Notebook Documentation

Release 0.1

Yesh Yendamuri

Sep 11, 2019

Contents

Contents:

CHAPTER 1

What is this about

Sphinx. is an amazing software and I always wanted an excuse to use it and, after seeing it being used for the amazing Python guide. and also by Pydanny for pydanny event notes, I decided to go ahead and set it up for myself.

This is home for most of notes I make while moving from one book to another. You can reach me at Yesh and check out the code samples accompanying the notes on my github profile.

CHAPTER 2

Haskell

Contents:

2.1 Haskell

All the notes from my exploration of Haskell.

Right now, I am working through the following:

- Learn you a Haskell
- Real World Haskell
- Erik Meijer's amazing Lecture series

You can find the snippets of code and their mutations from this Github repository.

2.2 Haskell Links

2.2.1 Books

- Learn You a Haskell
- Real World Haskell
- *Learn Haskell Fast and Hard*
- Wiki Book

2.2.2 Links

- How to Learn Haskell

- Build a Compiler
- 20 Intermediate Question
- 99 Problems
- Enter Haskell

2.3 Learn You a Haskell

Contents:

2.3.1 Chapter 2

Lists

- Can't mix various types in lists
- An **If** function should always have an else block
- List can be joined together using **++**; but have to noted that when we do that haskell has to traverse the whole list on the left hand side.
- we can use **":"** which is the **cons** operator

```
5:[1,2,3] -- [5,1,2,3]
```

- We can use ***"!!"*** to get the index of a list. (Strings are basically lists!!)
- **Lists**
  - **head** takes a list and returns the first element.
  - **tail** takes a list and returns the all the element except the first one.
  - **last** takes a list and returns the last element.
  - **init** takes a list and returns everything except the last element.
  - **reverse** reverses the list
  - **take** takes an number and a list and returns that mn=any elemnts from the list. If index doesn't exist throws an error.
  - **drop** does what take does but gives all elements minus what the index given

```
drop 3 [1,2,3,4,5,6] -- [4,5,6]

drop 100 [1,2,3] -- []
```

- **null** gives wheather is list is null or not
- **minimum**, **maximum**, **product** and **sum** work on lists.
- **Ranges**
  - **cycle** takes a list and cycles it infinitely:
  - **repeat** takes an element and produces it infinitely

```
[1..10] -- [1,2,3,4,5,6,7,8,9,10]

[2,4..10] -- [2,4,6,8,10]

take 5 (cycle[1,2]) -- [1,2,1,2,1]
```

- **List Comprehension**
  - **The last condition is called the predicate. It is also called filtering**

    ```
    [x * 2 | x <- [1..10], x * 12] -- [12,14,16,18,20]
    ```

  - We have created a function where we take odd numbers. If they are greater than 10 print "BANG!" else "BOOM!"

    ```
    boomBangs xs = [if x <10 then "BOOM!" else "BANG!!" | x <- xs, odd x]
    ```

  - We can give two inputs:

    ```
    [x*y | x <- [2,5,10], y <- [8,10,11], x*y > 60] -- [55,80,100,110]
    ```

  - Strings are lists too:

    ```
    removeNonUpperCase st = [c | c <- st, c `elem` [`A`..`Z`]]
    ```

  - In GHCI (Reference) :

    ```
    let removeNonUpperCase st = [c | c <- st, c `elem` [`A`..`Z`]]
    ```

Tuples

- **Tuples**
  - A tuple can contain different types, unlike a list.
  - But a tuple of fixed size is its own type. Which means, we cannot have:

    ```
    [(1,2), (3,4,5), (6,7)] -- Error
    ```

  - Like lists **fst** gives first element, **snd** gives second element. Does not works on triples, 4-tuples and soon
  - **zip** is a super cool function:

    ```
    zip [1,2,3] [4,5,6,7] -- [(1,4), (2,5), (3,6)]

    zip [1..] ["apples" "bananas" "cherry"] -- [(1,"apple"),(2,"bananas
    ↪"),(3,"cherry")]
    ```

- Prob, which right triangle has integers for all sides to or smaller than 10 and has a perimeter of 24. This problem is solved in steps:

  ```
  let triangle = [(a,b,c)| c <- [1..10], b <- [1..10], c <- [1..10]

  let rightTriangle = [(a,b,c) | c<-[1..10], b<-[1..c], a <- [1..b], a^2 +␣
  ↪b^2 = c^2, a+b+c == 24]
  ```

