SYLLABUS

100 DAYS OF CODE
COMPLETE
PROFESSIONAL
PYTHON BOOTCAMP

BEGINNER PYTHON

- Variables in Python
- String Manipulation
- Input and Print Functions
- Variable Naming Rules
- Mathematical Operations in Python
- DataTypes
- Converting types
- Conditionals IF/ELIF/ELSE
- Logical Operators
- Randomisation
- Error Handling
- Functions
- For Loops
- Code blocks and Indentation
- While Loops
- Flowchart Programming
- Positional and Keyword Arguments
- Python Dictionaries and Lists
- Nested Collections
- Returning Functions
- Return vs. Print
- Doc Strings vs. Comments
- Scope and Local/Global Variables
- Debugging Techniques

INTERMEDIATE PYTHON

- Local Development Environment Setup
- PyCharm Tips and Tricks
- Python Object Oriented Programming
- Creating Classes in Python
- Using External Python Modules/Import
- Getting / Setting Attributes
- Python Methods
- Class Initialisers
- Module Aliasing
- Optional, Required and Default Parameters
- Event Listenters
- Python Instances and State
- Python Turtle
- Game Development with Python and OOP
- Python Inheritance
- Python Slice Function
- File I/O Reading and Writing to Local Files
- File Directories
- Reading and Writing to CSV
- Introduction to the Pandas Framework
- List Comprehensions
- Dictionary Comprehensions

APP BREWERY

www.appbrewery.co

SYLLABUS

100 DAYS OF CODE
COMPLETE
PROFESSIONAL
PYTHON BOOTCAMP

- Packing and Unpacking Functions in Python
- Creating Desktop GUI Apps with Tkinter
- Strongly Dynamic Typing
- Error Handling and Exceptions
- Try / Except/ Raise
- Working with JSONs
- Local Persistence
- Sending Email with Python and SMTP
- Working with date and time
- Hosting Python Code Online with PythonAnywhere

INTERMEDIATE +

- APIs
- Making HTTP Requests with the Requests module
- Sending Parameters with the Request
- APIs with Authentication
- Sending SMS with Python
- Web Scraping with Beautiful Soup
- Browser Automation with Selenium Web Driver
- Automating Tinder
- Automating Twitter
- Automating LinkedIn
- Automating Instagram
- Web Development with Flask
- Command Line
- Python Decorators
- Templating with Jinja 2
- WTForms

ADVANCED PYTHON

- Build Your Own REST API with Python
- Build Your Own Blog
- Databases with SQLite
- Dataframe Inspection
- Data Cleaning
- Sorting Values in Dataframes
- Arithmatic Operations with Pandas
- Creating Pivot Tables
- Chaining Functions
- Smoothing Time Series Data
- Creating Line Charts with Matplotlib
- Using Jupyter Notebook
- HTML Markdown
- Creating Scatterplots with Matplotlib
- Relational Database Schemas
- Descriptive Statistics
- Creating Bar Charts, Pie Charts, Donut Charts, Box Plots with Plotly
- Creating NumPy NDArrays

APP BREWERY

www.appbrewery.co

SYLLABUS

100 DAYS OF CODE
COMPLETE
PROFESSIONAL
PYTHON BOOTCAMP

- Array Slicing and Subsetting
- Matrix Multiplication
- Bitwise and Operators in Pandas
- Creating Bubble Charts with Seaborn
- Running Regressions with Scikit-Learn
- Non-Parametric Regression
- Students T-Tests and Histograms with Scikit-Learn
- Multi-Variable Regression
- Log Transformations
- Residuals Analysis

PROFESSIONAL PORTFOLIO BUILDING INDEPENDENT ASSIGNMENTS

- Text to Morse Code Converter
- Portfolio Website
- Tic Tac Toe Game
- Image Watermarking App
- Typing Speed Test
- Breakout Game
- Cafe and Wifi Website
- Todo List Website
- Disappearing Text Writing App
- Image Color Palette Generator
- Custom Web Scraper
- Automating the Google Dinosaur Game
- Space Invaders Game
- Custom API Driven Website
- An Online Shop
- Custom Browser Automation
- Analyse and Visualise the Space Race
- Analyse Deaths Involving the Police in the US
- Predict Earnings using Multivariable Regression

APP BREWERY

www.appbrewery.co

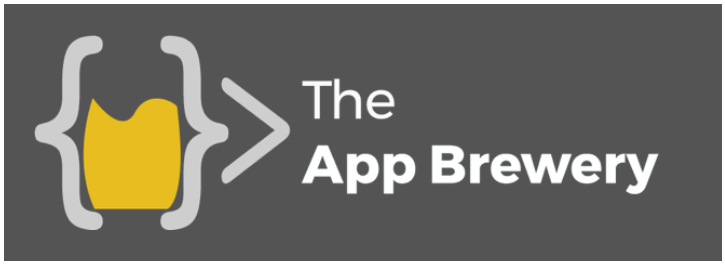
www.appbrewery.co

12 RULES TO

LEARN TO CODE

Everything we know from teaching
150,000 students

BY DR. ANGELA YU



12 Rules to Learn to Code

Dr. Angela Yu
iOS, WatchOS and Web Developer
Lead Instructor at The App Brewery

www.appbrewery.co

Contents

www.appbrewery.co

1

Trick Your Brain with the 20min Rule

Learning to code is a bit like going to the gym. Even if you max out and spent a whole weekend at the gym, you will not see a visible difference in your body. The more regularly you learn to code, the more likely it is that you'll start seeing your ripped coding muscles. (The irony is not lost on me).

But the problem is where do you find the time? Between working your full-time job, spending time with your family and life admin, when are you supposed to sit down and practice this "daily coding"?

While I was working as a doctor, I spent about 12 hours at the hospital, 1-hour commuting and approximately 2 hours on general life-sustaining stuff, such as eating. So that left me with only 9 hours remaining in my day. Theoretically, 2 hours could be allocated to coding practice and 7 hours on sleep. But there is nothing more difficult than trying to convince your work-saturated brain to sit down and learn when you could be watching Game of Thrones with a tub of ice-cream.

But then I found a trick.

As humans, we have a lot of inertia. This can be bad for us - I'm looking at you, "24" box set. However, we can also turn it to our advantage. I found

that once I got started coding and making things, I got so absorbed into the project, that I no longer cared about TV, food or sleep. There were quite a few weekends when I coded until sunrise.

