“Right now, I’m trying hard to become better at programming. I remember that you mentioned two books: Pragmatic Programming (Hunt & Thomas) and Refactoring (Martin Fowler). Is there another book of which you think a starting programmer must have?”

This is loaded question, so I will try and answer it as best I can. It may seem like I am circling around a direct answer to your question, but I need to place it in the wider context.

Computer science is the study of how electronic engineering, discrete mathematics and linguistics converge to make computers (and other programmable devices) work. That is to say how computers work, how programs run.

Software engineering on the other hand is the application of computer programming to solving real world problems in an efficient and financially viable way. Software engineering tends to be a bit removed from the low level details of how computers actually work and is as much a social process as it is a technical one. There is a strong emphasis on efficiency from a business perspective, the microeconomics of writing code if you will -> finding ways to make software applications more useful, more reliable, and less expensive to make.

You have probably already realized this already as you are able to program in python without a formal education in computer science. That said there is considerable overlap between the two disciplines. Many of the stated goals of software engineering are achieved through the application of computer science to a problem.

(The emergence of the git version control software is a typical example. To save time and money developers collaborate and share code through open source software, pooling resources to solve a common problem experienced by many organizations. Managing all the changes to this software required a distributed version control system. Git was the answer to this problem. Git incorporates many fundamental data structures and algorithms from computer science including hashes, graphs and trees.)

Having some foundational knowledge in computer science can help you be a better software engineer. You do not need a bachelor’s degree in the subject, almost all the relevant knowledge can be acquired through self study. The biggest problem you will face will be the demands on your time. You will not have time to learn everything and therefore you must be able to filter out what is important. Aim to acquire effective knowledge before acquiring transcendent knowledge on any given subject.

Computer science knowledge that is important:

Operating Systems

Networks
Computer Architecture
Data Structures and Algorithms
Encryption

Operating systems:

An understanding of the file system
An understanding of processes
An understanding of users and permissions
An understanding of virtual memory
An understanding of context switching and scheduling
An understanding of user space vs kernel space

A good theoretical understanding of these concepts can be obtained by reading any standard university textbook. Two very popular ones are:

“ Operating Systems Concepts” Silberschatz, Galvin and Gagne
“Modern Operating Systems” by Andrew S. Tanenbaum

Earlier pdf editions of these books can be obtained from:

http://freecomputerbooks.com

On the practical side the most immediately useful book I can recommend on this subject is:

“A practical guide to Linux commands, editors and shell programming” by Mark G. Sobell

This is really good because it is very hands-on, providing just enough theory and a lot of practice that will be useful to you in your day job. It is a lot more focused on how to *use* linux/unix than how linux/unix actually works. Here is the free online version of the second edition:

http://www.aem.umn.edu/~aem3100/spring2013/Prentice_Hall_A_Practical_Guide_to_Linux_Commands_Editors_and_Shell_Programming_2nd.pdf

For a more thorough overview of how to use and configure Linux I would also recommend:

“The Unix and Linux system administration handbook, 4th edition” by Nemith, Snyder Hein and Walley.

Here is book’s website:

http://www.admin.com/

A lot of the narratives in this book are very descriptive but there is also a hell of lot that you will not need unless you are a system administrator.

For a more rigorous understanding of the internals of Linux and Unix I would recommend:

“How Linux Works, 2nd edition” by Brian Ward

https://www.nostarch.com/howlinuxworks2

No point reading this book unless you are already familiar with using Linux and/or Unix. Really useful stuff is how network configuration works and the contents of the /etc directory. Does not do what it says on the cover however. It only explains how parts of Linux work, and those parts do not include the core operating system.

For an understanding of how Linux and/or Unix really work read the following

“Advanced Programming in the Unix Environment, 3rd edition” by Stevens and Rago

http://www.apuebook.com/cover3e.html

This book is very applied, and gives very descriptive narratives next to excerpts from the operating system source code itself. It is the best book on the subject if you really want to understand the internals. It does however require that you are able to program in C. It is also very long, so reading it should be hobbyist pursuit, not a career priority.

