک باشه ولی مثلا SetAccess اش پرایوت شده باشه (Read only باشه).

به طور کلی متغییرهایی که تعریف می‌شوند در دو دسته Value type‌ و Reference Type‌ قرار می‌گیرند. Value ها که مشخصه. مثلا a = 10‌ یا b = a . تو این دومی تغییر a روی b تاثیر نداره چون در مکان های مختلفی از حافظه هستند. از a یک کپی گرفته شده و جای b هست. حالا اگربا تغییر a مقدار b هم تغییر میکرد یعنی انگار یک وجود هستند، این میشه رفرنس تایپ.

متغییرهای رفرنس تایپ باید از کلاسی به نام handle ارث گرفته بشه، هندل یک سوپر کلاس هست و نمیشه ازش نمونه گرفت؛ فقط میشه ازش وراثت بگیری. تنها تفاوت تعریف چنان کلاسی هم تو تعریف خط اولش هست:



حالا نمونه های این کلاس رفرنس تایپ اند و یک وجود هستند در واقع، با تغییر یکی یکی دیگه هم تغییر میکنه



Value تایپ خودش کپی تهیه میکنه و میشه روی نمونه های مختلف کار کرد. کار باهاش راحت تره. مثلا یک متغییر رو میخواهیم دوبرابر کنیم (و یک فانکشن براش نوشتیم) و به جای همون متغییر بذاریم. اگر ولیو تایپ باشه باید یک خروجی هم داشته باشیم. مثلا a=f(a) ولی اگه رفرنس باشه دیگه خروجی نمیخواد؛ فقط مینویسیم f(a)