So how do we take advantage of this inertia? First, you must understand that task-switching is very difficult. It requires a lot of motivation. If as soon as you get home, you slump on the sofa and switch on the TV, you've already lost that evening. This is because the amount of motivation required to task-switch and do something not driven by evolution like eating or sleeping is a Herculean task.

This is why the moment you enter the door and change to a new environment is the most crucial moment. If at this moment, you tell yourself that you are just going to do 20 minutes of coding practice, you will most likely succeed and use your own inertia to end up learning for an hour or more. No brain will perceive a 20-minute task as a lot of effort and you end up tricking your brain to take advantage of your evening.

The next step is to develop a habit. Research suggests that in order to develop a new habit, you have to carry out the task daily for a month. I've used this next trick for loads of different things, from exercising to coding, it invariably works like a charm. To preface this trick, I want you to imagine a wall with five paintings hanging on it, four of which are perfectly aligned, perfectly horizontal, but one is crooked. Now really imagine it, is there a part of you that wants to fix it?

Now let's imagine a monthly calendar with boxes representing individual days. If you nurtured that new habit on a particular day, then you make a line through that day. If you continued your streak the next day then you extend that line and so on and so forth. There is something about not breaking a continuous line that motivates most people to continue to

develop a habit. As strange as it sounds, there are many times when I would have given up, but compelled to continue because of a long, continuous line.

2

Code for a Purpose

When I first started learning how to code, there were countless times when I picked it up then gave up, again and again. This is a common story amongst self-taught coders. Looking back, after teaching so many students, I finally realise what's going on. A lot of beginners start learning to code by picking an arbitrary language and follow along with a bunch of tutorials. Copying code, line by line, sometimes writing code to work out prime numbers, other times to find all the even numbers. But you know what? I can find prime numbers a lot faster by Googling for it and picking out even numbers is really not all that interesting.

Here's the truth. If you are learning to code for the sake of learning to code, it'll be pretty difficult for you to get good at it. Skills that require a lot of time to hone, like programming, will eat into your pool of internal motivation. Something from within that makes you forget to eat and sleep. I can honestly say that coding on my own projects is one of the most enjoyable things I do. It combines logical thinking with creativity, and at the end, you will have made something. In most cases, something that the world has never seen. Something that could make your life easier or more enjoyable. Something that could make loads of people's lives easier and more enjoyable. It's like making a crazy-beautiful custom motorbike in your garage, without needing the garage or spending a cent on the components.

This is what motivates most people. The creating part. The making part. So I urge you to start learning to code by following a tutorial that makes

something, anything. Of course, it's unlikely that at the beginning you'll be able to code up Clash of Clans or League of Legends. But you'll be able to make something interesting. It could be a dice game or a flash-card app. But as long at the end of the tutorial, you'll have made something you can use and play with, then you'll be far more motivated to code to the end.

During all our courses, we always tell our students to think up of a simple app that they want to make. Something that uses the skills that they've learnt during the course but will also stretch them a little because they have to find out how to include some new functionality.

We had a student who went on to make an app that wakes them up a minute earlier every day to ease the transition to an earlier waking time. There's a student who made a custom slideshow app as a mother's day present. Someone else made an app that is a timer for making perfect steaks based on its weight and thickness.

There are no limits on your imagination. It will be difficult when you start working on your own app because there are no step-by-step instructions, but it will also bring about the biggest improvement in your coding ability.

3

There is No "Perfect Language to Learn"

Whenever I do large talks, there will always be one person who asks me "which programming language should I start learning first"? There is this common perception that somewhere out there lies a perfect language for beginner programmers. Some argue it's Python, some say it's Swift.

But I say they're all wrong.

A programming language is simply a tool. It is no different from any other tool in your hardware box. If you want to hammer a nail, you should be using a hammer. If you want to fix your water pipes, you'll probably need a spanner. Yes, it's possible to hammer in a nail using the side of the spanner and the same programming language can be used to solve different types of problems. The carpenter will tell you that his favourite tool is a hammer and the plumber will say it's the spanner, but it still doesn't make it the "best tool to fix things".

A web developer will tell you that JavaScript is the best language to learn for a beginner. A statistician will advise you that you'll be best served with the R programming language. But at the end of the day, all that matters is what you are trying to do with your tool. If you want to make iOS apps, then learn Swift. If you want to make websites, you'll need JavaScript. But the good news is the core programming concepts: loops, conditionals, functions, etc. they're all the same. The difference is mostly syntactical. In English, we have werewolves, in German they have Werwölfe. It's still the same

shirt-ripping mammal that comes out during a full moon, it's just spelt differently.

Printing to the console in Swift:

```
print("Hello Werewolves")
```

Printing to the console in Java:

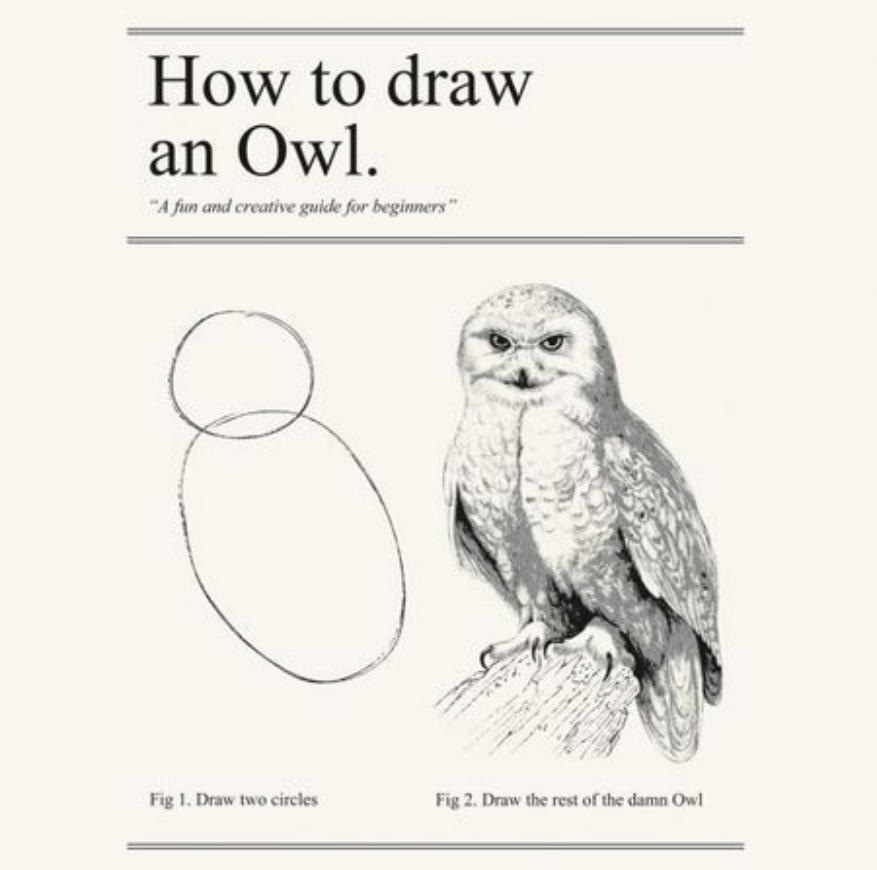
```
println("Hello Werwölfe")
```

So, decide on the task that you are trying to accomplish, then pick the best tool for that task.

4

Understand What You're Writing

I have an issue with the way that most programming tutorials are written. There are far too many tutorials where you see the "this is how you draw an owl" phenomenon.



It’s almost as if the programmer had good intentions and started off by showing you how to do everything, step-by-step. But then, at some point, he realises that he has embarked on a Sisyphean task and gives up. I’ve seen tutorials where the author starts off with an excruciating level of detail then mid-way reverts to “now you simply set up a cloud database”. Bearing in mind that this is a tutorial aimed at beginners!

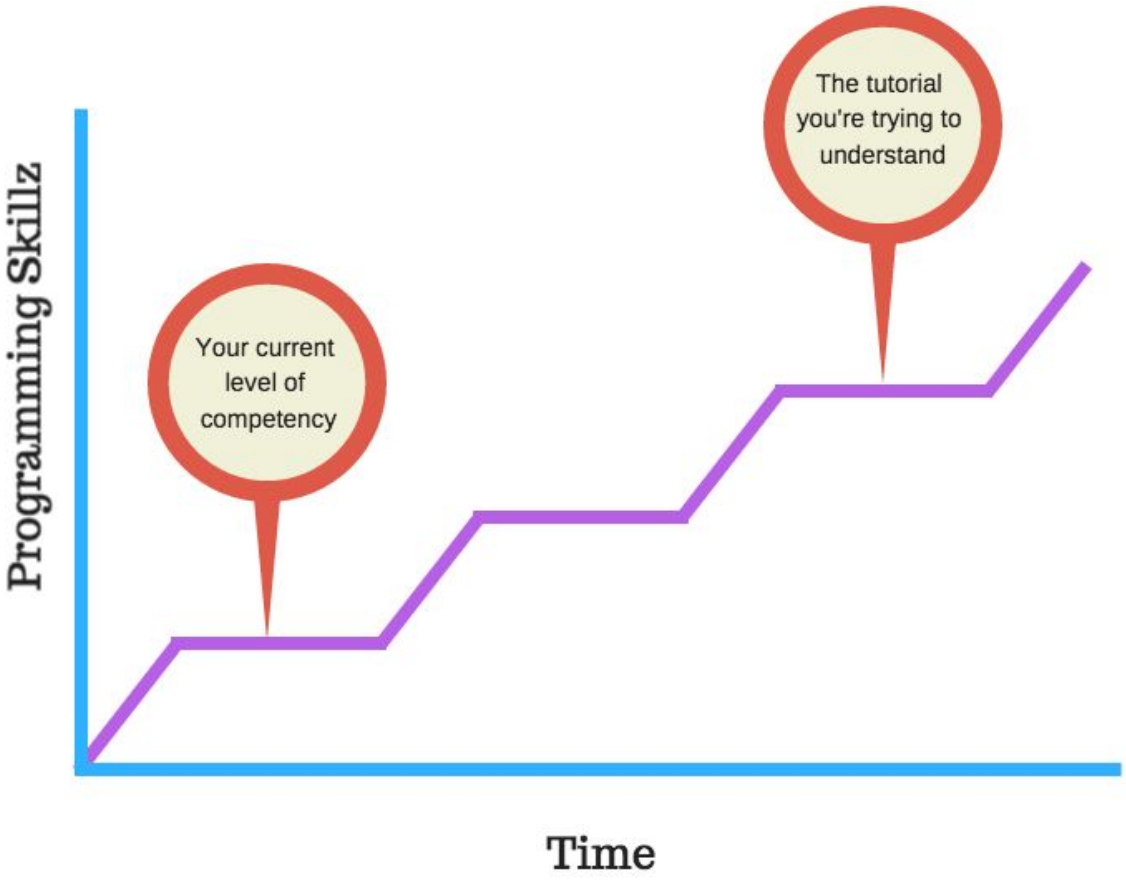
This leads to a number of problems. The most common problem is a student who just copies the code in the tutorial and has no clue what any of it does. Why did he add that extra line after parsing the JSON? Why is he making this dictionary differently from the last one?

It’s very easy to get knees deep in one of these types of tutorials because it promises to teach you how to build “Flappy Bird” or “Candy Crush”. But two-thirds of the way in, none of the things you’re typing makes sense and you start seeing red all over the screen. Bugs. Loads of them. Why? No idea. Nothing runs. The last 3 hours were spent copying code and you learnt nothing other than maybe that coding sucks.

Don’t get into this trap. If you see a tutorial that has jumps from beginner to advanced after line 3 or uses the word “simply” too liberally or doesn’t explain any of their code, then stop. Leave that tutorial.

There’s plenty of fish in the sea.

Other times, the author does try to explain what they’re doing. But you still don’t understand a thing that they’re saying, then you’re in an advanced tutorial that won’t improve your programming. It can be tempting to build grand things, especially when the blog is promising that anyone can make it. But if you can’t work out what’s going on, you’re be better served by building a better foundation.