There is another s a book in which the reader “creates” a mini Unix-like operating system themselves:

“Operating Systems: design and implementation” by Andrew S. Tanenbaum.

http://index-of.es/eBooks/Operating%20Systems%20Design%20&%20Implementation%203rd%20Edition(1).pdf

I should mention that Andrew S. Tanenbaum lectures at the VU in Amsterdam and is a world authority on Operating Systems and networks.

Networking:

An understanding of computer networks is also “must-have” knowledge for a software engineer, whether you are setting up a production environment, or just trying to figure out if an application is running on your own work station by testing a certain port. Try and obtain the following knowledge:

An understating of the TCP/IP protocol stack.
Well known ports
Public and private IP addresses
IPv4 vs IPv6
An understanding of DNS and routing
An understanding of SSL/TLS, know what tunneling is and how to use ssh
Certificate signing, Certification chains , CAs and PKI
Http, REST

The best practical introduction to networks that I can think of is:

Sams Teach yourself TCP/IP in 24 hours by Joe Casad

This book is not too dense on theory, but gives you enough to get an idea of what is going on, and then introduces you to common networking command line tools, which are an essential part of a developer's toolkit, such as:

ping, netstat, curl, ifconfig, hosts (formerly nslookup), telnet, traceroute

For a theoretical understanding of networks once again any university textbook will do. Popular options include:

Computer networking, a top-down approach by Kurose and Ross
Computer Networks by Andrew S. Tanenbaum

For in depth study on networking (not essential, only if you are curious):

The illustrated network by Goralski
TCP/IP illustrated, Volume 1 and Volume 2 by Fall, Stevens and Right
TCP/IP Guide by Kozierok

Also a good idea to learn about internet governance:

ICANN, IANA and RIPE NCC (the latter is located in central Amsterdam)
IETF and RFCs (formal protocol definitions)

Checkout their websites or just use wikipedia.

Data structures and algorithms:

Data structures and algorithms is one of those foundational subjects that you find everywhere. Main subject areas include:

Basics of lists, queues, stacks, graphs and trees
Basics of hashes and hash tables
Basics of searching and sorting

Real world examples of their use:

Network routing: uses shortest path graph algorithms
Calling functions in a running program: uses stacks
Storing data of indefinite size: uses linked lists
File system on your Mac: uses trees, both in the path names and inodes.
Listing names alphabetically: uses sorting algorithms
Using git bisect to pinpoint a bug in your code: binary search

Data structures and algorithms are ubiquitous in software engineering, even though one is often unaware of their presence as they will be hidden inside an api or library that you are using. That said I think learning this subject is well worth the investment.

There are innumerable books on this subject, so I will only name a few:

Data structures and Algorithms in Python by Goodrich, Tamassia and Goldwasser

This book should be very accessible to you because the example code is in python.

A book which I find is very rigorous without being overwhelming is:

Algorithms by Sedgewick and Wayne

This is the standard text book at Princeton. The exercises really force you to think. The only problem is that it takes a while to get through, so difficult to do part-time.

For in depth theoretical study the bible of algorithms is:

Introduction to Algorithms by Corman, Leiserson, Rivest and Stein

It is neither practical nor easy to digest.

Encryption:

Encryption is a huge topic so I will limit my discussion to what is important:

Understand the difference between symmetric cryptography and asymmetric cryptography. This is very important for securing computer network traffic using SSL/TLS. Once you have grasped the concepts it is worth reading about how the SSL/TLS handshake works, as it uses asymmetric encryption to exchange exchange a keys that are then used for symmetric encryption.

Understand how asymmetric cryptography can be used to transport encrypted information or digitally sign unencrypted information, in both cases using key-pairs, one of which is designated private and the other one public.

Figure out how to use “trusted” keys to make your ssh logins easier, such that they do not require password prompts.

Understand what a one way hash is and why it is better to store the hash of a password rather than the password itself.

Familiarize yourself with popular hash algorithms (i.e. MD5, SHA1). You do not need to know how they work, but more their general properties and which ones to use in a given situation.

A good exercise is to find out why git uses SHA-1. Hint: to minimize the probability of collisions.

Sams Teach yourself TCP/IP by Joe Casad, which I mentioned in the networking section covers the difference between systemic and asymmetric encryption

Understanding Cryptography by Paar Peizi and Preneel is book which is both rigorous and accessible if you want to deepen your knowledge.