2.3.2 Chapter 3

Types

- Haskell has ***type inference***. This just means that haskell can infer on its own without us explicitly telling it what the type of a function or a expression is.
- It is good practice to define the type of a function. Except for small fucntions.
- We can use **:t** in GHCI to find out the type:

```
> :t "Hello!" -- "Hello!" :: [Char]
```

- A string is a **[Char]** type because it is a list of characters:

```
> : "a" -- "a" :: Char
```

- Taking the removeNonUpperCase fn from Chapter 2:

```
removeNonUpperCase st = [c | c <- st, c `elem` [`A`..`Z`]]
```

Its type would be:

```
removeNonUpperCase :: [Char] -> [Char]
```

It means that it maps from a string to a string. * What happens when we have multiple parameters?:

```
addThree :: x y z = x + y + z
```

Type:

```
addThree :: Int -> Int -> Int -> Int
```

- **Common types are:**
  - **Int** used for whole numbers. It is bounded, which means it has a min and max.
  - **Integer** unbounded. **Int** is more efficient
  - **Float** and **Double**
  - **Bool**
  - **Char**

Generics in haskell

- :t head

```
head :: [a] -> a
```

which means that given any "a" it would output a single "a"

- ^^^ "a" is a ***type variable***

Typeclasses

- :t (==)

```
(==) :: (Eq a) => a -> a -> Bool
```

- Everything before => is called ***class constant***
- Complex topic. Extra Reading

2.3.3 Chapter 4

- Pattern matching is shown with the example in **factorial.hs**
- We should always try to have a catch-all pattern in order to avoid errors.
- Catch-all examples

```
first :: (a,b,c) -> a
first(x,_,_)    = x

second :: (a,b,c) -> b
first(_,y,_)    = y
```

- We can pattern match in list comprehensions. If a match fails, it ignores and moves to the next one:

```
let xs = [(1,3), (4,5), (7,8)]
[a+b | (a,b) <- xs]
```

- You can write patterns for breaking up something into names whilst still keeping a reference to the whole thing:

```
capital :: String -> String
capital "" = "Empty string"
capital all@(x:xs) = "The first letter of" ++ all ++ "is" ++ x
```

- Pipes | in functions which are akin to if statements:

```
bmiTell :: (RealFloat a) => a -> String
bmiTell weight height
    | weight / height ^ 2<= 18.5 = "You're underweight"
    | weight / height ^ 2<= 28 = "Normal"
    | otherwie = "You're are a whale."
```

- There is no = after the function name and its parameters before the first guard.
- Instead of repeating across the code base we can use **where**:

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= skinny = "You're underweight"
    | bmi <= normal = "Normal"
    | otherwie = "You're are a whale."
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
```

- The **where** does not pollute other namespaces. And pattern matching can be used in **where**
- **let <bindings> in <expression> are similar to where except**

  - they are expressions themselves.
  - they can be used by functions in local scope:

```
(let (a,b,c) = (1,2,3) in a+b+c) * 100 -- 600

[let square x = x * x in (square 5, square 3, square 2)] -- [(25,9,4)]
```

  - You can put **let** in list comprehensions:

```
calcBmis :: (RealFLoat a) => [(a,a)] -> [a]
calcBmis xs = [bmi | (w,h) <- xs, let bmi = w/h ^ 2, bmi > 25.0]
```

- We can do pattern matching for **case expressions**:

```
describeList :: [a] -> String
describeList xs = "The list is" ++ case xs of [] -> "empty"
                                              [x] -> "singleton list"
                                              [xs] -> lomger list
```

2.3.4 Chapter 5

Recursion

- In haskell, recursion is important because we do computations by declaring what something is instead if declaring how you get it.
- In recursive functions we should always identify the edge case and try to count down to that.
- All the examples can be found at recursive fns

2.3.5 Chapter 6

Curried Functions

- In haskell, every function officially takes only one parameter. The way we get to give several parametes to function is by having *curried function*

```
> max 4 5
> (max 4) 5

max :: (Ord a) => a-> a -> a
max :: (Ord a) => a-> (a -> a)
```

- The space in between is *funciton application*
- You can write *partially applied* functions. By doing so, we are creating functions on the fly.