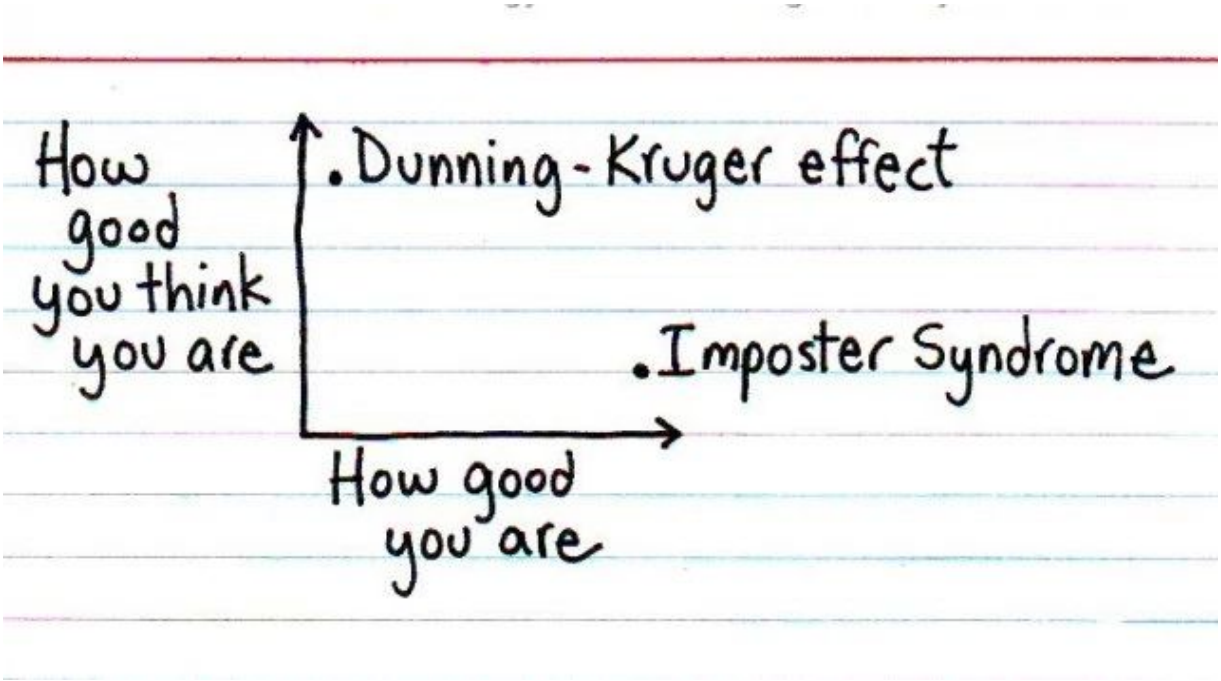


The key to learning to code is all about ramping. You want to be stretched over and over again and for knowledge to be built on previous knowledge. If that ramp is too steep, you'll get lost. If that ramp is too shallow, you'll get bored. The right gradient is different for everyone. That's why we encourage students to use the speed change functionality liberally on our tutorials. This way, you can listen at double speed if you're comfortable with the concepts and slow down to half speed if it's something unfamiliar and you need time to understand and absorb.

5

It's Ok to Not Know

Software engineers are purportedly the profession that has the largest population of Imposter Syndrome sufferers. Imposter Syndrome is a psychological phenomenon where people feel like frauds and massively underestimate their own skills and abilities.



Programmers tend to be self-critical and constantly feel that everyone else is better at programming than they. If you've ever felt this way, you're not alone, as studies show that a massive 70 percent of people have imposter syndrome.

I recently saw a post on the Q&A site Quora where somebody asked: "Would I get fired at Google (or another big tech firm if I got caught using StackOverflow as a reference?"

He got a bunch of really great answers from engineers working at Google, Amazon and other major tech companies. Anybody who has worked as a software engineer will tell you that not looking at references is far more frowned upon. In fact, I challenge you to find a single Google programmer who has not used Stack Overflow. (If you're not familiar, StackOverflow is a collaborative Q&A site for programmers).

A lot of new programmers are afraid that by checking references and asking people for help they will out themselves as a fraud who doesn't know to programme. Nobody can hold all the relevant information in their head. For example, this is the name of an iOS method:

```
- (id)initWithBitmapDataPlanes:(unsigned char **)planes pixelsWide:(NSInteger)width
pixelsHigh:(NSInteger)height bitsPerSample:(NSInteger)bps
samplesPerPixel:(NSInteger)spp hasAlpha:(BOOL)alpha isPlanar:(BOOL)isPlanar
colorSpaceName:(NSString *)colorSpaceName
bitmapFormat:(NSBitmapFormat)bitmapFormat bytesPerRow:(NSInteger)rowBytes
bitsPerPixel:(NSInteger)pixelBits;
```

It's almost 400 characters!

In iOS programming, there are over 800 classes, 9000 methods and growing. In web development, there's a new framework every week. No one will expect you to be able to remember the code. This is the precise reason why we are programmers, we can get the computer to do the boring stuff

for us. For example, the code for recording sound is only a short search away, why would you need to memorise it?

The skill that most employers look for when recruiting is the ability to think. Knowledge is valued in a world where information is hard to come by. In the 1800s, only the rich had access to good books and good teachers. Now, everyone has all the information they had and more at the tap of a mouse. Information is losing value, the ability to think is the stock to buy. So don't be afraid to search, to ask on StackOverflow or to find resources to help you solve your issues. The best programmers do it.

The skill you need to hone is in asking good questions and understanding the answer. There is no point copy-pasting code from a StackOverflow answer if you have no clue how it works. Because StackOverflow works on a reputation system, it's in their interest to be as clear as possible in their answer in order to be marked as correct and collect upvotes.

In most cases, it doesn't make sense to start searching StackOverflow whenever you get stuck. The first option should always be trying to figure it out yourself. So your program doesn't do what you expected it to, but before I typed the last 3 lines of code, it was working fine. So let's figure out what in those last 3 lines broke my app?

If you really can't figure it out, start with Google. Search for your query or if you have a bug paste the error codes and the error message. Chances are that as a beginner, your programming woes will be very common and somebody might have even taken the time to write a clear and concise tutorial to help you understand your bug. As you grow more skilled in programming, the problems you'll encounter get more and more obscure, but hopefully, if you followed the other 11 rules, you will also be a more

capable programmer and figure it out yourself or know exactly where to get help.

The other reason why you should start with Google is that StackOverflow's search algorithm organises questions and answers by recency and not popularity. A lot of the problems you will encounter while starting out will have been asked and answered years ago but still massively popular.

So ask wisely and you will reap the benefits from the community. One day when you yourself become a code master, you'll be giving back to that same community and helping the next generation of programmers.