Another really good source of information on this subject are the books by Bruce Schneier such as:

“Secrets and Lies: Digital Security in a networked world” -> easy to read
“Cryptography Engineering” -> medium difficulty, with the technical reader in mind
“Applied Cryptography” -> really difficult in depth

Bruce Schneier is a world authority on computer and network security and written many books on the subject both for engineers and non-technical people - well worth having a look at.

General programming knowledge that is useful:

Understand different types of programming models:

Imperative (C programming)
Object-oriented (java, javascript)
Functional (Haskell)
Hybrid models such as object-functional (scala)
(python can actually be used in an imperative, objected-oriented and object-functional way)

Type systems: know the difference between a strong and a weak type system, and the difference between and static and a dynamic type system. Wikipedia has some good disambiguation on this subject.

Important object oriented concepts:

Abstraction , encapsulation, inheritance, separation of concerns

A hands on guide on object orientation is:

Head First Object Oriented Analysis and Design by McLaughlin, Police and West

A very long and theoretical, but still effective book on object orientation:

A Touch of Class by Bertrand Meyer

The problem with this book is that it uses Eiffel for the example code. You have to learn another language just to read the book!

Important functional programming concepts:

Deferred or lazy evaluation, currying, memorization, tail call optimization.

A hands-on guide to these concepts is:

Functional thinking: paradigm over syntax by Neal Ford

A deeper dive into functional programming is to study Haskell by reading:

Learn you a Haskell for great good by Miran Lipovaca

Available for free online: http://learnyouahaskell.com/chapters

You will realize that certain problems are better tackled in an object oriented way and other problems are better tackled in a functional way, for example mapping is easier than looping. Object-functional languages such as python and scala let you choose depending on your needs.

Scope: understand what closure is.

Read the wikipedia article: https://en.wikipedia.org/wiki/Closure_(computer_programming)

Read: You Don’t Know Js: scope and closures by Kyle Simpson

Memory management: Know the difference between explicit (manual) memory management and implicit (automatic) management, also known as garbage collection.

There is no better way to understand this then by learning to program in C. C demands explicit memory management. Try and write a C program which dereferences but does not deallocate it’s data inside an infinite loop. It should crash after a while and bring your computer to a standstill. Then try de-allocating the memory using the free() function before dereferencing it in each iteration of the loop. This time the program should run forever and not run out of memory.

Most of the higher level languages that we program in such as python are actually created using C. Operating systems and network protocol implementations also tend to be written in C. Although it is unlikely that you will ever end up using C in your job, it is a good language to learn for the purpose of deepening your knowledge of how computers work.

Higher level languages have garbage collectors that handle the freeing of memory for you when your program is running, so you do not have to think about it while you are coding. For example if you assign a python variable to one object and then assign it to another, the first object is no longer needed and the memory it occupies must be freed. You never have to think about this when programming in python because the python garbage collector tracks the number of variables pointing to a given object. When this number drops to zero the object is “removed” from memory.

Runtimes and execution models:

Strive to obtain a high level, abstract understanding of what these things are:

Know the difference between compiling a program and running it
Know the difference between a compiled and interpreted programming language.
Know the difference between pre-execution compilation and just-in-time compilation.
Get an idea of the process of compiling then running a program (pre-processing => compilation => assembly => linking => loading)
Know how the stack and the heap is used to hold the program’s memory and state.

To learn C programming:

Sams Teach your C in one hour a day, by Jones, Aitken and Miller
The C Programming Language by Kerrighan and Ritchie (Also known as “K&R”)
Understanding and using C pointers by Richard Reece

To learn about computer systems:

Computer Systems: a Programmers perspective by Bryant and O’Halleron

I cannot understate how useful the Bryant and O’Halleron book is for a complete understanding of how a computer works and how programs are compiled and run. Some of the older editions of this book are available for free online

Concurrency:

Know what threads are
Know what shared memory is (hint: heap memory)
Know that each thread has it’s own execution stack (memory on the stack is local to that thread and not available to other threads)
Know what a race condition is and how one can use locking to prevent them
Know what locking is, and what re-entrant locks are
Know what semaphores are
Know what deadlock conditions and live-locks are

Know that the mathematical foundations of concurrent programs amount to governing access to shared mutable state. These mathematical “rules” are the same across different programming languages.
Know how to avoid concurrency problems altogether (hint: actors, functional programming)

I wont recommend any concurrency books right now as this really depends on which programming language you are using. That said bare in mind that the underlying principles of concurrency are the same in all programming languages.