```
multiThree :: (Num a) => a-> a-> a
multiThree x y z = x * y * z

> let multiTwoWithNine = multiThree 9
> multiTwoWithNine 2 3
> 54
```

2.4 Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

CHAPTER 4

Elm

Contents:

4.1 Elm

```
isNegative n = n < 0

isNegative 4 -- true

-- Anon fns
\n -> n < 0

over9000 powerLevel = if powerLevel > 9000 then "It's over 9000!!!" else "meh"
```

4.1.1 Lists

```
names = [ "Alice", "Bob", "Chuck" ]

List.isEmpty names

List.length names

List.reverse names

double n = n * 2

nums = [2, 3, 5]

List.map double nums
```

4.1.2 Tuples

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

CHAPTER 6

Startup School (2013)

Contents:

6.1 Introduction

This is my attempt at trying to write notes for the Startup School conference.

The current speaker list(2013) for is:

- Chase Adam Founder, Watsi
- Nate Blecharczyk Founder, Airbnb
- Ron Conway Partner, SV Angel
- Chris Dixon Partner, Andreessen Horowitz; Founder, Hunch, SiteAdvisor
- Jack Dorsey Founder, Square, Twitter
- Diane Greene Founder, VMWare
- Phil Libin Founder, Evernote, CoreStreet
- Dan Siroker Founder, Optimizely
- Balaji Srinivasan Founder, Counsyl; Lecturer, Stanford
- Mark Zuckerberg Founder, Facebook

6.2 Phil Libin (Founder, Evernote)

- Great co founders are very important. Should always cultivate a group.
- Don't make friends with people who you wouldn't want to start a company with.
- Its all about networking.

- Being your own boss sucks if you don't build anything of value.
- Building a product you love is crucial.
- Build something where you are the target audience.
- Build a company you want to keep.
- Don't complicate your work and hierarchy structure.
- In the early days, be clever only about your idea.
- It gets better but it also gets harder.

6.3 Dan Siroker (Founder, Optimizely)

- Value of the feedback loop is underrated.
- Focus on one or two things which will make you unique.
- Hire a person better than the mean of the entire company
- Pick an enemy to drive the team. (Adobe)

6.4 Ron Cosway

- Unusual entrepreneurs always catch the eye. Passion is want makes the difference.
- Motto: Is the product the best it can be. And do the users love it.
- Find the investor who can add most value.
- Product focus is the most important.
- Growth of the company doesn't lie.

6.5 Chase Adam (Watsi)

This had to be one of the most inspiring talks in this year's startup school. Watching his talk once its is up would be the best way to experience it. Nonetheless, these are few of the many things he talked about.

- Fuck Non-profits
- Problem with being a Non-Profit. When you fundraise, no one says NO.
- Find something to work on that you care about, more than yourself.

I'll post a link to the talk in its entirety once its up.

6.6 Jack Dorsey (Founder, Twitter, Square)

Very unique talk where he spoke about the books and music that inspires him everyday.

Books:

- The Art Spirit - Robert Henri

- The Score Takes Care Of Itself - Bill Walsh

Quotes: * Shared common sense of purpose for a team, company is crucial. * Good planning is good luck. * He showed us a small exampe of a do's and don't list that he maintains and look at everyday.

Song:

He played the following song in its entirety saying how he goes back to this song through out a day to give him focus and to keep himself going.

You know what, it is a pretty inspiring song.

- Angoisse - Serge Gainsbourg

6.7 Mark Zuckerberg (Founder, Facebook)

This was a fireside chat with Paul Graham. Mark Zuckerberg did have a few things to say amidst a sea of camera phones taking his pictures.

- Always build stuff you want.
- Spoke of a few hacks he did back in harvard. Basically, he pushed for people to be creative and just build.
- Lockdown - When ever a competitor or a clone build a feature they felt threathened them. They would basically shut themselves indoors until they fixed the problem.
- There might be 100 things you want to do. Pick one or two things you deem important.

6.8 Nate Blecharczyk (Founder, AirBnB)

- You will fail more than you succeed.
- Choosing the right partners to start off on your dream is very important.
- Try to make every experience addictive.
- Know your user base. A small base of users who like you a lot is better than lots of user who like you just a little.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

CHAPTER 8

ViM Everything

Contents:

8.1 Why

I realized I learn something new from ViM everyday. I just felt I needed a central place where I can keep a track of everything ViM

8.2 Zen Of ViM

- Less is Good
- Being Lazy is Crucial

8.3 Practical Vim Notes

Contents:

8.3.1 Introduction

Vim is an awesome piece of software. Many people love to hate it or worse, use emacs(;-)).

Practical Vim is one of those amazing technical books which are page-turners and you always want to keep reading ahead to find out what new thing you can learn to improve your overall vim skills. But, the book is packed in with so much information that I feel compelled to start making notes.

The whole book is primarily focuses on so called *tips* and I shall be making my notes along those lines.

UPDATE: My plan initially had been to keep writing notes as I went through Practical Vim. What is that they say about great plans and good intentions.

I have decided to make this my one stop place to put everything I learn about ViM.

8.3.2 The Dot command

This command which is the **"."** on the keyboard basically repeats the last change. But, from the time I came across this in the book I have quickly come to realize this is for too powerful, because it does exacty what it advertises.

The **"."** records the actions after we enter the *insert* mode.

Another cool thing in this tip is the usage of **A**. What this basically does is, moves the cursor to the end of the line and switch into *insert* mode.

The above two commands in combination can save one a lot of keystrokes.

Another important way to handle repeated commands with the use of the *dot* command is by using f{char}

For example:

```
var foo = "method("+argument1+","+argument2+")";
```

Using *f{char}* which here would be f+ would immediately jump to the first instance of the + and after doing any changes we want using *;* would help us find the next instance of + and the dot command would help repeat our changes.

When these two are used in tandem, can reduce the repetetive work a developer needs to do.

8.4 Indices and tables

- genindex
- modindex
- search

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

CHAPTER 10

Working with Unix Processes

By Jesse Storimer

The book uses ruby to explore UNIX processes. Jesse also gives the corresponding man page the command maps to.

10.1 PID

- On any UNIX system. Just type *man <command_name>*
- Each process has a **PID**:

```
puts Process.pid
```

- maps to *getpid(2)*
- A global (albeit implicit) way to retreive the current pid is using *$$*

10.2 PPID

- Each process has a Parent process **PPID**:

```
puts Process.ppid
```

*maps to *getppid(2)*

10.3 File Descriptor

- In UNIX everything is treated as a file(resource). Any time a resource is opened within a process it is assigned a **file descriptor** number.

- File descriptor are *NOT* shared between unrelated processes. They live and die with the process they are bound to:

```
passwd = File.open('/etc/passwd')
puts passwd.fileno                  => 3
```

- In ruby, open resources are represented by IO class
- The fileno is the way kernel keeps track of the resource:

```
passwd = File.open('/etc/passwd')
puts passwd.fileno                  => 3

hosts = File.open('/etc/hosts')
puts hosts.fileno                   => 4

passwd.close

null = File.open('/dev/null')
puts null.fileno                    => 3
```

- **Key points from the above example**
  - File descriptors take the lowest unused value.
  - Once a descriptor is closed, the number becomes available again.
- *What* happened to 0,1 and 2 file descriptors?

```
puts STDIN.fileno       => 0
puts STDOUT.fileno      => 1
puts STDERR.fileno      => 2
```

- **Ruby IO maps to *open(2)*, *close(2)*, *read(2)*, *write(2)*,** *pipe(2)*, *fsync(2)*, *stat(2)* etc

10.4 Resource Limit

- Limits are imposed by the kernel:

```
Process.getrlimit(:NOFILE) => [2560, big_number]
```

- **getrlimit return a 2-element array**
  - First number is the soft limit
  - Second number is hard limit
- The soft limit can be bumped by:

```
Process.setrlimit(:NOFILE, 4096)
```

To set soft limit to the hard limit:

```
Process.setrlimit(:NOFILE, Process.getrlimit(:NOFILE)[1])
```

- If you exceed the soft limit, an exception will be raised (Errno::EMFILE)
- **Real world examples:**
  - If you want to handle thousands of sumultaneous network connections

    - Restrain system resources when executing third part code
- Maps to *getrlimit(2)* and *setrlimit(2)*

10.5 Environment of the Process

- A little ruby heavy chapter
- Every process inherits environment variable from its parent. Its set per process and global to each process.
- Ruby ENV used hash-style accessor, but doesn't implement all the **Hash** API:

```
puts ENV['EDITOR']
puts ENV.has_key?('PATH')
```

- Access to special Array called **ARGV**
- You can change the name of the process:

```
puts $PROGRAM_NAME

10.downto(1) do |num|
    $PROGRAM_NAME = "Process: #{num}"
    puts $PROGRAM_NAME
end
```

- Somewhat maps to *setenv(2)* and *getenv(2)*. Also, *environ(2)*

CHAPTER 11

Indices and tables

- genindex
- modindex
- search