6

Be a Copycat

At the beginning of my coding journey, I thought the way to learn to code was to read a whole bunch of books. I bought books on C++, C#, Java and loads more. You name it, I had it. But they didn't do very much other than making me confused.

I read. I highlighted. I forgot. I fell asleep.

Books are good as references. If you want to dive deep into delegates and protocols, read the chapter on that. But if you want to learn, **make something**.

But what do you make?

Lacking in ideas? Be a copycat. Make your own notepad, make your own MSPaint, make your own piano. If you're into games, make minesweeper, make Tetris, make Flappy Bird. Not only will they be sort-of useful, but they'll also be the perfect opportunity for you to figure out how to do things and get experience in finding help. Something that is brand new to the world like holographic smartphone projections, no one will be able to help you with. By making copycat apps or programs, you'll be treading in the path that many have walked before you. This way you maximise the chances that someone will be able to offer you help and advice when you get stuck.

7

Be Accountable

Be accountable to someone. Show your work.

The biggest problem with online coding courses is the lack of accountability. No doubt there are loads of great Massive Open Online Courses (MOOCs), such as Coursera, Udacity, Udemy, Skillshare. But what are the consequences of not doing your homework or missing a month's worth of lectures? Nothing. Nobody cares.

Let's face it, internal motivation is not strong in any of us. We can always find a reason why we *deserve* to "Netflix and chill". I can't even count how many online courses I've signed up to and subsequently not listened to a single lecture or completed a single piece of coursework.

You need accountability and commitment to learning. Think back to your university days, would have bothered to finish that essay at 3 AM if nothing depended on it? Would you have gone to any of the lectures if you didn't care about passing or failing?

This is why we try to introduce accountability into our courses. We've realised that matching up students with a buddy helps. Someone else who is a beginner, at the same level as you who sometimes helps you and other times needs your help. Sometimes, as people's learning rates diverge or if you're paired up with a lazy bugger, you can swap it up and get a new buddy. Because this system is entirely voluntary, there a degree of self-selection for people who work well in teams and are motivated by

others. Just as you're more likely to go to the gym if you sign up with your partner, you're more likely to learn if you have a coding buddy.

So if you're not on our course then find your own. There's plenty of Facebook groups dedicated to those who are learning to code. There's an entire subreddit (r/learnprogramming) dedicated to this, I'm sure you'll find like-minded people somewhere online or offline.

The next thing I'm going to tell you will be controversial. We believe that people don't value things that don't have a value. This is the reason why Coursera is taking down a large number of their free courses. They saw that millions of people were signing up for it but no one was taking any of the classes let alone complete any of the projects. It was actually detrimental to students' learning to offer a free course. We all have a degree of hoarding tendencies and it's very easy to signup for a bunch of stuff that the future-you can suffer through. There's always tomorrow, she says.

So if you are driven more by external motivation than internal, try to use a little bit of financial motivation to drive your learning. Think about how much a life skill is worth to you and put your money where your intentions are. See if you're engaging with the course content more with or without the financial commitment. There are plenty of places where you can pay something affordable to motivate yourself to start a regular learning habit.

The final part of this rule is to try and find ways of getting assessed. Ok, so getting assessed is right up there with death and taxes in terms of how much people enjoy it. But when learning anything, it's always important to get feedback. You will get an objective assessment of your current skill level, instead of feeling like an imposter or brimming with false confidence. Coursera has a system where the students mark each other's' work. At the App Brewery, we use Github education to test your code and look for bugs

and problems with your code. But if you’re on a coding course that doesn't have a system like this, then it’ll be worth your while to find a code mentor who can review your code and give you feedback. Only what’s measured can be improved.

8

Keep Learning

Being a good programmer is a bit like being Madonna.

Don't run out and buy your cone-shaped bras just yet. What I mean is programming will keep evolving. In order to stay relevant, you have to keep re-inventing yourself.

There's always new trends, new technologies and new languages. Great programmers relish in learning new things, even if it means they become a beginner again.

The world will keep moving, if you stay in one place, you'll eventually be left behind. I know programmers who never learnt anything else apart from Fortran. I know Objective-C programmers who can't persuade themselves to make the leap and learn Swift, even though Apple is telling developers that Objective-C will be phased out. We all know that Apple never makes threats that they don't carry out, just look at the optical drive (and soon the headphone jack?).

Don't be the optical drive. Or rather, don't be the laptop that's still trying to play CDs. If your needs change, learn to use a new tool. Keep learning, stay relevant.

Are you a web developer who always wanted to get into mobile development? Pick a platform and learn iOS or Android. Are you a front-end developer who is tempted by the full-stack? Pick up web development with

Node. If you already understand the core programming concepts, picking up a few more languages will be a lot easier than starting from scratch.

"Learn x in y minutes" is a great resource for existing programmers to learn new programming languages. Check out their resources here: learnxinyminutes.com

Learn X in Y minutes

Share this page

Where X=swift

Get the code: learnswift.swift

Swift is a programming language for iOS and OS X development created by Apple. Designed to coexist with Objective-C and to be more resilient against erroneous code, Swift was introduced in 2014 at Apple's developer conference WWDC. It is built with the LLVM compiler included in Xcode 6+.

The official Swift Programming Language book from Apple is now available via iBooks.

See also Apple's getting started guide, which has a complete tutorial on Swift.

```
// import a module
import UIKit

//
// MARK: Basics
//

// Xcode supports landmarks to annotate your code and lists them in
the jump bar
// MARK: Section mark
// MARK: - Section mark with a separator line
// TODO: Do something soon
// FIXME: Fix this code
```

9

Play Foosball

When you see Hollywood movies about programmers, they're usually sat in front of a laptop, mashing the keyboard like they're in some sort of high-stakes "smash the mole" game.

When you see **real** programmers working. They tend to look like this:

Yep, that's right. No typing. Just staring. A lot of staring.

In a company, people tend to complain that the programmers are always playing foosball or doing something else that doesn't look like work. People might not be able to tell, but they are in fact working.

When you see them enjoying their foosball game, laughing and joking, they're probably suffering inside. For there's a bug, there's always a bug. Or there's something mysterious about their code that they can't work out. Maybe the code is working perfectly, but unexpectedly (programmers don't like anything unexpected by the way). Like if they just typed out a thousand lines in one go, and *unexpectedly* there are no errors.

Other people might not understand, but in these situations, it's almost always worth stepping away from your code and giving it some time and distance.