Ok enough computer science, now onto the software engineering side of things:

1) Try and have a main programming language and a scripting language. In your case, you main programming language can be used as a scripting language. Consider learning another language. Javascript would be very useful for front-end programming at work. C would be very useful for understanding how computers work. Go combines elements of python and C. Java is also a good one with a lot of job opportunities. Scala is in vogue at the moment but IMHO overrated.

2) Get good at using the command line. Learn to use shell utilities such as find, grep, xargs, cat, cut, sed, awk. Use git from the command line. Chain shell commands together using pipes. (eg find . -type f -name ‘*.py’ | xargs grep “def”). Consider installing “bropages”, which is easier to read than the man pages, and has practical examples.

3) Learn unix/linux command line network tools such as ping, telnet, netstat, host, ifconfig, ssh, scp, curl

4) Learn a command line based text editor such as vim, emacs or even sublime text. Generally the harder it is to learn the more you can do with it. Try vimtutor.

5) Learn to use your IDE effectively. Learn the keyboard shortcuts. Intellij Idea has a plugin called KeyPromoter which prompts you with the keyboard shortcut every-time you use the mouse. I do not know if it will work with PyCharm.

6) Master Test Driven Development. Know the difference between a unit test, an integration test and an automated acceptance test. Know the pyramid or ice berg model => lots of fast running unit tests close to the code, compared to a few slow running end-to-end tests that test the application as a whole. Get into the red-green-refactor flow. Understand how continuous integration and test-automation minimize the number of bugs in software. Understand that TDD is as much about design as it is about testing. Testable code tends to be well designed code.

   TDD books:

   Test Driven Development by Kent Beck (Introduction, a bit out of date)
   Test Driven Development with Python by Percival (Useful to you)

Growing object oriented software guided by tests by Freeman and Pryce (This book shows how test actually influence the design of the code, and how to write code that is highly readable and highly testable. This book is more for someone who is already used to TDD, who wants to do it better.)

7) Learn design patterns. Most important are the “Gang of four”. Some of these patterns are now redundant, as programming languages have evolved to incorporate them into the grammar of the respective languages themselves. Hint: design patterns in current programming languages often portend changes in future programming languages. In so far as web programming and relational databases are concerned, learn the “Patterns of Enterprise Application Architecture”, PoEAA for short, named after the book by Fowler. Many of these patterns are also out of date now but many of them are still used, particularly the object-relational patterns. There are also the IEP (Enterprise Integration Patterns), though in practice you will hardly ever use these. Your time would be better spent learning about REST and micro-services than IEP => IEP tends to be used on monolithic systems which are being disrupted by the emergence of micro-services.

Design patterns books:

Head first design patterns by Freeman and Bates is very accessible

Design Patterns: Elements of Reusable Object-oriented software by Gamma, Helm et al :this is the original design patterns book, not that readable though)

Patterns of Enterprise Application Architecture by Martin Fowler: although many of these patterns are out of date the patterns on object-relational mapping are invaluable

Read Rest in Practice by Ian Robinson

Read http://martinfowler.com/articles/microservices.html

8) Learn the dependency injection principle. Know the difference between tight coupling and loose coupling. Know why loose coupling is preferable. Know that loose coupling is achieved through dependency injection. Know the difference between dependency injection and inversion of control. Hint: Dependency injection achieves loose coupling between classes where as inversion of control achieves loose coupling between modules. Understand that DI makes classes easier to test, especially with mocks.

Read Clean Code by Robert C. Martin

9) Learn the Law of Demeter or “tell, don’t ask”. Understand why getters are evil, and why it is important to encapsulate behavior as well as state. Understand how

Demeter makes code more readable and testable by chunking code more uniformly across your classes, and making behavior easier to mock in unit tests.