Do you have a bug in your code that you can’t work out? Sleep on it, play foosball, go for a walk. In 9 out of 10 cases, the solution will become apparent. In the remaining 1 out of 10 cases, you’re just screwed.

This may sound unintuitive, but my advice is always to code less, think more. Once the poorly thought-out code is written and brought into the world, you’ll inevitably have to go back and comb through your code, line-by-line, refactoring and deleting things. This is always a painful experience.

So remember, the easiest code to get rid of is the code that was never written.

10

Get a Mentor

When I was learning French, I came across a method that resulted in the greatest leap in my speaking abilities. That was having language exchanges over Skype. I would pair up with a native French speaker who wanted to learn English. We would spend half an hour speaking French and half an hour speaking English. We would both dedicate an hour each week to improving the language that we were trying to learn.

While we were having a conversation in French, he would correct my pronunciation or grammar and suggest the ways that I could construct my sentences to sound more native.

Pair programming is an agile software development technique that's based on very similar principles. For example, a learner and a mentor would sit down at the same workstation and work on a problem. The learner is in charge of writing code and the mentor reviews the code line-by-line as they are written.

It can be uncomfortable at first because it's a bit embarrassing making mistakes and having them pointed out to you. But if you have a mentor who is a good teacher then they will offer you decades of accumulated wisdom that can lead to massive improvements in your own ability, all within a few hours.

You'll get to tap into someone who's had the time to hone their skills, find efficient ways of doing things and show you how they program and approach problems.

Good mentors don't solve your problems, rather they practice the Socratic method of asking good questions that get you to think for yourself. If you ask me how to write a networking call, of course, I can simply type it all out and get you to copy it. But that doesn't help you. Instead, if you show me how you approach the problem and I show you how I approach the problem then you can learn so much more than just following a recipe.

The next time you encounter a different problem, you can apply the same approach and start solving it yourself. Always remember that information is cheap. A century ago, if I wanted to learn about the causes of disease, I probably had to be an aristocrat, or chop wood and carry water for a master and become their apprentice. Nowadays I can search Google and get my answer in a few seconds.

So don't get hung up on information. Learn to think instead. How to approach a problem. How to break down the problem. How to frame the problem. These skills will take you much further than simple memorisation and regurgitation.

But where do you find a mentor?

There are programming related Meetups happening in almost every city in the world. Go to www.meetup.com and find one related to a language you're trying to learn. Attend the meetups, get to know people. Exchange your expertise for their expertise. Maybe someone needs an accountant, maybe someone needs legal advice. Exchange your time for their time. Don't say to someone, "will you be my mentor?". No one wants to throw away their free

time for some stranger. Instead, offer your help in return for their help and you'll be successful in finding a mentor 95% of the time.

11

Get into the Habit of Chunking

So you have an awesome app idea. But it's way-way-way too complicated for your current skill level. What do you do? You join the Chunking Express.

Nope, I'm not talking about the art house movie. I'm talking about breaking down your programming problem.

Let's say that you're trying to make a robot that can butter toast. (If anyone is working on one of these I'd happily fund your Kickstarter!) The robot doesn't know anything about toast or butter or knives. Believe it or not, it actually takes pretty sophisticated circuitry in our brains to be able to achieve something as simple as buttering a slice of toast. (This is probably why I can't seem to do it without coffee).

So creating a robot that does all of that autonomously is really complicated and difficult. But as we're good programmers, we can do some chunking and break down the problem.

The robot doesn't really need to know what is toast and what is butter, we're not making Skynet here, so let's just stick to the practical things. There are three things we need the robot to do:

1. Pick up and arrange the piece of toast in the ideal buttering position.
2. Pick up a serving of butter.

3. Place butter on toast with decent coverage (this is the part I find most difficult).

Next, you break each module down even further. In the process, you can think about alternate ways of solving the problem. For example, does the robot need to "spread" the butter? Or can it just melt the butter onto the toast? Does it need to learn to pick up a knife? Or can it have some sort of inbuilt knife-arm, like some sort of prison shiv pirate?

The more that you break down problems and define the issue that you're trying to solve, the easier it is to package your code into bite-sized chunks. The simpler the chunk, the easier it is to tackle.

So the next time that you're trying to make that "cross between Snapchat and Evernote", remember to break down the problem into solvable chunks.

12

Break someone else's code

One of the most important steps to take in order to make the jump from learner coder to a fully fledged programmer is understanding how to get help. Everyone needs help. Everyone, including those so-called “God Level Programmers”.

But what you do with the help will determine how fast you progress as a coder. On a site like StackOverflow, it can be very tempting to just copy and paste the code that someone has provided. Your program works exactly as you hoped it would and off you go on your merry programming ways. This exercise didn’t teach you anything other than code reliance. Because the next time you encounter the same problem but in a different situation, that same code snippet that someone provided may not work anymore. Then what do you do? You’re stuck.

That’s why there’s a rule in programming that says “never copypaste code that you don’t understand”. So what should you do when you’re confronted with a block of code that solves your problem but you have no clue how it works? Break it down.

Step 1 - Copy and paste the code into your program. (yes, yes, I know I just said not to do that, patience, patience).

Step 2 - Make sure that your program or application is functioning as expected. I.e. confirm that block of code really did solve your problem.

Step 3 - Delete the copy and pasted block of code line by line.

Step 4 - Each time you delete a line, check to see what's been broken. Does the app still run? What are the error codes? What has deleting that line of code done to your program?

Step 5 - Even if you think you know what a line of code does, delete it anyway. The most important task as a programmer is to always test your assumptions against the outcome. For the most enjoyable feeling as a programmer is for the real world to validate your assumptions. You know how nice it is when your boyfriend/girlfriend/husband/wife says those magical three words?

"You were right".

It's like that, but better.

Step 6 - Swap some of the lines around. Can the same functionality be achieved with a different order of lines? Why were they written in the order they were written in?

By breaking the solution code, line-by-line, you'll learn and understand what each line does and why it's been written. This is a far better way to use code from other people than just pasting it in and hoping for the best. Once you understand why each of those lines was necessary, the next time you encounter a similar problem, you'll be able to tease out the problem and solve it yourself.

Once you've mastered breaking code from StackOverflow, the next resource to target is GitHub. It's a tool used by programmers for collaboration but it is also one of the largest repositories of open source code.

So how can you use it to become a better programmer? Let's say that you want to make an Instagram clone. But unfortunately, you don't know how to do that. So you head over to github.com and search "Instagram" or "photo app".

Inevitably, there will be something written in Swift/Objective-C/Java that you can download and take a look at.

Think about the structure of their program. Take a look at all the classes, the constants, the interplay. Make some modifications to the code. Does it still work or have you broken it? Why did you break it? Is there a link that you didn't identify? Ask yourself a bunch of questions, learn through the Socratic method. Tear down the project and understand how it was built.

When you start getting really good at this, the next thing you can try is reverse engineering. Find a small project on GitHub made by a reputable programmer, download the app. Run it and see all of its functionality. Play around with it.

Then build it from scratch and once you're done, compare your code to their code. Are there efficiency gains that you could have made? Are there solutions to things you couldn't figure out? Now you're really getting into the big leagues.

That’s all for now folks. What are you still waiting for? The night’s still young! Code something, make something, learn something today! Head over to www.appbrewery.co to jump start your coding journey!

www.appberwery.co

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

BASICS

Print

Prints a string into the console.

```
print("Hello World")
```

Input

Prints a string into the console,
and asks the user for a string input.

```
input("What's your name")
```

Comments

Adding a # symbol in font of text
lets you make comments on a line of code.
The computer will ignore your comments.

```
#This is a comment
print("This is code")
```

Variables

A variable give a name to a piece of data.
Like a box with a label, it tells you what's
inside the box.

```
my_name = "Angela"
my_age = 12
```

The += Operator

This is a convient way of saying: "take the
previous value and add to it.

```
my_age = 12
my_age += 4
#my_age is now 16
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

DATA TYPES

Integers

Integers are whole numbers.

```
my_number = 354
```

Floating Point Numbers

Floats are numbers with decimal places.
When you do a calculation that results in a fraction e.g. 4 ÷ 3 the result will always be a floating point number.

```
my_float = 3.14159
```

Strings

A string is just a string of characters.
It should be surrounded by double quotes.

```
my_string = "Hello"
```

String Concatenation

You can add strings to string to create a new string. This is called concatenation. It results in a new string.

```
"Hello" + "Angela"
#becomes "HelloAngela"
```

Escaping a String

Because the double quote is special, it denotes a string, if you want to use it in a string, you need to escape it with a "\"

```
speech = "She said: \"Hi\""
print(speech)
#prints: She said: "Hi"
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

F-Strings

You can insert a variable into a string using f-strings.
The syntax is simple, just insert the variable in-between a set of curly braces {}.

```
days = 365
print(f"There are {days}
in a year")
```

Converting Data Types

You can convert a variable from 1 data type to another.
Converting to float:
`float()`
Converting to int:
`int()`
Converting to string:
`str()`

```
n = 354
new_n  = float(n)
print(new_n) #result 354.0
```

Checking Data Types

You can use the type() function to check what is the data type of a particular variable.

```
n = 3.14159
type(n) #result float
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

MATHS

Arithmetic Operators

You can do mathematical calculations with Python as long as you know the right operators.

```
3+2 #Add
4-1 #Subtract
2*3 #Multiply
5/2 #Divide
5**2 #Exponent
```

The += Operator

This is a convenient way to modify a variable. It takes the existing value in a variable and adds to it.
You can also use any of the other mathematical operators e.g. -= or *=

```
my_number = 4
my_number += 2
#result is 6
```

The Modulo Operator

Often you'll want to know what is the remainder after a division.
e.g. 4 ÷ 2 = 2 with no remainder
but 5 ÷ 2 = 2 with 1 remainder
The modulo does not give you the result of the division, just the remainder.
It can be really helpful in certain situations, e.g. figuring out if a number is odd or even.

```
5 % 2
#result is 1
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

ERRORS

Syntax Error

Syntax errors happen when your code does not make any sense to the computer. This can happen because you've misspelt something or there's too many brackets or a missing comma.

```
print(12 + 4))
  File "<stdin>", line 1
    print(12 + 4))
                  ^
SyntaxError: unmatched ')'
```

Name Error

This happens when there is a variable with a name that the computer does not recognise. It's usually because you've misspelt the name of a variable you created earlier.
Note: variable names are case sensitive!

```
my_number = 4
my_Number + 2
Traceback (most recent call
last): File "<stdin>", line 1,
NameError: name 'my_Number'
is not defined
```

Zero Division Error

This happens when you try to divide by zero, This is something that is mathematically impossible so Python will also complain.

```
5 % 0
Traceback (most recent call
last): File "<stdin>", line 1,
ZeroDivisionError: integer
division or modulo by zero
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

FUNCTIONS

Creating Functions

This is the basic syntax for a function in Python. It allows you to give a set of instructions a name, so you can trigger it multiple times without having to re-write or copy-paste it. The contents of the function must be indented to signal that it's inside.

```
def my_function():
    print("Hello")
    name = input("Your name:")
    print("Hello")
```

Calling Functions

You activate the function by calling it. This is simply done by writing the name of the function followed by a set of round brackets. This allows you to determine when to trigger the function and how many times.

```
my_function()
my_function()
#The function my_function
#will run twice.
```

Functions with Inputs

In addition to simple functions, you can give the function an input, this way, each time the function can do something different depending on the input. It makes your function more useful and re-usable.

```
def add(n1, n2):
    print(n1 + n2)

add(2, 3)
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

Functions with Outputs

In addition to inputs, a function can also have an output. The output value is proceeded by the keyword "return".
This allows you to store the result from a function.

```
def add(n1, n2):
    return n1 + n2

result = add(2, 3)
```

Variable Scope

Variables created inside a function are destroyed once the function has executed.
The location (line of code) that you use a variable will determine its value.
Here n is 2 but inside my_function() n is 3.
So printing n inside and outside the function will determine its value.

```
n = 2
def my_function():
    n = 3
    print(n)

print(n) #Prints 2
my_function() #Prints 3
```

Keyword Arguments

When calling a function, you can provide a keyword argument or simply just the value.
Using a keyword argument means that you don't have to follow any order when providing the inputs.

```
def divide(n1, n2):
    result = n1 / n2
#Option 1:
divide(10, 5)
#Option 2:
divide(n2=5, n1=10)
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

CONDITIONALS

If

This is the basic syntax to test if a condition is true. If so, the indented code will be executed, if not it will be skipped.