Law of Demeter is discussed in the Pragmatic Programmer

10) Learn about Domain Driven Design (DDD). Learn how to separate code that contains business logic from code that uses framework specific classes. Understand why this makes code easier to test. Keep business logic in a separate module away from persistence layers and web layers. Use inversion of control and dependency injection to achieve this. Realize that Domain Driven Design, Test Driven Development and Dependency Injection form the “holy trinity” of modern enterprise software development, with each one complementing the other.

Domain Driven Design by Eric Evans is the original DDD book, a bit long winded

Implementing Domain Driven Design by Vince Vaughn -> a bit more hands on than the Evans book

Here is a free e-book called Domain Driven Design Quickly -> shorter than the Evans book, requires infoq account:

http://www.infoq.com/minibooks/domain-driven-design-quickly

It may not be necessary to read an entire book on DDD as long as you know what the following are:

Services
Repositories
Entities
Value Objects
Bounded Context

11) Understand the principles of continuous delivery. See software deployments as being a bit like a factory production line, from your workstation to the continuous integration server, to QA, to staging, and then finally to production. Try and automate as many of the steps as possible, without compromising quality. Understand release management, tagging and versioning.

Continuous Delivery by Jez Humble

12) Understand the difference between virtualization and containerization. Try using at least one of the tools and libraries that allow for the rapid creation of VMs and containers (such as Vagrant and Docker). Understand the advantages of scripting environments, when it comes to horizontal scaling and load-balancing. Understand the advantages of scripting environments when it comes to creating

a deployment pipeline where the workstation, QA, Staging and production are almost exactly the same.

13) Develop an affinity with agile development processes. Find out who W. Edwards Demming was and find out a little about Japanese lean manufacturing. Find out how the software industry has tried to use these ideas to solve the Software Productivity Crisis. Important points are focusing on quality (refactoring, TDD) at the micro-level to improve productivity at the macro-level (rapid, reliable software releases). Identifying bottlenecks and waste in a process by taking measurements (such as how long does it take to get from a git push to a production release?). Understand that measurements are subject to statistical variation and may not be accurate in the short term. This is particularly the case for user story points during a sprint. It is necessary to aggregate measurements to get useful information (i.e. "what is the average story point velocity of the team over 8 sprints?" is a better question to ask than "how long did Rosa take to do task X?"). Look at the similarities between a Demming cycle and a unit test. Try and see everything thing as repeated cycles of varying duration (from running a unit test, to daily stand-ups, to sprints). Each iteration is a chance to improve on the one that came before it. Unlike building a bridge or developing an oil platform, software is malleable, you can change it after you have made it, and therefore lends itself to short iterative cycles rather than phased (waterfall) development models used in other engineering disciplines. Realize that scrum is just a business process and does nothing to address the technical side of agile software development. Many companies abuse scrum and use it as a way of micro-managing their staff, focusing exclusively on how fast features are delivered while neglecting the accretion of technical debt, resulting in misplaced blame and failed projects. Scrum has spawned a cottage industry with certifications and consultants peddling scrum as the secret sauce which will save your project. It is useful but it is not a panacea. It is outright harmful when misused in the way I have described above. Fowler refers to this is as "flaccid scrum".

The problem:

https://en.wikipedia.org/wiki/Software_crisis

The solution:

Demming:

https://www.youtube.com/watch?v=GHvnIm9UEoQ
https://www.youtube.com/watch?v=mKFGj8sK5R8
https://www.youtube.com/watch?v=6WeTaLRb-Bs

Agile Manifesto:

http://agilemanifesto.org/

Books:

Agile Estimating and planning by Mike Cohn
The Art of agile development by Shore and Warden

When the solution becomes the problem:

http://martinfowler.com/bliki/FlaccidScrum.html

What is technical debt anyway:
https://www.youtube.com/watch?v=pqeJFYwnkjE
http://www.laputan.org/mud/

14) Get active in the online community:

Create stackoverflow.com account if you don't already have one. Don't be afraid to ask questions through stackoverflow. There are no stupid questions, only stupid answers.

Create a github account if you don't already have one. Create a toolbox on github that you can take from one project or company to another.Clone a project that you are interested in

15) Enjoy yourself, don't exhaust yourself :)