```
n = 5
if n > 2:
   print("Larger than 2")
```

Else

This is a way to specify some code that will be executed if a condition is false.

```
age = 18
if age > 16:
   print("Can drive")
else:
   print("Don't drive")
```

Elif

In addition to the initial If statement condition, you can add extra conditions to test if the first condition is false.
Once an elif condition is true, the rest of the elif conditions are no longer checked and are skipped.

```
weather = "sunny"
if weather == "rain":
   print("bring umbrella")
elif weather == "sunny":
   print("bring sunglasses")
elif weather == "snow":
   print("bring gloves")
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

and

This expects both conditions either side of the and to be true.

```
s = 58
if s < 60 and s > 50:
   print("Your grade is C")
```

or

This expects either of the conditions either side of the or to be true. Basically, both conditions cannot be false.

```
age = 12
if age < 16 or age > 200:
   print("Can't drive")
```

not

This will flip the original result of the condition. e.g. if it was true then it's now false.

```
if not 3 > 1:
   print("something")
   #Will not be printed.
```

comparison operators

These mathematical comparison operators allow you to refine your conditional checks.

```
> Greater than
< Lesser than
>= Greater than or equal to
<= Lesser than or equal to
== Is equal to
!= Is not equal to
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

LOOPS

While Loop

This is a loop that will keep repeating itself until the while condition becomes false.

```
n = 1
while n < 100:
   n += 1
```

For Loop

For loops give you more control than while loops. You can loop through anything that is iterable. e.g. a range, a list, a dictionary or tuple.

```
all_fruits = ["apple",
   "banana", "orange"]
for fruit in all_fruits:
   print(fruit)
```

_ in a For Loop

If the value your for loop is iterating through, e.g. the number in the range, or the item in the list is not needed, you can replace it with an underscore.

```
for _ in range(100):
   #Do something 100 times.
```

break

This keyword allows you to break free of the loop. You can use it in a for or while loop.

```
scores = [34, 67, 99, 105]
for s in scores:
   if s > 100:
      print("Invalid")
      break
   print(s)
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

continue

This keyword allows you to skip this iteration of the loop and go to the next. The loop will still continue, but it will start from the top.

```
n = 1
while n < 100:
   if n % 2 == 0:
      continue
   print(n)
#Prints all the odd numbers
```

Infinite Loops

Sometimes, the condition you are checking to see if the loop should continue never becomes false. In this case, the loop will continue for eternity (or until your computer stops it). This is more common with while loops.

```
while 5 > 1:
   print("I'm a survivor")
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

LIST METHODS

Adding Lists Together

You can extend a list with another list by using the extend keyword, or the + symbol.

```
list1 = [1, 2, 3]
list2 = [9, 8, 7]
new_list = list1 + list2
list1 += list2
```

Adding an Item to a List

If you just want to add a single item to a list, you need to use the .append() method.

```
all_fruits = ["apple",
   "banana", "orange"]
all_fruits.append("pear")
```

List Index

To get hold of a particular item from a list you can use its index number.
This number can also be negative, if you want to start counting from the end of the list.

```
letters = ["a", "b", "c"]
letters[0]
#Result:"a"
letters[-1]
#Result: "c"
```

List Slicing

Using the list index and the colon symbol you can slice up a list to get only the portion you want.
Start is included, but end is not.

```
#list[start:end]
letters = ["a","b","c","d"]
letters[1:3]
#Result: ["b", "c"]
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

BUILT IN FUNCTIONS

Range

Often you will want to generate a range of numbers. You can specify the start, end and step.
Start is included, but end is excluded:
start >= range < end

```
# range(start, end, step)
for i in range(6, 0, -2):
   print(i)

# result: 6, 4, 2
# 0 is not included.
```

Randomisation

The random functions come from the random module which needs to be imported.
In this case, the start and end are both included
start <= randint <= end

```
import random
# randint(start, end)
n = random.randint(2, 5)
#n can be 2, 3, 4 or 5.
```

Round

This does a mathematical round.
So 3.1 becomes 3, 4.5 becomes 5 and 5.8 becomes 6.

```
round(4.6)
# result 5
```

abs

This returns the absolute value.
Basically removing any -ve signs.

```
abs(-4.6)
# result 4.6
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

MODULES

Importing

Some modules are pre-installed with python e.g. random/datetime
Other modules need to be installed from pypi.org

```
import random
n = random.randint(3, 10)
```

Aliasing

You can use the as keyword to give your module a different name.

```
import random as r
n = r.randint(1, 5)
```

Importing from modules

You can import a specific thing from a module. e.g. a function/class/constant
You do this with the from keyword.
It can save you from having to type the same thing many times.

```
from random import randint
n = randint(1, 5)
```

Importing Everything

You can use the wildcard (*) to import everything from a module. Beware, this usually reduces code readability.

```
from random import *
list = [1, 2, 3]
choice(list)
# More readable/understood
#random.choice(list)
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

CLASSES & OBJECTS

Creating a Python Class

You create a class using the class keyword. Note, class names in Python are PascalCased. So to create an empty class

```
class MyClass:
   #define class
```

Creating an Object from a Class

You can create a new instance of an object by using the class name + ()

```
class Car:
   pass

my_toyota = Car()
```

Class Methods

You can create a function that belongs to a class, this is known as a method.

```
class Car:
   def drive(self):
      print("move")
my_honda = Car()
my_honda.drive()
```

Class Variables

You can create a varaiable in a class. The value of the variable will be available to all objects created from the class.

```
class Car:
   colour = "black"
car1 = Car()
print(car1.colour) #black
```

PYTHON CHEAT SHEET

100 DAYS OF CODE
COMPLETE PROFESSIONAL
PYTHON BOOTCAMP

The __init__ method

The init method is called every time a new object is created from the class.

```
class Car:
    def __init__(self):
        print("Building car")
my_toyota = Car()
#You will see "building car"
#printed.
```

Class Properties

You can create a variable in the init() of a class so that all objects created from the class has access to that variable.

```
class Car:
    def __init__(self, name):
        self.name = "Jimmy"
```

Class Inheritance

When you create a new class, you can inherit the methods and properties of another class.

```
class Animal:
    def breathe(self):
        print("breathing")
class Fish(Animal):
    def breathe(self):
        super.breathe()
        print("underwater")
nemo = Fish()
nemo.breathe()
#Result:
#breathing
#underwater
```